MOD2-SCM: Experiences with co-evolving models when designing a modular SCM system

Thomas Buchmann, Alexander Dotor, and Bernhard Westfechtel

Lehrstuhl Angewandte Informatik 1, University of Bayreuth
D-95440 Bayreuth
firstname.lastname@uni-bayreuth.de

Abstract. Software configuration management (SCM) is the discipline of controlling the evolution of large and complex software systems. Many tools and systems for SCM have been developed which are based on a variety of different version models. Usually, an SCM application is a single software system whose underlying architecture is implicitly defined by its implementation. MOD2-SCM, instead, is a modular and extendable SCM system whose components have been explicitly modeled. Each component is defined by a separate model which addresses a specific area of the SCM domain. These components are loosely coupled by extending a common core model. We describe how the architecture of MOD2-SCM has been designed to support the co-evolution of the various component models. Additionally, we discuss the difficulties we experienced when it comes to the co-evolution of the common core model and the component models.

1 Introduction

Software configuration management (SCM) is the discipline of controlling the evolution of large and complex software systems. A wide variety of SCM tools and systems has been implemented, ranging from small tools such as RCS [1] over medium-sized systems such as CVS [2] or Subversion [3] to large-scale industrial systems such as Adele [4] and ClearCase [5].

Version control is a core function of any SCM system. Version control is based on *version models*, many of which have been implemented in research prototypes, open source products, and commercial systems [6]. While there are considerable differences among these version models, it is also true that similar concepts such as revisions, variants, state- and change-based versioning appear over and over again. Unfortunately, version models are usually implicitly contained in the implemented systems.

Thus, the SCM domain is characterized by a large number of systems with more or less similar features, incorporating hard-wired version models which have been implemented with considerable effort. This observation has motivated us to set up a project dedicated to a *model-driven modular SCM system* (abbreviated as *MOD2-SCM*):

1. *Version models* are defined *explicitly* rather than implicitly in the code. This makes it easier to communicate and reason about version models.
2. Modeling comprises both *structure* and *behavior*. Furthermore, behavioral models are executable.

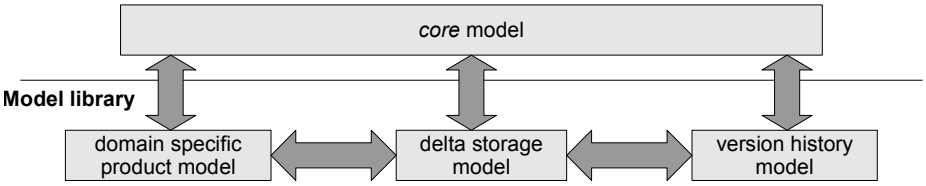

Fig. 1. Two types of co-evolution: horizontal and vertical

3. Productivity is improved by replacing programming with the creation of *executable models*.
4. Version models are not created from scratch. Rather, reuse is performed on the modeling level by extending a core model.
5. All models contribute to a *model library* which is composed of reusable and loosely coupled models.

This leads to two kinds of co-evolution as depicted in figure 1:

1. **horizontal:** i.e. between different models of the model library. Ideally, we can cope with the co-evolution as MOD2-SCM decouples the various models of the library. We show how this is achieved in section 4.
2. **vertical:** i.e. between the core model and the various models of the library, which is the common dependency between an abstract base model and its derivatives.

We used UML class diagrams to model the architecture of MOD2-SCM as well as Fujaba story diagrams to model the behavior. We discovered that some of our architectural decisions are lost when mapping UML onto source code. These issues are discussed in section 5.

2 Background

In this paper, we focus on *version control*, which constitutes a core function of SCM. In order to support version control, an appropriate underlying *version model* is required which defines the data to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. Many version models has been proposed; see [6] for a survey. Below, we recall some basic terminology and discuss our selection of *core modules* by comparison of already existing version models.

A *version* v represents a state of an evolving *item* i. The term "item" covers files and directories, objects in object-oriented databases, etc. A *versioned item* is an item that evolves into multiple versions maintained in a repository. In contrast, only one state is maintained for an *unversioned item*.

A version v is characterized by a pair $v = (ps, vs)$, where ps and vs denote a state in the product space and a point in the version space, respectively. The *product space* contains the items to be versioned, the *version space* arranges their versions in structures such as e.g. version graphs (see below).

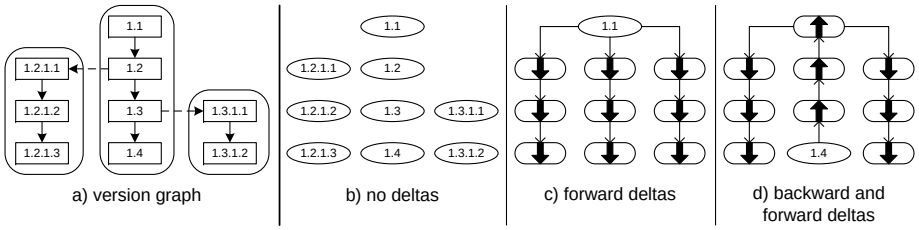

Fig. 2. Logical organization of a version graph (a) and alternative physical organizations (b–d)

The difference between two versions of a versioned item is called a *delta*. Using *directed deltas*, a version is constructed by applying a sequence of changes to some *baseline*. Alternatively, all versions are stored in an overlapping manner so that common fragments (e.g., sequences of text lines) are shared. Either each version points to its fragments (*embedded deltas*), or the fragments are decorated with *visibilities* (control expressions) for determining the versions in which they are contained (*selective deltas*).

According to the kind of evolution, versions are classified into revisions and variants. Sequential versions (e.g., for bug fixes) that evolve along the time dimension are called *revisions*. Parallel versions coexisting at a given time are called *variants*.

Above, a version has been defined as a state of an evolving item. Version models which focus on the states of versioned items are called *state-based*. In the case of state-based versioning, versions are described in terms of revisions and variants. Changes provide an alternative way of characterizing versions. In *change-based* models, a version is described in terms of changes applied to some baseline. To this end, changes are assigned change identifiers and potentially further attributes to characterize the reasons and nature of a change.

To represent the version space, the versions of an item are often organized into a *version graph*. In many systems, version graphs are composed of *branches* (for variants or concurrent work), each of which consists of a sequence of revisions. An example is given in Figure 2a, which shows an RCS/CVS-like version graph consisting of a *main trunk* (in the middle) and *side branches*.

To perform development and maintenance, the user has to establish a uni-version *workspace* on the versioned *repository*. Moreover, the workspace provides the data in a form (usually files) on which external tools operate. Typically, data are transferred between repository and workspace by *checkout* and *checkin* operations.

When multiple users operate on a shared repository, their work must be synchronized. In the case of *pessimistic concurrency control*, locks are set to prevent inadvertent concurrent changes. *Optimistic concurrency control* avoids locks, but checks for conflicts at checkin time and enforces conflict resolution.

Version models realized in SCM systems differ considerably from each other. The taxonomy introduced in [6] identifies the *core modules* and the typical elements they contain, by analyzing what version models have in common and where they have been designed and implemented in different ways. Our selection of *core modules* is presented with examples of typical elements in Table 1.

Table 1. Identification of core modules and typical elements

product model	files, directories, ...
versioned items	versioned files, versioned directories, ...
delta storage	baselines, (directed) deltas, ...
versions	sequence, tree, dag, ... revisions, variants, changes, ...
concurrency control	locks, users, ...

In contrast to the monolithic SCM systems, MOD2-SCM consists of several modules. By examining various evolution steps common to SCM systems we can identify now several co-evolving modules:

product model and versioned items: Whenever a new type of product model is added, it requires new versioned items. Most VCS are therefore restricted to file systems (i.e. CVS, Subversion or ClearCase).

product model and delta storage: Whenever a new type of product model items is added it requires its own "deltification" algorithm. This is already the case for file system based VCS (deltas for different file types, i.e. text and binary files) and becomes more apparent when it comes to completely different product models (i.e. UML diagrams).

delta graph and version history: Whenever the version history is changed the delta storage is affected, as in most VCS the version graph is traversed to apply the deltas (i.e. CVS uses forward deltas on branches).

version history and repository access methods: Whenever the version history is changed the repository access methods have to reflect these changes. If, for example, branches are introduced there is now a branch name that has to be taken into account (i.e. as parameter in commit/update operations). This also the case when the version identifieres evolve (i.e. introduction of tags).

workspace and repository: The evolution of a user's workspace and the following synchronization process can be considered as co-evolution. In contractdiction to the cases above, it is explicitly addressed by most VCS. It is not part of this paper but discussed in [8].

In section 4.3 we show our approach to minimize the effects of co-evolution in our system.

3 Approach

On the one hand, an analysis of the version models realized in SCM systems reveals that similar concepts appear over and over again. On the other hand, the degree of variations should not be underestimated. As a consequence, it is inherently difficult - maybe even impossible - to construct a *uniform version model* from which all specific version models may be derived (see also Section 6).

For this reason, we follow a less ambitious approach: We try to build customized SCM systems from reusable assets which are organized into a *model library*. An underlying *model architecture* assists in creating an SCM system from a set of reusable

components. The modeler using the model library is aware of the design decisions to be performed at previously defined variation points. As the model library grows, the modeler may expect to reuse more and more already existing components realizing the design decisions. However, the overall process goes beyond pure selection and configuration and may require the construction of new model components. Altogether, we follow an *open development approach* in the sense that the modeler is not restricted to a set of pre-defined configuration options. Rather, the desired version model is realized by reusing as many components from the model library as possible. The "rest" is supplied by hand-written model code.

Design for change is crucial for MOD2-SCM, as the number of co-evolving models increases whenever a new one is added to the library. It has to be taken care that the component models in the model library are decoupled as far as possible. In this way, implications of design decisions may be localized. For example, the structure of the version graph is orthogonal to the selected delta storage. This is illustrated on the right-hand side of Figure 2: (b) no deltas are used at all, each version is stored as a *baseline*; (c) only the root version is stored as a baseline, all other versions have to be reconstructed by *forward deltas*; (d) the most recent version on the main trunk is stored as a base line, all other versions are restored by *backward deltas* and forward deltas (the RCS/CVS solution). Thus, the component for the version graph should not depend on a specific delta storage and vice versa.

To realize our MOD2-SCM, we have selected the object-oriented modeling language and environment *Fujaba* [7]. In Fujaba, the structural model is defined with the help of *class diagrams*. The behavior is modeled with story patterns and story diagrams. Models written in Fujaba are executable; the Fujaba compiler translates them into Java code. *Story patterns* are UML-like communication diagrams. A story pattern consists of a graph of objects and links. Graph transformations are expressed by marking graph elements as deleted or created. Furthermore, conditions may be defined for attribute values, and the values of attributes may be modified. Finally, methods may be called on objects of the story patterns. Thus, story patterns constitute a uniform language construct for defining graph queries, graph transformations, and method calls. Programming with story patterns is supported by *story diagrams*, which are similar to interaction overview diagrams in UML 2.0. A story diagram consists of story patterns which are arranged into a control flow graph. Each story diagram is used to implement a method defined in the class diagram.

A carefully designed *model architecture* plays a crucial role in the context of a model-driven modular SCM system. We use *package diagrams* as a notation for the model architecture. Unfortunately, Fujaba does not support package diagrams. The package diagrams presented in the next section were reverse engineered from the generated code (in Fujaba, classes may be assigned to packages, but the package structure may not be visualized in a package diagram).

Typical SCM systems are monolithic systems which makes it difficult to reason about co-evolution. By selecting various evolution steps common to SCM systems and applying these steps to MOD2SCM we can identify co-evolving modules. This makes the co-evolution of the various aspects of SCM systems explicit and allows us to eval-

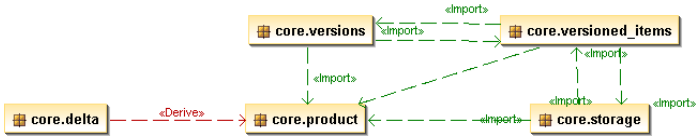

Fig. 3. Package diagram of the MOD2-SCM core

uate our architecture: the less modules co-evolve in a step the better we succeeded in decoupling the modules.

4 SCM Model

In the following section, we describe the current architecture of MOD2-SCM. The first subsection deals with our core model which provides the basic features of SCM systems introduced in [6]. We explain the design decisions that were necessary to ensure the library models are as loosely coupled as possible. The main goal of our approach is building a model library with reusable components which is designed for changes and extensions.

The second subsection gives an example of a CVS-like extension of the core model. It demonstrates how our core model is extended to model an SCM system with commonly known features on top of it. Please note that our current work is still under development and certain modules, like concurrency control, have to be added in the near future. Due to space restrictions, we can not describe the synchronization of workspaces and the repository. Please refer to [8] for further details on this topic.

In the last subsection we discuss problems of co-evolution that occur in SCM systems and explain how we solved and addressed them.

4.1 The core model

The *core model* of the MOD2-SCM consists of five distinct *kernel models* based on the modules identified in section 2: *product model*, *versioned items*, *versions* and *delta storage*. Each module is mapped onto a single kernel model (`core.<modulename>`), except the module delta storage which is mapped onto two different kernel models. Each kernel model is defined by an UML class diagram, whose classes are grouped within a single package of the same name as the kernel model, as well as various Fujaba story diagrams (one for each method). The relations between the five kernel models are shown by the package diagram in Figure 3.

Each kernel model makes only minimal assumptions about its context and provides an interface which is extended by the more concrete *library models*, i.e. a concrete filesystem product model extending the *core.product* model. This way each library model shares the following qualities:

– Models extending the same kernel model are *exchangeable* as they implement the same interface.

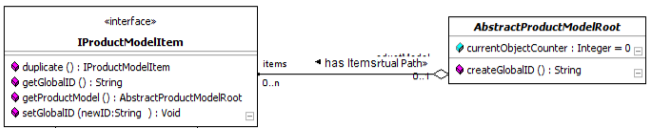

Fig. 4. Class diagram of `core.product`

- Models extending different kernel models are *loosely coupled* since each kernel model makes only minimal assumptions about its context.
- The model library become *extendable* as the introduction of new models is intended by extending the appropriate kernel model.

In order to obtain a complete SCM model, each kernel model has to be implemented by a library model. Once this is done, the library models have to be combined by a repository server model, which instantiates the selected library models and provides a specific interface for the server. Take the CVS repository server model as an example (p. 12).

core.product (see Fig. 4): Each product model instance consists of a root node which contains all product model items. Each item can be uniquely identified.

Minimal Assumptions: Each product model item needs a unique (and unmodifiable) *global id* to identify different versions of the same item. It also requires an operation to duplicate itself as it is necessary to store a copy of an item and not only a reference. In the later case each modification of an already versioned item will modify the stored version, too! The association between the product model root and the item is a *virtual path*[1], i.e., the concrete association is specified separately for each concrete product space model (and may even be derived from a more complex data structure).

Basic Functions: none

Dependencies: none

Usage: Each class of product model items must implement the interface *IProductModelItem*. A class must be chosen as product model root, and an association to all items must be defined.

core.versioned_items (see Fig. 5): Each versioned item model instance consists of versioned items that can be stored in multiple versions (each identified by a *version id*). A root node object provides access to all versioned items.

Minimal Assumptions: Each versioned item has a unique *object id* and is both associated with a set of versions and a storage for product model items. Each versioned item can be associated with a product model item by using its global id as object id.

Basic Functions: Each versioned item manager provides operations for adding and retrieving product model items. A *version id* is necessary for retrieval, and returned when a product model item is added. Computation of the version id, however, is delegated to `core.versions`.

[1] A *virtual path* resembles a *derived association* in UML. Instead of being generated, the navigation methods are defined by story patterns. This way a direct relationship can be derived from a more complex data structure.

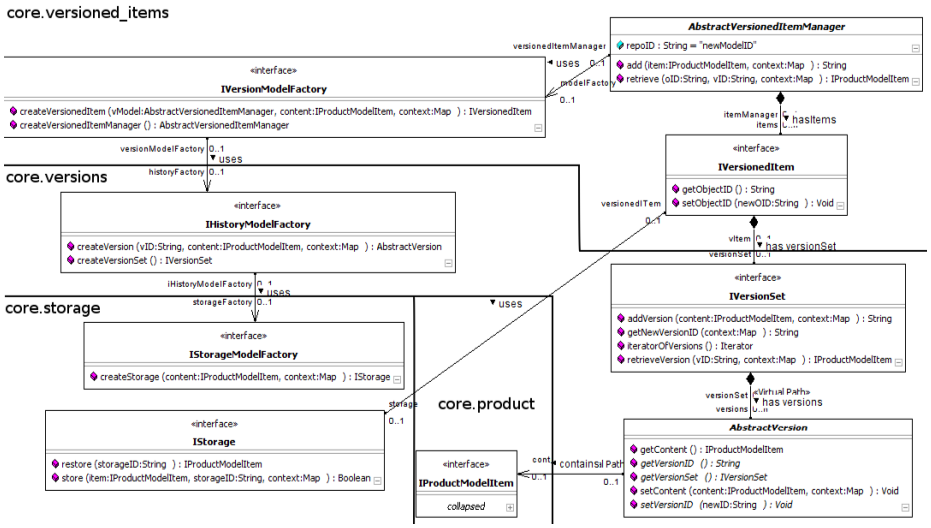

Fig. 5. Class diagram of `core.versioned_items`, `core.versions` and `core.storage`

Dependencies: Loosely coupled with `core.product`, as it requires a global id for each product space item. It is also coupled as loosely as possible with `core.versions`: versioned item and versions are separated, so different version models (i.e. version graphs) become easily exchangeable.

To decouple the versioned item model further from the version and storage model, a *context*-parameter is introduced. The context is used in a complete SCM model to pass further name-value-pairs to the version or storage model without changing the signature of the add/retrieve operations of the versioned item model. Take the *predecessor id* in case of the CVS version model as example (p. 12).

Usage: Each class of versioned items must implement *IVersionedItem*, and a versioned item manager must be added. A factory implementing *IVersionModelFactory* must be added to create instances of these newly modeled classes. To support exchangeability of different concrete versioned item models, each must have a factory for creating a new versioned item manager and a versioned item.

core.versions (see Fig. 5): Each version model instance consists of versions of product model items maintained in a set.

Minimal Assumptions: Each version resembles a product model item identified by a version id that is unique within the set. As each versioned item has exactly one version set, each version is uniquely identified within an SCM model by the tuple *(object id, version id)*. By using a set as container for the versions, we have defined the version history as abstract as possible. More precisely, we do not not introduce any concept of a version history at all. This way each concrete version model can define its own data structures, which allows us to support a large number of version models (i.e. version graphs).

Fig. 6. Class diagram of `core.delta`

Basic Functions: Each version set supports three operations: adding new versions, retrieving versions and creating a new version id. Each version references its product model item which is stored in a *storage*.

Dependencies: Dependent on `core.product` as each version is ultimately a product space item. To support the exchangeability of various version history models, a factory is used to create the concrete version set and version instances. The storage of the concrete product model item is delegated to `core.storage` and so decoupled from the version itself.

Usage: Each concrete version model must have at least two classes: one acting as version set and another as version by extending *IVersionSet* and *AbstractVersion*. Finally, the association between those two classes must be defined in concrete version model terms. Take the CVS version graph model as an example (p. 10).

core.storage (see Fig. 5): Each storage model instance consists of a single storage for product model items. It may even transform the stored items to save storage space.

Minimal Assumptions: A storage contains a number of product model items stored under a specific *storage id*.

Basic Functions: A storage supports operations for storing and restoring as described above. If storing transforms the item, the inverse transformation must be performed during restoration.

Dependencies: none

Usage: Each concrete storage model must define a storage which implements *IStorage* and provide a factory for creating instances of it.

core.delta (see Fig. 6): Each delta model instance consists of deltifiable items and deltas, to reduce the storage space of the deltifiable items.

Minimal Assumptions: A deltifiable item is a special type of product model item.

Basic Functions: Two deltifiable items can be compared in order to produce a delta. A delta can be applied to a deltifiable (product model) item in order to produce a new deltifiable (product model) item.

Dependencies: Depends on `core.product` as each deltifiable item is a product model item, too. Decoupling the delta model from the product model completely proved unfeasible once the deltas are used with a storage model. Each version stores a product model item so a restored delta item has to be a product model item, too. If not, it is no valid content of a version. See p. 11 how to combine the abstract storage and delta models to create a concrete delta storage model.

Usage: Each type of product model items that shall benefit from delta storage must implement *IDeltaItem*. Additionally, a type specific delta must be defined that implements *IDelta*. This means the complete creation and application of the delta mechanism is specified in the product model instance.

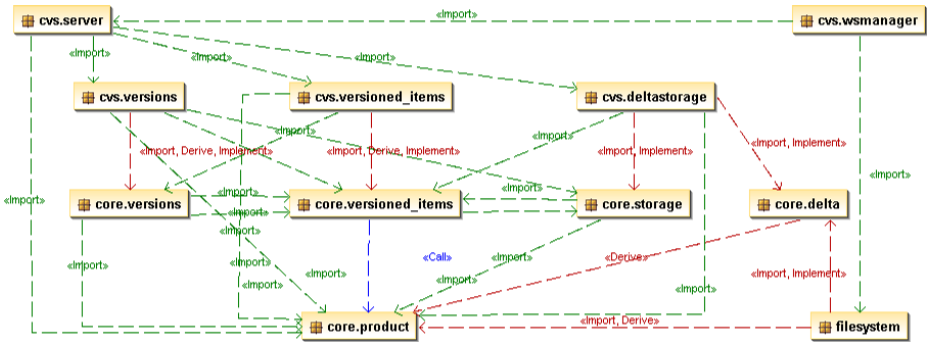

Fig. 7. The package diagram of the core model with the CVS-like extensions

4.2 The CVS-like extension

In this section we present an extension of the core model which provides CVS-like features. In particular, these features cover the CVS version graph including the development history and the delta storage mechanism using directed deltas. The basic idea was to extend the existing library with CVS-like functions.

The necessary steps and extensions to build the CVS-like version control system on top of the core model of MOD2-SCM are described in the following section.

Several new models have been introduced which extend the core model. Due to space restrictions, we focus on the extension to the versioning part (`cvs.versioned_items`, `cvs.versions`, `cvs.delta` and `cvs.server`). The extension of the product model (`filesystem`) will be omitted. The relationships between the single models can be seen in the corresponding package diagram in Figure 7.

Since the models for version, history and delta storage are orthogonal, we could also use the `cvs.versioned_items` model with a different history. As you can see in Figure 7, there is no dependency between `cvs.versioned_items`, `cvs.versions` and `cvs.deltastorage`. This shows that theses models have been decoupled successfully.

cvs.versioned_items: This concrete version model uses the same concepts as the core model does. Therefore, the only task was to derive and implement the abstract classes and interfaces given in the kernel model described on page 7.

cvs.versions (see Fig. 8): This model adds the CVS concept of branches, tags and time ordered revisions to our system. Branches are used to store parallel versions (variants) of versioned items. In a CVS repository, there is always a main branch called *trunk*. Tags are used to mark special versions, in order to set specific points in the development history, e.g. releases, bug-fixes and so on. Having time ordered revisions implies that each revision has at most one predecessor and successor, respectively.

In the previous section we mentioned the use of a virtual path between `IVersionSet` and `AbstractVersion` (see p. 8). We demonstrate how this specific modeling construct is used in our concrete model. Figure 8 shows a cutout

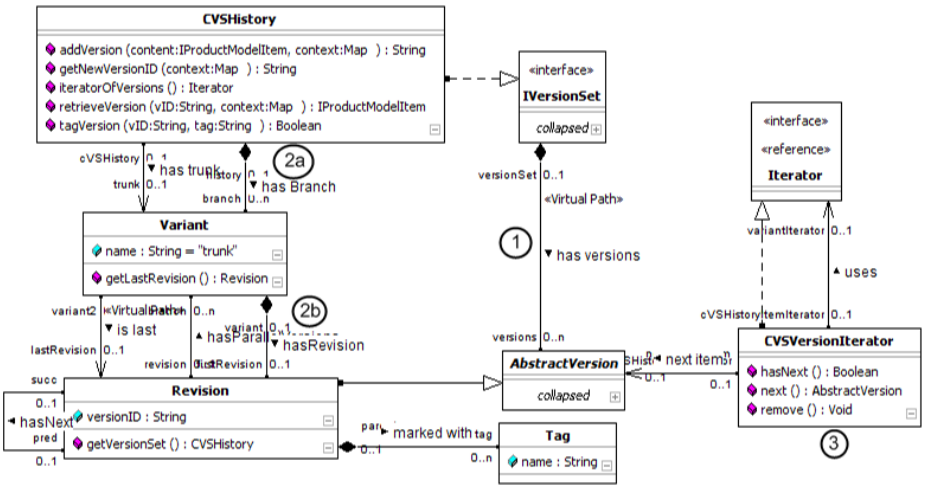

Fig. 8. Class diagram of the CVS-like history model

of the class diagram of `cvs.versions` (please note that the factory classes are omitted due to space restrictions). (1) shows the virtual path as it is defined in `core.versioned_items`. Since the `Revisions` are stored in a CVS version graph, which is able to express parallel and time ordered versions in our concrete implementation, a different mechanism is needed to retrieve all history items associated to the history. We use a custom iterator (3) which processes the associations (2a) and (2b) to achieve this. Now, every time the virtual path association is used in a model, our custom iterator is used.

cvs.deltastorage (see Fig. 9): This model covers the delta storage mechanism as used in CVS by extending the kernel models `core.delta` and `core.storage`. In CVS the latest version on the main branch (the trunk) is always stored as a baseline. Previous versions on the trunk are stored using backward deltas, whereas forward deltas are used on branches (see Figure 2d).

In our model this is done by `DeltaStorage` which implements the store and restore methods from the `IStorage` interface and covers the logic to store baselines (which contain the product model items) and forward/backward deltas respectively. Since `cvs.deltastorage` and `cvs.versions` are orthogonal, it is easily possible to exchange these models and use the `cvs.deltastorage` with a completely different history and vice versa. There is only an import dependency to `core.versioned_items`, because the `DeltaStorage::store` method requires access to the version set and versioned items.

In order to use the delta mechanism, product model items must also implement the `IDeltaItem` interface, which provides a method to compare these items and retrieve a delta. Resulting deltas are connected using a predecessor/successor relationship, where every delta has an optional predecessor and (possibly) many successors (e.g. when branches are used, a delta can have more than one successor). Please note,

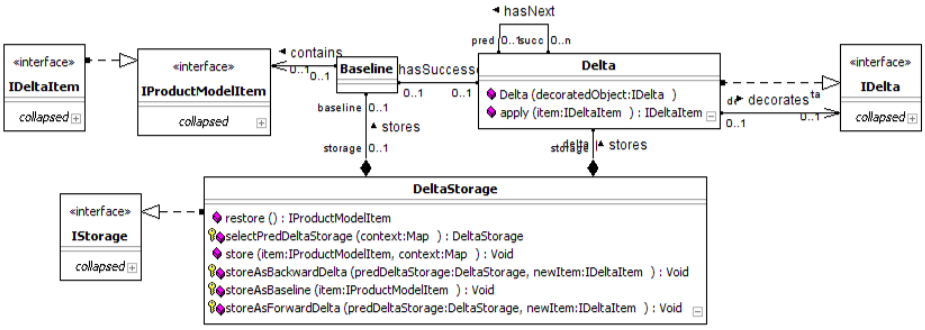

Fig. 9. Class diagram of the CVS-like delta model

that in case of CVS the ordering of the deltas is reverse to the ordering of the revisions on the trunk.

cvs.server: The different models described above are used and assembled in `CVSRepoServer`. It covers functions to commit items into the repository and to check them out again. CVSRepoServer uses the factories of `cvs.versioned_items`, `cvs.versions` and `cvs.deltastorage` to create the corresponding objects. Due to space restriction we have to omit the figure of the class diagram.

The flexibility of our approach will be demonstrated with a concrete example. Assume we want to commit a new version to our CVS-like SCM system. Since our version model supports branches and ordered revisions this information is mandatory when new revisions should be stored in the version graph. Therefore, the `commit` method[2] in class `CVSRepoServer` requires three parameters: the branch name where the item should be committed to, the predecessor version id, and the item itself.

As described above, the signature of the methods in the generic parts of our versioning system do not cover ordering of versions (with predecessor and successor relationships) or having parallel versions (like branches in CVS). Therefore, these additional parameters have to be put into the context object that is passed to the `add` method of the version model and the history model respectively.

Since we use methods complying to the signature of the kernel models, it is easily possible to replace our concrete models.

The Server holds references to the concrete implementations of `AbstractVersionedItemManager` and `IVersionModelFactory` respectively. In the following, we want to give a short listing of the steps and method calls which are required when committing to the repository.

1. **add** method of the associated version model is called
2. call **addVersion** method of the used history model
3. store the content of the committed item using the **setContent** method of the associated storage model

[2] Please note that this method does not correspond directly to the `commit` method in CVS. Rather it is similar to the `checkin` method, since only a single item is stored in the repository.

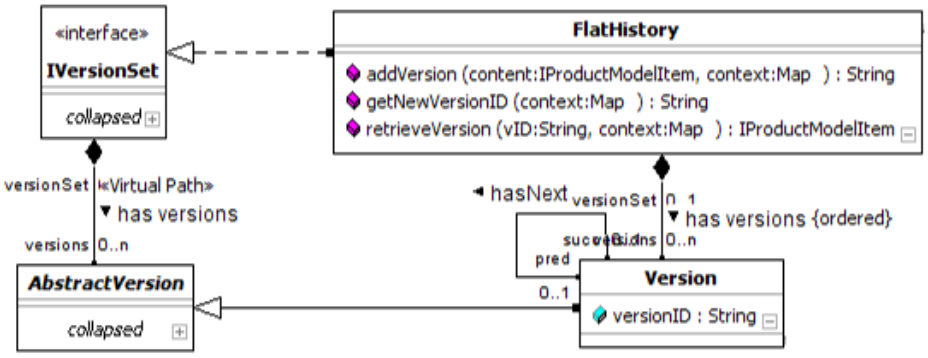

Fig. 10. class diagram of a flat history implementation

Please note, that all of the above methods are specified in the corresponding interfaces in the kernel modules. That means that each of these models could be replaced by a different kernel model extension without affecting the rest (since we only use methods complying to the signature of the kernel models). For example, we could replace the CVS-like history model with a flat history [3] as shown in figure 10.

Also, the history model does not need any knowledge about the storage mechanism which is used in the extension of the corresponding kernel model. Since these models are orthogonal as well, it is easily possible to replace the CVS-like storage mechanism with a different one (e.g. a mechanism which only uses forward deltas, or one could use a mechanism which doesn't use deltas at all).

4.3 Co-evolution: problems and solutions

As already stated in chapter 2, we want to minimize the number of modules, which are affected by certain evolution steps.

product model and versioned items: As long as a new product model item complies to the specified interface, the versioned items do not require a change. On the other hand, changes to the versioned items do not affect the product model at all, because the history model itself only holds a reference to the `IProductModelItem` interface, as shown in Figure 5.

product model and delta storage: As mentioned in chapter 2, new product model items require new "deltification" algorithms, in order to store them efficiently. That means, that if new product model items are introduced, and delta storage is needed, the modeler has to specify new delta algorithms, because each type of product model item has to implement its own algorithms for delta computation.

delta graph and version history: In our system, the delta graph and the version history are independent. That means, changing e.g. from a time-ordered history to a history supporting branches as well, does not affect the delta storage mechanism.

[3] This kind of history model only supports ordered revisions on one branch. No variants are supported.

version history and repository access methods: The core system provides the possibility to extend the submitted data by specifying a context parameter. That means changes in the version history have to be reflected only in the server and in the workspace manager. The signature of the server methods is not affected, because these methods comply to the generic interface. This works only to a limited extent. If a new operation is required, this approach does not work anymore.

5 Difficulties with co-evolution of models and source code

When modeling MOD2-SCM with UML class and story diagrams we encountered several difficulties that are located in the domain of co-evolution between models and source code and we also encountered problems related to UML. By investigating the corresponding specifications, we can show that these are in fact more general problems.

The goal of our work was to develop a model driven product line for software configuration management systems. To realize this, we had to investigate the impacts of evolution of a single component on the other modules of an SCM system. In the previous section we demonstrated our approach to keep these impacts as minimal as possible, and to support a product line. From the modelers perspective the separation of the whole system into single modules with variation points brings up some problems related to UML. The *import*-relationship in UML is more loosely defined than on the source code level ([10], p. 143). UML allows modification of imported elements, in contrast to the mere *uses*-relationship on the source code level. This leads to a mismatch between the imports in model and source code. For example, a class B is imported on the model level into package p (containing class A). Now a bi-directional association between A and B is added. This inserts a cyclic import into the source code as both classes need a reference of the opposite type. This leads to decoupled models that are (re)coupled by the generated code. This is also the cause why we generated our package diagrams from the source code. On the other hand, UML package diagrams do not show, which classes of a package are used in the importing package.

6 Related Work

SCM systems usually incorporate hard-wired version models [6]. In our MOD2-SCM project, we do not intend to develop yet another version model. Rather, we focus on a new way of constructing SCM systems using a model-driven product line.

Nearly a decade ago, the common principles underlying the different version models of SCM systems were investigated in a few research projects which were dedicated to the development of a *uniform version model*. Both ICE [11] and UVM [12] proposed generalizations of conditional compilation, where fragments are decorated with logical expressions defining their visibility. In both projects, it was demonstrated that different version models may be simulated with logical expressions for controlling visibilities and expressing constraints. For example, revision chains may be expressed by constraints of the form $\Delta_i \Rightarrow \Delta_{i-1}$, reading: "When you apply some delta of the chain, you must apply the preceding delta, as well." Please note that the expressive power of the logical calculi employed corresponds to propositional rather than predicate logic.

For example, it would not be possible to state constraints which exclude cycles of delta implications or enforce revision sequences.

Our MOD2-SCM project follows similar goals as ICE and UVM. However, from a technical perspective the approach is completely different: Version models are explicitly defined in an object-oriented modeling language; executable code is generated from these models. In contrast, both ICE and UVM provide a uniform base mechanism for version control. However, version models have to be programmed (rather than modeled) on top of this base mechanism — which implies a high customization effort. Furthermore, in order to perform the customization of the base layer correctly, a formally defined version model would be required, anyway.

To date, only a few approaches have been dedicated to *model-driven development of versioning systems*. In Bamboo [13, 14], version models are defined in an extended ER data model (Containment Modeling Framework). An SCM system is generated from a CMF model. However, the model defines only the structure, but not the behavior (which is programmed in Java). From the publications, it is not clear what kind of functionality is actually provided by a generated SCM system. Furthermore, each version model is defined from scratch, while we employ reuse in a model-driven product line.

The PhD thesis of Kovŝe [15], which was carried out in the context of database systems rather than SCM systems, investigates a product line approach to the model-driven development of versioning systems. The user of the product line defines a version model in a UML profile, where stereotypes and tagged values are used to parameterize the behavior of modeling elements. This implies that the underlying version model is fixed by the framework and can be customized only at some pre-defined variation points. On the one hand, this approach reduces the customization effort; on the other hand, the flexibility is restricted severely. In contrast, we follow an open approach in the sense that the designer of a version model may reuse whatever is already in the model library, and add extensions wherever this is required. Another difference concerns the modeling of behavior: In the approach followed by Kovŝe, the user of the product line merely customizes pre-defined and programmed behavior. In our approach, the behavior is modeled (with story diagrams) rather than programmed.

7 Conclusion

We have presented a novel approach to developing SCM systems. We separated the different parts of SCM systems in modular packages. These packages are described with the help of executable models which are created with the help of Fujaba, an object-oriented modeling language and environment. Model reuse is supported by a model library which is composed of loosely coupled packages. Design for change is realized through the use of interfaces, abstract classes, design principles such as “favor composition over inheritance”, and design patterns such as abstract factories. Unidirectional links and derived associations are suitable modeling techniques to achieve decoupling of components and to minimize imports of other packages. The concept of co-evolution helped us to verify our modular design, as the number of changes to dependent modules was minimized. Method bodies are implemented by story diagrams from which executable code is generated. Story patterns support the graphical definition of complex

graph queries and graph transformations such that development effort may be reduced significantly.

Our work on a modular framework for SCM systems provides an application-oriented contribution to model-driven engineering. However, model reuse cannot be demonstrated by just a single show case. Therefore, in future work we will explore further systems, initially focusing on systems based on version graphs (e.g., Subversion or ClearCase). Throughout our work, we also evaluate experiences gained in applying the used modeling language and environment. For example, we need improved support for *modeling-in-the-large*, which in particular allows to express variation points in a product line architecture. That means, that current modeling tools like Rational Software Architect, Rational Rose or also Fujaba do not provide support for models on the architecture level.

References

1. Tichy, W.F.: RCS – A system for version control. Software: Practice and Experience **15**(7) (July 1985) 637–654
2. Vesperman, J.: Essential CVS. O'Reilly & Associates, Sebastopol, California (2006)
3. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly & Associates, Sebastopol, California (2004)
4. Estublier, J., Casallas, R.: The Adele configuration manager. In Tichy, W.F., ed.: Configuration Management. Volume 2 of Trends in Software. John Wiley & Sons, New York (1994) 99–134
5. White, B.A.: Software Configuration Management Strategies and Rational ClearCase. Object Technology Series. Addison-Wesley, Reading, Massachusetts (2003)
6. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys **30**(2) (June 1998) 232–282
7. Zündorf, A.: Rigorous object oriented software development. Technical report, University of Paderborn, Germany (2001)
8. Buchmann, T., Dotor, A., Westfechtel, B.: Triple graph grammars or triple graph transformation systems? a case study from software configuration management,1st international workshop on model co-evolution and consistency management, mccm 08, toulouse, france, september 30th, 2008. (2008)
9. OMG: MOF 2.0/XMI Mapping. OMG. (December 2007) Version 2.1.1.
10. OMG: OMG Unified Modeling Language (OMG UML), Superstructure. OMG. (November 2007) Version 2.1.2.
11. Zeller, A., Snelting, G.: Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology **6**(4) (October 1997) 397–440
12. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. IEEE Transactions on Software Engineering **27**(12) (December 2001) 1111–1133
13. Whitehead, E.J., Gordon, D.: Uniform comparison of configuration management data models. In Westfechtel, B., van der Hoek, A., eds.: Software Configuration Management: ICSE Workshops SCM 2001 and SCM 2003. LNCS 2649, Portland, Oregon, Springer-Verlag (2003) 70–85
14. Whitehead, E.J., Ge, G., Pan, K.: Automatic generation of hypertext system repositories: a model driven approach. In: 15th ACM Conference on Hypertext and Hypermedia, Santa Cruz, CA, ACM Press (August 2004) 205–214
15. Kovŝe, J.: Model-Driven Development of Versioning Systems. PhD thesis, University of Kaiserslautern, Kaiserslautern, Germany (August 2005)