Towards the Integration of Model-Driven Engineering, Software Product Line Engineering, and Software Configuration Management

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

Keywords: Model-Driven Software Engineering, Software Product Lines, Software Configuration Management.

Abstract: Model-Driven Software Engineering (MDSE), Software Product Line Engineering (SPLE) and Software Configuration Management (SCM) have been established as independent disciplines to ease different aspects of software development. The usage of models as high-level abstractions promises to increase productivity, while software product lines manage variability within a family of similar software products; software configuration management systems manage evolution and support collaborative development. In this paper, we explore the state of the art regarding the pairwise combinations MDSE/SPLE, SPLE/SCM, and MDSE/SCM and show that an integrated solution combining all three disciplines is missing. We present a conceptual framework to integrate MDSE, SPLE and SCM uniformly based on a filtered editing model. The framework implies a number of advantages, namely unconstrained variability, a reduction of cognitive complexity, improved consistency, tool independence, and a higher level of automation. Our formalism is based on a uniform versioning model for temporal, cooperative, and logical versioning of models. By an example, we show the feasibility of our approach.

1 INTRODUCTION

The discipline *Model-Driven Software Engineering* (*MDSE*) (Völter et al., 2006) 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) (Steinberg et al., 2009) provides the technological foundation for many model-driven applications.

Software Product Line Engineering (*SPLE*) (Pohl et al., 2005) enforces an organized reuse of software artifacts in order to allow for 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* (Kang et al., 1990). 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 forms the *superimposition* of all product variants. In order to automatically derive a single-variant product, the variability within the feature model needs to be resolved by specifying a *feature configuration*.

Software Configuration Management (*SCM*) is a well-established discipline to manage the *evolution* of software artifacts[1]. A sequence of product *revisions* is shared among a *repository*. Besides storage, traditional SCM systems (Chacon, 2009; Collins-Sussman et al., 2004; Vesperman, 2006) 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 (Rochkind, 1975) constitute a *superimposition* of all revisions, annotated with *revision visibilities*.

[1]Throughout this paper, we use the terms *software configuration management* and *version control* as synonyms.

Schwägerl F., Buchmann T., Uhrig S. and Westfechtel B..
Towards the Integration of Model-Driven Engineering, Software Product Line Engineering, and Software Configuration Management.
DOI: 10.5220/0005195000050018
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 5-18
ISBN: 978-989-758-083-3

A combination of three mutually independent, traditional tools, one for each discipline, seems satisfactory at first glance. An arbitrary MDSE tool might be used for the creation of a software model (e.g., a set of UML class diagrams), a preprocessor language for managing the variability of the software model (or the source code generated from it), and a line-oriented version control system to manage the evolution of the software model (or the source code).

In this paper, we present a conceptual framework which realizes an integrated combination of the three disciplines. It assumes an editing model similar to 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 that persists all existing product versions. In addition to revision graphs, the high-level variability mechanism of feature models is provided to 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 brings the advantage of unconstrained variability: Single-version constraints do not affect the multi-version product in the repository. By providing an integrated mechanism, the distinction between variability in time and variability in space is blurred. Our conceptual framework will allow to postpone the decision, whether a change to a product constitutes a temporal evolution step or a new product variant, until the commit.

In the subsequent section, we outline the state of the art with respect to the pair-wise combinations MDSE/SPLE, MDSE/SCM and SPLE/SCM. Section 3 clarifies the contribution of our paper, before we describe a conceptual framework (Section 4) that allows for a combination of MDSE, SPLE and SCM in an integrated way, by providing a uniform versioning concept and a multi-version product model for the repository. In Section 5, an example is conducted. Section 6 includes a critical discussion, before the paper is concluded.

2 STATE OF THE ART

2.1 MDPLE/MDSE Integration

Model-Driven Product Line Engineering (*MDPLE*) is motivated by a common goal of MDSE and SPLE — increased productivity. Lifting up variability management to the abstraction layer of modeling seems adequate: Both disciplines consider models as primary artifacts. *Feature models* (Kang et al., 1990) have a well-defined syntax and semantics. The *platform* of the product line is provided as a *domain model* with a fixed or variable meta-model.

There exist a number of approaches to MDPLE based on *positive variability*. They require specific tools in order to *compose* variable realization fragments with the common core, typically using *model transformations* (Jayaraman et al., 2007; Ziadi and Jézéquel, 2007) or *aspect-oriented programming* techniques (Völter and Groher, 2007). This way, the core model is kept small and concise, but conflicts arise as soon as transformations or aspects are combined to realize several features.

Approaches based on *negative variability* assume a *multi-variant domain model* that realizes all features of the product domain in a place. It must be syntactically well-formed, but need not be semantically meaningful (i.e., neither executable nor translatable into a target language). Existing approaches differ in the way how features are mapped to realization artifacts. On the one hand, mapping information may be stored within the domain model, e.g., using *annotations* (Gomaa, 2004; Buchmann and Westfechtel, 2012), which correspond to *preprocessor directives* at source code level (Kästner et al., 2009). On the other hand, mapping information can be made explicit by using a distinct *mapping model* (Heidenreich et al., 2008; Buchmann and Schwägerl, 2012).

A common assumption of the mentioned approaches based on negative variability is that the user operates in a *multi-variant* view. When modifying the superimposition, all variants are visible at a time, and mapping information is added manually. The approach described in (Westfechtel and Conradi, 2009) deviates from this editing model. Like in *version control* systems, the user operates in a *single-version view* (*filtered editing*, see (Sarnak et al., 1988)) and need not map model elements to revision visibilities manually. The framework presented in this paper ties on this approach, providing higher-level abstractions for the feature model and the domain model.

2.2 MDPLE/SCM Integration

Model Version Control subsumes the combination of MDSE and version control (Altmanninger et al., 2009), with the goal of lifting existing version control metaphors (*commit*, *update*, etc.) up to the abstraction level of models. Rather than calculating deltas on the low-level physical representation (e.g., lines of text within the XMI serialization), existing

approaches to model versioning (Koegel et al., 2010; Taentzer et al., 2014; Westfechtel, 2014) define operations such as *object insertion* or *attribute change*. This improves both accuracy and consistency to a significant extent. Special emphasis is put on three-way merging, which is necessary to reconcile concurrent modifications to versioned artifacts. Model-specific kinds of *merge conflicts* can occur, which must be resolved in order to produce a consistent result. For typical conflict types, the reader is referred to (Altmanninger et al., 2010; Koshima and Englebert, 2014; Westfechtel, 2014). The three-way merge tool BTMerge (Schwägerl et al., 2013) guarantees a consistent merge result, given valid EMF model versions as inputs.

For storing and merging model versions, *change logs* are may be used to describe the performed modifications. Approaches such as (Koegel et al., 2010; Schneider et al., 2004) record the user's changes, but lack generality because they require a custom editor or at least extensions to existing editors. In case no change log is available, it needs to be reconstructed by *matching* and *differencing*, as realized in (Oliveira et al., 2005; Brun and Pierantonio, 2008; Taentzer et al., 2014). The quality of the matching can be significantly improved by the use of *universally unique identifiers* (UUIDs) that remain stable within subsequent revisions of model elements.

2.3 SPLE/SCM Integration

Software Product Line Evolution deals with common problems that occur 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 (Laguna and Crespo, 2013). In contrast to approaches discussed in this paper, platform and variability model are represented as artifacts on the same conceptual level, i.e., there is no "versioning" relationship as in software configuration management. When lifting product line evolution up to the modeling level, model transformations or decomposition techniques (Heider et al., 2012) play an important role.

As mentioned above, both SPLE and SCM manage variability within software artifacts. While SCM is focused on *variability in time*, i.e., evolution, SPLE provides mechanisms for *variability in space*, i.e., the co-existence of similar products in terms of logical variants. All approaches mentioned below study variability in space at a level that does not meet the requirements of software product lines, which are developed in a systematic way. Furthermore, their primarily targeted product space is text files.

With *branches*, traditional version control systems (Chacon, 2009; Collins-Sussman et al., 2004) offer logical variants to a limited extent. It is only possible to restore variants that have been committed earlier (*extensional* versioning, see (Conradi and Westfechtel, 1998)), but not to create new variants based on a predicate on variant options, i.e., feature configurations (*intensional* versioning).

The commercial SCM system Adele (Estublier and Casallas, 1994) has logical variants built into its object-oriented data model as symmetric deltas, which are exposed to the user. Temporal variability is realized by a versioning layer on top which relies on directed deltas. Thus, logical and temporal versioning are not integrated at the same conceptual level.

In (Reichenberger, 1995), an approach for *orthogonal* version management is proposed. A version cube is formed by product, revision, and variant space. Albeit, this approach does not consider that the variant space may be subject to temporal evolution.

In (Zeller and Snelting, 1997), an approach to *unified versioning* based on *feature logic* is presented. 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.

The *uniform version model* (UVM) presented in (Westfechtel et al., 2001) defines a number of basic concepts (*options*, *visibilities*, *constraints*) for version control, which have been initially introduced in the context of *change-oriented versioning* (Munch, 1993). UVM is designed to support both intensional and extensional versioning and does not assume a concrete product model, i.e., it is applicable to the MDSE context. It will provide the theoretical foundation for the framework presented in this paper.

3 CONTRIBUTIONS

Despite the increased variability gained with it, intensional versioning has never become popular for industrial applications. In our opinion, this is due to the high cognitive overhead resulting from both the low-level representation of both the version space (boolean variables and predicates) and the the product space (sequences of text lines).

Our conceptual framework provides an integrated solution to MDSE, SPLE and SCM, which addresses several drawbacks such as the constrained variability which is common to the MDPLE approaches, the lack of logical versioning concepts in model version control, and inadequacies concerning the low-level representation of both the variant space and product space

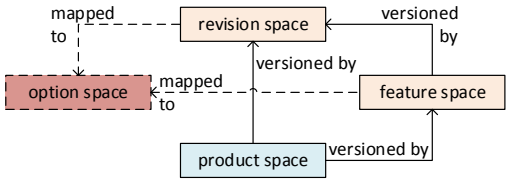

Figure 1: The roles of the revision, the feature and the product space within the repository. Abstractions represented by dashed lines are invisible to the user.

in integrated SPLE/SCM solutions. Our framework uses the UVM described in the previous section as its theoretical foundation and provides higher-level abstractions for it. A prototype called *SuperMod* (*Super*imposition of *Mod*els) is currently under development, which will use EMF both for its own implementation and as the primarily targeted product space.

3.1 The Conceptual Framework

As depicted in Figure 1, our conceptual framework consists of elements which are defined by set theory and distributed over three spaces: A *revision space*, which controls the temporal evolution of both the *product space* and the *feature space*. Elements of the product space are also versioned with respect to the feature space in order to achieve logical variability. Both the revision space and the feature space abstract from a low-level *option space*.

The *product space* is represented as a *superimposition* of product versions. In SPLE, this corresponds to *negative variability*, and in SCM to *symmetric deltas*, respectively. The superimposition allows for *extrinsic variability* (Westfechtel and Conradi, 2009); single-version restrictions do not apply for the multi-version representation. The connection between the product space and the version space is established by *visibilities* (also known as *presence conditions* in SPLE, see (Czarnecki and Kim, 2005)), boolean expressions on the options of the version space, which are assigned to elements of the product space and the feature space. The decision which elements are variable, i.e., to which elements visibilities may be assigned, depends on the specific product space implementation. This allows for an adjustable level of granularity with respect to versioned elements. The prototype SuperMod will support heterogeneous file systems as product spaces, consisting of EMF models and further contents such as plain text or XML files.

The *revision space* is inherited from revision control systems and utilizes a *directed acyclic revision graph*. Logical variability is managed by a *feature model*, which provides a high-level representation of logical properties of a system. Internally, elements of both the revision and the feature space are mapped onto *options* which are managed by the *option space* that realizes UVM as described in (Westfechtel et al., 2001). This mapping happens behind the scene, while only the higher-level abstractions of revision graph and feature model/configuration are presented to the user for version selection and version space editing.

The feature model plays a dual role: For the revision space, it is versioned the same way as the product space; for the product space, it incorporates an additional variability model.

3.2 An Integrated Editing Model

As illustrated in Figure 2, our 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:

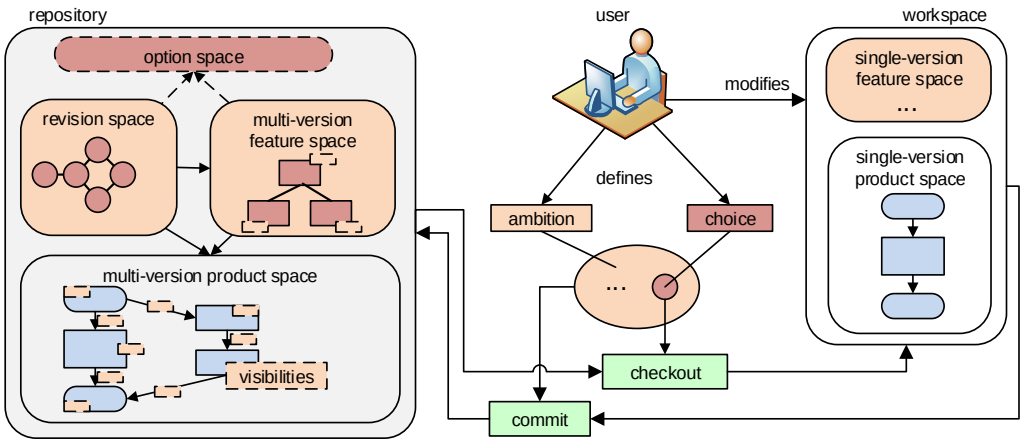

Figure 2: The integrated editing model underlying our conceptual framework.

1. *Checkout:* The user performs a *version selection* (a *choice*) in the repository. In the revision graph, the selection comprises a single *revision*. In the feature space, a *feature configuation* has to be specified. 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.

3.3 Benefits of an Integrated Approach

Taking into account the remarks given in the previous and current section, our conceptual framework advances the state of the art by the following aspects:

Unconstrained Variability. Typically, SPLE tools based on negative variability require that the multi-variant domain model is valid with respect to its meta-model. This may impede variability to a certain extent. E.g., in an *intrinsic* multi-variant UML model, it is not allowed that a class has different names in different versions. By means of an *extrinsic* product space model, we allow models to vary arbitrarily within the multi-version repository; single-version workspace models are still constrained by their respective meta-models.

Non-Destructive Updates. Combined SPLE/SCM solutions that assume orthogonality between the revision and the variant space, e.g., (Reichenberger, 1995), are faced with *destructive updates*: A change to the logical visibility of an element is not limited to the current revision, but global, which may lead to an inconsistent reproduction of old revisions. Our conceptual framework guarantees the *immutability* of revisions.

Reduction of Cognitive Complexity. By the adoption of typical version control metaphors (e.g., *update* or *commit*), our approach reduces the cognitive complexity for the development of a model-driven software product line. The user applies his/her local changes in a *single-version view*, and *visibilities* are updated automatically. The user is neither faced with difficult architectural decisions with respect to a multi-variant domain model (a common problem with negative variability), nor with the necessity to describe feature realizations by means of model transformations (as in SPLE approaches based on positive variability).

Uniform Versioning. Our conceptual framework provides a uniform versioning concept for logical and temporal changes. Not until writing back a change to the repository, the user has to decide if it incorporates a new revision of an existing variant or a new variant that co-exists in parallel.

Multi-Resource EMF Models. Many MDPLE tools, e.g. (Heidenreich et al., 2008; Buchmann and Schwägerl, 2012), as well as model versioning tools such as (Schwägerl et al., 2013), assume that a model is a self-contained single-resource entity. Our conceptual framework preserves the *integrity* of cross-resource links in case multiple EMF resources are submitted to version control. This applies even for references to the *meta-model*, which may vary, too. Furthermore, non-model (plain text or XML) files may be versioned.

Tool Independence. Our conceptual framework assumes a purely *state-based* environment; it does not require change logs which would in turn need an integration of custom recording mechanisms into the tools which are used to modify elements of the workspace. Changes are reconstructed by differencing, using UUIDs if available.

4 FORMALIZED APPROACH

In this section, we detail our contributed conceptual framework. The versioning mechanism and the editing model described in the previous subsection are formalized based on the notions introduced by the uniform version model (Westfechtel et al., 2001). An extrinsic meta-model for the product space is formalized by means of several Ecore models.

4.1 The Option Space

The *option space* – which is mapped by both the *revision space* and the *feature space* – is defined by a set of concepts described in (Westfechtel et al., 2001) using set theory and propositional logic. In Sections 4.2 and 4.3, we will show how the mapping between feature/revision space and option space is realized, before an integration is described in Section 4.4.

Options. An *option* represents a (logical or temporal) property of a software product which is either

present or absent. The option space defines a global *option set*:

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

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

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

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

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

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 option 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 $\rho_1, \ldots, \rho_m$ 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*:

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

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:

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

Visibilities. Each element e of the versioned product space may define a *visibility* $v(e)$ — a boolean expression over a subset of defined options. An element e is *visible* under a choice c iff its visibility is implied by the choice, i.e., it evaluates to *true* given the option bindings of the choice:

$$c \Rightarrow v(e) \quad (8)$$

Filtering. The operation of *filtering* a product space by a choice c can be realized as a conditional copy, where elements e that do not satisfy the choice ($c \not\Rightarrow v(e)$) are omitted.

In case an element e does not define a visibility, we implicitly assume $v(e) = true$ (global visibility). The visibility $false$, in turn, would correspond to a deletion from the repository.

4.2 The Feature Space

The option space introduced in Section 4.1 is applicable to *intensional versioning*. However, concepts such as options, constraints and choices should not be exposed to the user directly because they are represented at a too low conceptual level. *Feature models* (Kang et al., 1990) meet the requirements of SPLE in a satisfactory way. We show how feature modeling concepts can be mapped to the low-level option space. In Section 4.7, we will revisit the feature space in its role of an additional product space.

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 are either 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).

Table 1: Mapping feature models to option space constraints.

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 $f_1, \ldots f_n$	$f \Rightarrow (f_1 \otimes \ldots \otimes f_n)$
f_1 *excludes* f_2	$\neg(f_1 \wedge f_2)$
f_1 *requires* f_2	$f_1 \Rightarrow f_2$

Table 2: Mapping revision graphs to option space constraints.

Pattern	Transformation
initial revision r_0	r_0
new revision r_i after revision r_{i-1}	$r_i \Rightarrow r_{i-1}$
mutually exclusive branches with origins r_L and r_R	$\neg(r_L \wedge r_R)$

Additionally, cross-tree relationships may be defined: *requires* and *excludes* constraints.

It is straightforward to map feature models to propositional logic (see Table 1).

Version Selection. *Feature configuations* describe the characteristics of a single product of the product line and thus may be considered as a *version selection* within the feature space. 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:

$$b_i = \begin{cases} f_i & \text{if feature } f_i \text{ is selected.} \\ \neg f_i & \text{if feature } f_i \text{ is deselected.} \\ true & \text{if no selection is provided for } f_i. \end{cases} \tag{9}$$

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

4.3 The Revision Space

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 revisions that have been committed earlier may be checked out. Second, revisions are *immutable*: Once committed, they are expected to be permanently available, and should not be affected by *destructive updates* (see Section 3.3).

Revision Options. Temporal versioning can be realized by *revision options* (called transactions options in (Westfechtel et al., 2001)) $r \in O_r$, 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 Constraints. Automatically derived *revision constraints* may reduce the size of the revision space considerably. As summarized in Table 2, implications are introduced for consecutive revisions transparently. Furthermore, *exclusion constraints* are introduced for *branches* intended to co-exist in parallel.

Choice Selection. A version in the revision space 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 space is created 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:

$$b_i = \begin{cases} r_i & \text{if } r_i \text{ is the selected revision } r_c \\ & \text{or a predecessor of it.} \\ \neg r_i & \text{else.} \end{cases} \tag{10}$$

Choices referring to the revision space are necessarily complete since the binding $true$ may never appear. The number of selectable versions equals the number of available revision options (extensional versioning).

Ambition Selection. In contrast to choices, ambitions in the revision space only consist of one bound option, namely a newly introduced revision option which is a successor of the previously selected revision choice. As a consequence, within a *revision ambition* a_r, exactly one revision r_n occurs in a positive state. Bindings for other revision options are implicitly set to $true$.

$$a_r = r_n, \quad r_n \text{ is a successor of } r_c. \tag{11}$$

Rather than including all predecessors of the selected revision, in an ambition, only the new revision itself occurs. This results in much shorter visibilities when taking into account the editing model shown in Section 4.8. Ambitions of this form are weakly consistent since constraints ρ_i of the form $r_i \Rightarrow r_{i-1}$ (see Table 2) will be evaluated as follows: $r_n \wedge (r_n \Rightarrow r_c) = r_n \wedge r_c \neq false$.

4.4 Hybrid Versioning

The combination of the revision and the feature space causes interactions between elements of the revision

space and the and variant space. Thus, we provide the following extensions to our framework.

Hybrid Option Space. In hybrid versioning, the option set is decomposed into two disjoint subsets, feature options and revision options:

$$O = O_f \dot{\cup} O_r \tag{12}$$

An analogous decomposition is applied to the rule base.

$$\mathcal{R} = \mathcal{R}_f \dot{\cup} \mathcal{R}_r \tag{13}$$

Hybrid Choice Selection. A *version selection* has to be performed in both the revision and the feature space. 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 the subset c_r of the choice that refers to the revision space.

$$c = c_r \wedge c_f \tag{14}$$

Hybrid Ambition Specification. From the user's perspective, the specification of a *hybrid ambition* does not differ from a specification in the feature space. A conjunction of the selected feature configuration a_f and a revision option r_n, which is introduced transparently as a successor of the revision r_c selected for the choice, is formed automatically.

$$a = r_n \wedge a_f \tag{15}$$

Since a revision ambition is always weakly consistent (see above), an ambition in the hybrid version space only needs to be *weakly consistent* with respect to the feature part of the rule base:

$$\mathcal{R}_f \wedge a_f \neq false \tag{16}$$

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

$$c_f \Rightarrow a_f \tag{17}$$

4.5 Versioned File Systems

The product space of our conceptual framework is formalized by means of several Ecore class diagrams. In order to provide support for *multi-resource models* (see Section 3.3), complete sub-trees of file systems (e.g., an Eclipse project) will be submitted to version control. As shown in the class diagram in Figure 3, files and folders are organized hierarchically using the *composite* design pattern. The abstract class `VersionedElement` provides versioned elements with an optional visibility.

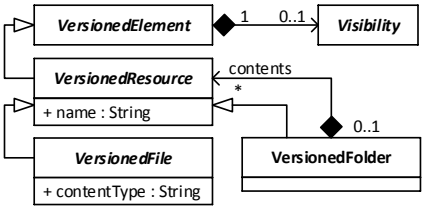

Figure 3: Ecore class diagram for the meta-model of versioned file systems in our conceptual framework.

Our conceptual framework supports different *file content types*. For each content type, a specific pair of *import/export* model transformations needs to be defined in order to convert the intrinsic multi-version representation into a single-version file in the workspace, which may be modified by the user with his/her favorite editor (in order to ensure tool independence, see Section 3.3). As one representative for file content types, EMF is discussed in the next subsection. Similarily, plain text files and XML files will be supported in the prototype SuperMod.

4.6 Multi-vrsion EMF Models

The meta-model shown in Figure 4 incorporates the following design decisions with respect to multi-variant EMF models: (a) unconstrained variability (see Section 3.3), (b) versioned meta-data, (c) variable object classes, and (d) variable object containers.

Considering the EMF product space meta-model, all elements of the EMF file content type are variable (a). Due to (d), an object must be able to have different containers in different versions. Thus, the containment hierarchy of objects inside a resource is flattened. We assume a unique identifier (`uuid`) assigned to each object. Objects may vary in their class (c) and in the values for their structural features (attributes and references). 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.

In order to meet design decision (b), classes and structural features are divided into the categories internal and external, too. 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. This way, the conformance relationship between objects and their corresponding classes is variable (c). Technically, a multi-version EMF model represents two modeling layers (model and meta-model) at the same intrinsic modeling level, which enables co-versioning of models, meta-models, and conformance relationships between those.

The *import/export* transformation pair for the

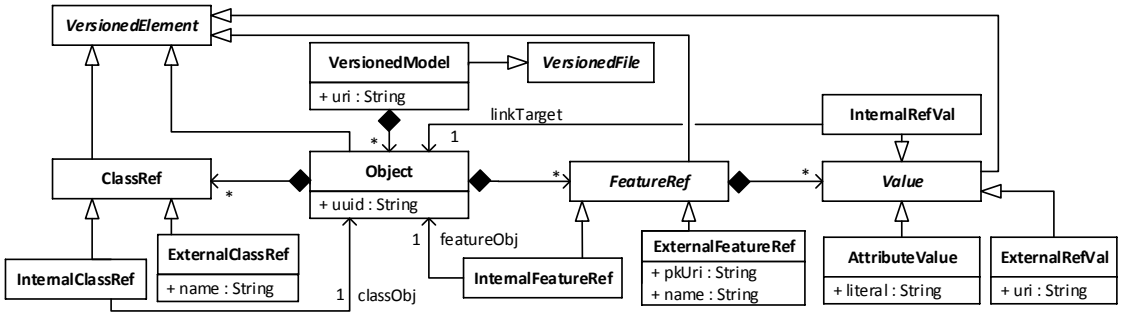

Figure 4: Simplified Ecore class diagram for the meta-model of the multi-version EMF product space.

EMF file content type 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.

4.7 Multi-version Feature Models

Our conceptual framework controls the evolution of an additional product space, the feature model which describes the feature space introduced in Section 4.2 at a higher level of abstraction. The meta-model shown in Figure 5 assumes that the following modifications are supported: (a) *insertion* of a new (mandatory or optional, AND or OR) feature below a fixed parent feature, (b) *deletion* of an existing feature, and (c) insertion or (d) deletion of a *requires* or *excludes* relationship. Within the visibilities of elements of the feature models, only *revision options* are allowed. Additionally to their higher-level abstractions, low-level options and constraints are versioned within the feature space. This is not shown in the figure, but implied in the example in Section 5.

4.8 The Formalized Editing Model

Below, we finalize our editing model sketched in Section 3.2. In contrast to (Westfechtel et al., 2001), the *ambition* is specified at commit-time; the decision whether a change corresponds to the realization of an existing or a new feature, or to a new revision of the checked-out product configuration, can be postponed.

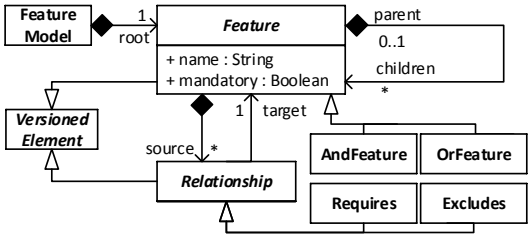

Figure 5: Ecore class diagram for the meta-model of multi-version feature models.

1. The user selects a revision r_c from the revision graph. The derived choice is $c_r = r_0 \wedge \cdots \wedge r_c$.
2. The feature model is *filtered* by selecting elements e_f which satisfy the revision choice ($c_r \Rightarrow v(e_f)$) and are exported into the workspace.
3. The user performs a 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: $b_i = \neg f_i$.
4. The *effective choice* c is calculated as the conjunction $c = c_r \wedge c_f$. It must be *strongly consistent* according to the rule base: $c \Rightarrow \mathcal{R}$.
5. The product space is *filtered* by selecting elements e_p which satisfy the choice ($c \Rightarrow v(e_p)$) and then are exported into the workspace.
6. Within the workspace, the user applies modifications to the filtered product space and/or to the filtered feature model. Updates are broken down to insertions and deletions of element versions. Both spaces are *imported* back into their multi-version representation and merged into the repository.
7. A new revision option r_n is added to O_r. The constraint $r_n \Rightarrow r_c$ is added to $\mathcal{R}_r$.
8. Next, the user specifies an (incomplete) feature configuration a_f that delineates the scope of the change. The ambition must be *weakly consistent* according to the rule base: $\mathcal{R}'_f \wedge a_f \neq false$. Furthermore, the ambition must be implied by the choice: $c_f \Rightarrow a_f$. [2]
9. The applied modifications are written back under the *effective ambition* a. For changes to the feature model, $a = r_n$; for changes to the product space, $a = r_n \wedge a_f$. Each modified (inserted or deleted) element e is processed as follows:

[2] For these checks, the modified option set and rule base O'_f and $\mathcal{R}'_f$ are taken into account. Furthermore, bindings for inserted features are omitted from c_f, and bindings for deleted features from a_f, respectively.

- Inserted elements e_{ins} are appended to the product space (or to the feature model). 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 multi-version product space (feature model). Their visibility is set to $v(e_{del}) := v_{old}(e_{del}) \wedge \neg a$.

5 EXAMPLE

We illustrate our approach by means of an example which refers to an evolving product line of *flow diagrams*. Flow diagrams are connected graphs with a single start node (no incoming, one outgoing control flow, cardinality 0/1), multiple activity nodes (1/1), binary decision nodes (1/2), join nodes (+/1), and end nodes (1/0). Start and end nodes are represented by rounded rectangles, decision nodes by diamonds, and join nodes by circles, respectively. We have chosen this meta-model as it significantly limits variability (e.g., multiple successors for an activity) within the workspace. Within the repository, the unconstrained multi-version representation (see Section 4.6) is used.

We will develop a product line of flow diagrams in subsequent steps: In revision 1, the product space is initialized by performing a global change that only affects the revision space. Next, two independent changes are applied that correspond to two mutually exclusive features. In the last revision, a change is reassigned to a new feature, mapping a temporal change to a logical feature retrospectively.

Initializing the Repository. In the beginning, the version space consists of a revision graph that only contains the initial revision r_0 and a feature model with a mandatory root feature R. The product space consists of an empty flow diagram.

Step 1. In order to populate our empty product space, we need to specify a choice. At the moment, there is only one possibility allowed by the rule base: $c = r_0 \wedge f_R$. Figure 6, step 1, shows the modifications applied: the insertion of a start node, the activity node `v` and an end node. The change is committed as a new revision r_1, resulting in the ambition $a = r_1 \wedge f_R$. A constraint $r_1 \Rightarrow r_0$ is added to $\mathcal{R}_r$ transparently[3].

Step 2. As there is no variability defined in the feature model yet (the selection of f_R is mandatory), it suffices to select a revision. A selection of revision 1 results in the choice $c = r_0 \wedge r_1 \wedge f_R$. After checkout, `v` is locally deleted and a sequence of activity nodes consisting of `w` and `x` is inserted (cf. Figure 6, step 2). Furthermore, an optional feature A is introduced, which automatically adds the constraint $f_A \Rightarrow f_R$ to $\mathcal{R}_f$. For the commit, we specify a feature configuration with A selected, resulting in $a = r_2 \wedge f_R \wedge f_A$. Please note that the performed change is not visible for products which do not include A.

Step 3. Once again, the latest revision is chosen. In the feature space, A is deselected, which generates $c = r_0 \wedge r_1 \wedge r_2 \wedge f_R \wedge \neg f_A$. Thus, the checked-out product version is not affected by the deletion of `v` performed in revision 2. As shown in Figure 6, step 3, an additional feature B is introduced, which excludes A due to an OR relationship. This results in an addition of the constraints $f_B \Rightarrow f_R$

[3] Analogous constraints are inserted for subsequent revisions.

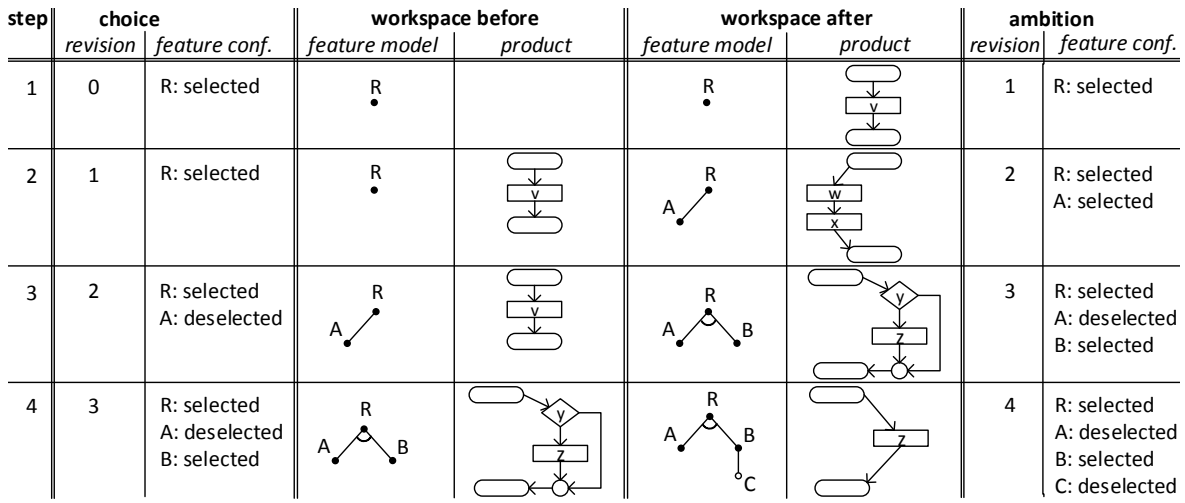

Figure 6: Specified choices and ambitions as well as local modifications performed during subsequent update/commit cycles.

Table 3: Visibilities of the superimposed feature model and product space after revision 4 (hidden from the user). For the reasons of clarity and comprehensibility, visibilities of flow edges have been omitted.

	Element	Visibility Expression
Feature Model	Root feature R, option f_R and constraint f_R	r_0
	Feature A, option f_A and constraint $f_A \Rightarrow f_R$	r_2
	Feature B, option f_B; constraints $f_B \Rightarrow f_R$ and $f_A \otimes f_B$	r_3
	Feature C, option f_C and constraint $f_C \Rightarrow f_B$	r_4
Product Space	Start and end node	$r_1 \wedge f_R$
	Activity node `v`	$r_1 \wedge f_R \wedge \neg(r_2 \wedge f_A) \wedge \neg(r_3 \wedge f_B)$
	Activity nodes `w` and `x`	$r_2 \wedge f_R \wedge f_A$
	Guard `y` and join node	$r_3 \wedge f_R \wedge \neg f_A \wedge f_B \wedge \neg(r_4 \wedge \neg f_C)$
	Activity node `z`	$r_3 \wedge f_R \wedge \neg f_A \wedge f_B$

and $f_A \otimes f_B$. Within the workspace, we perform a corresponding realization: the replacement of `v` by a new activity node `z` guarded by a conditional node `y`. Finally, we commit the change under a feature configuration in which A is deselected but B is selected: $a = r_3 \wedge f_R \wedge \neg f_A \wedge f_B$.

Step 4. After revision 3 has been committed, we come to the conclusion that the guard `y` should not realize feature B, but an optional child feature C. This retrospective feature assignment can be performed by checking out the latest revision with feature B included: $c = r_0 \wedge \cdots \wedge r_3 \wedge f_R \wedge \neg f_A \wedge f_B$. Then, a feature C is introduced and the guard as well as the join node are deleted (cf. Figure 6, step 4). The change is associated with a negation of C's feature option: $a = r_4 \wedge f_R \wedge \neg f_A \wedge f_B \wedge \neg f_C$. As a consequence, in revision 4, the guard `y` is only visible for products which include C. In case revision 3 is restored, `y`'s visibility still only depends on B (*immutability*).

Results. All modifications have been applied in a single-version view based on the filtered editing model defined in Section 4.8. Figure 7 shows the superimposition of the product space after revision 4. Table 3 shows visibilities of elements of both the feature model and the product space. Visibility changes, which affect elements such as `v` or `y`, are characterized by multiple revision options inside an expression.

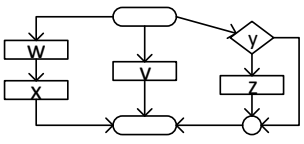

Figure 7: The superimposed product space in its extrinsic representation (hidden from the user).

Benefits Revisited. To conclude this example, we revisit the benefits claimed in Section 3.3.

The extrinsically represented model shown in Figure 7 could not have been created using a single-version editor because it violates the constraints imposed by the flow diagram meta-model (see above). Our approach, in contrast, allows for *unconstrained variability*. The management of visibilities was *fully automated*; we have focused on local modifications and the definition of the editing scope).

In an unfiltered editing model, the superimposition as well as the option expressions would have been necessarily specified manually by the user. In our example, all versioning information is created automatically based on the user-specified partial feature configurations for the ambitions (three in total). Thus, the filtered editing model brings a significant advantage in terms of *reduced cognitive complexity*. Within the workspace, a hypothetical editor for single-version flow diagrams is employed, which exemplifies the advantage of *tool independence*.

Within the last two steps, the *uniform representation* of revisions and features has been utilized to map an initially temporal change to a new feature. Furthermore, we have shown that the *update* performed in step 4 is *non-destructive*.

6 DISCUSSION

A thorough evaluation of our approach will be feasible as soon as a stable implementation of SuperMod is available. In advance, we discuss some potential (conceptual and/or technical) problems, for each of which we sketch possible solutions.

Limited Awareness of other Versions. An advantage adopted from UVM is that the user may operate in a single-version view. On the downside, filtered editing also reduces *awareness* of other versions that are invisible in the local workspace. The information which variants a modified element belongs to, i.e., its visibility,

is hidden from the user. This is in contrast to MDPLE approaches based on negative variability, where the whole superimposition is visible (and editable), including mapping information (i.e., visibilities). Thus, the advantage of reduced complexity is instantaneously linked to the disadvantage of limited awareness.

Mitigating the restriction of a *complete choice* might increase the editable subset of the superimposition, but then *product consistency control* becomes more important (see below). Furthermore, this does not yet solve the problem of hidden visibilities. As a solution, we might provide a specialized editor to support visibility-aware, partially filtered multi-version editing. Rather than actually filtering invisible elements, they could be shaded in gray, as realized in the tool MODPL (Buchmann and Westfechtel, 2012), and protected against local modifications. However, this would remove the benefit of tool independence.

Product Consistency Control. Currently, our conceptual framework does not ensure the consistency of extrinsically represented products. Considering multi-variant EMF models, consistency violations might concern referential integrity, the containment hierarchy, type correctness, or the cardinality of structural features. Additionally, each meta-model may define its own context-sensitive consistency constraints using the *Object Constraint Language* (*OCL*).

In Section 4.1, we have introduced *version rules*. Although they ensure that no inconsistent versions are specified (e.g., mutually exclusive features), they cannot make any assertions to an extrinsically represented product version, or a set thereof. For this purpose, additional *product rules* should be enforced before products are converted into their single-version representation using the *export* transformation. In (Westfechtel, 2014), generic EMF product rules have been investigated in the context of three-way merging. The tool FAMILE (Buchmann and Schwägerl, 2012) provides an OCL extension to check and repair the consistency of derived products.

Scalability. The representation of the product space (including visibilities) as a *superimposition* will result in a growing memory consumption. Due to the immutability of revisions, 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 commit and update runtimes. We propose three optimizations.

- *Hierarchical Propagation of Visibilities:* Product space elements are organized *hierarchically*. Due to existential dependency, the *effective visibility* of an element might be defined by conjunction with its parent visibility. When a tree of elements is modified, only its root element's visibility needs to be updated.
- *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. If these expressions become too long, they might be substituted by an artificial option Δ_a. The constraint $\Delta_a \Rightarrow a$ is then added to the rule base once, and the change is run under Δ_a, with the same effect.
- *Global Storage for Visibilities:* Visibilities are directly contained in product space elements, which will result in duplicates. Storing identical visibilities within a global data structure, and caching evaluation results for a given option binding, might reduce memory consumption and runtime. A technically sound solution is described in (Munch, 1993, Section 6.6).

Collaborative Editing. One of the greatest advantages of SCM systems is the support for collaborative development performed by multiple developers. In its current state, our editing model does not explicitly support collaboration. Concurrent changes are committed as multiple successors of a base revision. In order to achieve optimistic versioning, it should be permitted to create a new revision with several predecessors. This may be achieved by generalizing step 1 of the editing model presented in Section 4.8 inasmuch as a non-empty set of mutually unrelated revisions may be selected. As soon as mutually unrelated branches are *merged*, the corresponding exclusion constraint (see Table 2) needs to be removed.

In addition to adaptations to the version space and the editing model, collaborative editing will require support for three-way merging in the workspace. Tools and approaches for the detection and resolution of merge conflicts have been discussed in Section 2.2.

Added Value for the End User. We have described the integrated editing model in a very abstract way, and the pragmatic question arises, how end users, being SPL developers and SCM users, might gain advantage of our approach.

For SPL developers, the main advantage is the mentioned reduction of cognitive complexity, which may be achieved by (1) the fact that modi-

fications are performed to filtered versions of the product line rather than to a multi-variant domain model, (2) the concepts of choices and ambitions being expressed in a user-oriented way, making manual feature annotations obsolete, and (3) improved consistency support gained by version rules, which may avoid product line inconsistencies in advance. The development of an intuitive user interface will be necessary, especially in order to facilitate the specification of feature configurations, which is a frequent task in the filtered editing model.

For SCM users, the added value is the possibility of introducing logical options, i.e., features, in addition to temporal options (revisions). When compared to the widespread concept of branches, features allow for an exponential number of product versions: n branches represent n variants, while n features allow for up to 2^n variants (this number may be significantly constrained by the rule base).

7 CONCLUSION

We have explored the conceptual groundwork for an integrated solution to MDSE, SPLE and SCM. Our presented approach is purely *state-based*, has an *extrinsic* product space model allowing for *unconstrained variability*, and supports *intensional* and *extensional* versioning uniformly. By providing higher-level abstractions, version selection is eased significantly. In an example, we have shown that this uniform handling enables a *high level of automation* and a *reduction of cognitive complexity*.

Although it will be far from trivial to meet the specific requirements of the three disciplines by a single tool, we are convinced that the disciplines can profit from each other with respect to the way how tools are used. For instance, the filtered editing model borrowed from SCM can increase comprehensibility in MDPLE, while feature models provide a powerful logical extension to temporal versioning.

In addition to the technical realization of our approach, we think that in future, the definition of a *development process* would be advantageous in order to combine the filtered editing model with MDPLE. For evaluation purposes, we are looking for a case study of industrial scale, which would allow for a quantitative evaluation against approaches which rely on an unfiltered editing model.

REFERENCES

Altmanninger, K., Schwinger, W., and Kotsis, G. (2010). Semantics for accurate conflict detection in SMoVer: Specification, detection and presentation by example. *International Journal of Enterprise Information Systems*, 6(1):68–84.

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A survey on model versioning approaches. *International Journal of Web Information Systems (IJWIS)*, 5(3):271–304.

Brun, C. and Pierantonio, A. (2008). Model differences in the Eclipse Modelling Framework. *UPGRADE*, IX(2):29–34.

Buchmann, T. and Schwägerl, F. (2012). Ensuring well-formedness of configured domain models in model-driven product lines based on negative variability. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD 2012, pages 37–44, New York, NY, USA. ACM.

Buchmann, T. and Schwägerl, F. (2012). FAMILE: tool support for evolving model-driven product lines. In Störrle, H., Botterweck, G., Bourdells, M., Kolovos, D., Paige, R., Roubtsova, E., Rubin, J., and Tolvanen, J.-P., editors, *Joint Proceedings of co-located Events at the 8th European Conference on Modelling Foundations and Applications*, CEUR WS, pages 59–62, Building 321, DK-2800 Kongens Lyngby. Technical University of Denmark (DTU).

Buchmann, T. and Westfechtel, B. (2012). Mapping feature models onto domain models: ensuring consistency of configured domain models. *Software and Systems Modeling*.

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

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

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, Leipzig, Germany.

Heider, W., Rabiser, R., and Grünbacher, P. (2012). Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. *International Journal on Software Tools for Technology Transfer*, 14(5):613–630.

Jayaraman, P. K., Whittle, J., Elkhodary, A. M., and Gomaa, H. (2007). Model composition in product lines and feature interaction detection using critical pair analysis. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 151–165, Nashville, USA.

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., Apel, S., Trujillo, S., Kuhlemann, M., and Batory, D. S. (2009). Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *TOOLS (47)*, pages 175–194.

Koegel, M., Hermannsdoerfer, M., von Wesendonk, O., and Helming, J. (2010). Operation-based conflict detection. In Di Ruscio, D. and Kolovos, D. S., editors, *Proceedings of the 1st International Workshop on Model Comparison in Practice (IWMCP 2010)*, pages 21–30, Malaga, Spain.

Koshima, A. and Englebert, V. (2014). Collaborative editing of EMF / Ecore meta-models and models: Conflict detection, reconciliation, and merging in DiCoMEF. In Pires, L. F., Hammoudi, S., Filipe, J., and das Neves, R. C., editors, *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014)*, pages 55–66, Lisbon, Portugal. SCITEPRESS Science and Technology Publications, Portugal.

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

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

Oliveira, H. L. R., Murta, L. G. P., and Werner, C. (2005). Odyssey-VCS: a flexible version control system for UML model elements. In *SCM*, pages 1–16. ACM.

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

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

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

Schneider, C., Zündorf, A., and Niere, J. (2004). CoObRA - a small step for development tools to collaborative environments. In *Workshop on Directions in Software Engineering Environments in 26th international conference on software engineering*, Edinburgh, Scotland, UK.

Schwägerl, F., Uhrig, S., and Westfechtel, B. (2013). Model-based tool support for consistent three-way merging of EMF models. In *Proceedings of the workshop on ACadeMics Tooling with Eclipse*, ACME '13, pages 2:1–2:10, 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 edition.

Taentzer, G., Ermel, C., Langer, P., and Wimmer, M. (2014). A fundamental approach to model versioning based on graph modifications: From theory to implementation. *Software & Systems Modeling*, 13(1):239–272.

Vesperman, J. (2006). *Essential CVS*. O'Reilly, Sebastopol, CA.

Völter, M. and Groher, I. (2007). Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 233–242, Washington, DC, USA. IEEE Computer Society.

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

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

Westfechtel, B. and Conradi, R. (2009). Multi-variant modeling - concepts, issues and challenges. In Mezini, M., Beuche, D., and Moreira, A., editors, *Proceedings 1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009)*, pages 57–67. CTIT Proceedings.

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 Trans. Softw. Eng. Methodol.*, 6(4):398–441.

Ziadi, T. and Jézéquel, J.-M. (2007). PLiBS: an Eclipse-based tool for software product line behavior engineering. In *Proc. of 3rd Workshop on Managing Variability for Software Product Lines, SPLC 2007*, Kyoto, Japan.