A Layered Architecture for Uniform Version Management

Bernhard Westfechtel, Bjørn P. Munch, and Reidar Conradi, *Member*, *IEEE*

Abstract—Version management is a key part of software configuration management. A big variety of version models has been realized in both commercial systems and research prototypes. These version models differ with respect to the objects put under version control (files, directories, entities, objects), the organization of versions (version graphs versus multidimensional version spaces), the granularity of versioning (whole software products versus individual components), emphasis on states versus emphasis on changes (state- versus change-based versioning), rules for version selection, etc. We present a *uniform version model*—and its support architecture—for software configuration management. Unlike other unification approaches, such as UML for object-oriented modeling, we do not assemble all the concepts having been introduced in previous systems. Instead, we define a base model that is built on a small number of concepts. Specific version models may be expressed in terms of this base model. Our approach to uniform version management is distinguished by its underlying *layered architecture*. Unlike the main stream of software configuration management systems, our *instrumentable version engine* is completely orthogonal to the data model used for representing software objects and their relationships. In addition, we introduce version rules at the bottom of the layered architecture and employ them as a uniform mechanism for expressing different version models. This contrasts to the main stream solution, where a specific version model—usually version graphs—is deeply built into the system and version rules are dependent on this model.

Index Terms—Version model, version rules, software configuration management, software architecture, software repositories.

1 Introduction

Software configuration management (SCM) has been defined as the discipline of controlling the evolution of complex software systems [1]. Over their lifetime, software objects, such as requirements definitions, designs, program source code, documentations, and test data evolve into many versions. These differ with respect to the underlying operating system, window system, included bug fixes, and change requests. Thus, *version management* plays a central role in SCM. Versions have to be recorded and restored, the consistency relationships between versions of different objects have to be maintained, and new versions have to be created according to rule-based descriptions.

In order to support version management, an appropriate underlying *version model* is required. This defines the data (usually objects) to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. A big variety of version models has been proposed. Among others, the seminal papers of Katz [2] and Feiler [3] surveyed the state of the art as of the late 1980s and beginning 1990s. More recently (in 1998), we conducted a comprehensive and updated survey of version models realized both in commercial systems and research prototypes [4].

- *B. Westfechtel is with the Department of Computer Science III, Aachen University of Technology, D-52056 Aachen, Germany. E-mail: bernhard@i3.informatik.rwth-aachen.de.*
- *B.P. Munch is with Clustra AS, Haakon VIIs Gate 7, N-7485 Trondheim, Norway. E-mail: bjorn.munch@clustra.com.*
- *R. Conradi is with the Norwegian University of Science and Technology, N-7491 Trondheim, Norway. E-mail: conradi@idi.ntnu.no.*

Manuscript received 8 July 1998; revised 14 Jan. 2000; accepted 11 Sept. 2000. Recommended for acceptance by D. Perry and M. Shepperd. For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107117.

The study of version models for SCM reveals many mutual similarities: Virtually the same basic concepts appear over and over again. Triggered by this observation, we have developed a *U*niform *V*ersion *M*odel (*UVM*) [5]) that serves as a common base. UVM is realized by an *instrumentable version engine* that provides a uniform version storage which can be customized to specific version models.

Our approach offers the following *benefits*:

- By supporting the definition of version models in a *uniform framework*, UVM abstracts from notational differences and makes comparison of version models easier.
- Unlike other unification approaches, such as UML [6] for object-oriented modeling, UVM does not merely assemble all the concepts having been introduced in previous systems. Instead, UVM is based on a *small number of fundamental concepts* in terms of which a wide spectrum of version models may be expressed. (This is the reason why we call our model "uniform" rather than "unified.")
- The instrumentable version engine acts as a *reusable core component* for building SCM systems. In this respect, our work follows the lines of extensible database management systems [7], [8].
- An important feature of UVM consists in the *orthogonality between version model and data model*. That is, the version model is independent from the data model used for representing software objects and their relationships. The instrumentable version engine may be equally applied to file-based, relational, and object-oriented data. Conventional approaches either define the version model on top of the data model, or version and data model are deeply entangled.

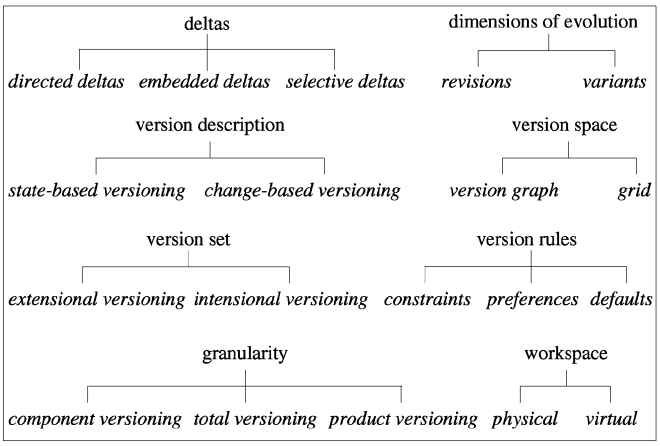

Fig. 1. Taxonomy of version-related terms.

- The instrumentable version engine supports the rule-based construction of configurations of software systems. It serves as a *deductive database system* [9], [10] for the application domain of version management. As we have shown in [11], the deductive capabilities of most SCM systems are severely restricted, even though flexible construction of consistent configurations from rule-based descriptions is urgently needed.

The instrumentable version engine makes building of SCM systems easier. A *vendor* building an SCM system may reuse the version engine as a key component, which obviates the necessity of building an entirely new one. The vendor will perform customization with respect to both product space (e.g., application to file-based data) and version space (e.g., version graphs).

The *end user* does not directly get in touch with the underlying version engine, which has to operate at a low-enough level to provide common base mechanisms. Rather, more high-level layers offering user-friendly interfaces are placed on top of the version storage [12]. Building these layers goes beyond the capabilities of the end user. Rather, we assume that they are provided by the vendor, or by the designers of the version engine, who may provide a library of predefined version models.

The rest of this paper is structured as follows: In Section 2, we summarize basic definitions to be used throughout the rest of this paper. In terms of these definitions, we state the requirements to a uniform version model in Section 3. Related work is compared in Section 4. Our uniform version model UVM is presented in Sections 5, 6, and 7. Experiences gained from a UVM implementation are described in Section 8. A brief discussion of our approach follows in Section 9. Finally, a short conclusion is given in Section 10.

2 Basic Definitions

In the following, we define the notions used throughout the rest of this paper. Most of these notions are summarized in the taxonomy of Fig. 1. For a more detailed discussion, the reader is referred to [4].

2.1 Versions, Versioned Items, and Configurations

A *version* v represents a state of an evolving *item* i. The term "item" covers, e.g., files (usually textual) and file directories, objects in object-oriented databases, entities, relationships and attributes in EER databases, etc.

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 version graphs. For example, ps and vs may denote the contents of a text file and the version number in a version graph, respectively.

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*, i.e., changes are done by overwriting.

A *configuration* is a version of a complex object. It is composed of versions of components. For example, a configuration of a software system is composed of versions of the requirements definition, the software architecture, the program source code, etc.

2.2 Deltas

All versions of a versioned item share some common parts and differ with respect to specific parts. The difference between two versions is called a *delta*. Using *directed deltas*, a version is constructed by applying a sequence of changes to some base version. 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*).

2.3 Revisions and Variants

According to the kind of evolution, versions are classified into revisions and variants. Sequential versions that evolve along the time dimension are called *revisions*. They are created to fix bugs or perform other enhancements. Parallel or alternative versions coexisting at a given time are called *variants*. While new revisions supercede old ones, variants do not replace each other. Rather, they are used concurrently in alternative configurations (or even in the same configuration).

2.4 State- and Change-Based Versioning

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 (possibly empty) baseline. To this end, changes are assigned change identifiers and potentially further attributes to characterize the reasons and nature of a change. In this way, it is possible to trace the changes that have gone into some version.

2.5 Version Graphs and Grids

To represent the version space, the versions of an item are often organized in a *version graph*. For example, in RCS [13] and ClearCase [14] version graphs are composed of branches (for variants), each of which consists of a sequence of revisions.

Alternatively, a *grid* may be used to arrange versions in an n-dimensional space, where each dimension corresponds e.g., to a (variant) selection-attribute, typically being of Boolean or enumerated type. For example, multidimensional variation may be represented this way, concerning, e.g., the language used in user dialogues, the window system, operating system, DBMS, etc.

2.6 Extensional and Intensional Versioning

A versioned item is a container, often called a *version group*, for the set V of all versions of this item. V may be defined either explicitly (by enumerating its members) or implicitly (by a version predicate). In the latter case, a version is a "potential" or "virtual" item [15], depending on the actual predicate, see below.

In the case of *extensional versioning*, V is defined by enumerating its members:

$$V = \{v_1, \ldots, v_n\}.$$

Version management based on extensional versioning supports the retrieval of previously created versions and the storage of new versions. To create a new version, the external user retrieves (checks out) some version v_i, performs changes to the retrieved version, and, finally, submits (checks in) the changed version as a new version v_{i+1}.

Intensional versioning supports flexible, automatic construction of consistent versions in a large version space. The term "potential versions" emphasizes that a certain version may not have been constructed explicitly before. In intensional versioning, the version set is defined by a legality-deciding *constraint, con*:

$$V = \{v | con(v)\}.$$

Thus, all versions v satisfying the constraint *con* will belong the version set V. A specific version v is then described by a selection predicate evd (user-defined or external *version description*), such that the following condition holds:

$$evd(v) \wedge con(v).$$

For example, both *evd* and *con* can be defined by expressions over a space of versioning-attributes A_i (called options in UVM). Examples of such attributes are an `OS`-attribute to determine the operating system, or a `WS`-attribute to denote the window system to be supported.

2.7 Version Rules: Global Constraints, Preferences, and Defaults

Intensional versioning is driven by (global) *version rules* which may be either stored in the versioned database or submitted as part of a query. A *constraint* is a legality-deciding, mandatory rule which must be satisfied. Any violation of a constraint indicates an inconsistency, e.g., the `SUN` and the `VAX` variant must not be selected simultaneously. A *preference* is an optional rule which is applied only when it may be satisfied, e.g., released module versions are preferred. Finally, a *default* is a weak preference. A default is applied only when no unique selection could be performed otherwise, e.g., the latest version of the main branch in a version graph may be selected by default. Thus, both preferences and defaults may further supplement a user vd with extra attribute bindings, going from an external and partially bound vd (an evd) to an internal and fully bound one (an ivd). Therefore, the rule part of a versioned database will check and possibly elaborate an initial evd, cf. the capabilities of *deductive databases* [9]. See more on version rules in Section 6.2.

2.8 Granularity

Seen from the user, versioned items have a minimal *granularity*. The user normally selects versions of *software objects*. While these are rather coarse-grained items, they are treated as atomic objects by the SCM system with respect to identification and version selection. On the other hand, an SCM system may operate at a much finer internal granularity—text lines or even syntactical tokens—to effectively store, compare, and construct versions of software objects.

From the user's point of view (*external granularity*), we distinguish between component, total, and product versioning:

- In the case of *component versioning*, versions of single components are maintained and assembled into composite configurations. For example, RCS maintains versions of text files, each of which has its own version graph. The relationships between these version graphs are defined by version rules for selecting configurations.
- In the case of *total versioning*, all items are versioned, including composites as well as components. For example, ClearCase maintains versions of both files and directories. Each item still has its own version space (version graph in ClearCase). The relationships between these version spaces may be defined either by rules (as for component versioning) or by versions of composites referring to versions of components.
- In contrast, *product versioning* establishes a total view of a software product and its processes etc., or even an entire database. This is done by arranging versions of all items in a uniform, global version space [16], [17]. For example, in Voodoo [18] there is one global revision number for the whole software product, while in RCS each file has its own revision numbers.

2.9 Workspaces and Transactions

To perform development and maintenance, the user has to establish a universion *workspace* on the versioned *repository*. Moreover, the workspace provides the data in a form (usually files) on which external tools operate (editors, compilers, debuggers, etc.).

A workspace may be set up by explicit check-out commands which copy data from the repository typically

Requirement 1 (Generality) *UVM must be general with respect to both product space and version space, i.e., it must put no restrictions on the items to be versioned and the way their version spaces are organized.*

Requirement 2 (Deltas) *UVM has to employ deltas to store versions efficiently, to compare them (`Diff` operations), and to combine them into new versions (`Merge` operations).*

Requirement 3 (Dimensions of evolution) *UVM has to support both revisions and variants.*

Requirement 4 (Version description) *In UVM, a version must be describable both in a state-based and in a change-based way.*

Requirement 5 (Version space representation) *The version space of a versioned item may be represented both by a version graph and a grid.*

Requirement 6 (Version set definition) *UVM must support both extensional and intensional versioning.*

Requirement 7 (Version rules) *For intensional versioning, UVM has to offer version rules in order to represent both internal constraints and external version descriptions.*

Requirement 8 (Selection granularity) *From the user's point of view, version selection may be performed at different levels of granularity (products, subsystems, components).*

Requirement 9 (Workspace management) *UVM must provide some means to establish a workspace in which users perform their work with the help of external tools.*

Fig. 2. Requirements to a uniform version model.

into the file system (*physical workspace* [19], [13]). Alternatively, an SCM system may offer a *virtual workspace* (virtual file system [20], [14]), where an *evd* dynamically serves as an access filter upon all repository accesses, so-called *transparent versioning*.

Finally, the work performed by one user (or a group thereof) may be embedded in a *transaction*. Throughout this paper, the term transaction will primarily refer to cooperative, long-lasting transactions [21], [22], which may be associated to change requests or change sets. We will not focus on classical short ACID transactions ("ACID" stands for atomicity, consistency, isolation, and durability; see [23]). Each transaction owns a *subdatabase*, which contains a part of the overall versioned database. Subdatabases are usually arranged in trees or graphs, the edges of which define the cooperation paths between transactions and the corresponding data exchange.

3 Requirements and Design Decisions

From the taxonomy presented in the previous section, we derive a set of requirements which our *U*niform *V*ersion *M*odel—called *UVM* below—has to meet (Fig. 2).

These requirements are generic and leave many degrees of freedom to the designer of an SCM system supporting a uniform version model. To determine the *system architecture* [24], design decisions have to be made in particular with respect to the following issues.

3.1 D1: Relationship between the Data Model and the Version Model

The *data model* (e.g., file-based, relational, object-oriented) defines the basic constructs for expressing the product data. The *version model* defines the identification of versions and their organization, as well as operations for retrieving existing versions and constructing new versions. Data model and version model may be related as follows:

- *Version model on top of the data model.* In this case, version management is seen as an ordinary database application. Thus, the version model is represented by a schema whose underlying data model is not aware of versioning. This solution is adopted, e.g., by PCTE [25] and O_2 [26].
- *Version model integrated with the data model.* The data model offers constructs for defining versioned object types. Furthermore, the query language is also aware of versioning. DAMOKLES [27], EXTRA-V [28], and Adele [29] follow this approach.
- *Data model on top of the version model.* In this case, the version model is completely orthogonal to the data model and can thus be combined with any data model (e.g., EER, object-oriented, or simply files). This solution is realized, e.g., in EPOS [15] and ICE [30].

3.2 D2: Selection of a Delta Representation

In the previous section, we have introduced different kinds of deltas, namely, directed, embedded, and selective ones. Each of these has been used successfully, e.g., directed deltas in RCS, embedded deltas in POEM [31], and selective deltas in DSEE [32] and its successor ClearCase [14].

Furthermore, the delta representation does not predetermine the version model implemented on top of it. For example, change-based versioning is implemented in PIE [33] with the help of directed deltas, while EPOS and ICE employ *selective* deltas.

3.3 D3: Definition of Basic Versioning Concepts

The term *basic concepts* refers to the fundamental elements on which a version model is grounded. Higher-level concepts are built on top of these basic concepts. The choice of basic concepts varies significantly from system to system. For example, Aide-de-Camp [17] is based on a single basic concept: the change. In contrast, ClearCase [14] relies on version graphs; change-based versioning may be realized on top of ClearCase [34]. Finally, Voodoo [35] considers revisions and variants as orthogonal concepts, while both are represented in version graphs in ClearCase.

3.4 D4: Relationship between Extensional and Intensional Versioning

Extensional and intensional versioning may be combined as follows:

- *Intensional versioning on top of extensional versioning.* This is realized in the *composition model* [3], where intensionally defined configurations are composed from extensionally versioned components.
- *Extensional versioning on top of intensional versioning.* Alternatively, we may support intensional versioning

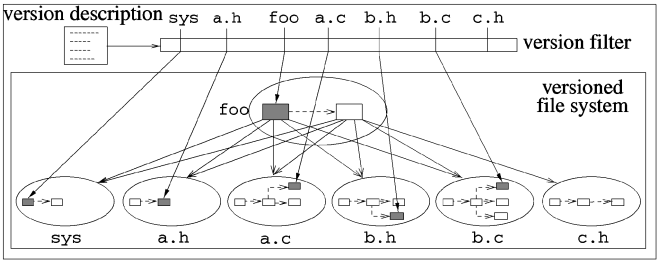

Fig. 3. ClearCase.

at the bottom, allowing for the construction of any conceivable version from a rule-based description at any level of granularity (as realized e.g., in conditional compilation). Extensional versioning may then be achieved by "remembering" (and freezing) designated version selections.

4 Related Work

A big variety of version models has been realized in different SCM systems. Below, we take a look at a few systems—both commercial systems and research prototypes—from the perspective of the requirements and design decisions introduced in Section 3. The systems included in the comparison are selected such that they demonstrate the spectrum of architectures that result from resolving design decisions D1–D4. The reader is referred to [4] for a much more comprehensive survey of SCM systems and the concepts behind them.

Our discussion includes four SCM systems: ClearCase, PCTE,[1] Adele, and ICE. These systems are described individually in Section 4.1. Subsequently, they are compared in Section 4.2 where we also draw some conclusions with respect to a layered architecture for uniform version management.

4.1 Version Models in SCM Systems

4.1.1 ClearCase

ClearCase [36], [14] is a successor of DSEE [37], [38], [32], which was developed in the mid 1980s. It is a commercial SCM system that versions file-based data. Both files and directories are versioned items (total versioning) whose version spaces are represented by version graphs. Selective deltas are used to store versions efficiently. Versions of files and directories are selected from their version graphs by means of version rules (composition model [3]). Applications access the versioned repository through a *virtual file system*. To this end, a version filter is set up which resolves each item access according to a version description (a sequence of rules). ClearCase uses Ramia, a fast network DBMS, as an underlying data storage.

In Fig. 3, a sample software system `foo` is stored in a directory which contains a system model `sys` as well as C source files (headers `.h` and bodies `.c`). The version description defines a version filter which hides the file `c.h` not contained in the selected configuration.

1. PCTE defines a public tool interface for software engineering environments. In the description below, we refer only to the version control part of PCTE.

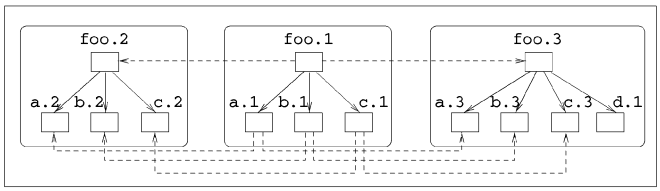

Fig. 4. Versions of complex objects in PCTE.

4.1.2 PCTE

PCTE [39] is a standard for open repositories and provides an interface for implementing software engineering tools. At the heart of PCTE, there is an *object management system*, based on a data model that combines concepts from the EER model and the Unix file system. Each object may have at most one long attribute for which Unix-like file operations are provided. Links between objects are classified into predefined categories, including composition links for representing complex objects.

The PCTE standard contains some basic versioning facilities that were originally developed in the PACT project [25]. The version model is defined on top of the data model. The versioned items are complex and atomic objects (total versioning). A version of a complex object has composition links to the component versions it is composed of. As in ClearCase, versions are arranged in version graphs.

Versioning of *complex objects* is illustrated in Fig. 4. When a new version of a complex object is created, a deep copy of the composition hierarchy underneath the root object is conceptually performed.[2] This ensures that navigation (along the composition links) is unique with respect to versioning. Therefore, versioning remains transparent, and existing applications can still navigate the database without having code added which is responsible for version selection.

4.1.3 Adele

Adele [29], [40] is a commercial SCM system that is based on an object-oriented data model. In contrast to PCTE, the version model is built into the data model. The schema definition language supports the definition of versioned object types. Furthermore, versioning is integrated into the query language that offers version rules for intensional version selection.

In contrast to both ClearCase and PCTE, Adele explicitly distinguishes between revisions and variants. A *versioned object type* stands for a sequence of revisions. The values of common (unversioned) attributes are shared by all revisions; each revision has its own value for a specific (versioned) attribute. Directed deltas are used for efficient storage of long attributes, e.g., text files. Variants are represented by attributes containing alternative values (e.g., the set of implementation variants implementing the same interface).

Fig. 5 gives an example of a versioned object type definition. A single type definition suffices; in contrast, defining the version model on top of the data model usually

2. Note that the copy operation may be implemented smartly, e.g., by delegation (see also [31]).

```
type Interface is Object;
   common
      system :  {unix, hp, vms};
      graphics :  {x11, open_win};
   modifiable
      belong-to :  Conf;
      bug-reports :  set_of Document;
   immutable
      header :  File;
      realization :  versioned Realization;
end;
```

Fig. 5. Definition of a versioned type in Adele.

involves the declaration of pairs of types (for versioned objects and their versions, respectively; see also [28]). The common attributes denote the platform for which the interface object is available. Versioned attributes are further classified into modifiable and immutable ones. For example, the header file is immutable, while the set of bug reports may grow gradually. That is, for a specific revision the header file cannot be updated (an update would create a new revision instead). In contrast, a bug report may be added without enforcing the creation of a new revision (which would be senseless since the bug refers to the already existing revision). Finally, there are multiple realization variants for each interface revision.

4.1.4 ICE

ICE [41], [30] is a research prototype which applies deductive database technology to SCM. The data model is orthogonal to the version model. So far, the ICE implementation supports file-based data that is accessible through a virtual file system interface. Selective deltas are used at the base layer of the ICE architecture. Version rules are introduced on top of the delta storage. Different version models may be defined with the help of version rules. This inverts the approach followed in ClearCase, where version rules are defined on top of version graphs.

The version rules layer is based on *feature logic*, where a feature represents an attribute. For example, `[ws:X11, os:Unix]` denotes the `X11` version of the window system (feature `ws`) and the `Unix` version of the operating system (feature `os`). Feature terms may be constructed from operators such as intersection, union, negation, etc. Intersection fails if the respective feature terms disagree on a common feature. For example, $[\texttt{ws} : \texttt{X11}, \texttt{os} : \texttt{Unix}] \sqcap [\texttt{ws} : \texttt{Windows}, \texttt{os} : \texttt{DOS}] = \bot$.

Since users of a configuration management system are not familiar with feature logic, a *featured file system* was built which hides feature terms from the users [30]. Internally, fragments of text lines are decorated with feature terms denoting their visibilities. As in ClearCase, a version filter is defined for both read and write operations. For reading, internal feature terms are intersected with the version filter and transformed into conditional compilation expressions. For writing, the inverse transformation is applied.

Feature logic is employed as a base mechanism on top of which different version models may be realized (*uniform version model*). In [42], [30], feature logic is used to realize the check-out/check-in model, the composition model, the long transaction model, and the change set model as introduced by Feiler [3].

4.2 Comparison

In the following, we compare the selected systems with respect to the requirements and design decisions introduced in Section 3. To this end, Table 1 classifies the considered systems according to the taxonomy of Fig. 1.[3] The rows of the table also correspond to the requirements given in Fig. 2.[4] Table 2 shows how the design decisions D1–D4 have been resolved in the selected systems. Finally, Fig. 6 displays the resulting layered architectures.

Note that all design alternatives introduced in Section 3 are covered. In particular, ClearCase and Adele offer version rules for selecting versions from extensionally versioned objects. ICE inverts this approach by using version rules to build different structures of the version space. In PCTE, the version model is realized on top of the data model. In Adele, version model and data model are entangled. In ICE, the version model is orthogonal to the data model.

Before proceeding with the comparison, let us emphasize the following points:

- The systems we have compared have different *validation levels*. ClearCase and Adele are commercial SCM systems. Furthermore, commercial PCTE implementations are also available. Thus, these systems have been validated through actual use by real customers. On the other hand, ICE is a research prototype with a much lower validation level.
- Here, we are interested in comparing the *internal architectures* of different systems. These architectures are evaluated against the requirements to a uniform version model. We are here not concerned with the external functionality of these systems (see e.g., [43] for an evaluation of the functionality of commercial SCM systems).
- Among the selected SCM systems, only ICE has been designed with the goal of offering a uniform version model. The authors of the other systems strive for providing useful, widely accepted version models, but they do not go as far as ICE with respect to uniformity and generality.

ClearCase is a well-established SCM system which provides sophisticated support for version management. Moreover, it has proven useful in practice. ClearCase supports revisions and variants, as well as extensional and intensional versioning. While ClearCase itself offers only state-based versioning, change-based versioning has been implemented in a layer above ClearCase [34].

The main restriction is that ClearCase commits itself to version graphs at a low level in the architecture. Version rules and change-based versioning are both located on top of version graphs. Version graphs do not support multi-dimensional variation in an adequate way. Moreover, they introduce constraints which work against the intents of change-based versioning: One of the goals of change-based

3. + indicates that a requirement is met; in the case of a blank entry the requirement is not fulfilled.

4. Requirement R1—generality with both respect to both product space and version space—is not covered in the table because it is not reflected in the taxonomy of Fig. 1.

TABLE 1
Classification of SCM Systems with Regard to Requirements

			ClearCase	PCTE	Adele	ICE
R2	**deltas**	*directed*			+	
		embedded		+		
		selective	+			+
R3	**dimensions of evolution**	*revisions*	+	+	+	+
		variants	+	+	+	+
R4	**version description**	*state-based*	+	+	+	+
		change-based				+
R5	**version space representation**	*version graph*	+	+		+
		grid			+	+
R6	**version set definition**	*extensional*	+	+	+	+
		intensional	+		+	+
R7	**version rules**	*constraints*			+	+
		preferences	+		+	+
		defaults			+	+
R8	**selection granularity**	*product*				+
		total	+	+	+	
		component				
R9	**workspace**	*physical*		+	+	
		virtual	+			+

TABLE 2
Design Decisions in Different SCM Systems

	D1: data model ↔ version model	*D2: delta representation*	*D3: basic versioning concepts*	*D4: extensional ↔ intensional versioning*
ClearCase	integrated	selective deltas (extensional)	selective deltas, version graphs, version rules	intensional on top of extensional versioning
PCTE	version model on top of data model	embedded deltas	embedded deltas, version graphs	(no intensional versioning)
Adele	integrated	directed deltas	directed deltas, revisions, variants	intensional on top of extensional versioning
ICE	data model orthogonal to version model	selective deltas (intensional)	selective deltas, version rules	extensional on top of intensional versioning

versioning is flexible application of changes. In version graphs, however, the deltas are committed to specific points in the history and cannot be moved around freely. Finally, rule-based version selection is confined to a coarse-grained level (no fine-grained selections at the level of text lines).

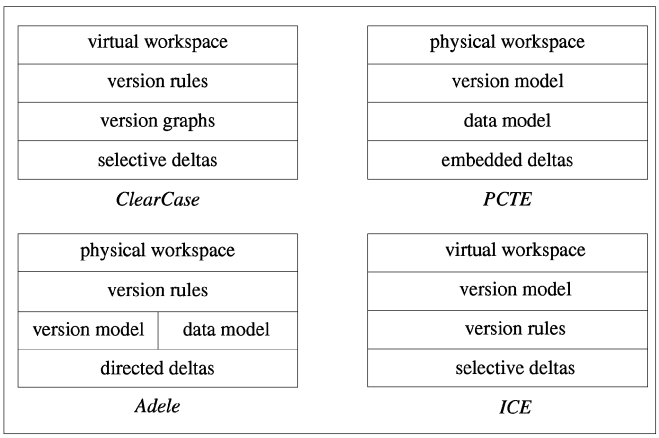

Fig. 6. Layered architectures in different SCM systems.

PCTE has been included mainly to demonstrate that the version model can be defined on top of the data model. However, the version management services are severely limited since they support only extensional versioning with version graphs.

Adele is based on long-term practical experience in SCM. The Adele system offers a sophisticated data and version model, including powerful rules for version selection. Adele supports revisions and variants, extensional and intensional versioning, as well as version rules which can be classified into constraints, preferences, and defaults.

Building the version model into the data model has its advantages (efficient implementation, more elegant schema definition and query language). On the other hand, this solution commits the user to a specific version model. Furthermore, Adele does not address change-based versioning (which could be implemented on top of Adele; see remarks on ClearCase).

ICE provides a uniform version model, since a wide spectrum of version structures can be constructed from feature terms. ICE can cover revisions and variants, extensional and intensional versioning, version graphs and grids, as well as state- and change-based versioning.

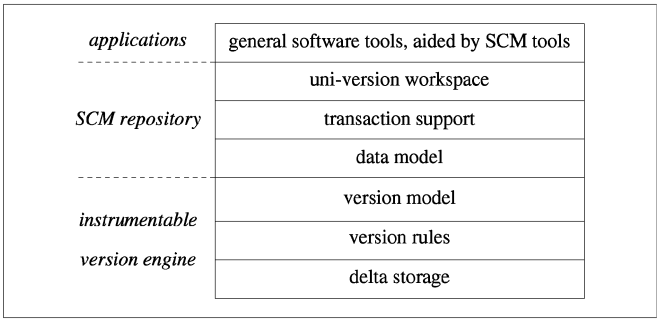

Fig. 7. Layered UVM architecture.

On the other hand, ICE is a research prototype, and practical experiences have been gained only to a limited extent. A major problem with ICE is *efficiency*: to decide the consistency of a feature unification is NP-complete. As reported in [30], naive application of feature unification results in check-in times which grow beyond all limits in the number of versions. In contrast, systems such as RCS, DSEE, and ClearCase, which rely on version graphs, can implement check-out and check-in much more efficiently. However, these systems do not provide for intensional versioning at the fine-grained level. While RCS, SCCS, Adele, and ClearCase employ what we will call *extensional deltas* with simple visibilities, ICE is based on *intensional deltas* with sophisticated visibilities.

5 System Architecture

To satisfy the requirements defined in Section 3, we have developed a layered *system architecture* which is shown Fig. 7. The layers are aggregated into two groups, namely, the instrumentable version engine and the SCM repository, respectively. Below, the layers are described briefly from the bottom to the top. We indicate in which ways requirements R1–R9 are met and how design decisions D1–D4 are resolved.

The *instrumentable version engine* provides basic versioning capabilities without constraining the product space. It is merely assumed that there are versioned items whose item identifiers are provided by the data model layer. The version model is orthogonal to the data model (D1), implying that the version engine can be used with any data model (R1). The *delta storage* offers selective deltas (R2, D2). For each versioned item, there is a set of different *fragments* each of which carries a *visibility* that is interpreted at the *version rules* layer. A visibility is a logical expression over variables called *options*. Each option may be true, false, or unset. The version rules layer provides intensional versioning (R6) based on a 3-valued logic. Extensional versioning is supported by remembering and freezing version selections. In addition to the visibilities, there is a rule base containing constraints, preferences, and defaults (R7).

Selective deltas and version rules constitute the sole base mechanisms of UVM (D3). With respect to the product space, the version engine operates at a low level of abstraction (the contents of fragments are regarded as byte streams). The main difference to other storage systems consists in its deductive capabilities for versioning. For example, the storage manager of EXODUS [7], an extensible database management system, is concerned with byte streams as well. The EXODUS storage manager supports delta storage for versions (embedded deltas), but it does not offer any version rules.

In the *version model* layer, any desired version model may be defined: revisions and variants (R3), state- and change-based versioning (R4), version graphs and grids (R5), and extensional and intensional versioning (R6). This is achieved by defining appropriate version rules. For example, a version graph can be built from implications of the form $\Delta_2 \Rightarrow \Delta_1$, meaning that the delta Δ_2 can be applied only when the preceding delta Δ_1 is applied as well.

Thus, UVM inverts the traditional composition model, where version rules are defined on top of version graphs. In UVM, extensional versioning is supported on top of intensional versioning (D4). This approach is more flexible than the composition model, which commits itself to version graphs from the very beginning. In UVM, any desired version model (not just version graphs) may be defined with the help of version rules. Thus, version rules are a more basic concept than version graphs.

Above the version engine, the *SCM repository* provides layers for the data model, transactions, and workspaces. The *data model layer* is responsible for offering a data model (e.g., object-oriented, EER, or simply files) to the upper layers that hides versioning. That is, clients of this layer may perform queries and updates as if the database were unversioned.

Transaction support provides cooperative, long-lasting transactions. Each software development or maintenance activity is performed in the context of some transaction. Each transaction is confined to a subset of both the product space and the version space. In particular, a transaction may operate at any level of granularity with respect to the product space (R8).

Finally, the *workspace layer* is concerned with the interface to software tools (editors, compilers, debuggers, CASE tools, etc.). The workspace layer provides a universion workspace (with the help of a virtual file system or simply check-out operations), i.e., software tools are shielded from versioning (R9). Each workspace is associated to some transaction.

The layer structure described above is idealized in several ways. For example, transaction handling is actually distributed over multiple layers.[5] Moreover, the data model layer does not depend on a specific version model; rather, it relies only on general functions provided by the instrumentable version engine. In spite of these considerations, we stick to the somewhat simplified layered architecture in order to convey our key points clearly.

The UVM architecture is not merely a proposal. To a great extent, it was actually implemented in 1989–1995 in *EPOS*, a research prototype of a software engineering environment supporting both SCM and process management [44]. Therefore, we will refer to the EPOS implementation at various points in the following sections. Our

5. For the sake of clarity, transaction handling will be discussed at one place, namely, in Section 7.2.

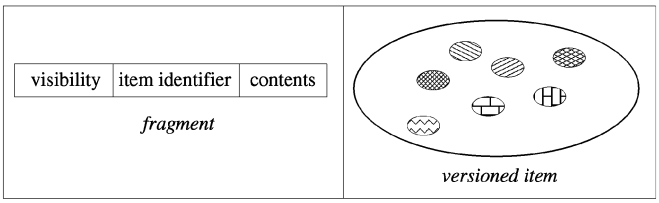

Fig. 8. Delta storage.

presentation will be primarily based on the revised implementation of the EPOS configuration management system (ECM) as described in [15], [45] (the initial implementation is described in [46], [16], [47]).

UVM is similar to ICE (see Section 4), which was developed much later than Munch's ECM implementation. Both share the same fundamental design decisions. However, UVM still differs from ICE with respect to the following points:

- UVM is based on a different kind of logic (propositional 3-valued logic instead of feature logic). In ECM, visibilities are implemented much more efficiently than in ICE (see Section 8).
- Transaction support lies at the borderline between configuration management and process management and has not been considered seriously in ICE so far.[6]
- The data model layer has not been addressed thoroughly in ICE. So far, ICE assumes (text) files and directories only.

6 Instrumentable Version Engine

6.1 Delta Storage

The delta storage constitutes the most basic layer of the architecture. UVM is based on *selective intensional deltas*, where each fragment is governed by a visibility expression. Thus, intensional versioning is directly supported. In addition, any desired version model may be realized with such deltas, as to be demonstrated in Section 6.3.

Please note that embedded deltas provide for sharing, but do not offer intensional versioning. Moreover, directed deltas are useful for implementing version graphs (as e.g., in RCS) and change-based versioning (as e.g., in PIE). However, they are less suited for variants, which can be implemented naturally with selective deltas (being ultimately derived from conditional compilation).

Thus, the versioned database consists of a collection of *fragments*. Each fragment has the following parts (left-hand side of Fig. 8):

- A *visibility* (version part of the fragment) controls the region in the version space where the fragment is legal, i.e., the set of versions in which it is included and can be selected. The visibility may be considered the version identifier (VID) of the fragment, though it (usually) covers a large set of potential versions.
- An *item identifier* indicates the corresponding versioned object (or more generally: versioned item, see Section 2). The item identifier (resembling an object identifier, OID) is provided by the product/data model layer.
- A *contents*—raw data—corresponds to the product part of the fragment (e.g., an attribute-value pair or a text line).

Note: As we shall see in Section 7.2, all fragments also have an associated *transaction identifier (TID)*, which may be regarded as an appended part on the visibility.

The large ellipsis on the right-hand side of Fig. 8 illustrates the version space of a versioned item, i.e., the space of all potential versions. Each of the small ellipses corresponds to one fragment. The area covered by such an ellipsis indicates the set of all versions to which the fragment belongs. These sets are always mutually disjoint.[7] However, their union need not cover the whole version space because the item may not belong to all potential versions (i.e., the uncovered region represents all versions in which the respective item is not included at all). A specific version corresponds to a single point in the version space.

The delta storage is still *dumb* in that it leaves the interpretation of information to the upper layers. In particular, it does not know which items are versioned, how items are identified, and how visibilities and contents are interpreted. The delta storage provides low-level functions for storing fragments and for retrieving them both sequentially (e.g., when a text file is composed from a sequence of matching lines) and via an index (e.g., when an object reference is resolved). In EPOS, the delta storage was realized by index-sequential files, as it is done in most DBMSs.

6.2 Fragment-Level Version Rules

The next layer adds *deductive capabilities* to the delta storage. It evaluates and writes the visibilities of fragments, but it leaves all other parts untouched (item identifier and contents). Thus, it considers only the version space without making any assumptions with respect to the product space, apart from the existence of unique item identifiers (OIDs). In fact, it is possible to implement the layer in such a way that it is unaware of the data model.

6.2.1 Formal Foundation

Propositional, 3-valued logic (see below) is used to formalize version rules. Thus, a version rule is a logical expression over global variables which are called *options* o_i. Each option defines a property which is either absent from or included in a version. The version space can therefore be represented by an n-dimensional grid, where each dimension corresponds to some option. A point in the version space is denoted by a complete *binding*, with each option being bound to either true or false. A binding is called incomplete if one or more options remain unset. An incomplete binding denotes a region of the version space

6. In [30], it is sketched how long transactions in ICE can be simulated by means of transaction features.

7. This is ensured by the version rule layer, see Section 6.2.

(an m-dimensional cube, where m is the number of unset options).

A *logical expression* is constructed from constants, variables (options), and operators such as $\neg$, $\wedge$, $\vee$, $\otimes$, and $\Rightarrow$. An expression is evaluated under some *binding* b, where

$$b : [1..n] \Rightarrow \{true, false, unset\}, \tag{1}$$

b_i denotes the binding of option o_i, and $b_i = unset$ means that o_i is unset. b is *complete* if no option is unset.

The logic is *3-valued* rather than 2-valued. Under some binding b, an expression e evaluates to true, false, or unset. For example, $true \wedge unset = unset$, $true \vee unset = true$, etc.

Rather than by a function, a binding may also be represented by a conjunction:

$$b = b_1 \wedge \ldots \wedge b_n,\ b_i \in \{o_i, \neg o_i, true\}\ (i \in \{1, \ldots, n\}). \tag{2}$$

As shown above, an unset option o_i in a binding is represented by $b_i = true$, so that it can be eliminated from the conjunction. Below, we stick to this definition because it is more convenient to use in the definitions to follow.

Options will be identified by their name. Examples could be:

- `GUI`: Set to true for versions of programs which supply a graphical user interface. Can also be used for the documentation.
- `Linux`: Set to true for program versions specially adapted for being run under the Linux operating system.
- `SpeedOpt`: Indicates whether or not the version is optimized for speed.

6.2.2 Visibilities and Version Filters

Visibilities of fragments need to be evaluated when reading from the database, and they have to be updated when writing to the database. Reads and writes go through *version filters*.

A *read filter*, also called a *choice, c*, determines a single version to be presented to the user (or an application). Thus, a choice is given by a complete option binding—at least for the accessed product subspace and its fragments. A *write filter*, also called an *ambition, a*, defines a set of versions which will be affected by the change. An ambition is specified by a (potentially) incomplete option binding.

The choice must lie within the ambition, i.e., choice and ambition must agree in all bound options:

$$\begin{aligned}(a = a_1 \wedge \ldots \wedge a_n) \wedge (c = c_1 \wedge \ldots \wedge c_n)\\ \wedge\, a_i \neq true \Rightarrow\\ a_i = c_i (i \in \{1, \ldots, n\}).\end{aligned} \tag{3}$$

This means that the choice implies the ambition:

$$c \Rightarrow a \qquad \text{or informally : } c \in a. \tag{4}$$

If all options remain unset, the ambition evaluates to true, i.e., all versions are affected by the change. Inversely, if no option is unset, ambition and choice coincide so that only a single version (denoted by $a = c$) is changed.

The distinction between read and write filter, which is also performed in the *multiversion editors* P-Edit [48] and MVPE [49], has the following advantages:

- We may edit multiple versions simultaneously without being bothered with visibilities. In any case, visibilities are hidden from the users.
- Reading is simplified because it does not involve general unification of logical expressions. Rather, we just have to evaluate visibilities under complete option bindings.

Note that we could have allowed for more general ambitions (general logical expressions). However, increased generality would result in increased complexity (see below).

A fragment f with visibility v passes the read filter c (the choice), iff the visibility v is implied by the choice, i.e., it evaluates to true given the option bindings of the choice:

$$c \Rightarrow v. \tag{5}$$

We must ensure that at most one fragment per item is visible under a given choice, or use a multiversion editor which is capable of displaying several versions simultaneously. Let v_i, v_j denote visibilities of two fragments (or versions) of the same item. Then, the following condition must hold to express that the fragment visibilities are *disjoint*:

$$\neg(v_i \wedge v_j). \tag{6}$$

Alternatively, this may be rephrased as follows: Let S_i, S_j denote the version sets in which the above mentioned fragments are visible. Each S_i, S_j corresponds to a set of points of the version space, where each point is characterized by a complete option binding. S_i, S_j must therefore be disjoint:

$$S_i \cap S_j = \emptyset. \tag{7}$$

This has already been illustrated earlier on the righthand side of Fig. 8, where the small ellipses represent different contents and the extent of each fragment—the set of versions in which it is visible—is represented by the region which it covers.

With respect to write operations, we distinguish between deletions, insertions, and updates. An *inserted* fragment f gets its visibility entirely from the ambition a:

$$v_{new} = a \quad (\text{for insertion}). \tag{8}$$

In the case of a *deletion*, the visibilities of all the "same" fragments—i.e., all fragments with the same item identifier—are constrained such that they would not be selected under the current ambition:

$$v_{new} = v_{old} \wedge \neg a \quad (\text{for deletion}). \tag{9}$$

Finally, an *update*—as well as a move operation—is reduced to a combined deletion and insertion operation, as described above. There is no additional "adjusted" fragment with visibility $v_{new} = v_{old} \wedge a$ since this would have violated the nonoverlap of versions.

And for write operations that do not touch the fragment, the visibility remains unchanged: $v_{new} = v_{old}$.

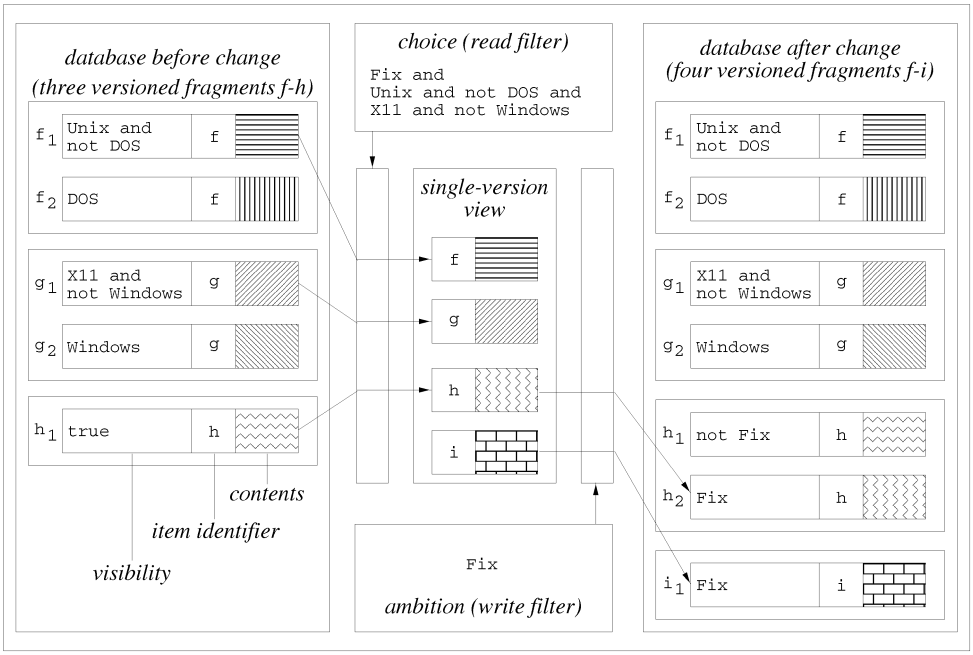

Fig. 9. Example of evolution of visibilities.

The write operations (8) and (9) maintain the invariant indicated above, that visibilities of different versions of the same fragment are mutually disjoint.

Example 1 (Ambition, choice, visibilities). Below, we illustrate the concepts introduced above by a simple example which is adapted from [11] (Fig. 9). The sample database varies with respect to the operating system (`Unix,DOS`) and the window system (`X11,Windows`). Each variant is represented by a corresponding option. On the lefthand side, the contents of the database before an update is shown. There are three versioned fragments, identified by `f`, `g`, and `h`. For convenience, the fragment versions are identified similarly (e.g., f_1 and f_2), but these identifiers are not present in the implementation. Each fragment version is illustrated in the same way as in Fig. 8, i.e., with the visibility at the leftmost slot.

Please note, that we assume that changes were performed in a certain order. For example, the `Unix` variant was introduced before the `DOS` variant. Thus, the visibility of fragment version f_1 was narrowed by $\neg$ `DOS`, when the DOS change was carried out.

In order to carry out a change that extends to all versions of operating system and window system, we define an option `Fix` which simultaneously serves as the ambition. Here, we can perform the change in any version; we just set up the choice so that we can carry it out in our preferred environment. To perform the changes under `Unix` and `X11`, we set the corresponding options in the ambition to true and all other options to false.[8] In our single-version view (in the middle), we modify `h` and add another fragment `i`.

The database in Fig. 9 is changed according to the rules given earlier: 1) At the top right in the figure, the fragments `f` and `g` remain untouched and, thus, unchanged. 2) At the middle right, the fragment `h` is split in two—an old version (i.e., "deleted" in this context) with visibility $\neg$`Fix`, while a new ("updated") version is assigned the visibility `Fix` being the ambition. 3) At the bottom right, the same `Fix` visibility is also attached to the inserted (previously unknown) fragment `i`.

8. This is clearly awkward. How the user can be assisted in setting up the default option bindings, will be discussed later (Section 8.2).

6.2.3 Complexity

Complexity has to be dealt with at different levels. At the *user level*, we are facing the problem of consistent (legal) selections from a large version space. This problem will be discussed later in Section 8.2. At the *system level*, we are concerned with efficiency.

At the end of Section 5, we have mentioned the efficiency problems encountered in ICE. These problems are due to the inefficient evaluation of logical expressions (*intensional deltas*). Classical systems such as SCCS, RCS, or Adele are not confronted with complex expression evaluation because they use embedded or directed deltas (*extensional deltas*). However, such SCM systems can only construct versions of components that were explicitly checked in before; while explicit merging must be used to create "new" version combinations. In Section 8.1, we will demonstrate that intensional deltas also can be implemented efficiently.

6.2.4 Metadatabase: Option Definitions and Rule Base

A UVM repository is composed of two parts: a fully versioned *contents database* containing a collection of fragments and a "semiversioned" *metadatabase* containing option definitions and global version rules. Both databases attach permanent transaction identifiers (TIDs) to all stored information items, primarily to facilitate traceability between change jobs and database changes. In addition, the TIDs serve as *timestamps* to realize a revision chain for "metalevel" evolution, see below and Section 7.2 on transactions.

The metadatabase contains two parts:

- *Option definitions*: These introduce named options which may be referred to in fragment visibilities, ambitions and choices (fragment-level version rules), and in global version rules, see next point.

  We can also define high-level version identifiers, such as `v2.1` or `Unix-v5.0`, usually representing (parts of) a choice or an ambition.

  Both options and high-level version identifiers can be used in the version rules below. See also Section 6.3 and Section 8.2.
- *Global version rules*, or just version rules: Without further constraints, n options result in 2^n potential choices (3^n ambitions), of which only a few may be expected to be consistent (legal) or indeed relevant. To manage the complexity of *legal* version selection, we therefore need a *rule base* of global version rules, which are not attached to specific fragments. They complement the choices and ambitions used to set up version filters for reading and writing, respectively. Further, the user will typically specify a high-level and partially bound external version description evd. This is then extended into a low-level and further bound, internal version description ivd (ambition or choice) with the help of global version rules—see Section 8.2 on the Versioning Assistant.

There are at least five types of global rules, cf. Section 2:[9]

- *General constraints (usually dependencies)*: Dependencies between options will typically mean that an option cannot be bound explicitly to either true or false in an ambition or choice, unless another option is bound to true. An example can be options for deciding layout of a graphical user interface, which make no sense unless GUI is selected in the first place.

  Note, that an option dependency is not merely stating that "this feature requires that feature," but rather that "this feature does not even make sense to talk about, unless we have that other feature."
- *Option groups (special constraints for variants)*: Option groups are used to express mutual exclusion among variants and resemble enumerated datatypes, see Section 6.3. They specify that at most one of a group of options may be bound to true. Typical examples include options for different hardware, OS, or natural languages in user dialogs.
- *Validity checks (special constraints to support baselines)*: Validities express that certain versions are "frozen," i.e., baselines that cannot be updated without adding more options in the associated ambition (see Section 6.3).
- *Preferences (weak dependencies)*: The rule base may contain user- or project-specific preferences, e.g., to fill in usual or trivial choices. This means that we can have high-level version identifiers (such as `v2.1` above) to express releases, e.g., covering a bundle of error fixes. These version identifiers (evds) are then translated to a group of more detailed option settings (ivds).
- *Defaults*: The rule base may also contain default selections, that we apply to resolve otherwise ambiguous choices.

For more details, see [50] or [12].

The metadatabase will itself be an *evolving entity*. First, new options need to be defined to represent logical changes, to support new platforms, etc. Second, version rules may be added to exclude combinations due to bad experiences. Conversely, they may also be removed, e.g., after corrective work has been performed in a set of merge-and-test transactions.

Then, how should evolution of the metadatabase be supported? In principle, we could handle the metadatabase in the same way as the contents database, adding a metalevel with its own metaoptions and metarules. However, it is hard to imagine who could manage the incurred conceptual complexity of multilevel version selection. Rather, we believe that simple, TID-based time-stamping to support *revisions* of the metadatabase will suffice. This allows for regeneration of old versions/configurations by using the corresponding old option definitions and rule base. "Newer" options in visibilities will consequently be set to false. A simple example is given below.

Example 2 (Evolution of the metadatabase). Let us assume a database which has evolved as follows:

1. Initially, the contents database is unversioned (all visibilities set to true; no options).
2. We introduce an option o_1 in the metadatabase and perform a change (in the contents database) under the ambition $a = o_1$. New fragments get the visibility $v = o_1$.
3. We introduce another option o_2 and perform a change under the ambition $a = o_1 \wedge o_2$. Updated fragments previously decorated with $v = o_1$ are split (visibilities $v' = o_1 \wedge \neg o_2$ and $v'' = o_2$, respectively).
4. We add a constraint $o_1 \Rightarrow o_2$ to ensure that the enhancement o_2 is always selected when o_1 is selected.

Now, we want to fix a bug in the version which was only denoted by o_1 after Step 2. The current version of the metadatabase prohibits this because it enforces the selection of o_2 as well (the enhancement o_2 may not be appropriate for the customer, e.g., because his license agreement includes only bug fixes).

To perform the change, we conceptually roll back the metadatabase to the state after Step 2. In addition, all options introduced afterwards are bound to false, so that old fragment versions are correctly selected. For example, in case of a fragment split as described in Step 3, the old version with visibility v' is selected rather than the new one with visibility v''.

6.3 Version Model

On top of the basic version rule layer, any desired version model can be defined. Note that, the version model is still

9. Earlier, we introduced three types of rules: constraints, preferences, and defaults. The classification below is more fine-grained since it distinguishes between different types of constraints.

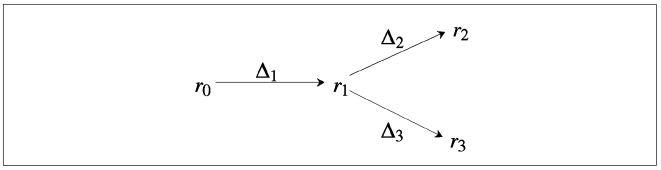

Fig. 10. Version graph and delta combinations.

concerned with the version space only; the product space will be handled in the data/product model layer (Section 7.1). So far, we have only assumed unique item identifiers, the rest of the data/product model has been left unspecified.

6.3.1 Version Graphs and Grids, Revisions

The option space can be represented as an n-dimensional *grid*. By adding constraints, we may model *version graphs* as well. As mentioned earlier, a revision chain can be defined by means of implications. Therefore, we introduce *delta options* (NB: not the delta changes themselves), named Δ_i, and implication rules from each delta to its predecessor in the version graph:

$$\Delta_i \Rightarrow \Delta_{i-1}. \tag{10}$$

A *revision* r_i can be defined by listing all included and excluded options, respectively the head and tail of a (sorted) conjunction:

$$r_i = \Delta_1 \wedge \ldots \wedge \Delta_i \wedge \neg\Delta_{i+1} \wedge \ldots \wedge \neg\Delta_n. \tag{11}$$

Example 3 (Version graph). Consider the version graph of Fig. 10. Note that the delta symbol here represents a change *option*, not a physical change. Delta sequences are represented by the following equations:

$$\Delta_2 \Rightarrow \Delta_1,\ \Delta_3 \Rightarrow \Delta_1. \tag{12}$$

The revisions here may be described as follows, where an increasing number of deltas are included:

$$r_0 = \neg\Delta_1 \wedge \neg\Delta_2 \wedge \neg\Delta_3,\ \ r_1 = \Delta_1 \wedge \neg\Delta_2 \wedge \neg\Delta_3,\ \ r_2 = \ldots. \tag{13}$$

6.3.2 Variants as Option Groups

We have already shown above how we can model (linear chains of) revisions. *Variants* can be represented by *option groups*. An option group consists of a set of mutually exclusive options. In a global version rule, the $\otimes$ (XOR) operator is thus used to indicate that at most one option of the group may be selected (be bound to true) simultaneously.

Example 4 (Variants). The branch in the version graph of Fig. 10 can be represented as follows:

$$\Delta_2 \otimes \Delta_3. \tag{14}$$

This assumes that Δ_2 and Δ_3 cannot be meaningfully combined (in case they can be combined, the above constraint has of course to be omitted, typically when merging has been completed).

Similarly, we may use option groups to express variant attributes ranging over an enumeration type. In Example 1, we have introduced a contents database varying with respect to the operating system (and the window system). Operating system variants may be represented by the following option group `OS`:

$$\texttt{OS} = \texttt{Unix} \otimes \texttt{DOS} \otimes \texttt{VMS}. \tag{15}$$

6.3.3 State- and Change-Based Versioning

It should be obvious from the above that both *state-* and *change-based versioning* can be supported in UVM. Really, there is not a sharp distinction. State- and change-based approaches differ with respect to the constraints they impose on change combinations. Taken to the extreme, we may say that change-based versioning enforces no constraints at all, while deltas are tied to fixed locations in version graphs in the case of state-based versioning—i.e., the version graph implicitly defines the constraints. However, in reality, we will rarely find these extremes. Change-based versioning gets lost in the version space without constraints. Moreover, even in classical state-based systems there are facilities to merge changes—e.g., by text-based merge tools—or to apply them at different locations in the version graph (the latter is supported, e.g., in SCCS). In addition, the composition model mentioned earlier allows for combining changes by rule-based selection of component versions. For example, if two changes were performed which affected two different files, these changes may be combined by selecting the new file versions in both cases.

Example 5 (Merging). Let us assume that we want to merge the changes Δ_2 and Δ_3 in the version graph of Fig. 10. To this end, we perform a change under the following ambition (and choice):

$$a = c = r_4 = \Delta_1 \wedge \Delta_2 \wedge \Delta_3. \tag{16}$$

Initially, we obtain a version r_4 which is merged automatically. Often, the result of the merge has to be adjusted manually. Writing the consolidated changes to the database establishes a merged version that we can trust later on.

For example, consider the following (cut-out of a) versioned database, where f,g and f_i, g_i denote versioned fragments (version groups) and individual fragment versions, respectively (the fragment contents are not shown):

OID	"Fragment-ID"	Visibility
f	f_1	$\Delta_1 \wedge \Delta_2$
	f_2	$\Delta_1 \wedge \neg\Delta_2$
g	g_1	$\Delta_1 \wedge \Delta_3$
	g_2	$\Delta_1 \wedge \neg\Delta_3$

Under the ambition a defined above, the fragments f_1 and g_1 are selected. The user tests out this "miniconfiguration" and detects an error, requiring a change to both fragments. After the change described by Δ_3, the database contains new fragment versions f_3 and g_3 as part of the adjusted revision r_4, and f_1 and g_1 have got changed visibilities:

OID	"Fragment-ID"	Visibility
f	f_1	$\Delta_1 \wedge \Delta_2 \wedge \neg\Delta_3$
	f_2	$\Delta_1 \wedge \neg\Delta_2$
	f_3	$\Delta_1 \wedge \Delta_2 \wedge \Delta_3$
g	g_1	$\Delta_1 \wedge \neg\Delta_2 \wedge \Delta_3$
	g_2	$\Delta_1 \wedge \neg\Delta_3$
	g_3	$\Delta_1 \wedge \Delta_2 \wedge \Delta_3$

The above example demonstrates that merging is a 2-stage process:

1. *Automatic, raw merge.* The user supplies a version description (choice), where the options for the changes to be merged are set to true. The version engine selects those fragment versions, that correspond to the actual choice.
2. *Manual adjustment of the merge.* The user must now check whether the result of the first step already delivers a meaningful version satisfying his requirements. For example, this may involve compiling and testing the merged program version. In case of errors, the user has to fix the result of the automatic merge manually. These updates will be given visibilities specific to the option combination in the ambition. Thus, we distinguish between individual changes specific to an option, and "merge changes" specific to their combination. For instance, SCCS cannot distinguish the extra merge changes.

Experiences from using commercial 3-way merge tools indicate that conflicting or meaningless merges happen quite rarely [14]. Thus, the merge process described above should be practically feasible, also by nonUVM SCM tools. The advantage of UVM over many commercial approaches is that illegal combinations can be detected beforehand by constraints, or be assigned a special `Raw` state (see below).

6.3.4 Extensional and Intensional Versioning, Validities

In UVM, *intensional versioning* constitutes the base on top of which *extensional versioning* can be realized. First of all, we introduce external *version identifiers* to identify versions that can be conveniently reconstructed later. Such a version identifier is simply a symbolic name which stands for a choice or a group of choices, such as `v2.1` or `Unix-v5.0`. So, instead of repeating long, extensional lists of option bindings, the user may refer to one compact, intensional version identifier. Again, see Section 8.2 on the Versioning Assistant.

Moreover, the user should be able to select a version which has some predefined property (e.g., `Stable`), or to use such properties as a guide to completing a version choice. This can be implemented by so-called *validities*,[10] where a validity is a disjunction of *validity terms*:

$$V = V_1 \vee V_2 \ldots \vee V_l. \tag{17}$$

Each validity term is a binding, i.e., a conjunction according to (2). In the case of complete option bindings, each V_i denotes a single version.

10. The name "validity" has been retained for historic reasons.

Thus, a validity simply enumerates a set of versions, i.e., extensional versioning. In addition, these versions share some desirable property, usually characterized by a value from a (predefined) set of version states, such as `Raw`, `Compiled`, `Inspected`, `Tested`, ... , `Stable`, `Released`, ... For instance, all "new" (previously not selected) versions will be assigned the `Raw` state.

In UVM, several consecutive transactions may use the same ambition, e.g., to bundle changes or to split update work over several products/subsystems. Later on, we need to define stable product baselines—for internal or external releases—to protect desired versions from further changes.

For this purpose, a *stable validity*, V_s, is introduced. Stable validities cover those validities that have `Stable` or `Released` states. Furthermore, we have added the constraint (global version rule) that a new ambition a must not overlap with a stable validity V_s. The possibly adjusted ambition a' must therefore have the following property:

$$a' \Rightarrow a \wedge \neg V_s. \tag{18}$$

The stable validity, unlike other validities, is therefore used to restrict a new ambition. That is, by adding further option bindings (using new options?), the ambition will eventually fall *outside* the stable validity. If this is impossible, the ambition is illegal.

Example 6 (Validities). Before the merge performed in Example 5, let us assume a stable validity V_s containing the revisions $r_0, \ldots, r_3$, i.e., each of these revisions are immutable:

$$V_s = r_0 \vee r_1 \vee r_2 \vee r_3. \tag{19}$$

Now, the change is performed under the ambition $a = r_4$ (see (16)). There is no overlap with V_s, e.g.,

$$r_4 \wedge r_2 = \Delta_1 \wedge \Delta_2 \wedge \Delta_3 \wedge \Delta_1 \wedge \Delta_2 \wedge \neg\Delta_3 = false. \tag{20}$$

Therefore, the ambition need not be constrained, i.e., $a \wedge \neg V_s = a$. After the change, we may extend the stable validity such that r_4 is included:

$$V_s = r_0 \vee r_1 \vee r_2 \vee r_3 \vee r_4. \tag{21}$$

7 SCM Repository

So far, we have discussed only the instrumentable version engine located at the bottom of the UVM architecture. The following subsections are devoted to the *SCM repository*, whose three layers have been sketched only briefly so far. These layers are from bottom to top: product/data model layer, transaction support, and workspace support. They will be presented in that order.

Lastly, we will sketch some SCM toolkit functionality for the application layer above.

7.1 Product/Data Model

The *product/data model layer* adds a data model, which has been missing so far. At the external interface of this layer, the database basically behaves like a conventional, unversioned database (consisting of objects identified by OIDs

and relationships, both decorated with nonversioned attributes). However, versioning is already taken care of *below* this layer rather than on top of it. Essentially, a version is selected first, then operation proceeds as in an unversioned database.

Thus, there is a clear separation between version selection and queries against a selected database version. In UVM, we distinguish between *version space queries* and *product space queries*, respectively. First, a version is selected (more precisely, an ambition and a choice); then, the user may issue product space queries without being bothered with version selection. For example, in the case of a relational data model standard SQL queries may be used for this purpose.

In contrast, queries need to consider both the product space and the version space, if version model and data model are entangled. For example, if a software system is configured by a transitive closure over use relationships, the respective query has to handle alternating product and version space selections. For some already selected module version, all outgoing use relationships have to be traversed; subsequently, a version is selected for each used module. Alternating product and version space selections make queries more complicated. In contrast, the version selection is factored out beforehand in UVM.

Since the data model is introduced only on top of the version model, the version model is *independent* of the data model. However, this statement also holds to a large extent in the opposite direction. The data model layer knows that versioning takes place, and it also knows about the existence of read and write filters (choices and ambitions, respectively). However, these filters are just passed down to the version engine (going straight through); the meaning of them is not understood by the data model layer.

The data model layer turns fragments into typed data. In particular, it determines the *granularity* at which versioning is performed, e.g., full objects, single attributes, or lines in textual attributes. It also provides more high-level product identifiers, usually *object identifiers*. That is, it defines the "sameness" criterion missing from the lower layers, where only "raw" *item identifiers* are used. Thus, the product/data model layer "knows" about item identifiers on version fragments. Inversely, the lower layers provide and use such identifiers for their own purpose, but also "know" that they will be interpreted and utilized by the upper layers.

There are several alternatives of mapping data model elements onto fragments. In determining the granularity of fragments, we have to balance efficiency against combinability: The smaller the fragments are, the more overhead has to be paid for storing and evaluating visibilities. On the other hand, small fragments can be combined much more flexibly than large ones. For example, the composition model introduced in [3]—realized in ClearCase [14]—treats whole text files as the fragments from which a configuration is built. In this way, changes at the level of text lines cannot be combined.

Before 1990, Adele-1 supported a predefined object model, with a special interface-body and subsystem structure. In Adele-2, a more general object model was chosen to accommodate a wider range of software and design objects.

In EPOS, a binary EER model was chosen which offers single inheritance on entity and relationship types. Entities, relationships, and attributes are mapped onto fragments in the following way:

- An *entity* is represented by one fragment for each type up to the root of the type hierarchy, as in many object-oriented DBMSes. In this case, the item identifier is composed of both the object and the type identifier.
- Short *attributes* are part of the fragment for the type in which the attribute is declared. However, this approach is too coarse-grained for long attributes (here, restricted to sequential files of variably-sized items), so these are treated in a special way, as instances of a special `longfield` type. Each text file is represented by a sequential database file, where each record (fragment) consists of a visibility and a text line. If a text file is written, a `Diff` is performed. The visibility of deleted lines is constrained, and inserted lines are made visible under the current ambition. Since lines do not have a unique identifier, it is not possible to represent multiple versions of the "same" line. Thus, lines cannot be updated—rather, an update is handled by a combined deletion and insertion.

  Nonversioned or read-only attributes (such as names, creators, and logs) may be put in a separate database to save space, but must then be merged with the versioned attributes upon access. Such a separation is anyhow normal for subtyped objects in OODBMSs.
- Finally, each *relationship* is mapped onto one fragment. Again, there cannot be multiple versions of the "same" fragment because relationships do not have unique identifiers.[11]

The *product model* for an application is then defined in terms of the data model by means of a database schema. Notice that the schema may be versioned as well. As a result of version selection, a specific *schema version* [51], [52] is obtained, consisting of a set of raw, universion fragments that represent (parts of) the type definitions.

The data model layer must then ensure *consistency* with respect to both the schema definition and the instances viewed under the schema version (see [15] for details):

- The *schema* must comply with the constraints of the data model. For example, a subtype may exist only if its supertype exists as well, a relationship type may exist only if source and target types exist, etc.
- Similarly, the *instances* must comply with the selected schema version. In particular, the schema version determines the extent of visible information. For example, an entity may be viewed as an instance of the visible types only, a relationship is visible only if source and target as well as its type are visible, etc.

11. It is beyond the scope of this paper to discuss the technical reasons for this, but the combination of subtyping, several instances of N:M relationship types, and versioning is complex.

7.2 Transaction Support

Transactions are used to coordinate cooperative work against a versioned database. Work is triggered by a *change request* which is then implemented by one or more transactions. A transaction defines the context in which developers perform their work. The work context determines which versions of which objects are read or written.

We have introduced a single transaction layer in the (simplified) UVM architecture of Fig. 7. However, transactions are actively planned, followed-up, and terminated also by the workspace and application layers on top of the transaction layer. Below, we mainly concentrate on the design issues concerning the layered UVM architecture. For further work on cooperative transactions carried out in EPOS, see [44], [53], [45].

At the level of delta storage, each fragment is decorated with a unique *transaction identifier* (TID). Whenever a fragment is written to the database, it is tagged with the TID of the transaction which performs the change. Thus, all changes performed by transactions may be traced later on. By means of TIDs, we may provide the functionality of a *temporal database* [54]. Also note, that transaction logs in classic databases resemble "deltas" in our terminology.

Such a TID becomes a high-level "visibility" of a fragment, selecting the accessible subdatabase for a given transaction. (In case of cooperating transactions, special propagation mechanisms and policies must be used.) Such a transaction visibility can also be achieved by so-called *transaction options* [55], effectively combining visibilities and transaction identifiers. So, for each transaction t, an extra, internal transaction option o_t is assigned automatically. During a transaction, an internal ambition a_{i_t} is obtained by constraining the user-specified external ambition a_{e_t} with o_t, i.e., $a_{i_t} = a_{e_t} \wedge o_t$. After commit, o_t is automatically set to true in subsequent version selections.

On the next layer (version rules), both a read filter (choice) and a write filter (ambition) are associated to a transaction. In this way, the cut-out of the *version space* is defined on which a transaction is operating. The ambition, which includes the transaction option mentioned above, defines the visibilities attached to fragments being written to the database.

In addition to the changes to the *contents database* (a collection of fragments), all changes to the *metadatabase* have to be traced. This can be achieved by implementing the metadatabase itself as a collection of fragments, each of which is tagged with the TID of the transaction by which it was written. In this way, revision chains of the metadatabase may be realized (with TIDs effectively functioning as time stamps).

Above we have described the basic functions for handling transactions that are provided by lower layers of the UVM architecture. Below, we describe what is added by the *transaction support* layer located on top of the product/data model layer.

So far, transactions are concerned only with the version space. However, the *product space* needs to be considered as well. In order to detect and resolve conflicts, transactions must know the objects on which the users are working. Only then is it possible to preset locks on these objects (pessimistic concurrency control). Alternatively, later, upon commit, we can detect direct or indirect update conflicts by time stamps (optimistic concurrency control, which in general involves interactive conflict resolution through merging). Such time stamps can be represented by TIDs.

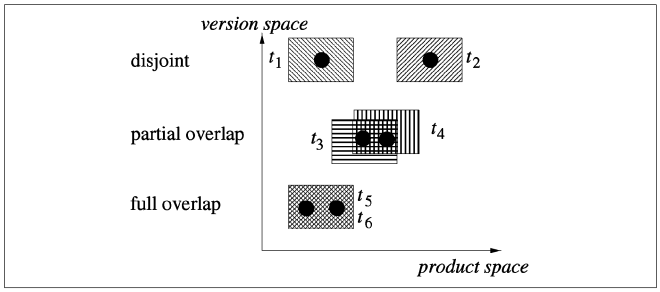

Fig. 11. Scopes of transactions.

Thus, each transaction operates on a *subdatabase* of the overall versioned database. The version space is defined by an ambition for writing and a choice for reading. The product space is typically defined by a transitive closure from some root object (a main program) over a set of relationship types (`Part-Of`, `Depends-On`, etc.).

Fig. 11 illustrates under which conditions two transactions come into potential *conflict*. Shaded regions and black points represent ambitions and choices, respectively. Transactions conflict only when they overlap with respect to *both* the product space and the version space. Transactions t_1 and t_2 affect the same versions, but are disjoint with respect to the product space, so no conflict. Conversely, t_1 and t_5 (or t_6) refer to the same objects, but operate on disjoint version sets, so again no conflict. However, t_3 and t_4 may conflict because they partly overlap in both spaces. Moreover, their choices lie within the common parts of their ambitions; thus, t_3 actually sees what t_4 modifies and vice versa. Lastly, t_5 and t_6 are in conflict since they have identical version and product spaces.

Formally, a conflict may be defined as follows: Let $O_i, a_i, c_i\ (i = 1, 2)$ denote the sets of objects and the ambitions and choices of transactions t_1 and t_2, respectively. Transactions t_1 and t_2 stand in potential conflict if the following condition holds for $i = 1, 2$:

$$(O_1 \cap O_2 \neq \emptyset) \wedge (a_1 \wedge a_2 \neq false) \wedge (c_i \in (a_1 \wedge a_2)). \quad (22)$$

That is, both the object sets and the ambitions overlap. Furthermore, the choices are defined such that the changes are mutually visible.

The transaction support layer is responsible for detecting, preventing, or resolving such conflicts. In addition, it provides mechanisms for defining work units (tasks), for assigning developers to such units, and for defining and executing rules regarding change notification, propagation, and reconciliation. Again, it will be up to the upper layers (Section 7.4) to instrument these mechanisms, as part of a process model or high-level work policy.

Whether the transaction support layer prescribes flat (ACID [23]), hierarchical, or graph-like transaction structures is not fixed in our UVM architecture. The same applies to whether or how intertransaction cooperation (before commit) is possibly organized.

We also urge for transactions (and workspaces below) to be considered *first-class, persistent objects*, that can be stored, accessed, and manipulated by normal database queries.

A final comment: In the EPOS database the transaction layer was originally below the versioning layer (one such). As part of a redesign, we switched these two layers.[12] This reversal lead to more orthogonal and simple handling of both transactions and versions since it turned out that versioning could be handled as a more fundamental and data model independent mechanism than transactions. The transactions establish a context for the versioning, not the other way around. In the UVM proposal, the transaction layer is put even higher, above the product/data layer, so that it can offer more high-level synchronization (i.e., locking) features.

7.3 Workspace Support

The last abstraction is achieved through the *workspace* support layer, which aims at providing applications with data to work on. Such applications can be a CASE tool, such as UML RationalRose, or more classical development tools, such as editors and compilers.

Workspace support can be done along the lines of SCCS and RCS. That is, the files held in the SCM database are checked out into the file system where they are manipulated by applications. Proper caching must be observed for adequate performance. As mentioned earlier, another alternative is a virtual file system [14] which avoids physical copying and provides applications with the illusion of working in an ordinary file system, even though they are in fact operating on a universioned slice of a versioned database. Various regimes for maintaining consistency between the repository and the workspace can be envisaged: busy (full synchrony), lazy (on demand), reconcile upon commit/check-in, or combinations depending on object type, etc. Elaborate and large-scale workspace management can be found in Adele [56].

7.4 SCM Toolkit Functionality

To make the UVM functionality really useful, we envisage an *SCM toolkit* as part of the application layer, or possibly in a separate layer between the workspace layer and the application layer. Here, more sophisticated product and version models can be offered, possibly combined with a graphical interface and visualization facilities [57].

For instance, the product and version space for certain subsystems can be clustered, e.g., using scope rules. We can also put more high-level version descriptions in this layer, and map these to more low-level representations by the version rule layer. See the Versioning Assistant in Section 8.2.

More elaborate system models, using an architecture description language (AML [24]) or a systems modeling language (SySL [58] or PROTEUS PTL [59]) can be also be offered by using a database schema defined in the underlying product/data layer.

We should also be able to explicitly model and support parts of the (cooperative) SCM process, possibly using instrumentable "hooks" into the below layers. See, for instance, the Work Unit Description Language in EPOS [45] and similarly in Adele and ClearCase.

Indeed, modern SCM tools offer a myriad of user-oriented functionality, but most of this falls outside the scope of this paper.

12. Bjørn Munch claims he got the idea in the shower!

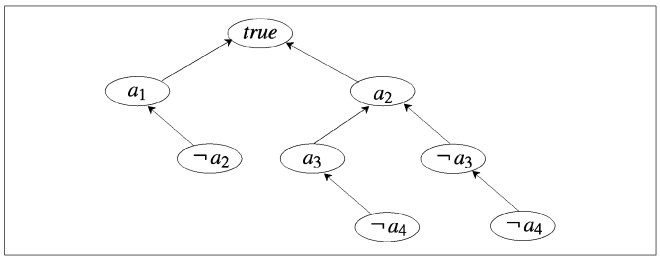

Fig. 12. Visibility tree, in general.

8 Experiences

We implemented most of the UVM model in EPOS [44], a research prototype of a software engineering environment supporting both SCM and process management. Two implementations of UVM were performed in EPOS. The initial one is reported in [16]; in this paper, we mainly refer to the revised implementation done in Munch's PhD thesis [15]. From these implementations, two problems became evident: efficiency and ease of use.

8.1 Efficiency of Version Visibilities and Version Selection

In the initial EPOS implementation, general-purpose evaluators (from the VLSI domain) for logical expressions were applied. It became apparent very soon that this was not the way to go. Storing large logical expressions as visibilities attached to fragments causes unacceptable overhead. Moreover, evaluation of these expressions is NP-complete, and standard algorithms which expand logical expressions into a normal form are too slow by orders of magnitude [46].

In the revised EPOS implementation, visibilities were handled much more efficiently with respect to both time and space. Inspection of the above equations for calculating visibilities reveals that each visibility can be expressed as a conjunction of positive and negative ambitions:

$$v = a_1 \wedge \ldots \wedge a_m \wedge \neg a_{m+1} \wedge \ldots \wedge \neg a_n. \tag{23}$$

Please recall also that each ambition is a conjunction of option bindings.

Visibilities are stored according to the above equation; they are never expanded into a normal form. Furthermore, many fragments share the same visibility. Therefore, the visibility field of a fragment contains a pointer into a global data structure called *visibility tree* (Fig. 12). Each tree node contains one (positive or negative) ambition and a pointer to its parent visibility.

The trees' leaves grow when insertions, deletions, or updates are performed. For example, when a fragment is deleted under an ambition a, a new visibility v_{new} is inserted to the tree whose node contains $\neg a$ and points to the old visibility v_{old}. In this way, *writes* can be performed

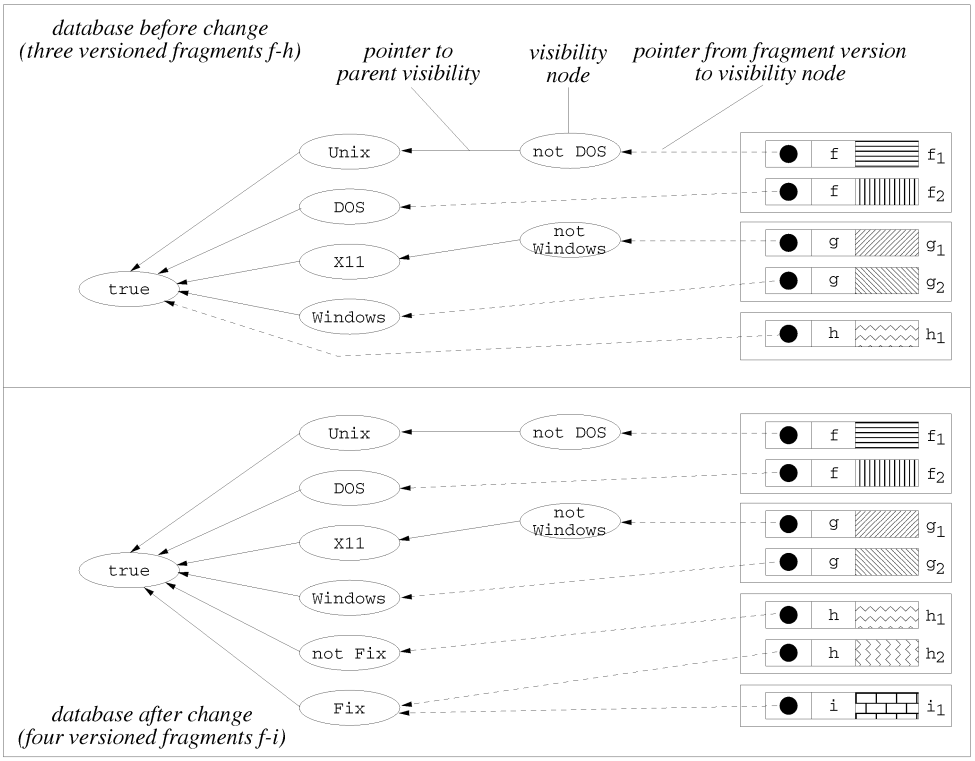

Fig. 13. An example of a visibility tree.

efficiently; no costly expansions and transformations of logical expressions need to be performed.

Reads can also be performed efficiently. Let us assume some choice c, which is a complete list of option bindings. For some fragment with visibility v, we have to decide whether $c \Rightarrow v$ holds. To this end, we process each ambition term in the visibility, from the bottom up. If it occurs positively in v, all option bindings must agree with those in c. If it is negated, at least one option binding must disagree. If the current term evaluates to false, evaluation stops (short-cut evaluation). Otherwise, the next term is considered until there is no more term to process.

Example 7 (Visibility tree). We return to the example given in Fig. 9 and display the visibility trees before and after the change carried out under the ambition `Fix` (Fig. 13). Before the change under the ambition $a =$ `Fix`, there is only one version of fragment `h` with visibility true. Since this fragment is updated, we have to introduce two visibility nodes corresponding to a and $\neg a$, respectively. Since the visibility of $\mathtt{h}_1$ is constrained by $\neg a$, the pointer to the visibility node is moved to the new node for $\neg a$. The new fragment versions $\mathtt{h}_2$ and $\mathtt{h}_1$ both point to the node for a.

Since many fragments share the same visibilities, we can speed up evaluation significantly by introducing a *cache* of evaluated visibilities, e.g., implemented as a hash table. Each visibility need only be evaluated once; next time, the value is retrieved from the cache. Since we work bottom-up in the visibility tree, we can also stop whenever we encounter an already evaluated visibility on the way up, i.e., *partial evaluation*. We also maintain a similar cache for evaluated ambitions. Both these caches are currently in use, and we do *incremental* (on demand) evaluation, so that unused terms are not evaluated. This is important for large option and product spaces.

Let us summarize the performance and complexity of this implementation:

- *Storage space.* For each fragment, there is a small constant overhead for storing its visibility ID, represented by a pointer into the visibility tree. Furthermore, the visibility tree grows linearly in the number of visibilities, which again increases with the number of changes to the database. The tree growth is usually one extra ambition term per visibility, due to shared subexpressions of such.

  However, the growth in visibilities may not be linear since the number of new visibilities created by a transaction depends on the number of different visibilities on fragments that are touched by it—see below.
- *Writes.* Inserting, deleting, or updating a fragment can be performed in constant time. Here, a new leaf node may have to be inserted into the visibility tree, and the new pointer has to be stored.
- *Reads.* Evaluation of a visibility against a choice is linear in the number of ambitions and the number of option bindings per ambition, i.e., linear in the total number of option bindings. However, each used visibility has to be evaluated only once when a cache is used, and only partially due to sharing.

The possible, nonlinear growth in the number of visibilities is the most serious problem and also the most difficult to evaluate since it depends heavily on the pattern

of transactions and ambitions being used. It will affect the storage space needed and, to some extent, also the evaluation time. However, we are not aware of any empirical studies of the intensity and/or locality of changes on single text files, and which is not limited to change-oriented versioning.

On the other hand, there are a few optimizations available which will greatly reduce the number of new visibilities (and new fragment versions) created by a transactions:

- Fragment visibilities are actually split into two parts: the *existence* of the fragment which is only affected by insert and delete and is common to all versions of the fragment, and the "variable" visibility of each version which is only affected by updates. This reduces the average size of the visibilities, but more importantly reduces the total number of them since more fragments are likely to share the same (simpler) visibilities.
- When deleting or updating a fragment according to (9), we only need to update those visibilities which have some overlap with the ambition; adding the negated ambition to one which has no effect. It doesn't help if the ambition consists only of a new option, but most ambitions will be more restricted.
- By sorting the versions of a fragment, so that the new version made by an update is read first, and making the version engine understand that the positive ambition in its visibility is *implicitly* negated in all the following versions of the same fragment, we don't actually have to update the visibilities at all.

The first two of these were designed into EPOS from the start, but the last one is a recent idea which hasn't been tested in theory or practice.

Example 8 (Efficiency of the EPOS implementation). In chapter 7 of Munch's PhD thesis [15], an experiment is described in which C source code with conditional compilation expressions is converted and stored in the EPOS version storage. It was assumed that `ifdefs` represent variants of the product.

In the first step, v2.3.3 of the `gcc` C/C++ compiler source, with 366 Klines—420 files and 3,166 `ifdefs`—were parsed. *All* variants (with `ifdefs` removed) of the 420 files were then checked into the client-server EPOSDB using a special-made tool. One hundred ninety-seven of the files were actually versioned with a total of 466 options and 639 different visibilities. Our tool deduced 518 ambitions which would generate those visibilities. On RCS, these were represented in a single version of each text file. Total CPU-time for check-in on the EPOSDB was 905 seconds on a Sun-4 and with 0.3 percent extra space compared to the original source. That is, 2.47 CPU-seconds of check-in time per 1Kline of potentially versioned source text. Most of the check-ins were coupled to automated workspace "group" check-ins, sharing the same ambition. Time usage for RCS was not recorded.

The second step was to upgrade `gcc` to v2.4.0 with 454 Klines, i.e., checking in new revisions and variants of the now 461 files with 3,580 `ifdefs`. Three hundred eighty-two of these files were now versioned and, in total, 604 new ambitions were checked in (again with `ifdefs` removed). Five hundred forty-two options, including one new for "revision 2.4.0" and 988 different visibilities were now registered. Total CPU-time for check-in on the EPOSDB was 1,048 seconds and space consumption 0.6 percent more than the original combination of RCS and conditional compilation. Check-in time per file was about the same, 2.31 CPU-second per Kline. Check-out times were not recorded, but I/O will anyhow dominate.

Again note, that EPOSDB was a research prototype where little performance optimization had been performed.

Our initial experiments [15] have shown that the overhead in storage space is moderate and can compete with RCS combined with conditional compilation. A more compact format of the visibility ID in the text files would have given us a 0.6 percent *saving* in total size in the above example. Moreover, compared to ICE, we have accelerated reads and writes so that evaluation of logical expressions is *dominated by I/O time*.

Due to the growth of the visibility tree, performance degrades with increasing age of the database. In [15], several *clean-up operations* are described that should be applied periodically to reorganize the database. These operations are designed to reduce the visibility tree, e.g., by searching for identical or overlapping ambitions in visibilities, by removing visibilities always evaluating to false (and deleting fragments decorated with these visibilities), etc.

To conclude, we have reduced the complexity from NP-completeness to "quite manageable" [60]. Compared to ICE, our approach has the following benefits:

- Reads and writes are performed in polynomial time. In contrast, feature unification in ICE is NP-complete. Only deductive shortcuts, implemented for restricted classes of feature terms, perform better.
- Only the evaluation of global version rules (Section 6.3), on top of the delta storage, is in theory NP-complete. However, these rules are evaluated only once at the beginning of a transaction. Once this is done, only ambition and choice are used for operating on the database. In contrast, in ICE there are no separately stored global version rules. All rules are embedded in the source text, as in conditional compilation.

8.2 A Versioning Assistant

It is no pleasure to be confronted with the raw version space. The user needs a tool to relieve him from all the selection details he has to consider.

A *Versioning Assistant* [50], [12], [55] must be considered part of an SCM toolkit (Section 7.4), providing considerable leverage by:

- Visualizing the version space, e.g., as version graphs.
- Checking version descriptions for consistency, using constraints stored in the rule base, e.g., for dependencies or validities.

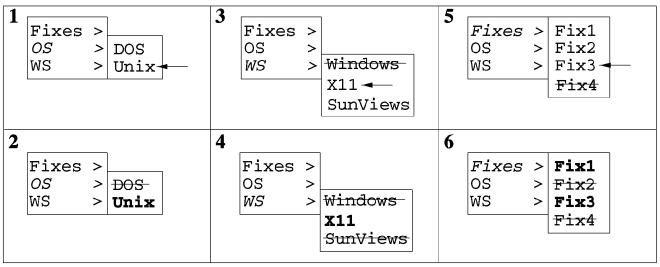

Fig. 14. Menu-based version selections.

- Remembering previous version selections.
- Offering scoped or *product-specific options*.
- Offering menu-based selections derived from global version rules, e.g., to implement option groups (variants).
- Deriving option bindings, so that the user needs to supply only a partial version description, e.g., using preferences or defaults.

The last two items are illustrated by the small example given below.

Example 9 (Versioning Assistant). Let us assume a software system which varies with respect to the operating system and the window system. Furthermore, several fixes have been applied. The following constraints exclude inconsistent version selections:

DOS $\otimes$ Unix
Windows $\otimes$ X11 $\otimes$ SunViews
DOS $\otimes$ X11
DOS $\otimes$ SunViews
Unix $\otimes$ Windows
Fix2 $\Rightarrow$ Fix1
Fix3 $\Rightarrow$ Fix1
Fix2 $\otimes$ Fix3
Fix4 $\Rightarrow$ DOS

Menu-based version selection is illustrated in Fig. 14. The sequence of actions is arranged column-wise from left to right (see numbers in the boxes). The upper row shows the states before the user makes selections, and the current selection is indicated by an arrow. The lower row shows the corresponding states *after* these selections. Options set to true are shown in bold face; options set to false are striked through. A hierarchical menu entry is indicated by ">". A selected hierarchical menu entry is emphasized by italic font.

The version selection proceeds as follows:

1. The user selects the option group `OS` in the top-level menu. A submenu is displayed to offer the alternatives `DOS` and `Unix`.
2. After the user has selected `Unix`, `DOS` is automatically disabled.
3. The user selects `WS` (for window system) in the top-level menu. In the submenu offering the available window systems, `Windows` has already been disabled automatically because it cannot be selected with `Unix`.
4. After the user has selected `X11`, `SunViews` is automatically disabled as well.
5. The user selects `Fixes` in the top-level menu. In the submenu offering the fixes, `Fix4` has already been disabled automatically because it assumes that the `DOS` variant of the operating system is selected.
6. After the user has selected `Fix3`, `Fix1` is automatically included because `Fix3` depends on `Fix1`. `Fix2` is excluded since `Fix2` and `Fix3` cannot be applied together.

After going through these steps, the choice is now complete; it binds `(Unix, X11, Fix3, Fix1)` to true and all other options to false.

8.3 Database and Workspace Issues

We built an experimental tool on top of the EPOS database, called ECM [45], which provided CVS-like functionality for managing versioned text files in checked-out workspaces and subworkspaces corresponding to transactions in EPOS. ECM also had knowledge of the product model, so you could check out a subproduct and associated files and subdirectories. ECM provided both a command-line and a Prolog interface; the latter would also check out entities and relationships into the user's environment.

The `gcc` source experiment above was not done through this tool as it hadn't been implemented by then, but we did use it to manage development of its own source code once it had become stable enough.

9 Discussion

In the previous section, we have already covered some crucial issues concerning the UVM approach, namely, efficiency and ease of use. We have pointed out how these issues have been dealt with in the EPOS implementation of UVM. Below, we briefly discuss further essential topics: the generality of the UVM approach and database design, respectively.

9.1 Generality of the UVM Approach

UVM radically differs from other approaches to versioning. Other systems implement a specific version model, and the version model depends on the data model. The generality of UVM is particularly achieved through the following design decisions:

- Version rules are used as the common base to realize specific version models. This inverts the approach followed in ClearCase and many other SCM systems, where version rules are defined on top of version graphs. In ClearCase, version graphs are deeply built into the system as a basic concept. In UVM, version graphs are not a basic concept. Rather, such graphs may be introduced on top of the uniform version storage—in the mentioned ECM toolkit.
- The version model is orthogonal to the data model. In PCTE, the version model is defined on top of the data model; in Adele, it is integrated into the data model. In both cases, the version model depends on the data model. In contrast, UVM may be applied to any data model.

Does the generality of UVM allow all other version models to be simulated on top of UVM? This statement holds insofar as both revisions and variants, both state- and change-based versioning, both version graphs and grids, and both extensional and intensional versioning can be expressed in UVM. We have shown in Section 6 that all these kinds of versioning may be defined with the help of version rules. However, these version rules are *global*, because the version space layers are located below the product space layers. As a consequence, the version rules cannot refer to specific product items (e.g., subsystems), because these are introduced in the upper layers of the UVM architecture (see discussion below).

For example, we can model both revision chains and alternative variants, as employed in Adele. However, we cannot express that some attributes are shared by all revisions and others have revision-specific values. This requires to handle updates to common and specific attributes in different ways. This distinction is not made in the instrumentable version engine as presented in Section 6. However, it can be accommodated by splitting a fragment into a versioned part and an unversioned part, as proposed in [5].

Moreover, we may simulate version graphs, as employed in ClearCase, but in UVM there would be one global (and virtual) version graph, rather than one version graph per versioned item (file or directory). We consider this an advantage over the ClearCase model because it is difficult to select a consistent set of versions from multiple version graphs that are structured in different ways.

In principle, the version space may be scoped by introducing *product-specific options*. Since this again breaks the orthogonality of data model and version model, we have to let a topmost SCM toolkit offer such "clustering" (Section 7.4) through some system modeling formalism. The toolkit's Versioning Assistant (Section 8.2) can be responsible for resolving and mapping the detailed product and version information onto the proper layers below. We can compare such multilayer approaches to the introduction of *modules* or object-oriented *classes* in Prolog. Here, we either have to resolve all "non-Prolog" name bindings before the basic inference engine can start its unification work, or such binding and inferencing have to be intertwined [61].

Our approach is not very different from the one being offered by virtual file systems, such as in ClearCase. Here, the file system (at least for some catalogs) is replaced by the ClearCase equivalent (technically done by API subroutine traps or by relinking of file system libraries). This provides a universion view of a specified part of the versioned database using appropriate version filter settings. As mentioned, ClearCase uses Ramia as an underlying DBMS, but this is hidden for the application programs and users. Of course, this extra layering provides some extra overhead, but performance seems satisfactory.

Similarly, the PCMS SCM tool uses Ingres as an underlying DBMS. Again, version-internal selection attributes and relationships are inaccessible at the application level. Also, in Adele, there is limited application access to version-internal attributes and relationships.

Our UVM proposal requires that *all* DBMS-internal fragments (buffers, tuples, etc.) are equipped with a prefixing visibility to serve as a supplementing "transaction identifier." This visibility is stripped off by a low-level SELECT operation guided by actual option settings. That is only those fragments are considered whose visibilities evaluate to true *before* any other interpretation or operation is performed on a fragment. However, this overhead is judged to be slight in terms of both space and time.

On the other hand, if such version selection is not done at the lower-levels of buffer or tuple management but, at the normal query level, the overhead may become intolerable. For instance, the "EPOSDB-1" implementation used Ingres as an underlying DBMS, but this resulted in a significant performance degradation because of the overhead of a full-fledged DBMS which really wasn't needed.

The best solution is probably to apply a *DBMS toolkit*, so that we can configure either versioned or unversioned DBMS support. For example, this assumes that buffer sizes are properly instrumentable to accommodate extra space for visibilities. We can mention the KIDS DBMS toolkit by Klaus Dittrich's group [8], [62], but this does not include versioning. On the other hand, transaction management is explicitly analyzed for possible functionality generation. However, the first paper from 1994 [8] says quite bluntly: "Up to now, no adequate, generic architecture model for DBMSes is known." To our knowledge, this is also the situation today. So, more research is needed—that is, motivating this paper.

The next best solution is to ignore, i.e., passivate, the versioning layer. That is, all visibilities are set to true, although this still renders a small overhead.

A final remark. Versioning is most useful for design and development work. It is unrealistic to equip major commercial DBMSes with full versioning as a default functionality. Therefore, the main target group of UVM is researchers and vendors of design databases, CAD tools, and SCM tools.

10 Conclusion

We have introduced a *uniform version model*, UVM and its support architecture for software version management, a subset of SCM. UVM differs from previous approaches in its support architecture, which inverts the ordering of certain layers. First, version graphs (or any other version model) are realized on top of version rules rather than vice versa. Second, the version model is completely orthogonal to the data model.

From the perspective of database management, the UVM architecture constitutes an attractive alternative to the main-stream architectures which define the version model on top of or integrated into the data model. The uniform version storage is highly reusable because it may be combined with any data model. Moreover, customized version models may be defined independently of the data model and may be provided as high-level libraries.

So far, practical experience in using the UVM architecture is still limited. However, we are convinced that our approach is promising and deserves further studies. Hopefully, this paper triggers further activities targeted at design, implementation, and evaluation of the UVM architecture, or parts of it.

ACKNOWLEDGMENTS

This work was partially carried out while Bernhard Westfechtel was conducting research at Norwegian University of Science and Technology (NTNU) in March 1998. Support from University of Aachen and NTNU is gratefully acknowledged.

REFERENCES

[1] W.F. Tichy, "Tools for Software Configuration Management," *Proc. Int'l Workshop Software Version and Configuration Control,* pp. 1–20, 1988.
[2] R.H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys,* vol. 22, no. 4, pp. 375–408, Dec. 1990.
[3] P.H. Feiler, "Configuration Management Models in Commercial Environments," Technical Report CMU/SEI-91-TR-7, Software Eng. Inst., Carnegie-Mellon Univ., Pittsburgh, Penn., Mar. 1991.
[4] R. Conradi and B. Westfechtel, "Version Models for Software Configuration Management," *ACM Computing Surveys,* vol. 30, no. 2, pp. 232–282, June 1998.
[5] R. Conradi and B. Westfechtel, "Towards a Uniform Version Model for Software Configuration Management," *Proc. Software Configuration Management Workshop (ICSE '97),* R. Conradi, ed., May 1997.
[6] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide.* Reading, Mass.: Addison Wesley, 1998.
[7] M.J. Carey, "The EXODUS Extensible DBMS Project: An Overview," *Readings in Object-Oriented Database Systems,* S.B. Zdonik and D. Maier, eds., pp. 477–499, 1990.
[8] A. Geppert and K.R. Dittrich, "Constructing the Next 100 Database Management Systems," *ACM SIGMOD Record,* vol. 23, no. 1, pp. 27–33, Mar. 1994.
[9] R. Ramakrishnan and J.D. Ullman, "A Survey of Deductive Database Systems," *The J. Logic Programming,* vol. 23, no. 2, pp. 125–149, May 1995.
[10] K. Ramamohanarao and J. Harland, "An Introduction to Deductive Database Languages and Systems," *The Very Large Data Bases J.,* vol. 3, no. 2, pp. 107–122, Apr. 1994.
[11] R. Conradi and B. Westfechtel, "Configuring Versioned Software Products," *Proc. Software Configuration Management Workshop (ICSE '96),* pp. 88–109, Mar. 1996.
[12] B. Munch, "HiCOV: Managing the Version Space," *Proc. Software Configuration Management Workshop (ICSE '96),* pp. 110–126, Mar. 1996.
[13] W.F. Tichy, "RCS—A System for Version Control," *Software—Practice and Experience,* vol. 15, no. 7, pp. 637–654, July 1985.
[14] D. Leblang, "The CM Challenge: Configuration Management That Works," *Configuration Management,* W.F. Tichy, ed., pp. 1–38, 1994.
[15] B.P. Munch, *Versioning in a Software Eng. Database—The Change Oriented Way.* PhD thesis, Norwegian Univ. of Science and Technology, Trondheim, Norway, 1993.
[16] A. Lie, R. Conradi, T. Didriksen, E.-A. Karlsson, S.O. Hallsteinsen, and P. Holager, "Change Oriented Versioning," *Proc. Second European Software Eng. Conf.,* pp. 191–202, 1989.
[17] Software Maintenance and Development Systems, Concord, Massachusetts, *Aide-de-Camp Product Overview,* 1990.
[18] C. Reichenberger, "Concepts and Techniques for Software Version Control," *Software—Concepts and Tools,* vol. 15, no. 3, pp. 97–104, July 1994.
[19] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.,* vol. 1, no. 4, pp. 364–370, Dec. 1975.
[20] G. Fowler, D. Korn, and H. Rao, "n-DFS: The Multiple Dimensional File System," *Configuration Management,* W.F. Tichy, ed., pp. 135–154, 1994.
[21] N.S. Barghouti and G.E. Kaiser, "Concurrency Control in Advanced Database Applications," *ACM Computing Surveys,* vol. 23, no. 3, pp. 269–317, Sept. 1991.
[22] G.E. Kaiser, "Cooperative Transactions for Multiuser Environments," *Modern Database Systems,* W. Kim, ed., pp. 409–433, 1995.
[23] J.N. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. Seventh Int'l Conf. Very Large Databases,* pp. 144–154, Sept. 1981.
[24] M. Shaw and D. Garlan, *Software Architecture—Perspectives on an Emerging Discipline.* Englewood Cliffs, N.J.: Prentice Hall, 1996.
[25] F. Oquendo, K. Berrado, F. Gallo, R. Minot, and I. Thomas, "Version Management in the PACT Integrated Software Engineering Environment," *Proc. Second European Software Eng. Conf.,* pp. 222–242, 1989.
[26] GOODSTEP, *The GOODSTEP Project—Final Report,* GOODSTEP ESPRIT Project 6115, Dec. 1995.
[27] K. Dittrich, W. Gotthard, and P. Lockemann, "DAMOKLES, a Database System for Software Engineering Environments," *Proc. Int'l Workshop Advanced Programming Environments,* R. Conradi, T.M. Didriksen, and D.H. Wanvik, eds., pp. 353–371, June 1986.
[28] E. Sciore, "Version and Configuration Management in an Object-Oriented Data Model," *Very Large Data Base J.,* vol. 3, no. 1, pp. 77–106, Jan. 1994.
[29] J. Estublier and R. Casallas, "The Adele Configuration Manager," *Configuration Management,* W.F. Tichy, ed., pp. 99–134, 1994.
[30] A. Zeller and G. Snelting, "Unified Versioning through Feature Logic," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 4, pp. 397–440, Oct. 1997.
[31] Y.-J. Lin and S.P. Reiss, "Configuration Management with Logical Structures," *Proc. 18th Int'l Conf. Software Eng.,* pp. 298–307, Mar. 1996.
[32] D.B. Leblang and G.D. McLean, "Configuration Management for Large-Scale Software Development Efforts," *Proc. Workshop Software Eng. Environments for Programming-in-the-Large,* pp. 122–127, June 1985.
[33] I.P. Goldstein and D.G. Bobrow, "A Layered Approach to Software Design," Technical Report CSL-80-5, XEROX PARC, Palo Alto, Calif., 1980.
[34] J. Micallef and G. Clemm, "The Asgard System: Activity-Based Configuration Management," *Proc. Software Configuration Management Workshop (ICSE '96),* pp. 175–186, 1996.
[35] C. Reichenberger, "VOODOO—A Tool for Orthogonal Version Management," *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* pp. 61–79, Apr. 1995.
[36] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner, "ClearCase MultiSite: Supporting Geographically-Distributed Software Development," *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* pp. 194–214, Apr. 1995.
[37] D.B. Leblang and R.P. Chase, "Computer-Aided Software Engineering in a Distributed Workstation Environment," *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. Practical Software Development Environments,* P. Henderson, ed., *ACM SIGPLAN Notices,* vol. 19, no. 5, pp. 104–112, May 1984.
[38] D.B. Leblang, R.P. Chase Jr., and H. Spilke, "Increasing Productivity with a Parallel Configuration Manager," *Proc. Int'l Workshop Software Version and Configuration Control,* pp. 21–37, 1988.
[39] L. Wakeman and J. Jowett, *PCTE—The Standard for Open Repositories.* Englewood Cliffs, N.J.: Prentice Hall, 1993.
[40] J. Estublier and R. Casallas, "Three Dimensional Versioning," *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* pp. 118–135, Apr. 1995.
[41] A. Zeller and G. Snelting, "Handling Version Sets Through Feature Logic," *Proc. Fifth European Software Eng. Conf.,* W. Schäfer and P. Botella, eds., pp. 191–204, Sept. 1995.
[42] A. Zeller, "A Unified Version Model for Configuration Management," *Proc. ACM SIGSOFT '95 Symp. Foundations of Software Eng., ACM Software Eng. Notes,* vol. 20, no. 4, pp. 151–160, Oct. 1995.
[43] W. Rigg, C. Burrows, and P. Ingram, *Configuration Management Tools.* London: Ovum Ltd., 1995.
[44] R. Conradi, "EPOS: Object-Oriented and Cooperative Process Modelling," *Software Process Modelling and Technology,* A. Finkelstein, J. Kramer, and B. Nuseibeh, eds., Advanced Software Development Series, pp. 33–70, 1994.
[45] A.I. Wang, J.-O. Larsen, R. Conradi, and B. Munch, "Improving Cooperation Support in the EPOS CM System," *Proc. Sixth European Workshop Software Process Technology (EWSPT '98)* V. Gruhn, ed., pp. 75–91, Sept. 1998.
[46] A. Lie, "Versioning in Software Engineering Databases," PhD thesis, Division of Computer Systems and Telematics, Norwegian Inst. of Technology, Trondheim, Norway, Jan. 1990.
[47] A. Lie, R. Conradi, T.M. Didriksen, E.-A. Karlsson, S.O. Hallsteinsen, and P. Holager, "Change Oriented Versioning in a Software Engineering Database," *Proc. Second Int'l Workshop Software Configuration Management,* W.F. Tichy, ed., pp. 56–65, Nov. 1989.
[48] V. Kruskal, "Managing Multiversion Programs with an Editor," *IBM J. Research and Development,* vol. 28, no. 1, pp. 74–81, Jan. 1984.

[49] N. Sarnak, R. Bernstein, and V. Kruskal, "Creation and Maintenance of Multiple Versions," *Proc. Int'l Workshop Software Version and Configuration Control,* pp. 264–275, 1998.
[50] B. Gulla, E.-A. Karlsson, and D. Yeh, "Change-Oriented Version Descriptions in EPOS," *Software Eng. J.,* vol. 6, no. 6, pp. 378–386, Nov. 1991.
[51] S.E. Bratsberg, "Unified Class Evolution by Object-Oriented Views," *Proc. 11th Int'l Conf. the Entity-Relationship Approach,* G. Pernul and A.M. Tjoa, eds., pp. 423–439, Oct. 1992.
[52] E. Odberg, "Category Classes: Flexible Classification and Evolution in Object-Oriented Databases," *Advanced Information Systems Eng.—Sixth Int'l Conf. (CAISE '94),* G. Wijers, S. Brinkkemper, and T. Wasserman, eds., pp. 406–420, June 1994.
[53] R. Conradi, C. Liu, and M. Hagaseth, "Planning Support for Cooperating Transactions in EPOS," *Information Systems,* vol. 20, no. 4, pp. 317–335, Dec. 1995.
[54] R.T. Snodgrass, "Temporal Databases," *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space,* A.U. Frank, I. Campari, and U. Formentini, eds., pp. 22–64, Sept. 1992.
[55] B. Gulla, *User Support Facilities for Software Configuration Management.* PhD thesis, Norwegian Univ. of Science and Technology, Trondheim, Norway, 2000.
[56] J. Estublier, "Distributed Objects for Concurrent Enginering," *Proc. System Configuration Management: SCM-9 Symp.,* J. Estublier, ed., pp. 172–186, Sept. 1999.
[57] B. Gulla, "Improved Maintenance Support By Multi-Version Visualizations," *Proc. Int'l Conf. Software Maintenance,* pp. 376–383, Nov. 1992.
[58] I. Sommerville and R. Thomson, "An Approach to the Support of Software Evolution," *The Computer J.,* vol. 32, no. 5, pp. 386–398, Dec. 1989.
[59] E. Tryggeseth, B. Gulla, and R. Conradi, "Modelling Systems with Variability Using the PROTEUS Configuration Language," *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* pp. 216–240,
[60] B.P. Munch, J.-O. Larsen, B. Gulla, R. Conradi, and E.-A. Karlsson, "Uniform Versioning: The Change-Oriented Model," *Proc. Fourth Int'l Workshop Software Configuration Management (Preprint),* S. Feldman, ed., pp. 188–196, May 1993.
[61] P. Schachte and G. Saab, "Efficient Object-Oriented Programming in Prolog," *Logic Programming: Formal Methods and Practical Applications,* C. Beierle and L. Plümer, eds., chapter 7, pp. 205–243, 1995.
[62] A. Geppert and K.R. Dittrich, "Strategies and Techniques: Reusable Artifacts for the Construction of Database Management Systems," *Proc. Seventh Int'l Conf. Advanced Information Systems Eng. (CAiSE '95),* J. Iivari, K. Lyytinen, and M. Rossi, eds., pp. 297–310, June 1995.
[63] J.F.H. Winkler, ed., *Proc. Int'l Workshop Software Version and Configuration Control,* 1988.
[64] *Proc. Software Configuration Management, Workshop SCM-6, (ICSE '96),* I. Sommerville, ed., Mar. 1996.
[65] *Configuration Management,* vol. 2 of *Trends in Software.* W.F. Tichy, ed., New York: John Wiley & Sons, 1994.
[66] *Proc. Second European Software Eng. Conf.,* C. Ghezzi and J.A. McDermid, eds., Sept. 1989.
[67] *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* J. Estublier, ed., Apr. 1995.

Bernhard Westfechtel received the diploma degree in 1983 from University of Erlangen-Nuremberg and the doctoral degree in 1991 from Aachen University of Technology, where he has been working as a senior researcher since then. He is interested in software engineering environments, software configuration management, process modeling, workflow management, object-oriented modeling, engineering/product data management, database systems for engineering applications, and software architectures. In 1995 and 1998, he spent two sabbaticals at the Norwegian University of Science and Technology (NTNU), Trondheim, where the research performed there produced several workshop and journal papers on software configuration management. In 1999, he published a book on models and tools for managing development processes.

Bjørn P. Munch received the DrIng (PhD) at the Norwegian Institute of Technology in 1993, based on work developing the EPOSDB and the version engine. He continued as a researcher, further developing the EPOSDB and working on related project until 1995. He then joined Telenor R&D as a software developer, and since 1997 he has been employed as a senior engineer at Clustra Systems' development center in Trondheim, where he has designed and implemented their in-house testing tool for the parallel Clustra DBMS.

Reidar Conradi received the MS degree in 1970 and the PhD degree in 1976, both in computer science from the Norwegian University of Science and Technology, Trondheim, Norway. He is a professor in the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU), Trondheim, Norway. He spent a sabbatical stay in 1999/2000 at the Fraunhofer Center for Experimental Software Engineering, Maryland and at the Politecnico di Milano, Italy. His interests include software quality, process modeling, software process improvement, software engineering databases, versioning, object-orientation, reuse, software architectures for distributed systems, and programming languages. He is a member of the IEEE.