SuperMod — A Model-Driven Tool that Combines Version Control and Software Product Line Engineering

Felix Schwägerl, Thomas Buchmann and Bernhard Westfechtel

Applied Computer Science I, University of Bayreuth, Universitätsstr. 30, 95440 Bayreuth, Germany

Keywords: Model-Driven Software Engineering, Software Product Lines, Version Control, Version Models, Software Configuration Management, Tool Support.

Abstract: Version control (VC) and Software Product Line Engineering (SPLE) are two software engineering disciplines to manage variability in time and variability in space. In this paper, a thorough comparison of VC and SPLE is provided, showing that both disciplines imply a number of desirable properties. As a proof of concept for the combination of VC and SPLE, we present SuperMod, a tool realizing an existing conceptual framework that transfers the iterative VC editing model to SPLE. The tool allows to develop a software product line in a single-version workspace step by step, while variability management is completely automated. It offers familiar version control metaphors such as check-out and commit, and in addition uses the SPLE concepts of feature models and feature configuration the definition of logical variability and to define the logical scope of a change. SuperMod has been implemented in a model-driven way and primarily targets EMF models as software artifacts. We successfully apply the tool to a standard SPLE example.

1 INTRODUCTION

Version control (VC) has become indispensable for software engineers to control software evolution and to coordinate changes among a team. Version control systems (VCS) such as Git (Chacon, 2009) or Subversion (Collins-Sussman et al., 2004) propose an iterative three-stage editing model (cf., Figure 1): (1) A developer checks out a revision of a software project from a repository. A copy of the project is created in the local workspace. (2) In the workspace, the developer modifies the project by implementing new functionality or by fixing bugs. (3) To make these modifications visible to others, the developer commits his/her changes to the repository. For the internal representation of version differences within the repository, two distinct approaches exist. Symmetric deltas (Rochkind, 1975) comprise a superimposition of all existing revisions, assigning version identifiers to each element. Using directed deltas (Tichy, 1985), change sequences reconstruct product revisions on demand, ensuing form a baseline revision, which is fully persisted.

Software Product Line Engineering (SPLE) aims at a systematic development of a family of software products by exploiting the variability among members thereof (Clements and Northrop, 2001; Pohl et al., 2005). Core assets of different products are provided as a platform. Commonalities and differences among products are captured in variability models, e.g., feature models (Kang et al., 1990). In literature, a two-stage SPLE process is proposed (cf., Figure 2): (1) During domain engineering, platform and variability model are defined. A mapping, e.g., presence condi-
Section 4, the implementation and user interface of SuperMod are sketched and the operations check-out and commit are formalized. Subsequently, the example is reconsidered. Section 6 outlines related work, before the paper is concluded.

2 MOTIVATING EXAMPLE

We introduce as running example a product line of different domain models for graphs, a common example in SPL literature (Lopez-Herrejon and Batory, 2001). In this section, we conduct the example using a “traditional” tool chain: A state-of-the-art VCS supports the development of the platform in multiple iterations. Next, a feature model is defined, and an MDPLE tool based on negative variability is used to annotate domain model elements with variability information. During application engineering, we derive one example product.

Variability Model. Figure 3 shows the underlying feature model, which consists of a root feature Graph with two mandatory sub-features Vertices and Edges. Vertices may optionally be colored. For edges, the optional sub-features weighted and labeled are defined. Furthermore, the features directed and undirected are mutually exclusive.

Platform. The superimposition is defined in the form of a multi-variant domain model (MVDM) (Gomaa, 2004). We realize the platform in multiple VC iterations, after each of which a commit is carried out. In Table 1, the performed modifications are listed. When referring to the feature model in Figure 3, one feature has been realized at a time. Figure 4 shows the resulting MVDM.

Feature Mapping. After having defined the variability model and the platform, they need to be connected. In an MDPLE approach based on negative variability, this requires assigning feature expressions to MVDM elements. A mapping for the example...
Table 1: History of the developed multi-variant domain model for the graph product line. Within each revision, the inserted elements are listed by their respective qualifier. For all associations, the corresponding member ends have been inserted in the same revision.

<table>
<thead>
<tr>
<th>Inserted elements</th>
<th>Commit message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 —</td>
<td>&quot;Initial commit.&quot;</td>
</tr>
<tr>
<td>2 Class Graph</td>
<td>&quot;Realization of root feature Graph.&quot;</td>
</tr>
<tr>
<td>3 Class Vertex, association has vertices</td>
<td>&quot;Realization of feature Vertices.&quot;</td>
</tr>
<tr>
<td>4 Class Edge, association has edges</td>
<td>&quot;Realization of feature Edges.&quot;</td>
</tr>
<tr>
<td>5 Property Edge::label</td>
<td>&quot;Realization of feature labeled.&quot;</td>
</tr>
<tr>
<td>6 Property Edge::weight</td>
<td>&quot;Realization of feature weighted.&quot;</td>
</tr>
<tr>
<td>7 Association connects</td>
<td>&quot;Realization of feature undirected.&quot;</td>
</tr>
<tr>
<td>8 Association starts at and ends at</td>
<td>&quot;Realization of feature directed.&quot;</td>
</tr>
<tr>
<td>9 Class Color, property Color::name, association has color</td>
<td>&quot;Realization of feature colored.&quot;</td>
</tr>
</tbody>
</table>

Figure 4: Multi-variant domain model, which realizes all features of the graph product line as a superimposed UML class diagram. The diagram has been created using the EMF-based UML modeling tool Valkyrie (Buchmann, 2012).

Figure 5: Mapping between the multi-variant domain model and the feature model, realized by feature expressions, logical expressions on the set of feature variables defined in the feature model in Figure 3. The mapping has been realized with the help of the MDPLE tool FAMILE (Buchmann and Schwägerl, 2012), where the mapping is defined on the abstract syntax tree of the MVDM. Feature expressions are highlighted.

Application Engineering. To automatically derive specific products from the product line, feature configurations are specified which bind each feature to a selection (either selected or deselected). For instance, the feature configuration from Figure 6 produces the product shown in Figure 7, a directed and weighted graph.

Drawbacks. Although having successfully applied an “off-the-shelf” combination of VC and MDPLE in the initial example, we raise several issues.

- Variability in time and variability in space are managed by means of two different mechanisms. Therefore, the user has to repeatedly specify versioning information (i.e., feature expressions) for...
In Section 4, the tool SuperMod is presented, which allows to overcome the presented drawbacks. We will revisit the graph example in Section 5.

3 COMPARISON: VC AND SPLE

In this section, we compare terms and notions of VC and SPLE that have been used in the previous two sections. This comparison motivates the prototype SuperMod, which is described in Section 4. Table 2 summarizes the discussion below.
Table 2: Differences and commonalities among the terms and notions of VC and SPLE.

<table>
<thead>
<tr>
<th>Equivalent Concepts</th>
<th>Version control</th>
<th>SPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Product versions</td>
<td>Repository</td>
<td>Platform</td>
</tr>
<tr>
<td>Single Product version</td>
<td>Workspace</td>
<td>Product configuration</td>
</tr>
<tr>
<td>Superimposition</td>
<td>Symmetric deltas</td>
<td>Negative variability</td>
</tr>
<tr>
<td>Transformations</td>
<td>Directed deltas</td>
<td>Positive variability</td>
</tr>
<tr>
<td>Visibilities</td>
<td>Version identifiers</td>
<td>Presence conditions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Similar Concepts</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability model</td>
<td>Revision graph</td>
<td>Feature model</td>
</tr>
<tr>
<td>Version</td>
<td>Revision</td>
<td>Feature configuration</td>
</tr>
<tr>
<td>Filter</td>
<td>Check-out</td>
<td>Product derivation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Differences</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Variability kind</td>
<td>Variability in time</td>
<td>Variability in space</td>
</tr>
<tr>
<td>Variation points</td>
<td>Hidden</td>
<td>Explicit</td>
</tr>
<tr>
<td>Product variability</td>
<td>Unconstrained</td>
<td>Constrained</td>
</tr>
<tr>
<td>Version specification</td>
<td>Extensional</td>
<td>Intensional</td>
</tr>
<tr>
<td>Editing model</td>
<td>Filtered</td>
<td>Unfiltered</td>
</tr>
<tr>
<td>Visibility management</td>
<td>Manual</td>
<td>Automatic</td>
</tr>
<tr>
<td>Version membership</td>
<td>Immutable</td>
<td>Mutable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Equivalence</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic visibility update</td>
<td>Commit</td>
<td>—</td>
</tr>
<tr>
<td>Manual visibility update</td>
<td>—</td>
<td>Edit feature mapping</td>
</tr>
</tbody>
</table>

committed earlier (intensional versioning). In VC, filtered editing is applied. After a check-out, the developer sees and may modify only elements belonging to the selected revision. As soon as a commit is issued, changes are detected in the local workspace, and written back to the repository, while visibilities are updated automatically. In contrast, SPLE typically requires the user to edit a multi-version view (unfiltered editing) and to manage visibilities (i.e., presence conditions) manually. VCS guarantee the immutability of version membership of an element: Once committed, it is not possible to remove an element from a revision. In contrast, it is allowed to modify the visibility of an element arbitrarily in SPLE.

No Equivalence. VCS and SPLE tools both offer operations that are not realized by the opposite. Table 2 lists two of each. The VCS operation commit detects differences in the workspace in order to write changes back to the repository automatically. No equivalent operation exists in SPLE tools, which would, e.g., allow to propagate product specific modifications back to the platform. Conversely, in SPLE, it is possible to directly modify the mapping between the variability model and the platform, i.e., the visibilities. To the best of our knowledge, there exists no VCS that would allow to retrospectively modify version identifiers (which would, indeed, destroy the property of immutable version membership).

Bottom Line. VC and SPLE share an unexpectedly large amount of similarities, particularly with respect to underlying data structures. Most differences are due to the underlying editing models. As we will explain within the subsequent section, SuperMod eliminates these differences by transferring the filtered VC editing model to SPLE. In the workspace, the distinction between variability in time and variability in space is blurred, offering the user new ways of versioning, such as committing a change against a dedicated feature. In particular, the desirable properties of unconstrained variability, intensional versioning, automatic visibility management, and immutability of (temporal) version membership are transferred.

4 THE TOOL SuperMod

This section sketches the model-driven implementation of SuperMod. First, we explain theoretical foundations developed in advance. Thereafter, the architecture is described at a coarse-grained level, before we detail the specification by means of a metamodel. Next, the operations check-out, modify and commit are specified. Last, we discuss current limitations and address future tool improvements. The tool is available for evaluation purposes as an Eclipse plug-in (see installation instructions at the end of this paper).

4.1 Underlying Principles

SuperMod realizes the conceptual framework presented in (Schwägerl et al., 2015), which aims at the integration of MDPLE, SPLE, and VC. The framework in turn is built upon the uniform version model
1. check-out
2. modify
3. commit

Repository
Multi-Version
Domain Model
Multi-Revision
Feature Model
Revision
Graph
Revision
Configuration
Choice
Ambition
Workspace

Visibility

Revision Graph
Multi-Revision
Feature Model
Multi-Version
Domain Model

Figure 8: SuperMod tool architecture and editing model.

Options. An option is a temporal or logical property of a software system, which may or may not be included in a specific version. In SuperMod, two kinds of options exist: revision options and feature options (see below).

Choices. A choice denotes a single version by assigning a selection (selected, deselected) to each of the existing options. Choices are used as read filters, i.e., they describe versions visible in the workspace.

Ambitions. An ambition denotes a set of versions as a subset of all available versions. Ambitions are used as write filters in order to delineate the scope of a change performed in the workspace. In contrast to a choice, an ambition may contain unbound options, to which the change is immaterial.

Version Rules. The set of available choices and ambitions is constrained by a set of version rules, logical expressions over the option set. Version rules are used, e.g., in order to implement constraints such as mutual exclusion within feature models, or to designate subsequent revisions.

Visibilities. A visibility is a logical expression over the option set, which is attached to an element of the feature or domain model. In order to test an element’s presence in a specific version, the bindings specified by the respective choice are applied. Visibilities are modified automatically during the commit operation (see below).

4.2 Tool Architecture/Editing Model

Both the architecture and the editing model of SuperMod are inspired by distributed VCS such as Git (Chacon, 2009). The tool offers the traditional version control metaphors check-out and commit and additionally offers SPL concepts such as feature models and feature configurations for the definition of the version space and specific versions. The traditional VCS architecture is extended: (1) The feature model is an additional, temporally variable, versioned artifact. (2) The domain model varies along two dimensions, the revision graph and the feature model. Figure 8 illustrates the remarks below.

Repository. A repository is a persistent storage linked to a software project under VC. Developers communicate with their private repository by means of the operations check-out and commit. A SuperMod repository consists of three layers.

- The revision graph is a directed acyclic graph that describes the temporal history of a SuperMod project. The graph is extended automatically each time a new revision has been committed. For each revision, a revision option is introduced transparently. Furthermore, version rules ensure that revision selections amend all predecessor revisions.
- The multi-version feature model plays a dual role: Firstly, its evolution is controlled by the revision graph. Secondly, each feature is mapped to a feature option, such that the feature model provides an additional version model. Feature model constraints are mapped to version rules transparently (Schwägerl et al., 2015).
- The multi-version domain model describes the superimposition of the versioned project. Although the term “domain model” is used here, the project may comprise a file hierarchy containing model
perMod Feature metamodel (Schwägerl et al., 2015). Technically, the repository is an instance of the SuperMod metamodel (see Section 4.3), representing multi-version models (and non-model artifacts) as a superimposition. Thus, from the VC perspective, SuperMod uses symmetric deltas, and from the SPLE perspective, it is based on negative variability.

Visibilities are represented in a memory-optimized way using a global data structure, the visibility forest. It contains each visibility occurring on any model element at most once. Furthermore, visibilities may reference each other to form several tree-like structures (hence, a forest). The visibility forest is updated during a commit transparently.

Workspace. A SuperMod workspace contains the currently selected version of the domain model, i.e., the derived product, in its domain-specific representation within an ordinary file system. EMF models are represented as instances of their custom Ecore-based metamodel(s). Plain text and XML files are represented in their custom format. This allows SuperMod users to utilize their single-version editing tools they are familiar with.

During the sub-process modify, the user may also edit the feature model arbitrarily, e.g., by introducing new features or constraints. For this purpose, the chosen revision of the feature model is made available for modification in the workspace. Technically, the feature model is represented as an instance of the SuperMod Feature metamodel (Schwägerl et al., 2015).

In addition to the single-version resources, metadata are managed transparently to the user. They augment workspace resources with VC details, such as the versioning state (versioned, non-versioned, added, removed, etc.). Furthermore, the current choice is persisted.

Choices and Ambitions. As mentioned above, feature configurations are used to specify choices and ambitions in addition to revision graphs as known from state-of-the-art VCS. A feature configuration is always specified on the current revision of the feature model. When specified as an ambition, the feature configuration may be partial and typically binds only few features. The effective choice/ambition is formed during check-out/commit as conjunction of the temporal and logical component, e.g., rev4 and labeled.

4.3 The SuperMod Metamodel

SuperMod has been developed in a model-driven way using the Eclipse Modeling Framework (Steinberg et al., 2009). Furthermore, the tool has been implemented modularly; it is is not restricted to the three-layer architecture shown in Figure 8, but flexible with respect to the underlying version space space model.

The metamodel realizes the sub-set of concepts presented in Sections 3 and 4.1, which are relevant to the repository. As shown in Figure 9, a SuperMod repository consists of a version space, a product space, and a visibility forest. The version space in turn is composed of several version dimensions, and the product space comprises a number of product dimensions, which in turn contain a hierarchy of versioned elements. These may reference a visibility, which is organized within a visibility forest. A visibility may be an option reference or a composed expression (e.g., and, or, not). Choices and ambitions, which occur in SuperMod as temporary data structures (except for the choice in the metadata section), are represented by OptionBinding, which maps options to selections.

On the right hand side of the class diagram, the three dimensions discussed in this paper are depicted. Obviously, the revision graph is a subclass of VersionDimension. Due to its dual role, the feature model is both a version and a product dimension. The metamodel of the primary product space, the versioned file system, is decomposed into three different resource types: EMF, plain text, and XML. As a representative for VersionedElements, the class Object represents multi-version EMF objects. More details on the EMF and Feature multi-version metamodels are provided in (Schwägerl et al., 2015).

4.4 Repository Operations

The user interface of SuperMod has been realized using the Eclipse Team Provider API. In this section, we sketch the available UI commands.

Check-Out. For check-out, the UI offers two distinct commands: Switch prompts the user for a choice in both the revision graph and the feature model, whereas Update automatically generates a new choice whose temporal component is updated to the latest available successor of the current revision. In general, the operation check-out has been realized as follows:

- The specified choice is recorded in the metadata.
- The feature model is filtered by the temporal component of the selected choice and copied into the local workspace.
- The domain model is filtered by the effective choice and exported into the local workspace. The export transformation has been implemented for each specific resource type to translate a
multi-version representation into its corresponding single-version representation.\textsuperscript{1}

**Modify.** The user may modify both the feature model and the domain model within the workspace. For domain model resources, arbitrary editors available in the current Eclipse installation may be used. For the feature model, the command Edit Version Space is offered by SuperMod, which opens an EMF tree editor for the current feature model revision. In addition to the modification of versioned resources, the commands Add/Remove to/from Version Control are provided, which adjust the corresponding entries within the metadata section accordingly.

**Commit.** The Commit command is defined as a counterpart to the check-out operation as follows:

- A new revision is created as the successor of the revision specified for the choice. Within the given revision of the feature model, the ambition is user-specified as a partial feature configuration. For consistency reasons, it is required that the ambition implies the previously specified choice.
- The original state of the workspace version is temporarily restored by applying the recorded choice to the repository. The new state is generated by importing (the inverse of export) the current workspace into its multi-version representation.
- Differences are computed by comparing both versions, the original and the new state.
- Inserted elements are copied into the repository.
- The visibilities of inserted/deleted feature model elements are updated automatically by adding/subtracting the temporal component of the ambition to/from the existing visibility.
- The visibilities of inserted/deleted domain model elements are updated by adding/subtracting the effective ambition.

As proposed in (Schwägerl et al., 2015), the update operations add and subtract have been implemented by the operators $\lor$ and $\land$.

### 4.5 Current Limitations and Outlook

The current version of SuperMod allows to answer research questions referring to the added value of transferring the VC editing model to SPLE. However, there are some questions that remain to be answered by future work:

- Currently, SuperMod is restricted to single-user operation — the repository is persisted locally. After having evaluated and improved the tool, it is planned to realize support for multi-user operation. A multi-user version of SuperMod will offer commands similar to the Push/Pull operations as defined in the distributed VCS Git (Chacon, 2009). With the number of its users, the size of the repository will grow, concerning both the feature and the domain model. A transactional storage will be necessary that scales with large model instances.
- Concurrent modifications will lead to conflicts in the domain/feature model, which must be resolved by a specific three-way merging component for models, e.g., (Schwägerl et al., 2013).
- The evolution of the feature model remains to be further evaluated. For instance, how does the deletion of a feature affect existing variants in the workspace?
- From the SPLE perspective, domain engineering and application engineering are not clearly separated. SuperMod does not allow to design a multi-variant domain model from scratch and switch to filtered editing in a later phase. Furthermore, derived products are currently considered as volatile

\textsuperscript{1}In the case of EMF models, the export transformation is generic, i.e., there is no necessity to define a custom transformation for each metamodel used.
artifacts in the workspace, which disappear as soon as the choice is changed. It may be desir-able to derive several products in a batch mode.

- The concepts of choices and ambitions are hard to grasp for VCS users. Concerning user inter-action, there is still room for improvement. For in-stance, a default ambition might be inferred from the choice and newly introduced features.

## 5 EXAMPLE REVISITED

To demonstrate the added value of SuperMod, we re-consider the example from Section 2. Now, we de-velop the platform, consisting of the domain model and the feature model (and the mapping in between, which is hidden now) together in multiple iterations, using feature configurations to describe the scope of changes to the domain model. Figure 10 shows the subsequent iterations in which the model-driven product line is developed as described below. In each step, revision $i$ is evolved to revision $i+1$.

**Initialization.** We create an empty Eclipse project and invoke the Share command, which puts it under SuperMod version control. Next, we create an empty UML class diagram and add it to version control. Ini-tially, the feature model is empty. The project is com-mitted to the repository as revision 1. Since there is no variability defined in the feature model yet, the user is not prompted for a feature configuration — the change is universal.

**Realization of Common Parts.** In step 2, both the feature model and the domain model are evolved. We add the root feature Graph and its mandatory subfeatures Vertices and Edges to the feature model. Within the domain model, a class Graph is created. Since this class realizes the identically named feature, we specify as ambition a partial feature configuration where Graph is selected. Transparently, the visibil-ity of the added features is set to rev2, whereas the visibility of the class Graph is set to rev2 and graph. Similarly, realizations of the mandatory features Vertices and Edges are provided in steps 3 and 4, where the feature model remains unmodified. The performed commit operations result in the visibilities rev3 and vertices for the class Vertex and the asso-ciation has vertices, and rev4 and edges for Edge and has edges.

**Realization of Optional Parts.** In steps 5 and 6, the optional features labeled and weighted are in-

---

SuperMod — A Model-Driven Tool that Combines Version Control and Software Product Line Engineering
Figure 10: Summary of all performed check-out/modify/commit iterations during the example. For the reason of compactness, certain parts of the feature model and domain model have been elided (…). For the development of the domain model, Valkyrie (Buchmann, 2012) has been used again. All feature models and feature configurations are represented in the tree representation provided by SuperMod.

the multi-version feature model only contain revision options, since the feature model is only versioned by the revision graph and not by itself. Visibilities of the MVDM are hybrid: they contain a revision part (which corresponds to the version history shown in Table 1) and a feature part (which is equivalent to the mapping shown in Figure 5).

**Outlook.** The presented example has only scratched the surface of SuperMod. Performed
modifications were restricted to element insertions, and the specified ambitions always include one selected feature. These simplifications have been applied for the reason of comprehensibility. We informally sketch a couple of extensions to the example, where SuperMod is used in a more advanced way:

- We may add different constructors for the class `Edge`, depending on the combination of features `labeled` and `weighted`. As shown in Figure 12, the features selected in the ambition match the corresponding parameters in different versions of the constructor.

- Hyper graphs are a generalization of graphs whose edges connect a number of vertices greater than two. For this purpose, we may add a feature `hyper` below `Edge`. The realization would consist in renaming of the class `Edge` to `HyperEdge` and changing the association name and multiplicities as shown in Figure 13. The co-existence of different class names is only allowed due to SuperMod’s property of unconstrained variability.

- A universal change may be retrospectively mapped to a new feature by reverting the change (i.e., by deleting all added elements and vice versa) and specifying the negation of the new feature in the ambition. An example is provided in (Schwägerl et al., 2015, Section 5).

- Our example has been restricted to a single class diagram. SuperMod can handle complete Eclipse projects, including interconnected EMF model resources, and non-model resources such as plain text or XML files. For instance, we may add generated Java code to version control, which would enable for variability in behavioral aspects.

**Added Value.** When compared to the first version of the example, the cognitive complexity of feature mapping has been significantly reduced. The domain model and the feature model have been developed step by step, while realizations for each feature have been specified directly in the workspace. This is enabled by the uniform version mechanism for temporal and logical variability in SuperMod, which removes the necessity of repeated annotations. For realizing changes, the developer has used an arbitrary single-version editor. This is in contrast to many SPLE tools, which require custom multi-variant editors or additional composition languages, which both disrupt the developers’ workflow. The additional advantage of unconstrained variability has been demonstrated above. In sum, SuperMod removes the drawbacks and limitations that have been identified for an “off-the-shelf” approach to combined VC/SPLE in Section 2.

### 6 RELATED WORK

In (Schwägerl et al., 2015), concepts and theory used for the realization of SuperMod have been described, including a comparison with the integrating disciplines Model-Driven Product Line Engineering (MD-PLE), Model Version Control, and Software Product Line Evolution. The current paper describes the added value from the end user’s perspective. Subsequently, we compare SuperMod to other tools and approaches that partially share VC and SPLE concepts.

With branches, many contemporary version control systems, e.g., Git (Chacon, 2009) and Subversion (Collins-Sussman et al., 2004) offer logical variants to a limited extent. However, the current state of the art only allows to restore variants that have been committed earlier (extensional versioning), but not to recombine new variants based on predicates similar to feature configurations (intensional versioning).

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 the variant space to be subject to temporal evolution.
Figure 12: Four additional changes, which add different constructors to the class `Edge`, each with a corresponding ambition.

Figure 13: Additional change realizing the feature `hyper`, associated with a corresponding ambition.

The tool SuperMod presented in this paper builds upon the uniform version model (UVM) presented in (Westfechtel et al., 2001). UVM’s basic concepts (options, visibilities, constraints) have been initially introduced in the context of change-oriented versioning (CoV) (Munch, 1993). The tool EPOS-DB is an implementation of CoV concepts. In contrast to SuperMod, both the version space and the product space are represented at a low level of abstraction (propositional formula and text files, respectively).

In (Zeller and Snelting, 1997), an approach to unified versioning based on feature logic is presented. Versions of text files are stored with selective deltas; visibilities are controlled by feature-logical expressions. Constraints on feature combinations are expressed by (low level) version rules.

In (Walkingshaw and Ostermann, 2014), an approach to filtered (projectional) editing of multi-variant programs is described. Like in our work, the motivation is a reduction of complexity gained by hiding variants not important for a specific change to a multi-variant model. Visibilities are managed automatically, but in contrast to our approach, the choice always equals the ambition. Furthermore, the restriction of a completely bound choice does not exist since the user operates on a partially filtered product which still contains variability. Our tool presented in this paper ensures a relaxed form of the edit isolation principle discussed in (Walkingshaw and Ostermann, 2014, Section 4): a change may affect only those variants that are included in the specified ambition.

Using Stepwise and Incremental Software Development (Apel and Kästner, 2009), a sub-discipline of Feature-Oriented Software Development, features are described as refinements or layers. This replaces the necessity of an explicit mapping in the form of presence conditions, but the increments need to be specified in a form that deviates from the “normal” implementation language, e.g., model transformations. In contrast, SuperMod enables for SISD using a familiar development environment.

The source-code centric tool CIDE (Colored IDE) (Kästner et al., 2008) generalizes preprocessors using a colored representation to distinguish features. CIDE is based on negative variability and offers the possibility to temporarily restrict a variational project to views on a specific feature or variant. Then, irrelevant source code fragments are hidden. The performed changes only affect the selected feature or variant, i.e., choice and ambition must be equal.

The MDPLE tool Feature Mapper (Heidenreich et al., 2008), which is based on negative variability, offers the possibility of change recording during domain engineering. Having selected one or several features and invoked the record operation, all changes performed are associated with a feature expression derived from the provided feature selection. However, only insertions are supported, and change recording is restricted to GMF-based editors.

7 CONCLUSION

In this paper, we have presented SuperMod, a model-driven tool that combines the management of variability in time and variability in space, i.e., version control (VC) and Software Product Line Engineering (SPLE). Typical SPLE processes distinguish be-
between **domain engineering**, where a platform and a variability model are defined, and **application engineering**, where variability is resolved to automatically derive specific products. In contrast, in VC, software is developed iteratively. SuperMod bridges this gap by transferring VC metaphors to SPL. For the selection of versions during check-out and commit, **feature configurations** are specified in addition to a selection among the revision graph. The mapping between the platform and the variability model is managed automatically.

In a running example, where a product line of **graph** domain models has been developed, we have demonstrated many advantages of the VC/SPLE integration. Due to the filtered editing model, the versioning overhead is notably small when compared to existing SPL approaches. For workspace modifications, the developer is not restricted by single-version constraints. Furthermore, a familiar development environment can be used. **Intensional** version specification allows for the definition of feature configurations as version descriptions. These advantages are boosted by using models as higher-level descriptions of the versioned software system.

Future work will address the development of a multi-user component, which will advance SuperMod to a full-fledged distributed VCS. The evolution of the feature model will be subject to research. Furthermore, a detailed evaluation against SPL tools will be conducted, using a real-world example. The obtained results will be important to understand the impact of the filtered SPL editing model on the underlying development processes and tool chains.

### TOOL AVAILABILITY

The research prototype **SuperMod** is available as a set of Eclipse plug-ins under the Eclipse Public License. The plug-ins may be installed into a clean Eclipse Luna Modeling distribution using the following update site (**Help — Install new Software**):

http://btnlx4.inf.uni-bayreuth.de/supermod/update

In order to reproduce the example provided in this paper, at least the items **SuperMod Core** and **SuperMod Revision+Feature Layered Version Model** should be selected for installation.

After having installed the plug-ins, SuperMod version control may be added to arbitrary Eclipse projects using the operation **Team — Share Project** and selecting the **SuperMod repository connector**. After that, the operations **Team — Commit** and **Team — Update/Switch** are available in order to communicate with the locally persisted repository. The feature model may be edited by **Team — Edit Version Space**.

### REFERENCES


