

# Realizing a Conceptual Framework to Integrate Model-Driven Engineering, Software Product Line Engineering, and Software Configuration Management

Felix Schwägerl<sup>(✉)</sup>, Thomas Buchmann, Sabrina Uhrig,  
and Bernhard Westfechtel

Applied Computer Science I, University of Bayreuth, 95440 Bayreuth, Germany  
{felix.schwaegerl,thomas.buchmann,sabrina.uhrig,  
bernhard.westfechtel}@uni-bayreuth.de

**Abstract.** Software engineering is a highly integrative computer science discipline, combining a plethora of different techniques to increase the quality of software development as well as the resulting software. The three sub-disciplines Model-Driven Software Engineering (MDSE), Software Product Line Engineering (SPLE) and Software Configuration Management (SCM) are well-explored, but literature still lacks an integrated solution. In this paper, we present the realization of a conceptual framework that integrates those three sub-disciplines uniformly based on a filtered editing model. The framework combines the check-out/modify/commit workflow known from SCM with the formalism of feature models and feature configurations known from SPLE. The implementation is model-driven and extensible with respect to different product and version space models. Important design decisions are formalized by means of Ecore metamodels. Furthermore, we propose several optimizations that increase the scalability of the conceptual framework.

**Keywords:** Model-driven software engineering · Software product line engineering · Software configuration management

## 1 Introduction

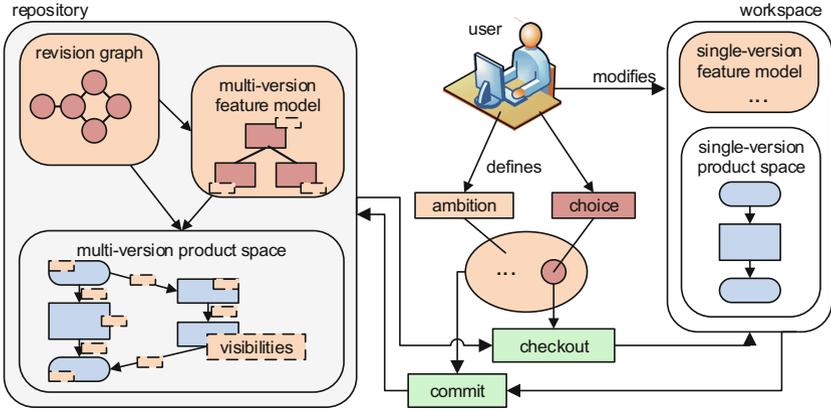
The discipline *Model-Driven Software Engineering (MDSE)* [25] is focused on the development of *models* as first-class artifacts in order to describe software systems at a higher level of abstraction and to automatically derive platform-specific source code. In this way, MDSE promises to increase the productivity of software engineers, who may focus on creative and intellectually challenging modeling tasks rather than on repeated activities at source-code level. Models are typically expressed in well-defined languages such as the *Unified Modeling Language (UML)*, which define the structure as well as the behavior of model elements. The *Eclipse Modeling Framework (EMF)* [21] provides the technological foundation for many model-driven applications.

*Software Product Line Engineering (SPLE)* [13] enforces an organized reuse of software artifacts in order to support the systematic development of a set of similar software products. Commonalities and differences among different members of the product line are typically captured in *variability models*, e.g., *feature models* [8]. Different methods exist to connect the variability model to a *platform*, which provides a non-functional implementation of the product domain. The concept of *negative variability* considers the platform as a multi-variant product, which constitutes the *superimposition* of all product variants. To automatically derive a single-variant product, the variability within the feature model needs to be resolved by specifying, e.g., a *feature configuration*.

*Software Configuration Management (SCM)* is a well-established discipline to manage the *evolution* of software artifacts. A sequence of product *revisions* is shared among a *repository*. Besides storage, traditional SCM systems [2, 3, 23] assist in the aspects of *collaboration* and *variability* to a limited extent, by providing operations like *diff*, *branch* and *merge*. Internally, the components of a versioned software artifact – most frequently, the lines of a text file – are represented as *deltas*. The most commonly used delta storage type are *directed deltas*, which consist of the differences between consecutive revisions in terms of change sequences, whereas *symmetric deltas* [15] constitute a *superimposition* of all revisions, annotated with *version identifiers*.

In literature, many approaches to pair-wise combinations of MDSE, SPLE, and SCM are described. *Model-Driven Product Line Engineering (MDPLE)* is motivated by a common goal of MDSE and SPLE — increased productivity. MDPLE may be realized using positive variability, e.g., by *composition techniques* [24], or using negative variability by creating a *multi-variant domain model* whose elements are mapped to corresponding feature model elements [6]. *Model Version Control* subsumes the combination of MDSE and version control [1], with the goal of lifting existing version control metaphors *check-out* and *commit* up to the model 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. A survey can be found in [11].

In [17], we have presented a conceptual framework to realize an integrated combination of the three disciplines. It is built around an editing model oriented towards version control systems, where the developer may use his/her preferred tool to perform changes to versioned software artifacts within a single-version workspace. The workspace is synchronized with a repository, which persists the entirety of product versions. In addition to revision graphs, feature models are provided to adequately define logical product variants. Version selection is performed in both the revision graph and the feature model. In the latter case, a feature configuration is selected, which allows for the combination of various logical properties in a consistent product variant. The adoption of a version control oriented editing model to SPL development implies advantages such as unconstrained variability: single-version constraints do not affect the multi-version repository. Furthermore, the distinction between variability in time and



**Fig. 1.** The integrated editing model underlying our conceptual framework.

variability in space is blurred: our conceptual framework allows to postpone the decision, whether a change to a product constitutes a temporal evolution step or a new product variant or feature, until the commit.

The current paper deals with the model-driven realization of a conceptual framework [17], discussing important design decisions using the formalism of Ecore models. Furthermore, extensions to the generated source codes are presented, which implement behavioral parts of the framework. In addition, this paper proposes several optimizations that improve the scalability of the framework’s implementation.

## 2 The Conceptual Framework

The conceptual framework presented in [17] provides an integrated solution to MDSE, SPLE, and SCM based on the *uniform version model* (UVM) presented in [27]. UVM defines a number of basic concepts (*options*, *visibilities*, *version rules*) for version control. A prototype of our conceptual framework, *SuperMod*, uses EMF both for its own implementation and as the primarily targeted product space. In [18], SuperMod is presented from the end user’s perspective and the added value of the integration of MDSE, SPLE and SCM is discussed.

### 2.1 Overview

As illustrated in Fig. 1, the conceptual framework defines an *editing model* oriented towards version control metaphors. The basic assumption is that the user edits a single version selected by a *choice* (the *read filter*), but the changes affect *multiple* versions, which are defined by a so called *ambition* (the *write filter*). Editing a product version consists of three – partly automated – steps:

1. *Check-out*: The user performs a *version selection* (a *choice*) in the repository. In the revision graph, the selection comprises a single *revision*. From the feature model, a *feature configuration* has to be derived. A single-version copy of the repository, filtered by the selected version, is loaded into the workspace.
2. *Modify*: The user applies a set of changes to the single-version product and/or to the feature model in the workspace.
3. *Commit*: The changes are written back to the repository. For this purpose, the user is prompted for an additional selection of a *partial* feature configuration (an *ambition*) to delineate the logical scope of the performed changes. Visibilities of versioned elements are updated automatically, and a newly created revision is submitted to the repository.

The conceptual framework proposes a three-layered hierarchy of version and product spaces. On top of the hierarchy is a *revision graph*, which controls the evolution of both the *product space* and the *feature model*, which plays a dual role: From the revision graph’s perspective, it is versioned the same way as the product space; for the product space, it incorporates an additional variability model.

The *product space* is represented as a *superimposition* of product versions. The connection between the product space and the version space is established by *visibilities*, which are assigned to elements of the product space and the feature model and in turn refer to the version space. The primary product space is a heterogeneous file system, consisting of EMF models and further contents such as plain text or XML files. The interaction between different spaces is described below.

Please note that the conceptual framework’s implementation shown in the current paper is restricted to single-user operation. Therefore, the repository is persisted locally in the user’s development environment. Collaborative versioning will be addressed by future research.

## 2.2 Version Space

The term *version space* subsumes the revision graph and the feature model. After introducing a set of general concepts, we show the mapping of those concepts to the feature model and to the revision graph, before an integration is described.

**General Concepts.** The version space is defined by a set of concepts described in [27] using set theory and propositional logic.

*Options.* An *option* represents a (logical or temporal) property of a software product that is either present or absent. The version space defines a global *option set*:

$$O = \{o_1, \dots, o_n\}. \quad (1)$$

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

$$c = b_1 \wedge \dots \wedge b_n, \quad b_i \in \{o_i, \neg o_i\} \quad (i \in \{1, \dots, n\}) \quad (2)$$

An *ambition* is an option binding that allows for unbound options ( $b_i = true$ , such that this component can be eliminated from the conjunction):

$$a = b_1 \wedge \dots \wedge b_n, b_i \in \{o_i, \neg o_i, true\} (i \in \{1, \dots, n\}) \quad (3)$$

Options occurring positively or negatively in the conjunction are *bound*. Thus, a choice is a *complete binding* and designates a specific version, whereas an ambition may have unbound options (*partial binding*) in order to describe a *set* of versions. The version specified by the choice is used for editing, whereas the change affects all versions specified by the ambition. The ambition must include the choice; otherwise, the change would be performed on a version located outside the scope of the change. Formally, this means that the choice must imply the ambition:

$$c \Rightarrow a. \quad (4)$$

*Version Rules.* The version space defines a set of *version rules* — boolean expressions over a subset of defined options. The *rule base*  $\mathcal{R}$  is composed of a set of rules  $\rho_1, \dots, \rho_m$  all of which have to be satisfied by an option binding in order to be consistent. Thus, we may view the rule base as a *conjunction*:

$$\mathcal{R} = \rho_1 \wedge \dots \wedge \rho_m \quad (5)$$

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

$$c \Rightarrow \mathcal{R} \quad (6)$$

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

$$\mathcal{R} \wedge a \neq false. \quad (7)$$

**Feature Model.** Concepts such as options, rules and choices should not be exposed to the user directly because they are represented at a too low conceptual level. *Feature models* [8] meet the requirements of SPLE in a satisfactory way.

*Feature Options.* A *feature* is a discriminating logical property of a software product. It is adequate to map each feature to a *feature option*  $f \in O_f$ , where  $O_f \subseteq O$ .

*Feature Dependencies and Constraints.* Feature models offer several high-level abstractions: First of all, features are organized in a tree, which makes them existentially depend on each other. Non-leaf features may be grouped as AND- or OR-features. If an AND-feature is selected, its mandatory child features have to be selected as well. In the case of an OR-feature, exactly one child has to be selected (exclusive disjunction). Additionally, cross-tree relationships may be defined: *requires* and *excludes* constraints. It is straightforward to map feature models to propositional logic (see Table 1).

**Table 1.** Mapping feature models to version space rules.

Pattern	Transformation
root feature $f_r$	$f_r$
child feature $f_c$ of parent feature $f$	$f_c \Rightarrow f$
AND feature $f$ and mandatory child $f_c$	$f \Rightarrow f_c$
OR feature $f$ and child features $f_1, \dots, f_n$	$f \Rightarrow (f_1 \otimes \dots \otimes f_n)$
$f_1$ excludes $f_2$	$\neg(f_1 \wedge f_2)$
$f_1$ requires $f_2$	$f_1 \Rightarrow f_2$

*Version Selection.* As aforementioned, *feature configurations* describe the characteristics of a single product of the product line and thus may be considered as a *version selection* within the feature model. A feature configuration is derived from a feature model by assigning a *selection state* to each feature. A feature configuration can be mapped to a choice or ambition by setting the binding  $b_i$  for a *feature option*  $f_i \in O_f$  as follows:

$$b_i = \begin{cases} f_i & \text{if feature } f_i \text{ is selected.} \\ \neg f_i & \text{if feature } f_i \text{ is deselected.} \\ true & \text{if no selection is provided for } f_i. \end{cases} \quad (8)$$

Please note that *partial* feature configurations allow for unbound options ( $b_i = true$ ). These are allowed only for ambitions, not for choices.

**Revision Graph.** In SCM, the *evolution* of a software product is addressed. The history of a repository is typically represented by a *revision graph*. Revision control deviates from variability management in two aspects. First, revisions are organized *extensionally*, i.e., only product revisions that have been committed earlier may be checked-out [4]. Second, revisions are *immutable*: Once committed, they are expected to be permanently available.

*Revision Options.* Temporal versioning can be realized by *revision options* (transactions options in [27])  $r \in O_r$ , where  $O_r \subseteq O$ . For each commit, a new revision option is introduced automatically. In order to achieve *immutability*, neither a revision option itself nor a visibility referring to it may ever be deleted.

*Revision Rules.* Automatically derived *revision rules* reduce the number of selectable versions within the revision graph considerably: it equals the number of available revision options. As summarized in Table 2, implications are introduced for consecutive revisions transparently.

*Choice Specification.* A version in the revision graph is selected as a single revision  $r_c$  by the user. Since each revision option requires the corresponding options of its predecessor revision ( $r_i \Rightarrow r_{i-1}$ ), a choice in the revision graph is created

**Table 2.** Mapping revision graphs to version space rules.

Pattern	Transformation
initial revision $r_0$	$r_0$
new revision $r_i$ as immediate successor of revision $r_{i-1}$	$r_i \Rightarrow r_{i-1}$

by conjunction of the selected revision with all of its predecessors. All other revisions appear in a negative binding. For each revision option  $r_i \in O_r$  within a revision choice  $c_r$ , the option binding  $b_i$  is determined as:

$$b_i = \begin{cases} r_i & \text{if } r_i \text{ is the selected revision } r_c \text{ or a predecessor of it.} \\ \neg r_i & \text{else.} \end{cases} \quad (9)$$

Choices referring to the revision graph are necessarily complete since the binding *true* may never appear.

*Ambition Specification.* In contrast to choices, ambitions in the revision graph only consist of one bound option, namely a transparently introduced revision option that is a successor of the previously selected revision choice. As a consequence, a *revision ambition*  $a_r$  consists of exactly one positive option  $r_n$ ; positive bindings for predecessors are set implicitly.

$$a_r = r_n, \quad r_n \text{ is a successor of } r_c. \quad (10)$$

**Hybrid Versioning.** The combination of the revision graph and the feature model causes interaction between elements of both spaces. Thus, we provide the following extensions to our framework.

*Hybrid Version Space.* Both the option set and the rule base are decomposed into two disjoint subsets for the feature model and for the revision graph, respectively:

$$O = O_f \dot{\cup} O_r \quad (11)$$

$$\mathcal{R} = \mathcal{R}_f \dot{\cup} \mathcal{R}_r. \quad (12)$$

*Hybrid Choice Selection.* A *version selection* has to be performed in both the revision graph and the feature model. Correspondingly, a *hybrid choice* is a complete option binding on  $O_f \dot{\cup} O_r$ . It must be ensured that each selected feature option is visible under  $c_r$ , the choice among the revision graph.

$$c = c_r \wedge c_f. \quad (13)$$

*Hybrid Ambition Specification.* From the user's perspective, the specification of a *hybrid ambition* does not differ from a specification in the feature model. A hybrid ambition  $a$  is a conjunction of the selected feature configuration  $a_f$  and a transparently introduced revision option  $r_n$ .

$$a = r_n \wedge a_f \quad (14)$$

Since a revision ambition is always weakly consistent, a hybrid ambition only needs to be *weakly consistent* with respect to the feature part of the rule base:

$$\mathcal{R}_f \wedge a_f \neq \text{false} \quad (15)$$

Similarly, it is sufficient to require that the feature part of the choice and the ambition imply each other:

$$c_f \Rightarrow a_f. \quad (16)$$

### 2.3 Product Space

Our conceptual framework makes only few assumptions with respect to the primary product space. These assumptions are discussed in this section, providing the basis for the realization of a multi-variant heterogeneous file system, consisting of EMF, XML, and plain text resources, in Sect. 3.

*Set-Theoretic Definition.* We assume that the product space consists of a base set of products. It depends on the implementation of the concrete product space, in which granularity elements are modeled, e.g., lines of code or model objects.

$$P = \{e_1, \dots, e_n\}. \quad (17)$$

*Hierarchical Organization.* Elements are arranged in a tree, and the product space defines a unique *root element*  $e_r$ . Furthermore, each non-root element  $e \in P \setminus \{e_r\}$  element has a unique *container* that is returned by the container function  $\text{cont}(e)$ . Basically, the hierarchy is invariant, i.e., the container of an element may not vary among multiple versions.

*Visibilities.* Each element  $e$  of the product space defines a *visibility*  $v(e)$ , a boolean expression over the variables defined in the option set. An element  $e$  is *visible* under a choice  $c$  if its visibility is implied by the choice, i.e., it evaluates to *true* given the option bindings of the choice:

$$c \Rightarrow v(e). \quad (18)$$

*Filtering.* The operation of *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.

$$\text{filter}(P, c) = P \setminus \{e \in P \mid c \not\Rightarrow v(e)\} \quad (19)$$

Unfortunately, we cannot assert any properties to the result of this function. In particular, it may be syntactically ill-formed, which raises new questions with respect to product consistency control. Those will be further discussed at the end of Sect. 3.2.

### 3 Model-Driven Realization

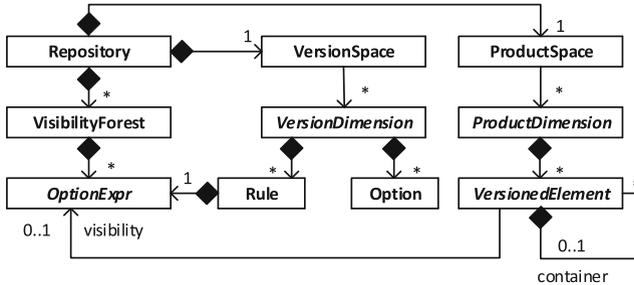
This section describes the underlying design decisions made in advance to the realization of the conceptual framework presented in Sect. 2. Unless specified differently, the presented implementation is included in our research prototype *SuperMod* [18]. First, we introduce a set of Ecore metamodels for the repository. Next, the local workspace and the synchronization between repository and workspace are addressed. Subsequently, the operations *check-out* and *commit* are specified. All behavioral components have been implemented in a modular and extensible way using the Google Guice [22] dependency injection framework.

#### 3.1 Metamodels for the Repository

In the realization of the conceptual framework, the repository, where products are contained in their multi-variant representation, is represented as an EMF model instance. This section explains underlying design decisions by presenting the repository’s core metamodel, which is divided into several Ecore models.

**The Core Metamodel.** The presented realization is highly configurable with respect to the concrete product and version space used in a specific versioning scenario. Figure 2 shows an Ecore Metamodel for the *core*, which is extended by specific product and version dimensions. Below, we assume a three-layer architecture consisting of a revision graph, a feature model, and a file system as primary product space (cf. Fig. 1).

A *repository* combines a version space and a product space, which are in turn divided up into several *product* and *version dimensions*. A product dimension contains a tree of *versioned elements*, to which a *visibility* may be assigned. Those visibilities are organized in an optimized data structure, the *visibility forest*, which is explained in Sect. 4.2. A version dimension contains *options* and (*version*) *rules*, which have been formalized in Sect. 2.2. Both visibilities and rules are represented as *option expressions*.



**Fig. 2.** The core metamodel implements the conceptual framework and is extensible with respect to specific product spaces and version spaces. [18]

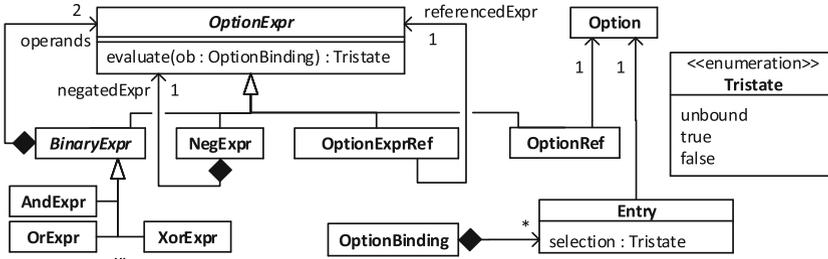


Fig. 3. Metamodel for option expressions and option bindings.

**Option Expressions and Option Bindings.** Option expressions are logical expressions on the option set. As shown in Fig. 3, there exist three categories of option expressions: *Option references* target an existing option. *Compound expressions* are used in order to combine option expressions (e.g., the negation  $\neg$  is represented by `NegExpr`, the conjunction  $\wedge$  by `AndExpr`). *Option expression references* re-use existing expressions in order to avoid their repeated duplication (see Sect. 4).

Choices and ambitions appear as temporary data structures; they are internally represented as *option bindings*, sets of entries binding an option to a selection state. The enumeration `Tristate` defines the three states allowed in *three-valued logic*. The value *unbound* indicates that no selection has been performed for a specific option, which is only allowed within ambitions. Option expressions may be evaluated with respect to a given option binding. This has been realized by corresponding implementations of the operation `evaluate` in the subclasses of `OptionExpr`. During evaluation, options are virtually replaced by the bound tristate value. Table 3 shows how three-valued literals are combined.

**Table 3.** Value table for three-valued logic and basic logical operators. Symmetric cases have been omitted. Complex logical operators such as  $\Rightarrow$  are derived by laws of propositional logic.

<i>a</i>	<i>b</i>	$\neg a$	$a \wedge b$	$a \vee b$
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>true</i>	<i>unknown</i>	<i>false</i>	<i>unknown</i>	<i>true</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>
<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>	<i>unknown</i>
<i>unknown</i>	<i>false</i>	<i>unknown</i>	<i>false</i>	<i>unknown</i>
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>

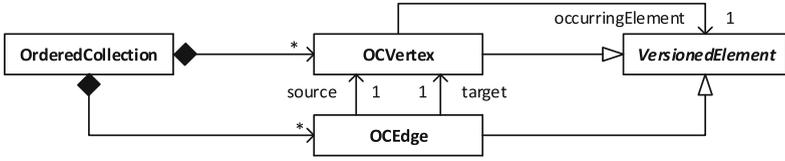


Fig. 4. Multi-version representation of ordered collections as directed graphs.

**Ordered Collections.** Ordered collections appear in several places in the product space. A text file is a sequence of lines; furthermore, in case a multi-valued structural feature is indicated as *ordered* in an Ecore metamodel, the order of its instances should be preserved whenever possible. The metamodel for multi-version ordered collections, shown in Fig. 4, is used in several concrete product space metamodels.

The underlying design decision is to represent a single version of a collection as a linear directed graph, where succeeding elements are connected by an edge. A corresponding multi-version representation in the repository may form an arbitrary directed graph, whose vertex set and edge set are variable. Furthermore, the vertices of a graph refer to *occurrences* of elements rather than to elements themselves. In order to convert a multi-version collection into a single-version representation (i.e., a *list*), this graph is linearized by a topological sort algorithm. This problem is discussed in detail in [20] in the context of three-way merging of ordered collections.

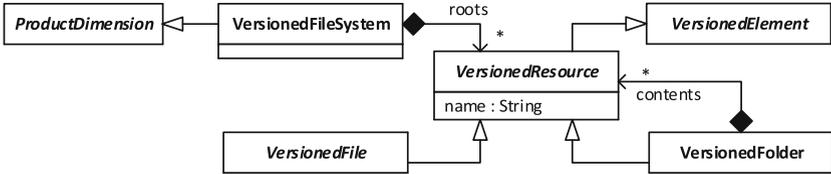
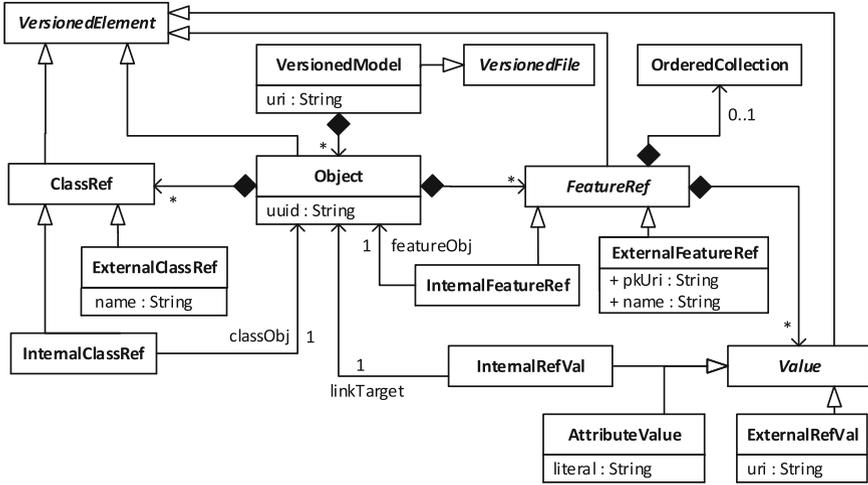


Fig. 5. Ecore class diagram for the metamodel of multi-version heterogeneous file systems.

**Heterogeneous File Systems.** The class diagram shown in Fig. 5 defines the primary product space, *heterogeneous file systems*, which organize files and folders in a tree. The presented realization supports different *file content types*. As one representative, EMF models are discussed below. In a similar way, support for plain text files and XML files has been realized.

**EMF Models.** EMF models are structured in a specific way: A (single-version) EMF resource contains a set of hierarchically organized objects. The state of each



**Fig. 6.** Simplified Ecore class diagram for the metamodel of the multi-version EMF product space.

object is encoded in the specific values of its structural features, which are divided up into attributes and references. For attribute values, primitive data values are allowed. The values of references are links to existing objects. The metamodel in Fig. 6 realizes the following design decisions with respect to multi-variant EMF models:

- (a) *Unconstrained Variability:* Each detail of an object may vary arbitrarily. For this purpose, the meta-classes for structural features and their values extend **VersionedElement**.
- (b) *Optionally Versioned Meta-data:* The metamodel of the versioned model may or may not be versioned itself. Classes and structural features are divided up into the categories internal and external. Internal classes/features define a reference to a co-versioned meta-object, while external classes/features are identified by their package URI and class name, or their feature name, respectively.
- (c) *Variable Object Classes:* An object may be instance of different classes in different versions. Therefore, the conformance relationship between objects and their corresponding classes is expressed by an ordinary object link that may vary among different versions. Technically, a multi-version EMF model represents two modeling layers (model and metamodel) at the same modeling level, which enables co-versioning of models and metamodels.
- (d) *Variable Object Containers:* An object may have different containers in different versions. Thus, the containment hierarchy of objects inside a resource is flattened.

Furthermore, we assume a unique identifier (`uuid`) assigned to each object. Attribute values are represented by string literals; reference values may be internal, by defining a link to an existing object, or external, by specifying a workspace-global object URI. Variability among the order of multi-valued features is achieved by an `OrderedCollection` (see above) that refers to instances of `Value` that must be contained in the corresponding `FeatureRef`.

**Revision Graphs.** Revision graphs are realized as a *version dimension* comprising a directed graph of *revisions*, for which details such as the revision number, the commit date and message, as well as the username are recorded (see metamodel in Fig. 7). In order to conceptually prepare three-way merging, a revision may have multiple predecessors and successors. Furthermore, specific references to low-level version space elements ensure that the mapping shown in Table 2 may be realized. On instance level, the corresponding options and rules are introduced transparently during the *commit* operation.

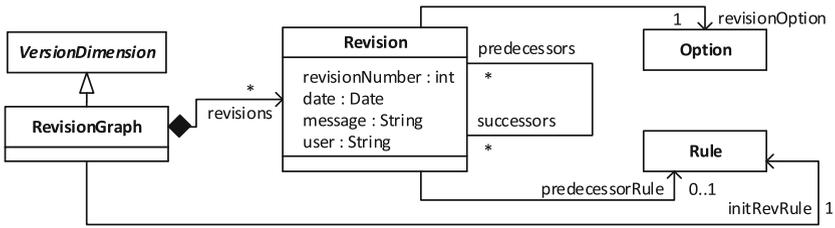


Fig. 7. Ecore metamodel for revision graphs in the repository.

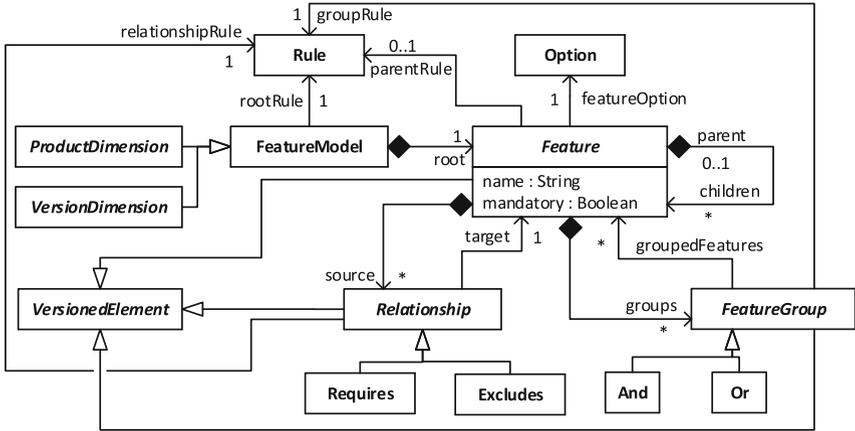


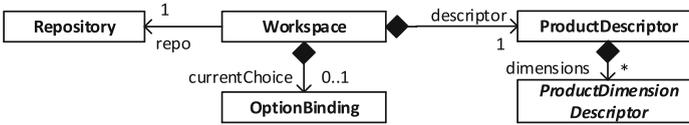
Fig. 8. Ecore class diagram for the metamodel of multi-version feature models.

**Feature Models.** As mentioned above, the feature model plays a dual role. Therefore, the corresponding class in the Ecore model in Fig. 8 extends both `ProductDimension` and `VersionDimension`. A feature model arranges features within a tree. Each feature is uniquely identified by its name. Multiple sibling features may be arranged in either an *AND* or *OR* group. Furthermore, cross-tree *requires* and *excludes* relationships are allowed. Since features, feature groups and cross-tree relationships are subject to evolution, the corresponding classes extend `VersionedElement`.

For each feature, a feature option is introduced transparently in advance to a *commit*. Furthermore, the semantics of feature groups and cross-tree relationships explained in Table 1 is enforced by links to corresponding *rules*, for which the metamodel provides corresponding references.

### 3.2 Workspace and Local Synchronization

After having explained the repository, whose internals are not directly exposed to the user, we now switch to workspace abstractions which allow the user to communicate with the repository. First, we describe the realization of the meta-data section of the workspace. Next, the synchronization between single- and multi-version representation is explained. Last, mechanisms for product consistency control are outlined.



**Fig. 9.** Simplified metamodel for workspace meta-data in Ecore class diagram representation.

**Workspace Meta-data.** Workspace meta-data keep track of local modifications and allow to restore the previous state of the workspace during the *commit* operation (see metamodel in Fig. 9). Meta-data comprise the *current choice*, which has been used for the latest check-out operation, a reference to the *repository*, and a *product descriptor*, which is composed of specific dimension descriptors, each referring to a product space dimension of the repository. We informally outline two specific subclasses of `ProductDimensionDescriptor` used for the primary product space and for the feature model.

- The *file system descriptor* keeps track of the versioning state (*added*, *removed*, *modified*, *unmodified*, *non-versioned*) of files and folders in the workspace. This descriptor is invisible to the user.
- The *feature model descriptor* contains a copy the revision of the feature model that has been selected in the previously specified choice. For the purpose of its modification, an EMF-generated *feature model* editor is provided.

**Import and Export Transformations.** Within the repository, the product space is represented in a multi-variant representation. In order to make the selected product version available for modification, it needs to be converted into a specific single-version representation. The conversion between single-variant and multi-variant products, and vice versa, is realized by two transformations, namely *import* and *export*. For those, the core module provides an interface, which needs to be implemented by each specific product dimension, e.g., for the file system and for the feature model.

- The import/export transformation pair for the *file system* converts between a physical file system and an instance of the metamodel shown in Fig. 5. The operations are further divided up into specific resource types, i.e., plain text, EMF, and XML. For instance, the EMF-specific *export* transformation maps each multi-version object of the repository to a corresponding EObject in the workspace, setting attribute and reference values accordingly. The flattened containment links are converted back into a hierarchical object tree.
- The *feature model* in the workspace (see meta-data) is represented as an instance of the multi-variant feature metamodel shown in Fig. 8. Therefore, the import/export transformations correspond to the *identity function*.

**Product Consistency Control.** Multi-version models within the repository are not restricted by single-version constraints and may therefore vary arbitrarily. *Version rules* introduced in Sect. 2.2 cannot guarantee that the outcome of the *export* transformation, i.e., the conversion between multi- and single-variant representation, is syntactically correct. Thus, a mechanism for *product consistency control* is required in addition.

For the purpose of conflict detection, we introduce an additional operation, which has to be implemented by specific product dimensions. The *validation* operation takes as input a filtered multi-version representation and returns a set of *conflicts*, which forbid the transformation into a corresponding single-version representation. It has been implemented for the file system and for the feature model as follows:

- Once again, the validation of a multi-version *file system* is passed to resource-specific validation operations. For multi-variant EMF models, generic constraints such as referential integrity, spanning containment hierarchy, type correctness, and the cardinality of structural features are checked. In the case of an *ordered collection*, a conflict is raised whenever a topological sort of the corresponding collection graph (cf. Fig. 4) does not produce a unique result.
- For the feature model, the following constraints are enforced: unique root feature, unique parent feature, unique feature name, non-contradicting requires/excludes relationships, and unique group membership.

After having detected conflicts, they must be *resolved* by the user, which is not in focus of this paper. An approach to *interactive conflict resolution* is described in [19] in the context of three-way merging of EMF models.

### 3.3 Realization of Check-Out and Commit

Below, we finalize our editing model sketched in Sect. 2.1 by detailing the operations *check-out* and *commit*. In the description below, we refer to the conceptual framework presented in Sect. 2 as well as to local synchronization operations defined in Sect. 3.2.

#### Check-Out

1. The user is prompted for a revision  $r_c$  from the revision graph, by default the latest revision. The derived *revision choice* is  $c_r = r_0 \wedge \dots \wedge r_c \wedge \neg r_{c+1} \wedge \dots \wedge \neg r_n$ .
2. The multi-version feature model is *filtered* by elements  $e_f$  that satisfy the revision choice ( $c_r \Rightarrow v(e_f)$ ). The filtered feature model is *exported* into the workspace and made available for modification.
3. The user performs a *feature choice*  $c_f$  on the filtered feature model by specifying a *completely bound* feature configuration. Options for invisible features  $f_i$  are automatically negatively bound:  $b_i = \neg f_i$ . The feature choice must be *strongly consistent* according to the rule base:  $c_f \Rightarrow \mathcal{R}_f$ .
4. The *effective choice*  $c$  is calculated as the conjunction  $c = c_r \wedge c_f$  and recorded within the meta-data section of the workspace.
5. The primary product space is *filtered* by selecting elements  $e_p$  that satisfy the effective choice ( $c \Rightarrow v(e_p)$ ). The filtered product space is *validated*; in the case of conflicts, the user is prompted for conflict resolution. Finally, the conflict-free filtered product space is *exported* into the workspace and provided for modification.

#### Modify

6. Between *check-out* and *commit*, the user may apply arbitrary modifications to the primary product space and/or to the filtered feature model provided in the local workspace. Model or non-model files within the workspace belonging to the primary product space may be modified with arbitrary editors, e.g., GMF-based graphical or Xtext-based textual editors. For the modification of the feature model, a generated EMF tree editor is provided, which ensures single-version constraints, although the feature model itself is represented in its multi-version metamodel.

#### Commit

7. The previous version of the product space is reproduced using the recorded choice. The current state of the product space is obtained by applying the *import* operation to the current workspace contents. Next, *differences* between the previous and the current state of the product space are detected. *Updates* are broken down to insertions and deletions of element versions.
8. A new revision option  $r_n$  is added to  $O_r$  transparently. The rule  $r_n \Rightarrow r_c$  is added to  $\mathcal{R}_r$ .

9. Next, the user specifies an *incomplete feature configuration*  $a_f$  that delineates the logical scope of the change. The feature ambition must be *weakly consistent* according to the rule base:  $\mathcal{R}_f \wedge a_f \neq \text{false}$ . Furthermore, the feature ambition must be *implied* by the feature choice:  $c_f \Rightarrow a_f$ .
10. The applied modifications are written back under the *effective ambition*  $a$ . For changes to the feature model,  $a = r_n$ ; for the primary product space, the *hybrid ambition*  $a = r_n \wedge a_f$  is applied. Each modified element  $e$  is processed as follows:
  - Inserted elements  $e_{ins}$  are appended to the primary product space or to the feature model, respectively. Their visibility is set to the ambition:  $v(e_{ins}) := a$ .
  - For re-inserted elements  $e_{reins}$ , which have not been visible under  $c$ , the visibility is modified as follows:  $v(e_{reins}) := v_{old}(e_{reins}) \vee a$ .
  - Deleted elements  $e_{del}$  remain in the repository. Their visibility is modified accordingly:  $v(e_{del}) := v_{old}(e_{del}) \wedge \neg a$ .

## 4 Optimization

The representation of the product space as a superimposition will inevitably result in a growing memory consumption. Since revisions are immutable, product space elements will never be effectively deleted from the repository. Furthermore, the evaluation of the constantly growing visibilities will be noticeable in terms of higher runtimes for *check-out* and *commit*. In this section, we present three mutually independent optimizations for the implementation of the conceptual framework, which significantly improve the scalability of the framework's implementation. All three optimizations have been realized in the tool SuperMod [18].

### 4.1 Hierarchical Evaluation of Visibilities

As explained in Sect. 2.3, one of the few assumptions with respect to the product space is that its elements are organized *hierarchically*. This inherently implies two drawbacks:

- *Duplication of Visibilities*: The insertion of a tree of new elements under the same logical ambition will result in multiple copies of the same visibility during the application of Sect. 3.3, step 10.
- *Consistency of Parent/Child Relationships*: Many modeling frameworks including EMF assume that non-root elements  $e$  *existentially depend* on their respective container element  $cont(e)$ , if any. This constraint should be ensured in any version described by the superimposition; conflicts may be avoided by requiring that the child element's visibility must imply the parent element's visibility:  $v(e) \Rightarrow v(cont(e))$ .

In order to compensate these drawbacks, we introduce the concept of *effective visibility*  $v_{eff}$  of an element, which is defined by conjunction with the visibility of its container element as follows:

$$v_{eff}(e) = \begin{cases} v(e) & \text{if } e \text{ is a root element.} \\ v(e) \wedge v_{eff}(cont(e)) & \text{otherwise.} \end{cases} \quad (20)$$

Furthermore, the visibility of an element is made *optional*; in case an element  $e$  does not define a visibility, we implicitly assume  $v(e) = true$ . This improvement has been conceptually prepared in the core metamodel: In Fig. 2, the cardinality of the reference `visibility` is 0..1.

The *editing model* shown in Sect. 3.3 is modified as follows: When a tree of elements is inserted/removed in the workspace, only the corresponding root element's visibility needs to be updated during step 10.

Replacing visibilities with effective visibilities improves scalability for the following reasons:

- *Reduced Commit Runtimes*: The above modification of the editing model significantly reduces the number of elements to be processed by visibility updates and therefore the entire runtime of a commit.
- *Reduced Check-out Runtimes*: Likewise, the number of visibilities to evaluate during the *filter* operation, which is applied during a check-out, is reduced. In case an element is filtered, all sub-elements must be filtered, too, removing the necessity to evaluate their visibilities.
- *Improved Consistency Control*: The constraint  $v(e) \Rightarrow v(cont(e))$  is ensured automatically for each non-root element  $e$ .

## 4.2 Visibility Forests

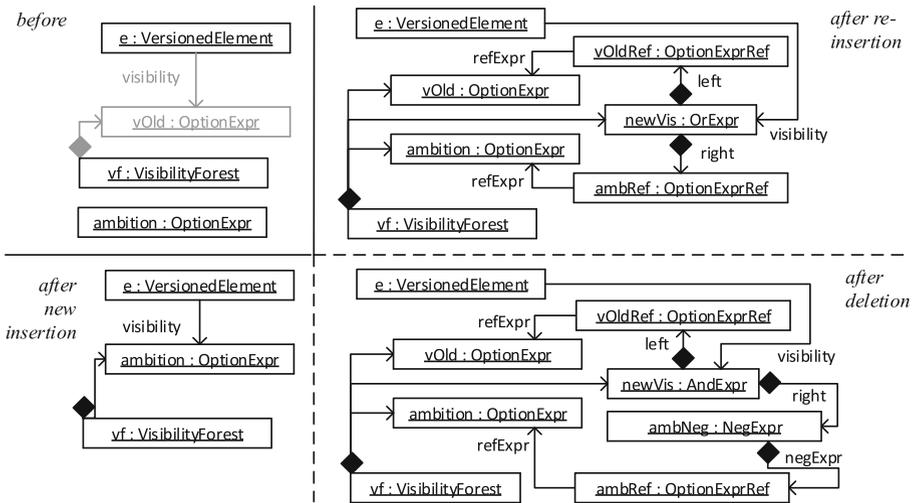
Now, we discuss how the connection between product space elements and visibilities can be realized. One possibility would be to make an element directly contain its visibility, which has two obvious drawbacks:

- *Duplication of Visibilities*: The insertion of a large set of new elements (not necessarily connected by containment) under the same logical ambition will result in repeated copies of the same visibility during the application of Sect. 3.3, step 10.
- *Repeated Evaluation of Equivalent Visibilities*: During the *filter* operation, the visibility of all product space elements is evaluated with respect to the specified choice. However, many elements share an equivalent visibility. Without any optimization, the filter operation would repeatedly evaluate the same visibility with respect to the same choice, causing additional runtime.

These drawbacks are removed by *visibility forests*, a global data structure for the storage of visibilities, which has been conceptually prepared in the core metamodel (cf. Fig. 2). Rather than subordinating an element's visibility by containment, a cross-reference is established between `VersionedElement` and `OptionExpr`. This allows several elements to share the same visibility. Furthermore, *option expression references* (see Fig. 3) allow to re-use existing visibilities in the case of element re-insertions or removals. By the mechanisms described below, several hierarchies of interconnected visibilities are created, giving the *visibility forest* its name.

The following modifications are applied to the editing model from Sect. 3.3:

- *Commit, step 10*: The visibility of an element is modified by adding corresponding new entries to the visibility forest and re-using the old visibility by means of expression references. The graph patterns presented in Fig. 10 describes how visibilities of inserted and deleted elements are updated. For instance, in the case of an element deletion, the old visibility is re-used as the first operand of an **AndExpr**, and the second operand consists of the negated ambition. Both the old visibility and the ambition are connected by *option expression references*, which ensure that these expressions may be re-used within different visibilities. As a consequence, no duplicate visibility will ever be inserted into the visibility forest.
- *Check-out, steps 2 and 5*: The visibility forest ensures that equivalent visibilities are represented by the same runtime object. Therefore, the runtime object’s identity is used to *cache* the evaluation result (i.e., the **Tristate** returned by **evaluate**, cf. Fig. 3) in a hash map. Before an element’s visibility is actually evaluated, a lookup is performed. In the case of a match, the cached result is returned.



**Fig. 10.** Object diagrams describing optimized visibility updates of newly inserted, re-inserted and deleted elements in the visibility forest as graph patterns. It is assumed that the **ambition** is passed as a parameter. In the case of a new insertion, the elements shaded in grey are omitted.

### 4.3 Substitution of Ambition Expressions

The mechanism of writing back changes using an ambition results in corresponding option expressions appearing in the visibility of all affected elements. These

expressions increase the size of the superimposition and/or the visibility forest, regardless of whether hierarchical visibilities are used.

In order to reduce the size of serialized feature expressions, we propose to introduce a third component to the option set and the rule base, namely *change options*  $\Delta \in O_\Delta$  and *change rules*  $\rho_\Delta \in \mathcal{R}_\Delta$  by redefining Eqs. 11 and 12 as follows:

$$O = O_f \dot{\cup} O_r \dot{\cup} O_\Delta \quad (11')$$

$$\mathcal{R} = \mathcal{R}_f \dot{\cup} \mathcal{R}_r \dot{\cup} \mathcal{R}_\Delta \quad (12')$$

The *change space* is completely invisible to the user and used transparently for optimization purposes. After a logical ambition  $a_f$  has been specified by the user (cf. Sect. 3.3, step 9), the editing model is modified as follows:

1. A new *change option*  $\Delta$  is introduced to  $O_\Delta$ .
2. The rule  $\Delta \Rightarrow a_f$  is added to  $\mathcal{R}_\Delta$ .
3. The change is committed to the repository under the ambition  $a'_f := \Delta$ .

Besides improved commit runtimes, this optimization brings an additional advantage: It becomes easier to modify a user-specified ambition a posteriori. In case the user has specified an erroneous ambition, it is only necessary to correct  $a_f$  in the rule base rather than in the visibility forest.

## 5 Related Work

In this paper, the implementation of the conceptual framework presented in [17] has been presented. The design decisions explained here have been realized in the research prototype *SuperMod*, which is presented in [18] from the user’s perspective. The conceptual framework itself is based on the *uniform version model* (UVM) presented in [27]. UVM’s basic concepts (*options*, *visibilities*, *version rules*) have been initially introduced in the context of *change-oriented versioning* (CoV) [12].

A detailed comparison of approaches to pairwise integration of MDSE/SPLE, MDSE/SCM, and SPLE/SCM, can be found in [17]. In the following, we confine our comparison to tools that address both temporal and logical versioning.

With *branches*, traditional version control systems [2, 3] offer logical variants to a limited extent. Albeit, it is only possible to restore variants that have been committed earlier (*extensional* versioning, see [4]). In contrast, our approach allows to create new variants based on a predicate on variant options, i.e., feature configurations (*intensional* versioning). This reduces the overhead of product derivation considerably.

The tool *EPOS-DB* [12] is an implementation of CoV concepts at a low level of abstraction when compared to our approach. Propositional formula are exposed to the user directly, e.g., to specify *choices* and *ambitions*. The product space is based on an *EER* (*Enhanced Entity-Relationship*) model, and is also capable of versioning plain text files. In [12], a global storage for visibilities

is introduced, which shares many conceptual similarities with *visibility forests* discussed in this paper.

In [14], an approach for *orthogonal* version management is proposed. In the version control tool VOODOO, a version cube is formed by product, revision, and variant space. The user interface is capable of versioning a complete file hierarchy, which may itself vary along all three dimensions. Albeit, the approach does not consider that the variant space may be subject to temporal evolution.

The commercial SCM system Adele [5] 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, which relies on directed deltas. Thus, logical and temporal versioning are not integrated at the same conceptual level.

In [28], an approach to *unified versioning* based on *feature logic* is presented. In the version control system ICE, versions of artifacts (i.e., text files) are stored with selective deltas; visibilities are controlled by feature-logical expressions. Constraints on feature combinations are expressed by version rules, which are enforced by means of *unification*. The editing model slightly differs from the approach presented in this paper: The user performs only a *partial* version selection. As a consequence, the workspace may still contain variability, which is exposed to the user in the form of C preprocessor directives [10]. Concurrent changes are orchestrated by means of a pessimistic versioning strategy, i.e., write locks.

In [26], an approach to filtered (*projectional*) editing of multi-variant programs is described. The motivation is a reduction of complexity gained by hiding variants not important for a specific change to a multi-variant model. As in our approach, visibilities are managed automatically. Conversely, the restriction of a *completely bound choice* does not exist since the user operates on a partially filtered product, which still contains variability. Temporal versioning is not addressed by the approach presented in [26].

In the field of SPLE, there exist several approaches to partially apply filtered editing to software product lines. These approaches can be considered as a conceptual extension to *multi-version editors* [16]. The source-code centric tool CIDE (*Colored IDE*) [9] generalizes preprocessors using a colored representation to distinguish features. The changes performed in a filtered view may only affect the selected feature or variant, i.e., *choice* and *ambition* must be equal. The MDPLE tool *Feature Mapper* [7] offers the possibility of *change recording* during domain engineering. However, only *insertions* are recognized while recording.

## 6 Conclusion

We have described the model-driven realization of a conceptual framework [17] that integrates MDSE, SPLE and SCM. The framework defines an editing model that is oriented towards version control metaphors and uses the operations *check-out* and *commit* in order to make a single-version workspace communicate with a multi-version repository. In addition to revision graphs, which manage temporal

variability, feature models and feature configurations are used in order to define logical variability and the logical scope of a change. With respect to the repository's architecture, the presented implementation is highly configurable. In this paper, we have focused a three-layered approach, where a revision graph is used to control the evolution of a feature model and a primary product space, which may consist of arbitrary model- or non-model resources. The feature model plays a dual role since it is also used as an additional variability model.

The conceptual framework is based on a *uniform version model*, which builds upon the formalisms of set theory and propositional logic. The static structure of the framework's core is defined by a basic metamodel that abstracts from concrete product and version dimensions. On top of the core metamodel, the model-driven realization of the tool has been presented by means of several concrete extending metamodels, e.g., for the revision graph, for the feature model, and for the heterogeneous file system. The operations *check-out* and *commit* have been fully specified. Last, we have presented three optimizations that increase the scalability of our approach: hierarchical visibilities, visibility forests, and substitution of ambition expressions.

Future work will address the realization of a multi-user component, which requires a mechanism to synchronize multiple, remotely distributed copies of a repository. This extension will advance our prototype *SuperMod* [18] to a full-fledged distributed version control system. Furthermore, conflict resolution still needs to be improved, especially with regard to collaborative versioning. For evaluation purposes, we are planning a case study of industrial scale, which will allow for a quantitative comparison with related SPLE and SCM approaches.

## References

1. Altmanninger, K., Seidl, M., Wimmer, M.: A survey on model versioning approaches. *Int. J. Web Inf. Syst. (IJWIS)* **5**(3), 271–304 (2009)
2. Chacon, S.: *Pro Git*, 1st edn. Apress, Berkely (2009)
3. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: *Version Control with Subversion*. O'Reilly, Sebastopol (2004)
4. Conradi, R., Westfechtel, B.: Version models for software configuration management. *ACM Comput. Surv.* **30**(2), 232–282 (1998)
5. Estublier, J., Casallas, R.: The Adele configuration manager. In: Tichy, W.F. (ed.) *Configuration Management, Trends in Software*, vol. 2, pp. 99–134. Wiley, Chichester (1994)
6. Gomaa, H.: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, Boston (2004)
7. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: mapping features to models. In: *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pp. 943–944. ACM, New York, May 2008
8. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: *Feature-oriented domain analysis (FODA) feasibility study*. Technical report CMU/SEI-90-TR-21, Carnegie-Mellon University, Software Engineering Institute, November 1990

9. Kästner, C., Trujillo, S., Apel, S.: Visualizing software product line variabilities in source code. In: Proceedings of the 2nd International SPLC Workshop on Visualisation in Software Product Line Engineering (ViSPLE), pp. 303–313, September 2008
10. Kernighan, B.W.: The C Programming Language, 2nd edn. Prentice Hall Professional Technical Reference, Upper Saddle River (1988)
11. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: from legacy system reengineering to product line refactoring. *Sci. Comput. Program.* **78**(8), 1010–1034 (2013). <http://dx.doi.org/10.1016/j.scico.2012.05.003>
12. Munch, B.P.: Versioning in a Software Engineering Database – The Change Oriented Way. Ph.D. thesis, Tekniske Høgskole Trondheim Norges (1993)
13. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations Principles and Techniques. Springer, Berlin (2005)
14. Reichenberger, C.: VooDoo a tool for orthogonal version management. In: Estublier, J. (ed.) ICSE-WS/SCM 1993/1995. LNCS, vol. 1005, pp. 61–79. Springer, Heidelberg (1995)
15. Rochkind, M.J.: The source code control system. *IEEE Trans. Software Eng.* **1**(4), 364–370 (1975)
16. Sarnak, N., Bernstein, R.L., Kruskal, V.: Creation and maintenance of multiple versions. In: Winkler, J.F.H. (ed.) SCM. Berichte des German Chapter of the ACM, vol. 30, pp. 264–275. Teubner (1988)
17. Schwägerl, F., Buchmann, T., Uhrig, S., Westfechtel, B.: Towards the integration of model-driven engineering, software product line engineering, and software configuration management. In: Hammoudi, S., Pires, L.F., Desfray, P., Filipe, J. (eds.) Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2015), pp. 5–18. SCITEPRESS Science and Technology Publications, Portugal (2015)
18. Schwägerl, F., Buchmann, T., Westfechtel, B.: 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 (ICSOFT-PT). SCITEPRESS Science and Technology Publications, Portugal, Colmar, France (2015, to be published, accepted for publication)
19. Schwägerl, F., Uhrig, S., Westfechtel, B.: Model-based tool support for consistent three-way merging of EMF models. In: Proceedings of the workshop on ACadeMics Tooling with Eclipse, ACME 2013, pp. 2:1–2:10. ACM, New York (2013)
20. Schwägerl, F., Uhrig, S., Westfechtel, B.: A graph-based algorithm for three-way merging of ordered collections in EMF models. *Science of Computer Programming* (2015, in press, accepted manuscript). <http://www.sciencedirect.com/science/article/pii/S0167642315000532>
21. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. The Eclipse Series, 2nd edn. Addison-Wesley, Upper Saddle River (2009)
22. Vanbrabant, R.: Google Guice: Agile Lightweight Dependency Injection Framework (Firstpress). APress, New York (2008)
23. Vesperman, J.: Essential CVS. O’Reilly, Sebastopol (2006)
24. Völter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: Proceedings of the 11th International Software Product Line Conference, SPLC 2007, pp. 233–242. IEEE Computer Society, Washington, DC (2007). <http://dx.doi.org/10.1109/SPLC.2007.28>

25. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
26. Walkingshaw, E., Ostermann, K.: Projectional editing of variational software. In: Generative Programming: Concepts and Experiences, GPCE 2014, Vasteras, Sweden, 15–16 September 2014, pp. 29–38 (2014). <http://doi.acm.org/10.1145/2658761.2658766>
27. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. *IEEE Trans. Softw. Eng.* **27**(12), 1111–1133 (2001)
28. Zeller, A., Snelting, G.: Unified versioning through feature logic. *ACM Trans. Softw. Eng. Methodol.* **6**(4), 398–441 (1997)



<http://www.springer.com/978-3-319-27868-1>

Model-Driven Engineering and Software Development  
Third International Conference, MODELSWARD 2015,  
Angers, France, February 9-11, 2015, Revised Selected  
Papers

Desfray, P.; Filipe, J.; Hammoudi, S.; Pires, L.F. (Eds.)

2015, XV, 438 p. 237 illus., Softcover

ISBN: 978-3-319-27868-1