Collaborative and Distributed Management of Versioned Model-driven Software Product Lines

Felix Schwägerl and Bernhard Westfechtel
Applied Computer Science I, University of Bayreuth, Universitätsstr. 30, 95440, Bayreuth, Germany

Keywords: Model-driven Software Engineering, Software Product Line Engineering, Configuration Management.

Abstract: Software Product Line Engineering promises a significant gain in productivity, yet with the addition of new challenges one of which is the increased complexity of collaborative development. This paper presents an approach to distributed and collaborative product line engineering based on a filtered editing framework that has been extended by multi-user support. Using filtered editing, the product line is developed in a single-version view in a local workspace and transparently organized in a local repository. The contributed conceptual extension orchestrates the evolution and synchronization of different copies of the repository. This way, local transactions realized by check-out and commit are complemented by remote transactions using the operations pull and push. The proposed approach has been implemented as an extension to the model-driven tool SuperMod backed by a REST-based web service, evolving the toolset to a distributed product line version control system. We discuss relevant design decisions and illustrate our contributions by several examples.

1 INTRODUCTION

Software Product Line Engineering (*SPLE*) emphasizes the methodical development of families of similar software products based on organized reuse and mass customization (Pohl et al., 2005). To manage variability, commonalities and differences among products are captured in *feature models* (Kang et al., 1990). Specific products are then derived in a preferably automated way based on *feature configurations* that resolve variability. This requires to connect implementation artifacts to specific features or combinations thereof, as realized by *presence conditions* (Czarnecki and Kim, 2005). Literature distinguishes *positive variability*, where a minimal *core* is defined to which specific features are added (Apel and Kästner, 2009), from *negative variability*, where product variants are derived by removing unnecessary artifacts from a *superimposition*, as realized, e.g., by *preprocessor languages* (Kästner et al., 2008).

Over the last decade, the importance of *Model-Driven Software Engineering* (*MDSE*) (Völter et al., 2006) has grown. *Models* are considered as first-class artifacts from which application code is generated. MDSE involves well-defined languages such as the *Unified Modeling Language* (*UML*) (OMG, 2011b). The combination of SPLE and MDSE yields the integrating discipline *Model-Driven Product Line Engineering* (*MDPLE*) (Gomaa, 2004), which promises increased productivity by offering high-level abstractions to describe both variability and products.

Revision control deals with the problems of evolution and collaboration in software development. *Distributed version control systems* as *Git* (Chacon, 2009) remove the bottleneck of a centralized server having to orchestrate changes immediately, and add off-line support. In Git, the commands *check-out* and *commit* are used to synchronize the workspace with a locally persisted repository; the operations *pull* and *push* synchronize different copies of a repository.

In (Schwägerl et al., 2015a), a conceptual framework for the integration of revision control, SPLE, and MDSE has been developed. Its model-driven implementation has been presented in (Schwägerl et al., 2016; Schwägerl et al., 2015b). The tool *SuperMod* supports the incremental development of a model-driven software product line in a single-version workspace using a filtered editing model that fully automates variability management using the revision control metaphors *check-out* and *commit*. In order to select a specific version and to delineate the scope of a change to be committed, the user makes selections in a *feature model* in addition to the revision graph. To date, both the conceptual framework and SuperMod suffered from being applicable only in a single-user environment.

Schwägerl, F. and Westfechtel, B.
Collaborative and Distributed Management of Versioned Model-driven Software Product Lines.
DOI: 10.5220/0005971300830094
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 2: ICSOFT-PT*, pages 83-94
ISBN: 978-989-758-194-6

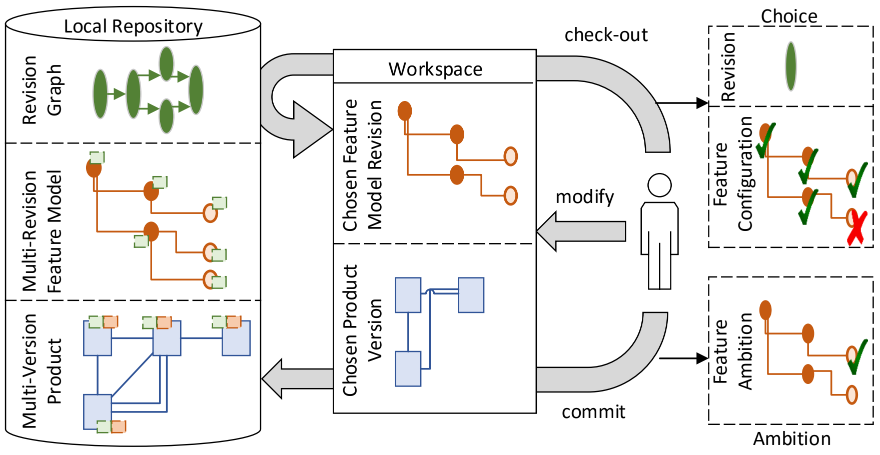

Figure 1: Filtered single-user MDPLE as originally defined by the conceptual framework and supported by SuperMod.

The contributions presented in the paper at hand remove this single-user restriction. Multiple copies of the product line are orchestrated by a *server-side repository*. Differences between multiple repositories are transferred using the operations *pull* and *push*. The theoretical contribution of this paper is an extension of the conceptual framework by *transaction support*, a distinction between *remote* (pull/push) and *local* transactions (check-out/commit), as well as a formal definition of the operations *pull* and *push* in the context of filtered MDPLE. The contributions have been implemented as a REST-based web service (Fielding, 2000), which reads and produces *symmetric deltas* (Rochkind, 1975). Being based on a model representation, the structure and calculation of deltas is considerably more complicated when compared to text-oriented version control systems like Git.

Section 2 revisits relevant parts of the conceptual framework. In Section 3, theoretical extensions based on collaborative revision graphs are presented, before client-server synchronization comes into focus (Section 4). Client-side implementation is discussed subsequently. Section 6 outlines related approaches focusing on filtered editing and collaborative SPLE.

2 FOUNDATIONS

The theoretical contributions of this paper are built upon a conceptual framework for the integration of historical and logical versioning, i.e., version control and SPLE. The original framework (Schwägerl et al., 2015a) supports *single-user filtered product line engineering*, combining version control concepts *workspace*, *repository*, *revision graph* with the SPLE concepts *feature model* and *(partial) feature configuration*. Moreover, the version control operations *check-out* and *commit* are extended by the specification of *choices* and *ambitions* (see Figure 1). More than ordinary version control systems, the underlying editing model encourages iterative software development. Each iteration consists of three steps:

Check-out. The user performs a *version selection* (a *choice*) in the local repository. In the revision graph, the choice comprises a single *revision*. In the feature model, a *feature configuration* has to be specified. The selected revision of the feature model and the selected version (revision *and* feature configuration) of the product are filtered and loaded into the workspace.

Modify. The user applies changes to the single-version product and/or to the feature model.

Commit. The changes are written back to the local repository. The user is prompted for a *feature ambition*, a partial selection in the feature model describing the set of variants to which the change applies. Visibilities of versioned elements are updated automatically, such that changes to the product become visible only for the current revision, and only for variants included in the partial feature configuration specified as ambition.

The contents of the local repository have been formally defined based on notions introduced by the *Uniform Version Model* (*UVM*) (Westfechtel et al., 2001). Moreover, the internal data structure has been defined by a set of metamodels (Schwägerl et al., 2016) serving for the model-driven realization of SuperMod (Schwägerl et al., 2015b). Following remarks recapitulate the theoretical foundations of the framework.

A *repository* R is a triple consisting of a *product space* P, a *version space* V, and a mapping *map* defining which elements of the version space are included in which version.

$$R = (P, V, map) \tag{1}$$

Elements of V are externally shown as higher-level abstractions such as *revisions* or *features* and internally mapped to low-level elements (*options* and *version rules*, see below). P contains elements of both the feature model and the domain model (which theoretically may be anything representable by set theory, e.g., text files or EMF models).

An *option* represents a (logical or temporal) property of a software product that can be either present or absent. The global *option set* is used to define the set of versions V *intensionally*.

$$O = \{o_1, \ldots, o_n\} \tag{2}$$

Each revision and each feature is mapped to one option transparently.

A *choice* is a conjunction over all options, each of which occurs in either positive or negated form:

$$c = b_1 \wedge \ldots \wedge b_n,\ b_i \in \{o_i, \neg o_i\}\ (i \in \{1, \ldots, n\}) \tag{3}$$

Choices uniquely identify elements of the version space V. In SuperMod, they are inferred automatically from a user-based selection of a revision and a completely bound feature configuration.

An *ambition* is an option binding that allows for unbound options and thus denotes a *subset* of the version space V.

$$a = b_1 \wedge \ldots \wedge b_n,\ b_i \in \{o_i, \neg o_i, true\} \tag{4}$$

Ambitions are inferred from a partial user-based selection in the feature model. Thus, an ambition may leave options unbound in order to describe a *set* of versions to which a change is applied.

Version rules are boolean expressions over a subset of options. The *rule base* $\mathcal{R}$ is a conjunction of version rules ρ_i all of which have to be satisfied by an option binding in order to be consistent:

$$\mathcal{R} = \rho_1 \wedge \ldots \wedge \rho_m \tag{5}$$

In SuperMod, version rules are managed transparently and kept invisible to the user. E.g., for a sequence of revisions, a rule of the form $r_i \Rightarrow r_{i-1}$ is introduced; the mapping between the revision graph and rule base constraints is redefined in Section 3. The semantics of feature models is mapped to propositional logical constraints (Schwägerl et al., 2015a).

A choice c is *strongly consistent* if it implies the rule base $\mathcal{R}$:

$$c \Rightarrow \mathcal{R} \tag{6}$$

In the case of ambitions, only the *existence* of a consistent version is required. An ambition is *weakly consistent* if it overlaps with the rule base:

$$\mathcal{R} \wedge a \neq false \tag{7}$$

Each element e of the product space carries a *visibility* $v(e)$, a boolean expression over the variables defined in the option set. This way, visibilities implement the function *map* declared in (1). An element e is *visible* under a choice $c \in V$ if its visibility is implied by the choice, i.e., it evaluates to *true* given the option bindings of the choice:

$$c \Rightarrow v(e) \tag{8}$$

The *filter* operation is applied during check-out. Filtering a product space P by a choice c can be realized as a conditional copy, where elements e that do not satisfy the choice are omitted.

$$filter(P, c) = P \setminus \{e \in P \,|\, c \not\Rightarrow v(e)\} \tag{9}$$

During commit, the visibility of product space elements is updated such that inserted elements become visible in all choices implied by the specified ambition a, and deleted elements remain invisible there. Accordingly, the *updated visibility* $v'(e)$ is defined:

$$v'(e) = \begin{cases} v(e) & \text{if } e \text{ remains unmodified.} \\ a & \text{if } e \text{ was newly inserted.} \\ v(e) \vee a & \text{if } e \text{ was re-inserted.} \\ v(e) \wedge \neg a & \text{if } e \text{ was deleted.} \end{cases} \tag{10}$$

For changes to the actual product, the ambition a referred to above binds both revision and feature options. However, for changes to the feature model, all feature options are stripped from a, since the feature model is merely versioned by the revision graph.

3 COLLABORATIVE SPLE

In this section, we explain extensions to the conceptual framework providing the basis for the definition of the synchronization mechanisms in Section 4, which enable collaborative MDPLE in the end.

3.1 Design Decisions

The metamodel and mapping explained subsequently are justified by the following design decisions.

Optimistic Synchronization. The conceptual framework does not require locks to coordinate collaborative versioning. Rather, developers may concurrently modify their copies of the repository.

Enforcement of a Linear Version History. Each revision committed to the repository is a successor of the latest revision available, the *head*. By intention, the conceptual framework does not support branches; as a replacement, variable parts of the product should be modeled using features.

Remote and Local Transactions. Different copies of the repository are synchronized not after each commit, but only at user-defined synchronization points. For this purpose, the extended framework distinguishes between remote transactions, which are started with a *pull* and finished with a *push* operation, from local transactions, which realize an *update-modify-commit* iteration. This reduces synchronization overhead and enables offline product line development. Moreover, it is not necessary to start a remote transaction explicitly; rather, all necessary bookkeeping steps are enforced after each *pull* transparently.

3.2 Metamodel and Rule Base Mapping

Figure 2 and Table 1 illustrate an Ecore metamodel for revision graphs as well as a mapping to low-level rule base elements for instances thereof. Both the figure and the table redefine the mapping from (Schwägerl et al., 2016, Table 2 and Figure 7).

A *revision graph* is a container for *public revisions*, which represent remote transactions and in turn contain *private revisions*, which express local transactions. Both inherit from an abstract base class Revision that defines an attribute for the revision number – private revisions are externally displayed using the nesting public revision as qualifier – and additional commit details (date, message, user). Secondly, a generic predecessors/successors relationship is defined. Last, references {private|public}{Init|Head} memorize corresponding revisions.

In addition to the structure of the revision graph, the metamodel shown in Figure 2 also defines references to low-level rule base elements (see Section 2), which are transparently derived using the transformation patterns defined in Table 1. Private revisions are mapped to a revision option in a straightforward way (pattern 2), whereas public revisions are mapped to two options, starting (1) and finishing (3) the transaction. Pattern (4) ensures that the initial revision is selected. In general, predecessor/successor relationships between two revisions r_{i-1} and r_i are mapped to version rules of the form $r_i \Rightarrow r_{i-1}$, i.e., when a revision is selected, all predecessors are enforced to be selected, too. There are five cases in which such relationships are created: (5) After finishing a remote transaction, a succeeding remote transaction is started immediately. (6) All private revisions follow the start option of the parent public revision. Pattern (7) ensures successorship of private revisions, and (8) is instantiated before finishing a remote transaction. Pattern (9) enforces a linear version history by automatically merging with remotely finished transactions.

3.3 Example

To illustrate the dynamic behavior of a collaborative revision graph, Figure 2 shows a sample version history involving two fictional developers, Alice and Bob. Alice creates the repository, which transparently introduces a revision option r_0 and a corresponding initial public revision rule (patterns 1 and 4 in Table 1). Next, she performs an initial commit, which introduces a nested private revision $r_{0.0}$ as successor of r_0 (6). By finishing the transaction through the *push* operation, $r_{0.\infty}$ (3) and rule $r_{0.\infty} \Rightarrow r_{0.0}$ (8) are introduced transparently. The next transaction is started automatically, introducing r_1 (1) and $r_1 \Rightarrow r_{0.\infty}$ (5). Concurrently, Bob clones the repository, which also starts a write transaction (option r_2, rule $r_2 \Rightarrow r_{0.\infty}$). Both developers commit private revisions to their local repositories and then finish their current remote transaction. Bob is the first to push, r_2 is closed, and r_3 is started immediately. Thereafter, Alice tries to push and receives an *out of date* error, enforcing a *pull* until the current head, such that the incoming $r_{2.\infty}$ is merged with $r_{1.\infty}$. When pushing again, pattern (9) is instantiated, adding $r_{1.\infty} \Rightarrow r_{2.\infty}$ to the rule base transparently. Moreover, a new remote transaction is begun by introducing r_4 and $r_4 \Rightarrow r_{1.\infty}$. Alice finishes the transaction in a straightforward way. After that, Bob starts his work forgetting to pull r_4. Thus, he receives an *out of date* error when trying to push $r_{3.\infty}$. The incoming $r_{4.\infty}$ is automatically merged before finishing r_3. Please note: Despite the underlying mechanisms defined in Section 3.2 and illustrated in this example being intrinsically complex, the fictional users are only exposed to familiar revision control metaphors (*commit*, *push*, *pull*, *out of date*).

4 DISTRIBUTED SPLE

So far, we have completely ignored the feature model and the product, both being versioned by the collaborative revision graph. Likewise, the evolution of the version graph itself must be orchestrated. Figure 4 complements Figure 1 by the architecture of the synchronization component, which is detailed below.

4.1 Design Decisions

The list of design decisions is extended by items referring to the synchronizing operations *pull* and *push*:

Symmetric deltas (Rochkind, 1975) correspond to a *superimposition* of all existing revisions, assigning *version identifiers* to each element. Using *directed deltas* (Tichy, 1985), *change sequences* re-

Table 1: Detailed mapping between collaborative revision graphs and low-level version rules.

Nr.	Pattern	Transformation	Metamodel Ref.
1	public revision r_i	option r_i	revisionOption
2	private revision $r_{i.j}$ nested in public revision r_i	option $r_{i.j}$	revisionOption
3	finished public revision r_i	option $r_{i.\infty}$	finishedOption
4	initial public revision r_0	rule r_0	initPublicRule
5	public revision r_i as successor of finished public revision r_{i-1}	rule $r_i \Rightarrow r_{i-1.\infty}$	predecessorRules
6	initial private revision $r_{i.0}$ nested in public revision r_i	rule $r_{i.0} \Rightarrow r_i$	initPrivateRule
7	private revision $r_{i.j}$ as successor of private revision $r_{i.j-1}$	rule $r_{i.j} \Rightarrow r_{i.j-1}$	predecessorRules
8	finished public revision r_i with private head $r_{i.h}$	rule $r_{i.\infty} \Rightarrow r_{i.h}$	predecessorRules
9	public revision r_i with incoming r_c causing *out of date*	rule $r_{i.\infty} \Rightarrow r_{c.\infty}$	predecessorRules

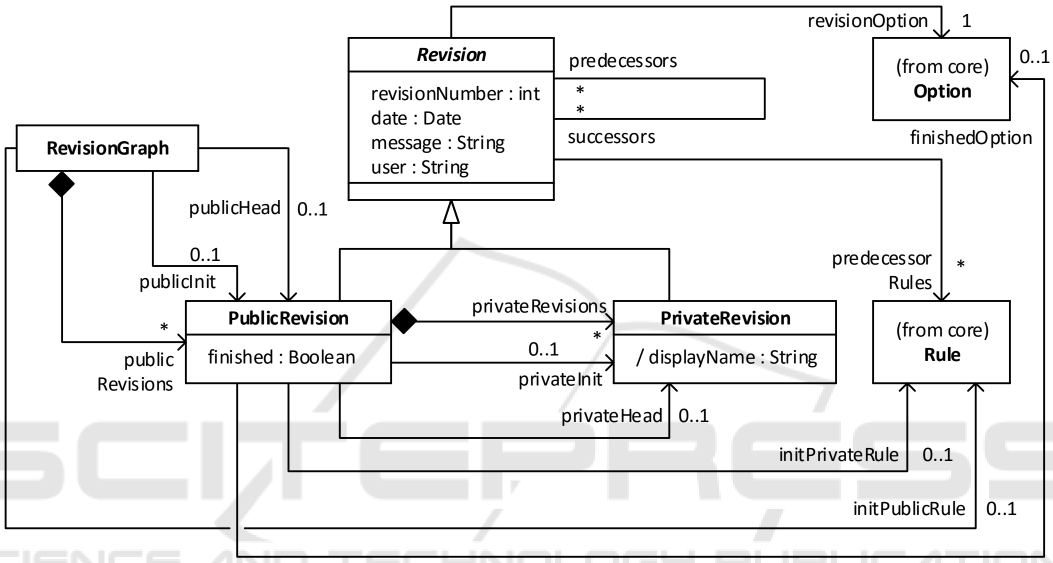

Figure 2: Metamodel for collaborative revision graphs. Classes core::Option and core::Rule refer to options and version rules, respectively, as introduced in Section 2.

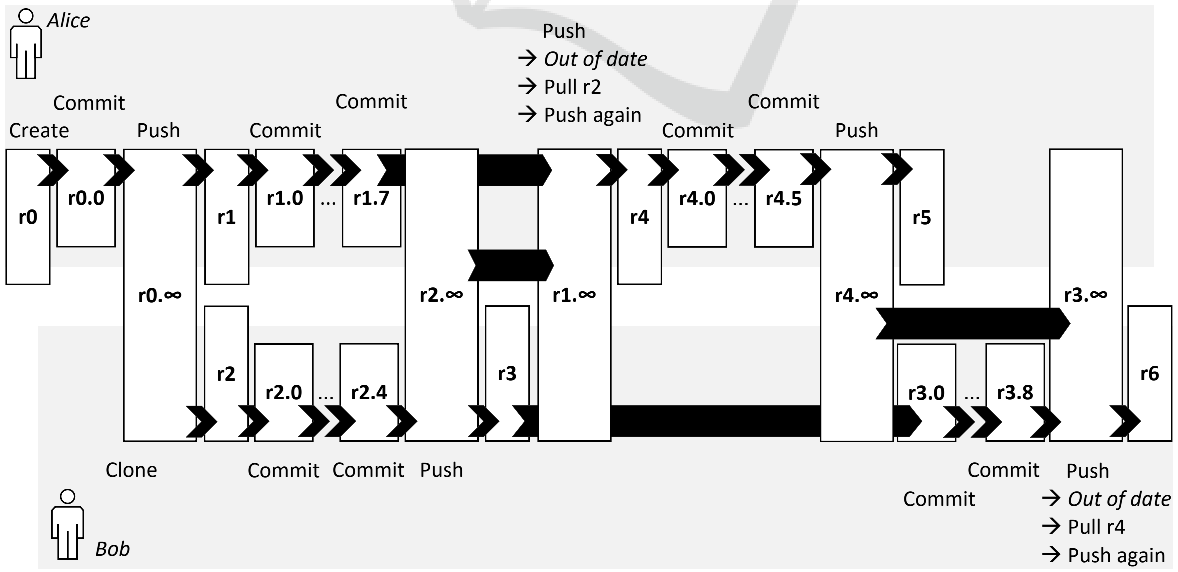

Figure 3: Example revision graph illustrating the interplay between public and private revisions. Boxes denote revisions with corresponding options. Black arrows starting at r_x and ending at r_y denote an instantiation of a predecessor rule $r_y \Rightarrow r_x$.

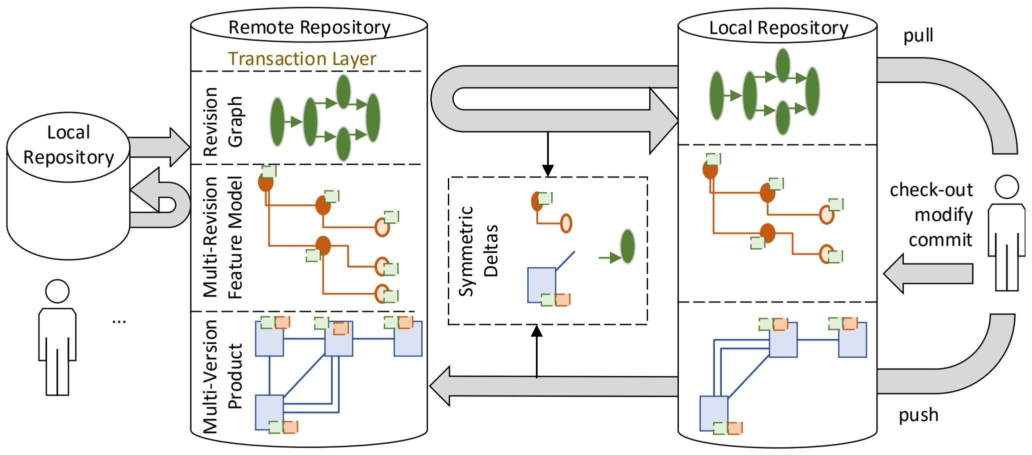

Figure 4: Synchronization of multiple local copies of the repository using a server-side repository and symmetric deltas.

construct product revisions on demand, ensuing from a fully persisted *baseline* revision. Although directed deltas consume less space, we decided for symmetric deltas, since they harmonize better with filtered editing and negative variability.

Cross-references. The conceptual framework abstracts from specific product space types; however, it is assumed that *cross-references* between elements of the product space need to be managed. Besides, references between higher-level version models and low-level rule base elements exist (e.g., references to core::Option and core::Rule in Figure 2). Last, through *visibilities*, product space elements refer to options defined in the version space. It is therefore important to correctly handle references from elements which are part of the delta to elements not included there.

Three-way Merging. According to our chosen optimistic versioning strategy, concurrent modifications may occur. Our solution relies on an automated three-way merging procedure that involves the user as late as possible, i.e., when trying to check-out a product that contains conflicts. Due to space restrictions, this paper does not deal with product conflict resolution in the workspace.

4.2 Low-Level Transaction Layer

In order to keep track of the elements modified by the current write transaction, a simple *transaction layer* has been added to the conceptual framework. It is realized by the following extensions:

- Each local repository carries a *read transaction number*, which indicates the revision number of the most recently pulled public revision, and a *write transaction number*, which equals the revision number of the public revision organizing the current remote transaction.
- Each element (i.e., product space element or version space element such as revision) carries a *transaction number* that indicates its most recent modification (i.e., insertion, deletion, or modification of a child element). The *visibility update* function (cf. Equation 10 in Section 2) is modified such that it assigns the current write transaction number to (re-)inserted and deleted elements. Furthermore, transaction numbers are updated during the creation of public and private revisions accordingly.
- The server-side repository manages a global *transaction log* to which transaction starting and finishing events are appended. Also from this log, new public revision numbers are generated. The transaction log for the example from Figure 3 is `o0 c0 o1 o2 c2 o3 c1 o4 c4 o5 c3 o6`, where `o` denote opened, `c` closed transactions. Transactions are opened, but not necessarily closed, in numerical order.

4.3 Symmetric Delta Calculation

The increments transferred along with push and pull operations are symmetric deltas; they are here considered as subsets of the product space, consisting of the feature model and the primary product. As pointed out in Section 4.1, special emphasis is put on the consistency of cross-references between different types of elements. Algorithm 1 describes a generic procedure for calculating symmetric deltas as a projection of the product space based on a given transaction number. The delta is returned as the sub-set of the product space that contains elements carrying the specified transaction number (Δ_0), elements referenced by elements in Δ_0 or by their visibilities (Δ_1), and the transitive closure ($^+$) over the containers (Δ_2) of elements in Δ_0 or Δ_1.

Algorithm 1: Symmetric Delta Projection.

```
procedure ProjectDelta($P$, tNr)
    $\Delta_0 \leftarrow$ elements of $P$ carrying tNr
    $\Delta_1 = \emptyset$
    for $e_0 \in \Delta_0$ do
        for $e_1 \in$ elements cross-referenced by $e_0$ do
            if $e_1 \notin \Delta_0$ then
                $\Delta_1 \leftarrow \Delta_1 \cup \{e_i\}$
        for $o_1 \in$ options referred to in $v(e_0)$ do
            $ve_1 \leftarrow$ high-level concept of $o_1$
            if $ve_1 \notin (\Delta_0 \cup \Delta_1)$ then
                $\Delta_1 \leftarrow \Delta_1 \cup \{ve_1\}$
    $\Delta_2 = \emptyset$
    for $e_{01} \in (\Delta_0 \cup \Delta_1)$ do
        for $e_2 \in container^+(e_{01})$ do
            if $e_2 \notin (\Delta_0 \cup \Delta_1 \cup \Delta_2)$ then
                $\Delta_2 \leftarrow \Delta_2 \cup \{e_2\}$
    return $\Delta_0 \cup \Delta_1 \cup \Delta_2$
```

4.4 Raw vs. Three-Way Merging

The counterpart to delta calculation is *merging*. This operation appends an incoming delta to a product space P, which is an "add-only" structure; no element is ever permanently removed.

Element Merging. When assuming a purely set-theoretic definition of the product space P, a merged product space P' including an incoming delta Δ can be easily calculated as

$$P' = P \cup \Delta \tag{11}$$

However, element merging imposes several new challenges such as the identification of "equal" elements, which are rather specific to the implementation and therefore discussed in Section 5.3.

Moreover, conflicts arise when two "equal" elements e carry different visibilities in P and Δ. Special attention is paid here, since visibilities encode insertions and deletions of elements. We have to distinguish between *raw merging* (an incoming delta is to be integrated into the local repository) and *three-way merging* (the local repository contains outgoing changes which conflict with an incoming delta, i.e., the local repository is *out of date*).

Raw Visibility Merging. In case no outgoing changes exist to redefine the visibility of an element $e \in P$, it is assumed that the "more recent" visibility is transferred from the delta to the local version.

$$v'(e) = \begin{cases} v_\Delta(e) & \text{if } e \in \Delta \text{ and } v_\Delta(e) \text{ is defined.} \\ v_P(e) & \text{otherwise.} \end{cases} \tag{12}$$

Three-Way Visibility Merging. If the local repository is *out of date*, the visibilities of its elements in P may conflict with the visibilities defined in Δ. In this case, the common base version b must be considered:

$$v'(e) = \begin{cases} v_P(e) & \text{if } e \notin \Delta \text{ or } v_\Delta(e) \text{ is undefined.} \\ v_\Delta(e) & \text{if } e \notin P \text{ or } v_P(e) \text{ is undefined.} \\ \mu(v_b(e), v_P(e), v_\Delta(e)) & \text{otherwise.} \end{cases} \tag{13}$$

Here, $v_b(e)$ is the visibility of e in the common *base* version of the considered revisions in the revision graph. Along with this, $\mu(v_b, v_1, v_2)$ is the *three-way visibility merge function* (Westfechtel, 2014):

$$\mu(v_b, v_1, v_2) = (v_1 \wedge v_2) \vee (v_1 \wedge \neg v_b) \vee (v_2 \wedge \neg v_b) \tag{14}$$

In case there is no difference between v_b and v_1, this function evaluates to v_2 (conversely for swapping v_1 and v_2). Otherwise, insertions and deletions made effective by the visibility update function (Equation 10) are combined.

4.5 Definition of Pull and Push

Based upon the definitions from the preceding subsections, we now formally define the operations *pull* and *push* referred to in Figure 4.

Pull. This operation fetches incoming changes from the remote and appends them to the local repository.

1. The current *read transaction number* is sent to the server as part of a pull request.
2. The server analyzes the *transaction log* to find all write transactions closed since the transmitted read transaction number.
3. If there are no newly closed transactions, cancel.
4. For each of the new transactions, a *symmetric delta* is calculated using Algorithm 1.
5. The deltas are combined and sent to the client.
6. The client merges the incoming delta with the local repository as shown in Section 4.4.
   (a) In case there are no outgoing changes, *raw merging* is applied. Then, a *check-out* is enforced in order to transfer incoming changes to the workspace.

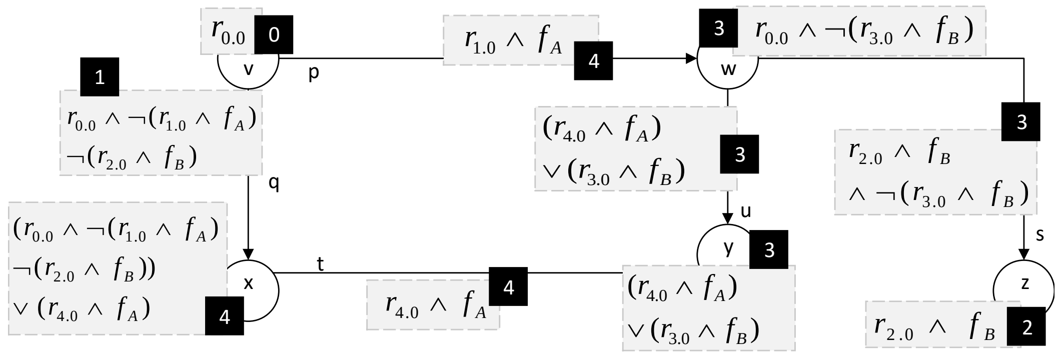
$r_{0.0}$
0
v
p
$r_{1.0} \wedge f_A$
4
3
$r_{0.0} \wedge \neg(r_{3.0} \wedge f_B)$
w
1
$r_{0.0} \wedge \neg(r_{1.0} \wedge f_A)$
$\neg(r_{2.0} \wedge f_B)$
q
$(r_{4.0} \wedge f_A)$
$\vee (r_{3.0} \wedge f_B)$
3
3
$r_{2.0} \wedge f_B$
$\wedge \neg(r_{3.0} \wedge f_B)$
u
s
$(r_{0.0} \wedge \neg(r_{1.0} \wedge f_A)$
$\neg(r_{2.0} \wedge f_B))$
$\vee (r_{4.0} \wedge f_A)$
4
x
t
$r_{4.0} \wedge f_A$
4
y
3
$(r_{4.0} \wedge f_A)$
$\vee (r_{3.0} \wedge f_B)$
z
$r_{2.0} \wedge f_B$
2

Figure 5: Product space of the running example, including visibilities (grey boxes), and transaction numbers (black boxes).

   (b) In case there are outgoing changes (i.e., pending commits in an unfinished public revision), *three-way merging* is applied, and the local workspace is checked-out anew.

7. The client's *read transaction number* is updated to the server's latest read transaction number.

Push. This operation finishes a remote transaction and transfers all committed changes to the server.

1. A *pull* is enforced. For incoming changes, corresponding public revisions are memorized.
2. The current remote transaction is finished, instantiating patterns 3 and 8 from Table 1. Revision details (user, date, commit message) are applied.
3. A *symmetric delta* is calculated using Algorithm 1 and the current write transaction number.
4. The delta is sent to the server as part of a push request.
5. The server *raw-merges* the incoming delta with the remote repository as shown in Section 4.4.
6. The *transaction log* is updated by closing the client's write transaction and opening a new write transaction immediately.
7. The new *write transaction number* is transferred to the client.
8. In the client revision graph, a new public revision is introduced, instantiating patterns 1 and 5 from Table 1. In addition, for each incoming revision memorized in step 1, pattern 9 is enforced.

4.6 Example

We refer back to the example introduced in Section 3.3 with a focus on delta calculation and three-way merging. To facilitate understanding the example, we make some simplifications: (1) One change is performed per commit. (2) For each remote transaction, only the first commit is shown. (3) The feature model consists of two optional features f_A and f_B, not imposing any further constraints. (4) The product space is represented as a simple *bubbles and arcs* model. Both bubbles and arcs carry a *label*, which also serves as equality criterion.

The final state of the remote repository's product space, including visibilities, is depicted in Figure 5. Below, the performed changes (each followed by *push* and *commit*) are ordered by push date.

Revision 0. Alice initializes the repository, creating the entire feature model and an initial product consisting of bubbles v, w, x and arc q. She does not associate her change with a specific feature, resulting in ambition *true*. The transferred delta contains $\{f_A, f_B, v, w, x, q\}$.

Revision 2. Ensuing from revision 0, Bob removes q and x, and inserts arc s and bubble z, all under ambition f_B. The delta is $\{f_B, q, x, s, z\}$.

Revision 1. Alice concurrently removes arc q and bubble x, and adds arc p, under ambition f_A. The delta is $\{f_A, q, x, p, w\}$. w is included as an element cross-referenced by the inserted arc p.

Merging Revisions 3 and 4. Bob was the first to finish, so Alice receives an *out of date* error when trying to push. When pulling to r_2, the incoming delta is merged. The visibility of arc q and bubble x has been concurrently modified, such that $v_R = r_{0.0} \wedge \neg(r_{1.0} \wedge f_A)$ and $v_\Delta = r_{0.0} \wedge \neg(r_{2.0} \wedge f_B)$. Using the base visibility $v_b = r_{0.0}$, the three-way visibility merge function (Equation 14) evaluates to $v'(q) = v'(x) = r_{0.0} \wedge \neg(r_{1.0} \wedge f_A) \wedge \neg(r_{2.0} \wedge f_B)$.

Revision 4. Alice re-inserts bubble x and adds bubble y, arcs t and u under ambition f_A. The delta is $\{f_A, y, t, u, w, x\}$

Revision 3. Ensuing from revision 1, Bob concurrently deletes bubble w and arc s, and inserts u and y, under ambition f_B. The transferred delta is calculated as $\{f_B, w, s, u, y, z\}$.

Merging Revisions 1 and 2. Bob receives an *out of date* error when trying to push. As a consequence, the incoming delta of r_4 is merged. This time, the visibilities of arc u and bubble y are merged: $\mu(true, r_{4.0} \wedge f_A, r_{3.0} \wedge f_B) = (r_{4.0} \wedge f_A) \vee (r_{3.0} \wedge f_B)$. Moreover, the visibility of q is raw-merged.

In this example, we have focused on the internals of low-level transaction management, delta calculation, visibility update, and visibility merging. It is important to notice that the end users Alice and Bob never get bothered with these details, since they operate in their local workspaces using the VCS abstractions *commit*, *push*, and *pull*, which hide complexity from them. The example also shows that visibility merging produces the intuitive result; the concurrent deletions of q and t as well as the concurrent insertions of u and y have been combined, while the logical scopes – the ambitions specified by the users – have been maintained. During delta calculation, the smallest subset necessary to describe the changes in a self-contained way is yielded, reducing synchronization traffic to a minimum.

5 IMPLEMENTATION

The extensions to the conceptual framework presented in Sections 3 and 4 have been implemented as plug-ins for the research prototype SuperMod (Schwägerl et al., 2015b), removing its single-user restriction. SuperMod has been developed in a model-driven way using the *Eclipse Modeling Framework* (*EMF*) (Steinberg et al., 2009). The tool is publicly available for evaluation purposes; installation instructions and an accompanying screencast video are referenced at the end of this paper. Figure 6 shows that SuperMod's client has been realized as a *team provider* plug-in for the Eclipse IDE. This section outlines the most important implementation details that enable collaborative MDPLE.

5.1 Server Architecture

The server component of SuperMod has been realized as a *REST*-based (*Representational State Transfer*) (Fielding, 2000) web-service hosted on an Apache Tomcat 7 servlet container. The remote repository is an instance of the SuperMod metamodel (Schwägerl et al., 2016) and persisted in the XMI (*XML Metadata Interchange*) (OMG, 2011a) format in the server's file system. Additionally, the *transaction log* (see Section 4.2) is stored within a simple text file.

Read and write transactions are protected by a server-side lock file, ensuring that push and pull operations never coincide. *Atomicity* of push and pull is ensured using EMF's resource framework. Only after finishing a write transaction successfully, all modified EMF resources are saved at a time; otherwise, in case the transaction has been canceled, all modifications are rolled back.

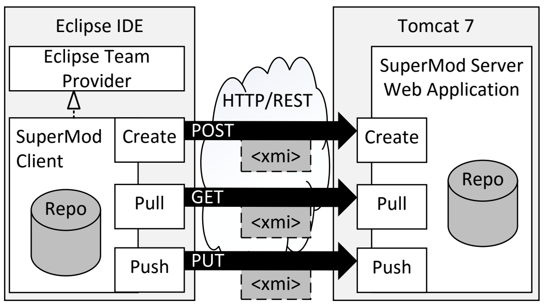

Figure 6: Coarse architecture of communication between client (left) and server (right).

5.2 Mapping VCS to REST Operations

The architectural style REST is based on the *Hyper Text Transfer Protocol* (*HTTP*). Modifiable and non-modifiable data is qualified by means of *resources* encoded in hierarchical *URLs*. The content transferred with and obtained by a HTTP request (i.e., the *entity*) may have binary or text format. Moreover, different *methods* are distinguished: GET serves for reading, POST for creation, and PUT for modification.

Below, the mapping of the distributed VCS operations *create*[1], *pull*, and *push*, to HTTP methods is explained, assuming that the servlet is running at `http://root.url/supermod/`. These methods are invoked by the SuperMod client transparently when the user demands the corresponding operation.

Create. The initialization of a new remote repository is available as a POST method on `.../supermod/repo/path/create?user=X`, where the entity contains the XMI serialization of the entire initial repository. Variable repo/path distinguishes several independent repositories. The initial write transaction number 1 is returned.

Pull. Server-side changes are requested using GET `...supermod/repo/path/pull?user=X&read TransactionNr=Y`, where the query entity is kept empty. An XMI-serialized delta is returned that includes changes referring to transactions closed after Y. [2]

Push. Transferring client-side changes to the remote repository is realized as a PUT method on

[1]Conceptually trivial, therefore omitted in Section 4.

[2]$Y = -1$ requests the entire repository (*clone*).

`.../supermod/repo/path/push?user=X&readTransactionNr=Y&writeTransactionNr=Z`. In case *Y* does not equal the most recently closed transaction number, the repository is *out of date*, returning an response code that signalizes to the client that a pull must be performed first. Otherwise, the local repository is merged with the delta XMI-serialized in the entity; a new write transaction is started, whose number is returned.

5.3 Product Space Merging

As aforementioned, *merging* the client or remote repository with incoming changes consists of *visibility merging*, which has been implemented as explained in Section 4.4, and *element merging*, details of which depend on the specific representation of the product and version space.

Being based on EMF, SuperMod specializes the set-theoretic definition of the repository (see Equation 1) by representing both version and product space as a *containment tree* of elements. Likewise, non-hierarchical cross-references are allowed between elements. Sequences, which occur, e.g., as text files or as values of ordered structural features in EMF models, are represented as a *directed graph*.

Product space merging has been implemented as a recursive procedure beginning at the model roots. To decide on equality of tree elements, specific *matching* strategies are applied. For complex elements such as EMF classes or features of the feature model, we rely on unique object identifiers. Atomic values such as EMF attribute values or lines in plain text files are matched by string equality. In case two elements of the repository and delta version are considered as equal, they are merged, and so are their children recursively. During sequence merging, *transitive* relationships between vertices, encoded in paths, are maintained (Schwägerl et al., 2015c).

6 RELATED WORK

The original contribution of the current paper is the addition of multi-user support to both a conceptual framework for the integration of MDSE, SPLE, and version control (Schwägerl et al., 2015a) and its implementation SuperMod (Schwägerl et al., 2015b).

Reconciliation of revision control and variability management has originated in the *Uniform Version Model* (*UVM*) (Westfechtel et al., 2001), a conceptual predecessor of our framework. UVM itself is derived from *change-oriented versioning* (Munch, 1993) (CoV), which was implemented in the relational data base version control system EPOS. The design of a collaborative component for EPOS was presented in (Conradi and Malm, 1991). EPOS and SuperMod have in common a low-level transaction layer assigning transaction numbers to versioned elements. Furthermore, the notions of choices, ambitions, and visibilities are shared, but CoV (and UVM) require that the ambition must be fixed at check-out time. As another difference, in EPOS, transactions may be nested in a tree; user modifications are allowed only in leaf transactions. There is technically no difference between *commit* and *push*. Synchronization is orchestrated by a *propagation* mechanism between different workspaces. In contrast, SuperMod separates filtered editing (check-out/commit) from synchronization (pull/push) and transfers symmetric deltas only on demand, leading to a less disruptive workflow. EPOS offers (but is not restricted to) *pessimistic synchronization* by inhibiting multiple transactions having overlapping ambitions. This cannot be applied in SuperMod, where the ambition is specified at commit time. Conversely, EPOS does not explicitly address three-way merging, such that "the last update wins".

UVM's *layered architecture* (Westfechtel et al., 2001) extends the low-level transaction layer of EPOS. Transactions are manifested in visibilities by means of *transaction options*, which coarsely correspond to *public revision options* but are never shown to the user in the form of a revision graph as realized in SuperMod. Similarly, the variant management provided by UVM is more complex than in SuperMod, since the user is exposed to *variant options* directly.

The distinction between local and remote transactions used in this paper was borrowed from distributed revision control systems such as *Git* (Chacon, 2009), which extend centralized revision control systems such as Subversion (Collins-Sussman et al., 2004), where, speaking in Git metaphors, each *commit* enforces a *pull*. SuperMod and the underlying framework advance the state of the art in distributed revision control in two ways. On the one hand, support for logical variants has been added by integrating SPLE metaphors; by providing the possibility of re-combining different features (i.e., *intensional versioning*), our approach goes considerably beyond *branches* and *forks* offered by Git. On the other hand, *models* are versioned as structured artifacts rather than taking their text-based serialization as a basis for line-oriented version control.

Few SCM systems have been extended with partial support for SPLE, or vice versa. Adele (Estublier and Casallas, 1994) has logical variants built into its object-oriented data model as symmetric deltas, which are exposed to the user. Temporal variability is realized by a versioning layer on top, enabling collaborative development. Conversely, in (Krueger, 2002), an extension to the SPLE toolchain *BigLever* is presented that deals with controlling the variability of different diverging products. In both approaches, logical and cooperative versioning are not integrated at the same conceptual level.

Software Product Line Evolution deals with common problems occurring during the management of the life-cycle of software product lines, for instance propagating changes from the variability model to the platform (Laguna and Crespo, 2013). Product and variability model are typically represented as artifacts on the same conceptual level, i.e., there is no "versioning" relationship as in software configuration management. In addition, there exists no approach or tool offering the full range of capabilities of state-of-the-art version control systems. The approach by (Thao, 2012) relies on change propagation at configuration level. In particular, the approach facilitates unfiltered collaborative SPLE. Though, change propagation, as opposed to state-based versioning, requires deep integration into the toolchain. The authors also present a solution for semi-automatic backward propagation of product-specific changes to the product line. However, instead of connecting visibilities of updated elements to an *ambition* as provided by SuperMod, manual visibility updates are required.

Approaches to *filtered editing* can be found in earlier literature (Sarnak et al., 1988) or in more recent related work (Walkingshaw and Ostermann, 2014). In the context of SPLE, *filtered editing* has only been employed partially in related work. For instance, the source-code centric tool *CIDE* (Kästner et al., 2008) offers the possibility to temporarily restrict a variational project to a *view* on a specific variant. Similarly, the MDPLE tool *Feature Mapper* (Heidenreich et al., 2008) includes a *change recording* mode. Albeit, these approaches are only designed for local filtered editing rather than for collaborative MDPLE.

7 CONCLUSION

We have presented a novel approach facilitating collaborative SPLE, and in particular Model-Driven Product Line Engineering. The key contribution is the extension of a conceptual framework that had so far enabled single-user MDPLE on the basis of a filtered editing model. To this end, the existing repertoire of operations *check-out* and *commit* has been extended by *pull* and *push*, which synchronize different copies of the product line repository. The operations are defined based upon a *collaborative revision graph*, which organizes remote and local transactions, as well as *symmetric deltas*. The conceptual contributions have been implemented as plug-ins for the Eclipse-based tool SuperMod, which has been advanced to a distributed version control system. The server component has been realized as a REST-based web service. The distributed VCS operations *pull* and *push* have been mapped to the HTTP methods `GET` and `PUT`; symmetric deltas are transferred in XMI format. A low-level transaction layer ensures that transactions are coordinated consistently. A comparison with related work reveals that the extensions to the framework and SuperMod advance the state of the art in both version control and SPLE. Through a simple example, the added value of the theoretical contributions has been demonstrated; the practical benefit is demonstrated by screencasts (see below).

As soon as tool users apply concurrent modifications to the same versioned artifact, conflicts at product space level may arise. This type of consistency control is orthogonal to the problem of collaborative editing. It has therefore been neglected in this paper an will be addressed by future research.

Tool and Screencasts. The tool *SuperMod* is available as several Eclipse plug-ins from the following update site: [3]. We recommend Eclipse Mars, Modeling edition. The server component requires a Tomcat 7 servlet container. It is available as a *web application archive*: [4]. A screencast illustrating the contributions of this paper is available here: [5]. Adobe Flash plug-in is required.

REFERENCES

Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84.

Chacon, S. (2009). *Pro Git*. Apress, Berkely, CA, USA, 1st edition.

Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2004). *Version Control with Subversion*. O'Reilly, Sebastopol, CA.

Conradi, R. and Malm, C. C. (1991). Cooperating transactions and workspaces in EPOS: Design and preliminary implementation. In *Proceedings of the 3rd In-*

[3]http://btn1x4.inf.uni-bayreuth.de/supermod/update

[4]http://btn1x4.inf.uni-bayreuth.de/supermod/webapp/supermod-server.war

[5]http://btn1x4.inf.uni-bayreuth.de/supermod/screencast (item *Collaborative MDPLE* based on the *Graph* example)

ternational Conference on Advanced Information Systems Engineering (CAiSE'91), pages 375–392.

Czarnecki, K. and Kim, C. H. P. (2005). Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories at OOPSLA'05*, San Diego, California, USA. ACM.

Estublier, J. and Casallas, R. (1994). The Adele configuration manager. In Tichy, W. F., editor, *Configuration Management*, volume 2 of *Trends in Software*, pages 99–134. John Wiley & Sons, Chichester, UK.

Fielding, R. T. (2000). *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

Gomaa, H. (2004). *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, Boston, MA.

Heidenreich, F., Kopcsek, J., and Wende, C. (2008). FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA. ACM.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute.

Kästner, C., Trujillo, S., and Apel, S. (2008). Visualizing software product line variabilities in source code. In *Proceedings of the 2nd International SPLC Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 303–313.

Krueger, C. W. (2002). Variation management for software production lines. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 37–48, London, UK, UK. Springer-Verlag.

Laguna, M. A. and Crespo, Y. (2013). A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. *Sci. Comput. Program.*, 78(8):1010–1034.

Munch, B. P. (1993). *Versioning in a Software Engineering Database — The Change Oriented Way*. PhD thesis, Tekniske Høgskole Trondheim Norges.

OMG (2011a). *OMG MOF 2 XMI Mapping Specification, Version 2.4.1*. Object Management Group.

OMG (2011b). *UML Infrastructure*. Object Management Group, Needham, MA, formal/2011-08-05 edition.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Germany.

Rochkind, M. J. (1975). The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370.

Sarnak, N., Bernstein, R. L., and Kruskal, V. (1988). Creation and maintenance of multiple versions. In Winkler, J. F. H., editor, *SCM*, volume 30 of *Berichte des German Chapter of the ACM*, pages 264–275. Teubner.

Schwägerl, F., Buchmann, T., Uhrig, S., and Westfechtel, B. (2015a). Towards the integration of model-driven engineering, software product line engineering, and software configuration management. In Hammoudi, S., Pires, L. F., Desfray, P., and Filipe, J., editors, *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015)*, pages 5–18, Angers, France. SCITEPRESS.

Schwägerl, F., Buchmann, T., Uhrig, S., and Westfechtel, B. (2016). Realizing a conceptual framework to integrate model-driven engineering, software product line engineering, and software configuration management. In Desfray, P., Filipe, J., Hammoudi, S., and Ferreira Pires, L., editors, *Model-Driven Engineering and Software Development*, volume 580 of *Communications in Computer and Information Science (CCIS)*, chapter 2, pages 21–44. Springer International Publishing. Revised Selected Papers from MODELSWARD 2015.

Schwägerl, F., Buchmann, T., and Westfechtel, B. (2015b). SuperMod - A model-driven tool that combines version control and software product line engineering. In *ICSOFT-PT 2015 - Proceedings of the 10th International Conference on Software Paradigm Trends*, pages 5–18, Colmar, Alsace, France. SCITEPRESS.

Schwägerl, F., Uhrig, S., and Westfechtel, B. (2015c). A graph-based algorithm for three-way merging of ordered collections in EMF models. *Science of Computer Programming*, 113, Part 1:51 – 81. Selected and Revised Papers from MODELSWARD 2014.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, Upper Saddle River, NJ, 2nd edition edition.

Thao, C. (2012). Managing evolution of software product line. In Glinz, M., Murphy, G. C., and Pezzè, M., editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1619–1621. IEEE.

Tichy, W. F. (1985). RCS — a system for version control. *Journal of Software: Practice and Experience*, 15(7):637–654.

Völter, M., Stahl, T., Bettin, J., Haase, A., and Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons.

Walkingshaw, E. and Ostermann, K. (2014). Projectional editing of variational software. In *Generative Programming: Concepts and Experiences, GPCE'14, Vasteras, Sweden, September 15-16, 2014*, pages 29–38.

Westfechtel, B. (2014). Merging of EMF models - formal foundations. *Software & Systems Modeling*, 13(2):757–788.

Westfechtel, B., Munch, B. P., and Conradi, R. (2001). A layered architecture for uniform version management. *IEEE Transactions on Software Engineering*, 27(12):1111–1133.