Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving 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, Version Control, Filtered Editing.

Abstract: Model-driven software product line engineering is complicated: In addition to defining a variability model, developers must deal with a multi-variant domain model. To reduce complexity, filtered editing, inspired by version control, was recently transferred to software product line engineering. On check-out, a single-variant model is derived based on a configuration of its features. On commit, the representatively applied change is scoped with the features to which it is relevant. The here considered dynamic editing model involves different kinds of evolution: The variability model and the domain model are edited concurrently. Features, which define the workspace contents or the scope of the change, may be introduced or deleted. Furthermore, the scope of a change may be revised until commit. The dynamism of this filtered editing model raises consistency problems concerning the evolving relationships between the variability model, the specified configuration, and the scope of the change. This paper formalizes these constraints and presents consistency-preserving algorithms for the workspace operations check-out, commit, as well as a new operation, migrate. This way, the evolution of model-driven software product lines is managed automatically, non-disruptively, and consistently.

1 INTRODUCTION

In *model-driven software engineering* (*MDSE*), software systems are developed from high level models, which are eventually transformed into executable code (Völter et al., 2006). *Models* are abstractions of systems; they may be located at different levels of detail, e.g., requirements, architecture, and design. A model is an instance of a metamodel, which defines the abstract syntax of the used modeling language.

Software configuration management is concerned with the evolution of software systems; *version control* lies at its heart (Conradi and Westfechtel, 1998). A *version* denotes a state of an evolving artifact. Evolution may occur along several dimensions, giving rise to historical, logical, and cooperative versioning (Estublier and Casallas, 1995). Versions with respect to the historical and the logical dimension are called *revisions* and *variants*, respectively. Cooperative versioning orchestrates the work of different software developers. Historical versioning is frequently supported by *revision graphs*, which allow for *branches* that support logical versioning to a limited extent.

Finally, *software product line engineering* (*SPLE*) is a paradigm to develop software applications based on the principle of *variability* (Pohl et al., 2005). A *platform* is a set of artifacts that form a common structure from which a set of derivative products can be efficiently developed and produced. A variability model, e.g., a *feature model* (Kang et al., 1990), describes common and discriminating features of the variants of a product line. The combination of MDPLE and SPLE, *model-driven product line engineering* (*MDPLE*) is a promising research field, since it may increase productivity in two ways (Gomaa, 2004). Yet, it requires to edit multi-variant models, which tends to be cognitively complex.

To this end, several approaches to *partially filtered software product line engineering* recently emerged (Kästner et al., 2008; Walkingshaw and Ostermann, 2014), where variants not important for a change are hidden. In this paper, we assume *fully filtered editing*, where the multi-variant domain model is entirely transparent to the developer. Instead, he/she modifies a single-variant model in several *edit sessions*; in each session, the specific variant to be edited is described using a *choice*, i.e., a *read filter*. The modifications are transferred back to the platform using an *ambition*, a *write filter* that defines the variants for which the changes apply.

The contributions presented here are based on a conceptual framework (Schwägerl et al., 2015a) that

SchwÃd'gerl F. and Westfechtel B.
Maintaining Workspace Consistency in Filtered Editing of Dynamically Evolving Model-driven Software Product Lines.
DOI: 10.5220/0006071800150028
In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, pages 15-28
ISBN: 978-989-758-210-3

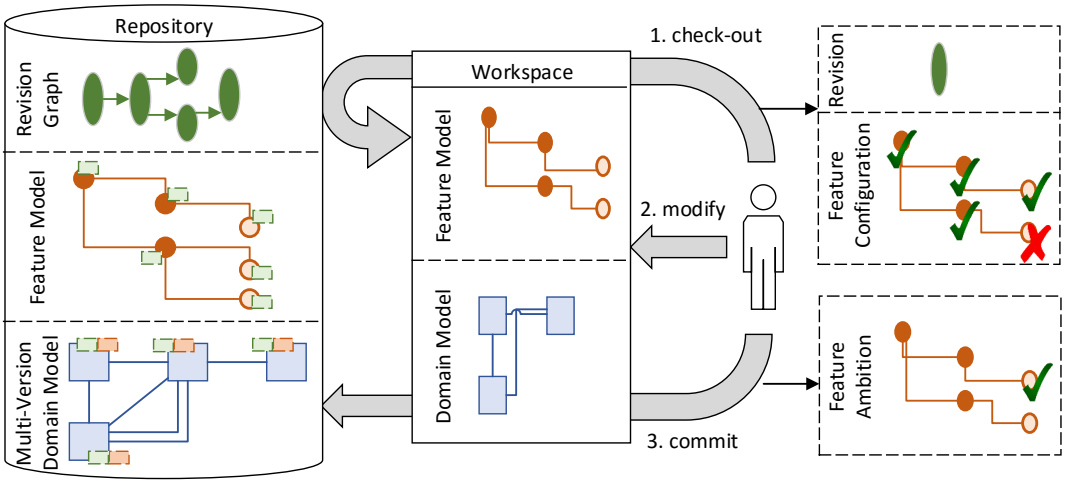

Figure 1: An architectural and functional sketch of the considered conceptual framework.

transfers the *check-out/modify/commit* workflow from version control to filtered MDPLE. As shown in Figure 1, in addition to a *revision graph*, which controls the historical evolution of the product line, a *feature model* is used for logical versioning. Selecting a version during *check-out* involves the selection of a revision as well as the definition of a *feature configuration*, which serves as read filter. A corresponding product version is transferred from the repository to a workspace. The write filter is provided as a *feature ambition*, a partial selection in the feature model.

Without loss of generality, the editing model is assumed to be *dynamic* in the sense that it supports co-evolution of feature and domain model and permits updates to the scope of a change during the edit session. Dynamic filtered editing was characterized earlier (Schwägerl et al., 2016b); however, formal definitions of the operations part of the editing model are missing, and consistency has been neglected.

In this paper, we show that dynamic filtered editing requires well-defined *workspace operations* complying to a set of *consistency constraints*. As main contributions, we formally define these constraints and present *algorithms* which preserve them. For example, the consistency-preserving commit operation ensures that there are no contradictions between the feature configuration, the feature ambition, and the feature model, which is subject to evolution. Furthermore, an extra operation is provided to *migrate* the old feature configuration such that it is consistent with a new feature model and obviates repeated check-outs.

Section 2 gives a brief overview on (dynamic) filtered MDPLE, introducing a running example. Section 3 sketches the addressed evolution scenario. Next, formal foundations are explained. Sections 5 and 6 present workspace consistency constraints and algorithms for consistency-preserving workspace operations, respectively. Section 7 describes the implementation. Section 8 evaluates and critically reflects our contribution. Section 9 presents related work.

2 FILTERED MDPLE

In the considered conceptual framework, all kinds of versioning, i.e., the revision graph as well as the feature model, are mapped internally to a common *uniform version model* (Westfechtel et al., 2001), which is not exposed to the end user. Behind the curtains, the framework relies on *negative variability*. The platform, which corresponds to the repository in SCM terms, contains a *superimposition* of all revisions and variants; the connection to the version model(s) is realized by *visibilities* (dashed boxes in Figure 1).

The repository relies on a three-layered architecture: A revision graph controls the evolution of both the feature model and the main artifact, the *domain model*. The feature model, whose role is dual, serves as an additional variability model. This way, the domain model is versioned by both the revision graph and the feature model, which is also reflected in the visibilities of the respective layers. All repository contents are transparent to the end users.

The *workspace*, in which the product line engineer evolves the product line, contains a selected revision of the feature model and a single-version domain model; the term “version” denotes a specific variant in the revision selected for the feature model.

On *check-out*, the workspace is populated through a staged selection process. In the first stage, a revision of the product line is selected; this selection uniquely determines a revision of the feature model.

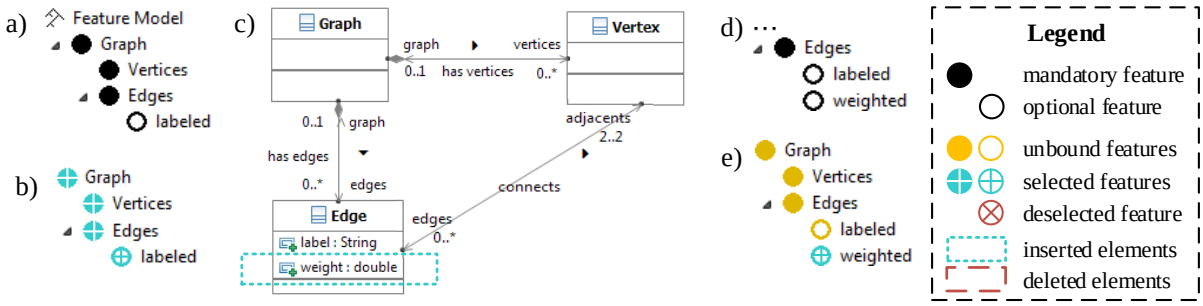

Figure 2: Example iteration of filtered editing: Attribute weight is introduced under ambition weighted.

In the second stage, a variant of the domain model is determined by configuring the feature model. In the resulting feature configuration, each feature is bound either to *true* (selected) or to *false* (deselected).

On *commit*, the repository contents are updated such that the changes performed in the workspace become persistent. However, the scope of the changes is rarely confined to the single variant in which they are performed. Therefore, the product line engineer is requested to specify a *feature ambition* when committing his or her changes. A feature ambition constitutes a partial feature configuration in the sense that some features are bound to either *true* or *false*, while the other features remain unbound. For the users, this is decisive, since they must identify those features the performed change is associated with. When taking into consideration the definition of feature as "increment of program functionality" (Batory, 2005), the number of features bound should be typically small.

We illustrate the conceptual framework using the well-known *graph* product line example (Lopez-Herrejon and Batory, 2001). The product line consists of a structural model, represented as a UML class diagram, where vertices and edges are realized. Furthermore, variable properties such as labels, weights, and directed vs. undirected edges, are defined.

Assuming that several iterations of the filtered editing model have been performed, Figure 2 illustrates one iteration where the feature weighted is both defined and realized, from the end user's perspective: First, the user selects a unique revision within the revision graph (not shown). Next, in the chosen revision of the feature model (a), the user selects a feature configuration where all available features, including the mandatory features Graph, Vertices, and Edges, and the optional feature labeled are selected (b). This results in a version of the domain model that is unique with respect to both the revision graph and the feature model; this version populates the workspace (c). Furthermore, the selected revision of the feature model is made available. In the edit session, the user introduces an optional feature for weighted edges (d). The feature is implemented in the same editing cycle by attaching the attribute weight to class Edge (c). Upon commit, the user defines the ambition of the change by selecting exclusively the feature weighted from the feature model (e). All other features remain unbound as the performed change is immaterial to them. As a consequence, weight becomes visible in future variants that include weighted.

3 EVOLUTION SCENARIO

As hinted in the preceding example, this paper assumes a *dynamic filtered editing model* where both the feature model and the domain model are available in the workspace for editing (e.g., the feature weighted has been both defined and realized at a time). Furthermore, the feature ambition may be specified and updated between check-out and commit.

When compared to *static* filtered editing (cf. related work), where the feature model may not evolve and the scope of a change must be defined at check-out (Westfechtel et al., 2001), the flexibility added by dynamic filtered editing implies new *consistency problems*. For example, for the sample iteration sketched in Figure 2, the feature weighted against which the change is committed was not present at check-out time. To this end, we develop a variety of *constraints*, redefine check-out and commit, and introduce an operation *migrate*, taking these constraints into account by enforcing them.

The generalized *evolution scenario*, forming the problem statement of this paper, is illustrated in Figure 3. The vertical dashed lines divide the figure into four parts, dedicated to check-out, modify, commit, and migrate, respectively. Blue arrows indicate evolution. Grey arrows represent consistency relationships and are labeled with the numbers of constraints introduced in Section 5. Unlike in Figure 1, the domain model is not shown, and versioning of the feature model is not illustrated explicitly.

Check-out. The editing cycle starts with an initial

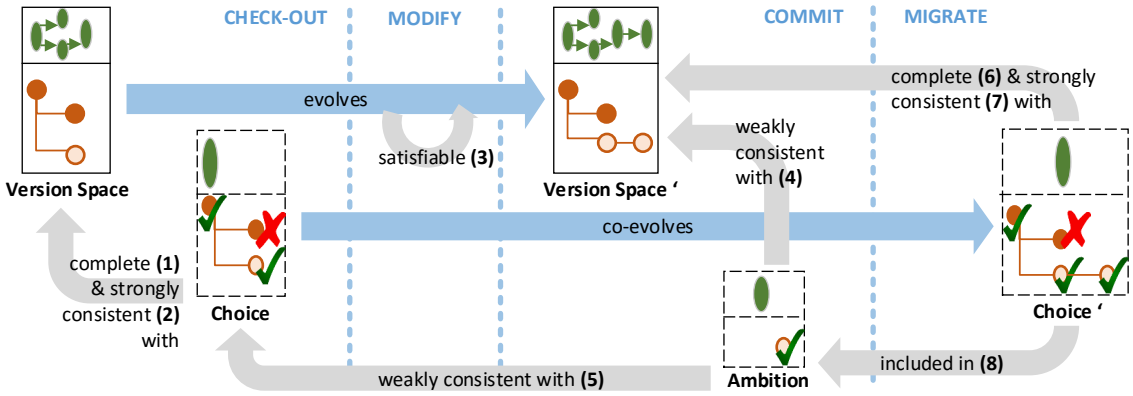

Figure 3: Evolution scenario summarizing the problem statement and contributions of this paper.

revision graph, from which a specific revision is selected. In the resulting revision of the feature model, all features need to be either selected or deselected. The unique version defined by the revision and feature configuration is referred to as the *choice*. The provided check-out operation has to ensure that the choice is *complete* (in particular, no features from the feature model may remain unbound) and *strongly consistent* (all version rules defined in the feature model must hold).

Modify. In the workspace, the feature model may be edited as required, provided that it remains *satisfiable*, i.e., non self-contradictory.

Commit. The commit operation will eventually result in a revised version space, where the revision graph is extended with a new revision for the performed change; the feature model may have evolved, as well. For commit, the user needs to define a *feature ambition*; the combination of feature ambition and the new revision is called *ambition*. This ambition must be checked for *weak consistency* with the new revision of the feature model, which means that there must be no contradiction to the version rules implied by the feature model (such that there exists at least one valid version within the set of versions described by the ambition). Furthermore, the ambition at commit time must be *weakly consistent* with the choice at check-out time, which means that there must be no contradiction between the old choice and the ambition (e.g., the ambition must not select a feature which was deselected in the old choice).

Migrate. Finally, the choice must *co-evolve* with the change: A new choice has to be established which satisfies the same constraints as the old choice, since the end of the previous cycle coincides with the start of the next cycle. To this end, we introduce an operation *choice migration*, which is performed immediately after commit. It essentially extends the old choice with the ambition, ensures *completeness* of the migrated choice and *strong consistency* with respect to the new feature model, as well as *inclusion* of the migrated choice in the ambition. Consequently, executing a new check-out under the migrated choice would result in a copy of the workspace at commit time.

Figure 4 summarizes the editing cycle as state diagram. Initially, the workspace is *Pending*, i.e., not populated yet. On check-out, a specific version is selected from the repository. After modifying the workspace and committing, the user may either continue with the subsequent iteration, requiring to *migrate* the choice, or migration is *canceled*, triggering a transition back into state *Pending*. To re-populate the workspace, a new choice must be specified then. The figure suggests two important properties of the editing model: (1) Except for the initial version, and as long as choice migration is never aborted, check-out is an optional operation. (2) Choice migration is a shortcut assuming that the user wants to perform the subsequent iteration using the current workspace as starting point, as usual in version control.

Let us resume the example from Figure 2 keeping the informally defined consistency constraints in mind. At check-out, it is ensured that the configuration of the selected feature model revision is complete and strongly consistent. In the edit session, the feature model is extended with a feature weighted, which

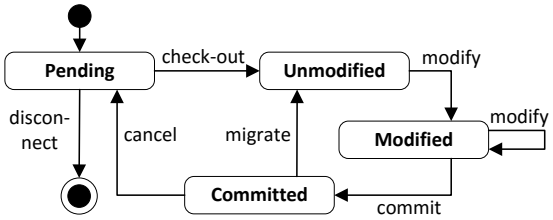

Figure 4: Workspace operations as state transitions.

serves as ambition for the commit. The ambition contradicts with neither the new feature model nor the old choice. Migration extends the old choice with the new feature. During choice migration, the choice is extended by a positive selection of weighted, such that it becomes complete, strongly consistent, and included in the ambition.

4 FORMAL FOUNDATIONS

Following (Westfechtel et al., 2001; Schwägerl et al., 2015a), we provide a formalization of the underlying conceptual framework. Internally, versioning concepts exposed at the user interface – revision graphs and feature models – are mapped to a generic base layer, the formal foundation of which is propositional logic. Workspace consistency constraints (Section 5) and consistency-preserving algorithms (Section 6) are formalized upon the base layer.

An *option* represents a (logical or historical) property of a software product that is either present or absent. The *option set* is defined globally:

$$O = \{o_1, \ldots, o_n\} \tag{1}$$

Internally, the conceptual framework maps revisions and features to options transparently.

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

$$c = b_1 \wedge \ldots \wedge b_n,\ b_i \in \{o_i, \neg o_i\} \tag{2}$$

It can also be represented as a *binding map*, i.e., a set of *binding tuples* (o_i, s_i), where $s_i \in \{true, false\}$ denotes the boolean selection state of an option o_i.

Choices are derived from a user-based selection of a revision and a feature configuration.

An *ambition* is an option binding that allows for unbound options:

$$a = b_1 \wedge \ldots \wedge b_n,\ b_i \in \{o_i, \neg o_i, true\} \tag{3}$$

When represented as a binding map, tuples for unbound options are omitted.

Ambitions are derived from a selection in the feature model, which may leave a number of features unbound in order to describe a *set* of variants to which a change is applied. In the revision graph, the management of ambitions is automated (see Section 6).

A *version rule* is a boolean expression over a subset of options. The *rule base* $\mathcal{R}$ is a conjunction of rules $\rho_1, \ldots, \rho_m$ all of which have to be satisfied by a choice in order to be consistent:

$$\mathcal{R} = \rho_1 \wedge \ldots \wedge \rho_m,\ \rho_i \text{ is an expression over } O \tag{4}$$

Version rules are derived automatically from revision graph and feature model (Schwägerl et al., 2015a).

Preferences and defaults have been introduced to ease version selection. A *preference* is a tuple of the form $p_i = (o_i, \pi_i)$, where π_i is an *initialization expression* for option o_i. *Defaults* $d_i = (o_i, s_i)$ define a fallback selection state $s_i \in \{true, false\}$. For each option, at most one preference and at most one default is allowed; preferences have a higher priority.

$$\mathcal{P} = \{(o_{i_1}, \pi_{i_1}), \ldots (o_{i_k}, \pi_{i_k})\},\ \pi_{i_j} \text{ is an expression over } O \tag{5}$$

$$\mathcal{D} = \{(o_{i_1}, s_{i_1}), \ldots (o_{i_l}, s_{i_l})\},\ s_{i_j} \in \{true, false\} \tag{6}$$

The conceptual framework infers preferences and defaults transparently in order to assist the user in selections in the version space. In particular, they automate the management of the revision graph.

Each element e of the product space (i.e., the union of feature and domain model) carries a *visibility* v_e, a boolean expression over the variables of O. Visibilities in the feature model are composed only of revision options; visibilities in the domain model are composed of both revision and feature options. An element e is *visible* under a given choice c iff applying c to its visibility v_e evaluates to $true$:

$$v_e(c) = true \tag{7}$$

The operation *filter* is applied during check-out. From a base element set E, those elements e that do not satisfy the choice are omitted.

$$filter(E, c) = E \setminus \{e \in E \mid v_e(c) = false\} \tag{8}$$

On commit, visibilities must be updated such that inserted (deleted) elements become (in)visible in all choices included in the ambition a. Accordingly, the *updated visibility* $v'(e)$ is defined:

$$v'(e) = \begin{cases} v(e) & \text{if } e \text{ is unmodified.} \\ a & \text{if } e \text{ has been inserted.} \\ v(e) \wedge \neg a & \text{if } e \text{ has been deleted.} \end{cases} \tag{9}$$

For updates to the domain model, the full ambition a is used; for updates to the feature model, bindings of feature options are omitted from the ambition, since the feature model evolves merely in the revision space.

5 WORKSPACE CONSISTENCY CONSTRAINTS

In this section, the consistency constraints mentioned in Section 3, Figure 3, are elaborated on the basis of the formal foundations from Section 4. To cope with evolution, we use superscripts (^{ch} = check-out, ^{mo} = modify, ^{cm} = commit, ^{mi} = migrate).

5.1 Check-out

In filtered editing, a choice designates a unique version used for check-out. Therefore, unbound options must not occur. Moreover, the choice must comply with the rules derived by, e.g., feature dependencies.

Constraint 1. *The binding map c^{ch} specified as choice during check-out must be* complete *with respect to the global option set O^{ch} at check-out time.*

$$\forall o \in O^{ch} : (\exists (o,s) \in c^{ch} : s \in \{true, false\}) \quad (10)$$

Constraint 2. *The choice c^{ch} at check-out time (satisfying Constraint 1) must be* strongly consistent *with the rule base $\mathcal{R}^{ch}$ at check-out time.*

$$\mathcal{R}^{ch}(c^{ch}) = true \quad (11)$$

Here, $\mathcal{R}^{ch}(c^{ch})$ denotes the evaluation of the rules contained in $\mathcal{R}^{ch}$ under the choice c^{ch}.

5.2 Modify

By editing the feature model, the user modifies parts of the option set and the rule base. It must be avoided that the user introduces rules disallowing consistent version selection in future check-outs.

Constraint 3. *After each modification to the version space, the rule base $\mathcal{R}^{mo}$ must be satisfiable:*

$$\exists c : (\mathcal{R}^{mo}(c) = true) \quad (12)$$

5.3 Commit

At commit time, the conjunction of the ambition and the rule base must be *satisfiable* (*weak consistency*): At least one choice c must exist which agrees with a^{cm} in all common option bindings ($c \Rightarrow a^{cm}$) such that all rules hold under c.

Constraint 4. *An ambition a^{cm} specified during commit must be* weakly consistent *with the rule base $\mathcal{R}^{cm}$ at commit time.*

$$\exists c : (c \Rightarrow a^{cm}) \wedge (\mathcal{R}^{cm}(c) = true) \quad (13)$$

In the version determined by the choice, a change is applied *representatively* for the ambition. Thus, there must not be any contradiction between option bindings of the check-out time choice and the commit time ambition inferred from feature selections:

Constraint 5. *The ambition a^{cm} must be* weakly consistent *with the check-out choice c^{ch}.*

$$\forall (o,s) \in a^{cm} : (o, \neg s) \notin c^{ch}, \quad s \in \{true, false\} \quad (14)$$

To support co-evolution (cf. Section 3), Constraint 5 allows to commit against a newly introduced feature which was not bound in c^{ch}.

5.4 Migrate

Due to modifications of the rule base, the choice c^{ch} specified at check-out time may become *incomplete* with respect to the option set O^{cm}, and/or *inconsistent* with the rule base $\mathcal{R}^{cm}$ at commit time. These temporary inconsistencies are explicitly allowed in order to support feature model evolution. However, before starting the subsequent iteration, it is required that the version being modified in the subsequent iteration must be uniquely and consistently identified by c^{mi}.

Constraint 6. *The binding map c^{mi} describing the choice after migration must be* complete *with respect to the commit time option set O^{cm}.*

$$\forall o \in O^{cm} : (\exists (o,s) \in c^{mi} : s \in \{true, false\}) \quad (15)$$

Constraint 7. *The migrated choice c^{mi} (satisfying Constraint 6) must remain* strongly consistent *with the rule base $\mathcal{R}^{cm}$ at commit time.*

$$\mathcal{R}^{cm}(c^{mi}) = true \quad (16)$$

Apart from this, it is required that the migrated choice c^{mi} must still comply with ambition a^{cm}, which represents changes applied to the current workspace. Since all newly introduced options are mandatory to be selected or deselected for the choice at migration time, total *inclusion* (realized by propositional logical implication in the opposite direction) is required.

Constraint 8. *An ambition a^{cm} must* include *the migrated choice c^{mi} describing the workspace contents for the subsequent iteration.*

$$c^{mi} \Rightarrow a^{cm} \quad (17)$$

6 CONSISTENCY-PRESERVING ALGORITHMS

In this section, we contribute detailed algorithms for operations *check-out*, *modify*, *commit*, and *migrate* mentioned in Figure 3. In addition, their properties are discussed and runtime considerations are made. For illustration, we refer back to the example from Figure 2 where adequate. The algorithms contain interactive steps, which have been underlined in the descriptions below. Subscripts ($_r$, $_f$) refer to disjoint subsets of revision graph or feature model.

6.1 Check-Out

According to Section 3, the purpose of *check-out* is to populate an empty workspace with a consistent product version uniquely defined by the user.

Algorithm 1: Check-Out.

$o_r^{ch} \leftarrow$ option in O_r^{ch} belonging to a selected revision
Apply revision preferences $\mathcal{P}_r^{ch}$ and revision defaults $\mathcal{D}_r^{ch}$ to $c_r^{ch} := o_r^{ch}$
Filter the feature model by c_r^{ch} and export it into the workspace ▷ Equation 8.
$c_f^{ch} \leftarrow$ select feature configuration in the filtered feature model
$c^{ch} \leftarrow c_r^{ch} \wedge c_f^{ch}$
Apply preferences $\mathcal{P}^{ch}$ and defaults $\mathcal{D}^{ch}$ to c^{ch}
if not $(\forall o \in O^{ch} : (\exists (o,s) \in c^{ch} : s \in \{true, false\}))$ **then** ▷ Ensure Constraint 1.
    return error "Choice is not complete."
if not $(\mathcal{R}^{ch}(c^{ch}) = true)$ **then** ▷ Ensure Constraint 2.
    return error "Choice is not strongly consistent."
Filter the domain model by c^{ch} and export it into the workspace ▷ Equation 8.
Memorize c^{ch} for the subsequent commit

Algorithm 1 first asks the user for a revision selection. Using preferences and defaults introduced during *commit* (cf. Section 6.3), it is ensured that options of the selected revision as well as all predecessors are bound to $true$, whereas remaining options are bound to $false$, making the revision choice complete. Next, the feature model, whose elements' visibilities only refer to revision options, is filtered by the revision choice. In the filtered feature model, the user specifies a configuration; invisible options for deleted features (cf. Section 6.2) are bound to $false$ by corresponding defaults. The effective choice c^{ch} is calculated by conjunction of revision and feature choice, before being checked for completeness and strong consistency. Finally, the workspace is populated with filtered versions of both the feature and the domain model.

Properties. If successful, Algorithm 1 transitions the workspace into state *Unmodified* and produces a choice both complete and strongly consistent with respect to the check-out time rule base, such that Constraints 1 and 2 are ensured. In case the user specifies an incomplete or inconsistent choice, the action is canceled and the workspace remains *Pending*.

Complexity. Applying preferences and defaults requires to iterate over the preference set $\mathcal{P}$ and the default set $\mathcal{D}$, the size of both is bounded by $|O|$. Ensuring Constraints 1 and 2 requires iterations over O and $\mathcal{R}$, respectively; the size of the latter is proportional to $|O|$. While filtering, each element of E is considered. The total complexity is therefore $O(|O| + |E|)$.

Example. Figure 2b depicts a valid feature choice. It would have been disallowed to leave any option unbound or to assign a negative selection state to one of the mandatory features Graph, Vertices, or Edges. Yet, the equivalent change could have been applied in a version where labeled was deselected.

6.2 Modify

The consistency of the domain model is supposed to be ensured by the respective single-version editing tools used for modification. For the feature model, Constraint 3 is actively ensured after each modification. Furthermore, *deletion* of features in the workspace version of the feature model is redefined:

Rather than persistently deleting a feature, it is merely hidden from the user interface and thus not available in future revisions of the feature model. Nevertheless, its feature option o_f, which still may occur in visibilities of domain model elements, remains in O_f. Deletion is only applicable to features bound to $false$ in the current choice c^{ch}. Otherwise, *choice migration* (see Section 6.4) would transfer the positive selection state to the choice for the next iteration, where o_f and corresponding realization artifacts are hidden. To maintain *completeness* of future choices, a default $(o_f, false)$ is introduced to $\mathcal{D}_f^{mo}$ transparently. This default also affects version rules referencing the feature; their evaluation is automated.

Properties. Constraint 3 is actively enforced. The workspace enters (or remains in) state *Modified*.

Complexity. The satisfiability check described by Constraint 3 performed after each save is *NP-complete*. The runtime of feature deletion can be neglected.

Example. Since all features are bound positively, feature deletion cannot be applied here. If labeled had been bound to $false$ in c^{ch} and then been deleted,

Algorithm 2: Commit.

$c^{ch} \leftarrow$ choice memorized during preceding check-out or migration
$o_r \leftarrow$ option of most recently committed revision
$o_{r+1} \leftarrow$ new revision option with user-specified details (commit message, etc.)
$O_r^{cm} \leftarrow O_r^{ch} \cup o_{r+1}$ ▷ New revision option.
$\mathcal{P}_r^{cm} \leftarrow \mathcal{P}_r^{ch} \cup (o_r, o_{r+1})$ ▷ Ensures that predecessor revisions are selected.
$\mathcal{D}_r^{cm} \leftarrow \mathcal{D}_r^{ch} \cup (o_{r+1}, false)$ ▷ Non-predecessor revisions must be deselected.
$ws^{ch} \leftarrow$ filter feature/domain model by c^{ch} ▷ Reproduce checked-out workspace.
$ws^{cm} \leftarrow$ current workspace version of feature and domain model
$a_f^{cm} \leftarrow$ select feature ambition in the current workspace version of the feature model
$a^{cm} \leftarrow a_f^{cm} \wedge o_{r+1}$
if not $(\exists c : (c \Rightarrow a^{cm}) \wedge (\mathcal{R}^{cm}(c) = true)$ **then**
  return error "Ambition is not weakly consistent." ▷ Ensure Constraint 4.
if not $(\forall (o,s) \in a^{cm} : (o, \neg s) \notin c^{ch})$ **then**
  return error "Ambition is inconsistent with choice." ▷ Ensure Constraint 5.
Differentiate ws^{ch} with ws^{cm} to find modified (inserted or deleted) elements
Update the visibilities of modified feature model elements using $a_r^{cm} := o_{r+1}$ ▷ Equation 9.
Update the visibilities of modified domain model elements using a^{cm} ▷ Equation 9.

a default $(o_{labeled}, false)$ would have been transparently introduced, which automatically deselects the invisible feature in future check-outs.

6.3 Commit

A consistency-preserving *commit* operation is formalized in Algorithm 2. The revision graph is handled automatically, introducing a new revision option along with a preference and default ensuring that selecting a single revision will lead to complete and consistent revision choices in future. Through repeated application of preferences of the form (o_r, o_{r+1}), the selection is propagated back until the initial revision. Defaults, having a lower priority, are applied to unbound revision options thereafter. Next, the check-out version of the workspace is reconstructed and differentiated with its commit time version.

Besides, the user specifies a *feature ambition*. It is ensured by corresponding checks that feature ambitions must be weakly consistent with the rule base (Constraint 4) and the previous choice (Constraint 5).

Visibilities of affected elements are updated as defined by Equation 9. For updates applied to the feature model, the new revision option serves as ambition. For the domain model, a conjunction of the new revision option and the chosen feature ambition is used.

Properties. If successful, Algorithm 2 transitions the workspace into the state *Committed*, while ensuring Constraints 4 and 5 for the specified ambition. Otherwise, the workspace remains in state *Modified*; in this case, the user may retry the commit.

Complexity. Reproducing the checked-out workspace and updating the visibilities both imply a runtime proportional to $|E|$. Constraint 5 checks a maximum of $|O|$ options. Runtime is dominated by the NP-complete satisfiability check in Constraint 4.

Example. The specified ambition (Figure 2e) is weakly consistent with both the rule base – a valid representative would be $c^{ch} \cup \{(o_{weighted}, true)\}$ – and the check-out time choice (which did not include a binding for the newly introduced feature). In the ambition, it would be disallowed to assign selection state $false$ to feature labeled being positively bound in c^{ch}.

6.4 Migrate

Migration prepares the workspace choice for the subsequent iteration. In contrast to *check-out* and *commit*, this operation is not triggered explicitly by the user, but invoked transparently after *commit*. Conversely, it makes the subsequent *check-out* optional, enabling a non-disruptive revision control workflow.

Algorithm 3 iterates over options unbound in the choice, assuming that the user stays in the current view. If a corresponding option has been bound in the ambition, the binding is transferred to the choice. Otherwise, preferences and defaults are triggered as far as applicable, with the aim to complete c^{mi} transparently. As a "last resort", a binding state is obtained non-deterministically. Since the new option has been ignored in the ambition, there cannot exist any reference to it in updated visibilities; therefore, it is immaterial for the subsequent choice whether the option is selected. At this point, it is not known how

Algorithm 3: Migrate.

$c^{mi} \leftarrow c^{ch}$
for $o \in O^{cm}$ **do**
  if $((o,true) \notin c^{mi}) \wedge ((o,false) \notin c^{mi})$ **then** ▷ Never override existing bindings.
    $s^{mi} \leftarrow undefined$
    if $\exists (o,s) \in a^{cm} : s \in \{true, false\}$ **then** $s^{mi} \leftarrow s$
    else if a preference $p \in \mathcal{P}^{cm}$ is applicable to o **then**
      $s^{mi} \leftarrow$ apply p to o
    else if a default $d \in \mathcal{D}^{cm}$ is applicable to o **then**
      $s^{mi} \leftarrow$ apply d to o
    else $s^{mi} \leftarrow$ user selection for o
    $c^{mi} \leftarrow c^{mi} \cup (o, s^{mi})$
if not $\mathcal{R}^{cm}(c^{mi}) = true$ **then** ▷ Ensure Constraint 7.
  return error "Cannot migrate to a consistent choice."
else Memorize c^{mi} for the subsequent commit ▷ Obviate check-out.

new (and therefore unbound) features will be incorporated in the next iteration. Therefore, the user may choose among the set of choices describing the current workspace equivalently.

Properties. *Completeness* (Constraint 6) is fulfilled by iterating over all options in O^{cm} and assigning $true$ or $false$ to missing bindings. A *strongly consistent choice* (Constraint 7) is enforced.

Being its descendant, c^{mi} includes c^{ch}. Moreover, a^{cm} is weakly consistent with c^{ch} (cf. Constraint 5). Thus, no contradictions exist between c^{mi} and a^{cm}. Bindings for missing options are transferred from the ambition, inferred from preferences or defaults, or if not applicable, requested from the user. Altogether, the migrated choice c^{mi} is included in the ambition a^{cm} (as required by Constraint 8).

If migration succeeds, the workspace directly enters state *Unmodified*. Otherwise or if the user cancels the operation, entering state *Pending* immediately triggers an exceptional check-out, forcing the user into specifying a new choice.

Complexity. Assuming that the binding maps underlying choices and ambitions have been implemented as an associative data structure, iterating over all options requires a maximum runtime proportional to $|O|$. Preferences and defaults, whose number is bounded to $|O|$, are taken into account at most two times per newly introduced option $o_{new} \in O_{new}$. Therefore, when neglecting user interaction, total complexity is $O(|O| \cdot |O_{new}|)$.

Example. Bindings for the new revision as well as for the new feature weighted are inferred automatically from the ambition. If an additional feature had been introduced but neither been bound in the ambition nor considered by a feature model rule, a selection would have been requested interactively.

7 IMPLEMENTATION

The consistency-preserving algorithms shown in Section 6 have been implemented as an extension to *SuperMod* (Schwägerl et al., 2015b), a prototype for filtered MDPLE. The tool has been handcrafted in a model-driven way using the *Eclipse Modeling Framework* (*EMF*) (Steinberg et al., 2009). The artifacts to be versioned, representing the "domain model", are *EMF models*, which may be instances of arbitrary metamodels, e.g., of the EMF-based metamodel for UML2 (OMG, 2011b). Being suitable for heterogeneous projects, the system may also manage text files, which are interpreted internally in the repository as EMF model instances.

Constraint validation has been implemented in a user-friendly and non-disruptive way providing specific dialogs (cf. cut-outs in Figure 2) for choice and ambition selection as well as choice migration. For modification of the feature model, the tool provides a dedicated editor that enforces satisfiability each time the modified feature model is saved. Constraint violations are reported using the EMF Validation Framework (Steinberg et al., 2009).

Internally, choices and ambitions are represented as *binding maps* (cf. Section 4), such that they can efficiently be passed to boolean expressions. For satisfiability checks, e.g., Constraints 3 and 4, the solver *Sat4j* (Berre and Parrain, 2010) is utilized. Moreover, optimized data structures ensure scalability; for instance, equivalent visibilities are re-used rather than being cloned (Schwägerl et al., 2016a).

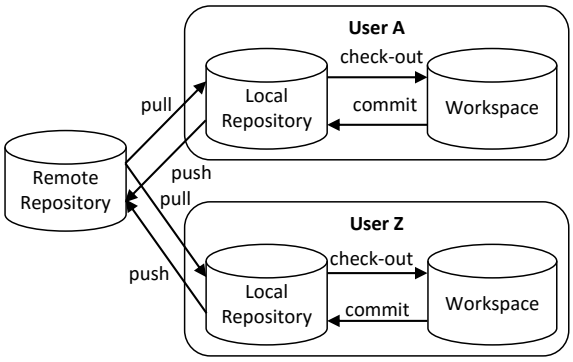

Figure 5: SuperMod's distributed architecture.

Cooperative versioning is supported through the *distributed architecture* displayed in Figure 5. Each user owns a *local repository*; check-out and commit operations are used to populate the workspace and to persist changes locally. Local repositories are synchronized through a *remote repository* by operations *pull* and *push* (Chacon, 2009). Technically, the synchronizing operations have been realized as REST-based (Fielding, 2000) web services. Symmetric deltas are transferred in XMI (OMG, 2011a) format; see (Schwägerl and Westfechtel, 2016).

8 EVALUATION

In order to evaluate the editing model in connection with the presented constraints and consistency-preserving algorithms, we report on two data sets extracted from case studies that refer to standard examples from SPL literature. The first example is the product line for *graphs* (cf. Sections 2 and 3); its adaptation to filtered product line editing has been shown in (Schwägerl et al., 2015b). The second case study refers to a product line for *Home Automation Systems* (*HAS*) originally introduced in (Pohl et al., 2005) and adapted in (Schwägerl et al., 2016b).

All results were obtained by analyzing recorded version histories. While the *Graph* product line was realized by the authors themselves, the *HAS* study was conducted by a master student with MDPLE background. In order to foster an incremental style of development, requirements were communicated to the modeler(s) in multiple interview sessions. Particularly in the HAS case study, hypothetical customer feedback was given, such that it became necessary to both revise the realization of existing features and to define and realize new features.

Table 1 summarizes key figures of both case studies. The bottom half demands for further explanation:

1. *Average ambition complexity*: The complexity of an ambition is defined as the number of features it binds. Negatively bound features are treated as two bindings (as feature expressions derived from them contain an additional syntax tree element for the negation). Note that an ambition can also have complexity 0 in case no feature is bound for a universal change (*true*).
2. *Migration interactions*: The ratio of editing model iterations in which at least one user interaction is required during *migration*.
3. *Explicit check-outs*: The ratio of iterations where the optional *check-out* operation is necessary since the previously applied migration does not produce the desired choice for the next iteration.
4. *Commits canceled*: The ratio of *commit* operations canceled due to violations of a consistency constraint, such that further domain or feature model editing was required.

In sum, the results show that (1) the complexity of ambitions is in average close to 1, supporting the proposition that ambitions frequently consist of only a single feature binding; (2) *migration* happens transparently for the larger part of iterations and significantly accelerates the development; (3) the *check-out* operation, which has been made optional by the dynamic editing model, is necessary only in a small number of iterations; (4) only in one of altogether 47 iterations was it necessary to cancel the commit (in this case, the feature the user intended to realize was accidentally missing in the feature model). In contrast, the larger part of iterations did neither require a *check-out* nor user interaction during *migrate*, underlining that the dynamic filtered editing model is non-disruptive while workspace consistency is maintained smoothly in the background.

Threats to Validity. The significance of the results derived from Table 1 is potentially limited by two factors. First, it was clearly communicated to the evaluators that the scope of the changes applied in one iteration should be equal, leading to comparably short-running iterations. However, in real-world scenarios, inexperienced users may accidentally realize several different features in one iteration, making it impossible to specify a valid global ambition. Second, cooperative versioning was faded out in both case studies as multi-user operation had not been implemented yet. We expect the number canceled commits to slightly increase with multi-user support due to problems such as doubly introduced features.

Table 1: Aggregate results quantifying the user complexity of the dynamic filtered editing model.

	Graph		**HAS**	
number of iterations	9		38	
feature model size	8		17	
domain model size	26		106	
average ambition complexity	$8/9$	$= 0.89$	$41/38$	$= 1.08$
% migration interactions	$1/9$	$= 0.11$	$5/38$	$= 0.13$
% explicit check-outs	$3/9$	$= 0.33$	$6/38$	$= 0.16$
% commits cancelled	$0/9$	$= 0$	$1/38$	$= 0.026$

9 RELATED WORK

This paper continues a series of previous work on SuperMod and its theoretical foundations published by the authors; see references section. Here, we refer to approaches that explicitly deal with workspace consistency problems appearing with filtered editing or related forms thereof. Furthermore, we supply references to related papers published in the meantime.

Version Control Systems. With *branches*, almost all contemporary revision control systems, including Subversion (Collins-Sussman et al., 2004) and Git (Chacon, 2009), offer co-existing product variants. Yet, existing approaches only allow to restore variants committed earlier (*extensional*), but not to re-combine new variants in a way as fine-grained as provided by feature models (*intensional versioning*). In addition, the state of the art lacks a version control system that (1) operates at a level of abstraction higher than text files (e.g., EMF models), and (2) supports intensional versioning. It is for these reasons why SuperMod has been handcrafted rather than relying on existing systems as low-level layer.

Model-Driven Product Line Engineering. As mentioned in the introduction, MDPLE (Gomaa, 2004) is motivated by a common goal of its subdisciplines, increased productivity. A specific product variant may be configured from a set of selected features in two ways. Approaches based on *positive variability*, e.g., (Zschaler et al., 2010), require specific tools, e.g., *model transformations*, to *compose* variable realization fragments. *Negative variability*, e.g., (Heidenreich et al., 2008) means that the product variant is constructed by filtering all elements not required for this variant from a superimposition. The approach at hand internally uses negative variability.

Integrated Historical and Logical Versioning. The *integration* of revision and variant control was targeted earlier. Approaches to *orthogonal* version management (Reichenberger, 1995) did not consider the variability model to be subject to historical evolution. In (Zeller and Snelting, 1997), an approach based on *feature logic* was presented; externally, variant annotations are mapped to low-level C preprocessor directives.

The *Uniform Version Model* (UVM) (Westfechtel et al., 2001) defines version control foundations parts of which had been introduced in the context of *change-oriented versioning* (Munch, 1993). Albeit, the low-level variability model is not provided as an explicit artifact in the workspace. The here considered framework represents both the product and the version space as high-level models. In (Westfechtel et al., 2001) an explicit check-out is necessary in advance to each iteration, since an operation corresponding to *choice migration* is missing. This results in a more disruptive and less automated workflow.

In (Seidl et al., 2014), a solution to integrated historical and logical versioning based on positive variability is presented. The approach relies on explicit historical versioning of the variability model. *Hyper feature models* record different versions of features and allow for version-aware constraints such as version range restrictions. Versioning is based upon a *delta language* for EMF models, such that for each increment added to the product line, a forward delta must be specified manually. To derive a specific version of the product, a revision must be selected for each active feature consistently. In contrast, the here considered conceptual framework manages deltas transparently, removing the necessity of specific editing tools; in SuperMod, familiar single-version EMF model editing tools may be used.

Filtered Editing. *Negative variability* assumes a *multi-variant domain model* realizing all features of the product domain. Approaches can be further distinguished by the degree of multi-version complexity passed on to the user.

In *unfiltered* approaches, mapping information,

regardless of whether stored within the domain model (Gomaa, 2004) or in a distinct *mapping model* (Heidenreich et al., 2008), must be maintained.

In *fully filtered* multi-variant editing tools (Sarnak et al., 1988; Westfechtel et al., 2001), a choice uniquely denotes a representative of the ambition. By presenting only a single-version view, cognitive complexity is reduced while familiar editing tools may be reused for multi-version editing. The approach at hand represents fully filtered editing.

Approaches to *partially filtered* editing (Walkingshaw and Ostermann, 2014; Zeller and Snelting, 1997) aim at hiding variants to which the current change is immaterial, without requiring the choice to be *complete*. However, there is only a single filter serving as choice and ambition simultaneously. Specific tools or preprocessor languages are still required, and the filtered multi-variant product is still constrained with single version rules.

A source-code centric approach to *temporarily filtered editing* is described in (Kästner et al., 2008). Here, a partial feature configuration is specified as *write filter*. Code fragments immaterial for the change are hidden. As approximation for a read filter, a *context* is derived an extended view on the write filter. The change recording mechanism provided in the MDPLE approach contributed by (Heidenreich et al., 2008) works in a similar way.

In the presented *dynamic* filtered editing model, the variability model may evolve during an iteration embraced by *update* and *commit*. This is in contrast to representatives of *static* filtered editing, e.g., UVM (Westfechtel et al., 2001) and EPOS (Munch, 1993), where the ambition is specified at check-out time; since the rule base does not evolve, constraints dealing with its evolution are unnecessary. Similarly, in (Walkingshaw and Ostermann, 2014), having a single filter requires that the scope of a change must be known beforehand, inhibiting the concurrent introduction of a feature and its realization (cf. example in Figure 2). To our knowledge, the flexibility implied by the *dynamic filtered editing model*, introduced in (Schwägerl et al., 2015a) and fully specified in this paper, is unique in literature.

Product Line Development Processes. The *development process* implied by the dynamic filtered editing approach significantly differs from phase-structured SPL processes such as (Pohl et al., 2005). First, it intentionally blurs *domain engineering*, i.e., the creation of the platform, and *application engineering*, i.e., the adaptation of derived products. Second, complex design decisions, such as defining *variation points* and *variants* in the product, are automated. Feature ambitions, where an arbitrary subset of features can be selected in order to define the scope of a change in the filtered editing model, are related *staged feature configurations* (Czarnecki et al., 2004), which are used for stepwise refinement during application engineering and require additional consistency properties such as parent-child co-selection.

Variation Control Systems. As *variation control systems*, we refer to tools that generalize the concept of version control systems by adding support for intensional variant management. *SuperMod* also falls into this category.

In (Mitschke and Eichberg, 2008), a logical versioning layer transparently mapping feature models, domain models, and traceability links to Subversion, is described. New variants are created locally by merging branches containing variability at the granularity of source files. Yet, it remains unclear how product specific changes may be written back to multiple variants; the concept *ambition* is missing.

In (Montalvillo and Díaz, 2015), an extension enabling SPLE management is presented. It allows to propagate product specific changes, performed in a merged branch, back to the trunk, such that domain and application engineering are blurred in a similar way as in the here considered framework. Rather than being integrated into the underlying version control system Git, the extension is built upon the hosting platform GitHub, resulting in a comparably loosely-coupled SPLE tool integration.

The configuration management platform presented in (Linsbauer et al., 2016) supports both variability and revision management. To this end, variability information is automatically extracted from different variants created by copying and adapting (clone-and-own approach). Once migrated to the platform, generalized check-out and commit operations, which are scoped by configurations, can be used. Rather than actually applying intensional versioning, the approach relies on *reference variants*, which constitutes the variant that "best matches" the selected configuration. This drastically reduces precision when compared to truly intensional approaches.

In (Stănciulescu et al., 2016), a prototypical variation control system relying on the editing model of (Walkingshaw and Ostermann, 2014) is presented. Like in our framework, the concept of *ambition* is used to scope changes during commit (here: *put*). In contrast, check-out (here: *get*) also allows for partial configurations. Unresolved variability remains visible to the user in the workspace, e.g., in the form of preprocessor directives.

Related Consistency Questions. While this paper is focused on workspace consistency, the orthogonal problem of *product consistency* must be distinguished. Since the multi-variant model managed in the repository is unconstrained with respect to its variability, product derivation may deliver conflicting results. For instance, a UML class may obtain two candidate names when two features that define candidates for the name are selected. Such problems occur much more often in multi-user development. Depending on the concrete representation used for the product space, specialized solutions exist; see (Thüm et al., 2014) for a survey.

10 CONCLUSION

We have presented an approach to maintaining workspace consistency in dynamic filtered editing of evolving model-driven software product lines. To this end, we have defined a set of workspace consistency constraints and have redefined workspace operations such that these constraints are preserved. On check-out, the choice describing the version populating the workspace is checked for completeness and strong consistency with the feature model. Within an edit session, modifications to the feature model are permitted only if they preserve its satisfiability. On commit, the ambition delineating the scope of the change is checked for weak consistency with both the choice specified at check-out time and the evolved feature model. Finally, the choice used for the subsequent iteration is migrated such that it is complete, strongly consistent with the commit-time feature model, and included in the ambition.

The approach at hand uses a conceptual framework whose application is in no way restricted to model-driven software product line engineering. While the majority of constraints and algorithms presented here are obsolete when following a *static*, waterfall-like development process, they become relevant whenever *dynamic*, i.e., iterative and incremental editing of (potentially) variational software artifacts is required, e.g., in *agile* software development. Furthermore, when confined to the logical dimension, our constraints and algorithms may support two-layer filtered SPLE architectures, which do not include a revision graph. Only slight adaptations are necessary in case a different variability model (as opposed to feature models) or product space representation (as opposed to EMF models) is to be supported.

The material presented in this paper has been implemented completely as extension to the Eclipse-based tool *SuperMod*. The user interacts with a local repository; multiple copies of local repositories are synchronized with a master repository by *pull* and *push*. The tool has been applied successfully to two standard case studies: the well-known *graph* example and a *Home Automation* product line. In this paper, we have deduced from the aggregated data sets that the dynamic filtered editing model keeps user complexity small and exposes a high degree of automation; user interaction is only necessary in a small number of development iterations, where ambiguities occur or when migration produces a non-intuitive result.

Future work will address the consistency of derived product variants in the workspace, where conflicts can arise due to concurrent modifications.

REFERENCES

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg. Springer.

Berre, D. L. and Parrain, A. (2010). The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6.

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 Westfechtel, B. (1998). Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282.

Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged configuration using feature models. In Nord, R., editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer Berlin Heidelberg.

Estublier, J. and Casallas, R. (1995). Three dimensional versioning. In Estublier, J., editor, *Software Configuration Management: Selected Papers SCM-4 and SCM-5*, volume 1005 of *Lecture Notes in Computer Science*, pages 118–135, Seattle, WA. Springer-Verlag.

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.

Linsbauer, L., Egyed, A., and Lopez-Herrejon, R. E. (2016). A variability aware configuration management and revision control platform. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 803–806, New York, NY, USA. ACM.

Lopez-Herrejon, R. E. and Batory, D. S. (2001). A standard problem for evaluating product-line methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, GCSE '01, pages 10–24, London, UK. Springer.

Mitschke, R. and Eichberg, M. (2008). Supporting the evolution of software product lines. In Oldevik, J., Olsen, G. K., Neple, T., and Paige, R., editors, *ECMDA Traceability Workshop Proceedings 2008*, pages 87–96, Berlin, Germany.

Montalvillo, L. and Díaz, O. (2015). Tuning GitHub for SPL development: branching models & repository operations for product engineers. In *Proceedings of the 19th International Conference on Software Product Line, SPLC 2015, Nashville, TN, USA, July 20-24, 2015*, pages 111–120.

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). OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1.

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

Reichenberger, C. (1995). VOODOO - a tool for orthogonal version management. In Estublier, J., editor, *SCM*, volume 1005 of *Lecture Notes in Computer Science*, pages 61–79. Springer.

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.

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*, pages 5–18, Angers, France. SCITEPRESS.

Schwägerl, F., Buchmann, T., Uhrig, S., and Westfechtel, B. (2016a). 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.

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

Schwägerl, F., Buchmann, T., and Westfechtel, B. (2016b). Filtered model-driven product line engineering with SuperMod: The home automation case. In Lorenz, P., Cardoso, J., Maciaszek, L. A., and van Sinderen, M., editors, *Software Technologies*, volume 586 of *Communications in Computer and Information Science (CCIS)*, chapter 2, pages 19 – 41. Springer.

Schwägerl, F. and Westfechtel, B. (2016). Collaborative and distributed management of versioned software product lines. In *Proceedings of the 11th International Conference on Software Paradigm Trends*, pages 83–94, Lisbon, Portugal. SCITEPRESS.

Seidl, C., Schaefer, I., and Aßmann, U. (2014). Integrated management of variability in space and time in software families. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 22–31, New York, NY, USA. ACM.

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.

Stănciulescu, Ş., Berger, T., Walkingshaw, E., and Wąsowski, A. (2016). Concepts, operations and feasibility of a projection-based variation control systems. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*, ICSME'16.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45.

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

Zeller, A. and Snelting, G. (1997). Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398–441.

Zschaler, S., Sánchez, P., Santos, J., Alférez, M., Rashid, A., Fuentes, L., Moreira, A., Araújo, J., and Kulesza, U. (2010). VML* — a family of languages for variability management in software product lines. In van den Brand, M., Gaševic, D., and Gray, J., editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 82–102. Springer Berlin / Heidelberg, Denver, CO, USA.