Version Models for Software Configuration Management

REIDAR CONRADI

Norwegian University of Science and Technology, Trondheim

AND

BERNHARD WESTFECHTEL

Aachen University of Technology

After more than 20 years of research and practice in software configuration management (SCM), constructing consistent configurations of versioned software products still remains a challenge. This article focuses on the version models underlying both commercial systems and research prototypes. It provides an overview and classification of different versioning paradigms and defines and relates fundamental concepts such as revisions, variants, configurations, and changes. In particular, we focus on intensional versioning, that is, construction of versions based on configuration rules. Finally, we provide an overview of systems that have had significant impact on the development of the SCM discipline and classify them according to a detailed taxonomy.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*computer-aided software engineering;* D.2.6 [**Software Engineering**]: Programming Environments; D.2.9 [**Software Engineering**]: Management—*software configuration management;* H.2.3 [**Database Management**]: Languages—*database (persistent) programming languages;* H.2.8 [**Database Management**]: Database Applications; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*deduction, logic programming*

General Terms: Languages, Management

Additional Key Words and Phrases: Changes, configuration rules, configurations, revisions, variants, versions

1. INTRODUCTION

Software configuration management (SCM) is the discipline of managing the evolution of large and complex software systems [Tichy 1988]. The importance of SCM has been widely recognized, as reflected in particular in the Capability Maturity Model (CMM) developed by the Software Engineering Institute (SEI) [Humphrey 1989; Paulk et al. 1997]. CMM defines levels of maturity in order to assess software development processes in organizations. Here SCM is seen as one of the key elements for

This work was partially performed during a sabbatical in which the second author spent August through October 1995 in Trondheim. Support from NTNU is gratefully acknowledged.

Authors' addresses: R. Conradi, Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU), N-7034 Trondheim, Norway; B. Westfechtel, Department of Computer Science III, Aachen University of Technology, D-52056 Aachen, Germany.

CONTENTS

moving from "initial" (undefined process) to "repeatable" (project management, SCM, and quality assurance have come into operation). Furthermore, SCM plays an important role in achieving ISO 9000 conformance.

SCM serves different needs [Feiler 1991a].

—As a *management support discipline,* SCM is concerned with controlling changes to software products. It is this view of SCM that is addressed in the classical textbook by Bersoff et al. [1980] and the IEEE standard [IEEE 1983; IEEE 1988]. According to the latter, SCM covers functionalities such as identification of product components and their versions, change control (by establishing strict procedures to be followed when performing a change), status accounting (recording and reporting the status of components and change requests), and audit and review (quality assurance functions to preserve product consistency). Thus SCM is seen as a support discipline for project managers.

—As a *development support discipline,* SCM provides functions that assist developers in performing coordinated changes to software products. This view of SCM is described, for example, in the textbook by Babich [1986]. To support developers, SCM is in charge of accurately recording the composition of versioned software products evolving into many revisions and variants, maintaining consistency between interdependent components, reconstructing previously recorded software configurations, building derived objects (compiled code and executables) from their sources (program text), and constructing new configurations based on descriptions of their properties.

In this article, SCM is primarily considered a development support discipline. We provide an overview of *version models* implemented both in commercial systems and research prototypes. A version model defines the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. Software objects and their relationships constitute the *product space,* their versions are organized in the *version space.* A *versioned object base* combines product and version space. A specific version model is characterized by the way the version space is structured, by the decision of which objects are versioned both externally (from the user's point of view) and internally (within the versioned object base), by the relationships among version spaces of different objects, and by the way reconstruction of old and construction of new versions are supported.

SCM systems express version models in varying ways. Many systems, including most commercial ones [Rigg et al. 1995], are *file-based* and apply versioning to files and directories [Rochkind 1975; Tichy 1985; Fowler et al. 1994]. Various *language-based* approaches have been developed as well based on modular programming languages [Lampson and Schmidt 1983b], module interconnection languages [Prieto-Diaz and Neighbors 1986], or system modeling languages [Marzullo and Wiebe 1986]. These languages are typically used to represent versions of modules and relationships such as imports, include dependencies, and the like. Finally, several SCM systems are founded on databases and manage versions of objects and relationships stored in the database. Different data models have been used, including EER [Dittrich et al. 1986; Oquendo et al. 1989], object-oriented [Estublier and Casallas 1994], and deductive [Zeller and Snelting 1995] ones.

The variety of formalisms makes it difficult to compare the version models of different SCM systems with one another. In addition, each system comes with its own terminology. On the other hand, the underlying concepts are often very similar. In order to reveal these concepts, we introduce a unified terminology. Furthermore, to describe version models in a uniform, "canonical" formalism, we use *graphs* at many places in the article. A graph consists of nodes and edges representing entities and (binary) relationships, both of which may be decorated with attributes. Graphs are well suited to represent the organization of a versioned object base, even if the corresponding system is not graph-based. For example, SCCS [Rochkind 1975] and RCS [Tichy 1985] are both file-based, but the version space of a text file may be represented naturally as a version graph. Other formalisms are used as required, such as textual languages for expressing configuration rules.

The main contribution of this article is to give definitions and introduce a taxonomy. Furthermore, it provides a survey of the current state of the art by describing SCM systems in a unified terminology and classifying them according to the taxonomy. In this way, it prepares the ground for developing a uniform version model, that is, a common framework in which specific version models may be expressed. A uniform model would assist developers not only in constructing SCM systems but also in tailoring them more flexibly to the needs of their users. Multiple paradigms could be supported in parallel, allowing users to switch back and forth as required. Furthermore, as noted in Brown et al. [1991], a uniform model would constitute a common foundation for integrating heterogeneous SCM systems in a federated architecture.

In the interest of a thorough discussion, we focus on core issues of versioning, namely, the organization of the version space, the interrelations of product space and version space, and the construction of consistent configurations. Other issues considered essential parts of SCM are only discussed briefly, in particular, management of workspaces, construction of derived objects, cooperation, and distribution: all these issues are related to version management, but elaborating on them goes beyond the scope of this article.

The article is structured as follows. Before introducing versions, the product space is described in Section 2. Subsequently, we discuss the version space without making any assumptions about the product space (Section 3). The interplay of product and version space is addressed in Section 4. Section 5 is devoted to *intensional versioning* (i.e., construction of versions based on rules describing consistent combinations). Section 6 provides an overview of systems that have had significant impact on the development of the SCM discipline. Related work is discussed in Section 7, and a short conclusion is given in Section 8.

2. PRODUCT SPACE

The product space describes the structure of a software product without taking versioning into account (in other words, we assume only one version of a software product). The product space can be represented by a *product graph* whose nodes and edges correspond to software objects and their relationships, respectively. Different version models vary in their assumptions with respect to the product space. These differences refer to the types of software objects and relationships, to the granularity of object representations, and to the semantic levels of modeling.

2.1 Software Objects

A *software object* records the result of a development or maintenance activity. An SCM system has to manage all kinds of software objects created throughout the software life cycle, including requirements specifications, designs, documentations, program code, test plans, test cases, user manuals, project plans, and the like.

Identification is an essential function provided by SCM. Thus, each software object carries an *object identifier* (OID) that serves to identify it uniquely within a certain context. An external OID is a name assigned by the user, whereas a system-generated, unique OID may be used internally.

Software objects are rather *coarse-grained* units that are structured internally. For example, a program module is composed of declarations and statements, and a documentation consists of sections and paragraphs. Thus a software object is composed of more *fine-grained* units.

The data model used for representing the product space may or may not distinguish explicitly between coarse-grained and fine-grained units. For example, file systems make this distinction, whereas many object-oriented data models represent coarse- and fine-grained units in a uniform way. In the following, we represent the contents of software objects in long attributes attached to nodes of the product graph.

Software objects may have different representations, depending on the types of tools operating on them. In toolkit environments, software objects are stored as text files [Rochkind 1975]. In contrast, syntax trees [Habermann and Notkin 1986] or graphs [Nagl 1996] are used in structure-oriented environments. As discussed later, these representations influence the functionality of an SCM system. For example, a diff command for comparing two versions of a program module returns differing text lines in the case of text files and differing syntactic units in the case of syntax trees, respectively.

Independently of the representation chosen for software objects, we may distinguish between domain-independent and domain-specific models of the product space. *Domain-independent models* make no assumptions about the types of software objects to be maintained. All software objects produced throughout the whole software lifecycle project are subject to version control [Tichy 1985]. *Domain-specific models* are tailored towards specific types of software objects (e.g., abstract data types in algebraic specifications) [Ehrig et al. 1989].

2.2 Relationships

Software objects are connected by various types of relationships. *Composition relationships* are used to organize software objects with respect to their granularity. For example, a software product may be composed of subsystems, which in turn consist of modules. Objects that are decomposed are called *composite objects* or *configurations*. Objects residing at the leaves of the composition hierarchy are denoted *atomic objects*. Note that an “atomic” software object is still structured internally; that is, it has a fine-grained content. The root of a composition hierarchy is called the *(software) product*.

The semantics of composite objects

varies significantly across different modeling approaches. Furthermore, a specific approach may support customizable semantics. As a least common denominator, a composite object is defined as an object *o* that represents a subgraph of the product graph. All objects that are transitively reachable from *o* via composition relationships belong to this subgraph. There may be structural constraints with respect to the composition hierarchy (trees, DAGs), and the existence of a component may depend on the existence of its superobject(s). Furthermore, there may be constraints with respect to long attributes (e.g., long attributes may be attached only to leaves of the composition hierarchy). Finally, a composite object may act as a unit with respect to structural operations (e.g., copy or delete), abstraction (encapsulation of components), concurrency control (locking of a composite object includes locking of its components), and version control.

Dependency relationships (simply called dependencies in the following) establish directed connections between objects that are orthogonal to composition relationships. They include, for example, lifecycle dependencies between requirements specifications, designs, and module implementations, import or include dependencies between modules, and build dependencies between compiled code and source code.

The source and the target of a dependency correspond to a *dependent* and a *master object,* respectively. A dependency implies that the contents of the dependent must be kept consistent with the contents of the master. Thus the dependent may have to be changed when the master is modified.

Software objects are further classified into *source objects* and *derived objects.* A source object is created by a human who is supported by interactive tools, for example, text editors or graphical editors. A derived object is created automatically by a tool, for example, a compiler or linker. Note that the classification of a software object as a source or derived object depends on the available tool support. Furthermore, software objects may be partially derived and be partially constructed manually. For example, the skeleton of a module body may be created automatically from its interface and subsequently filled in by a programmer.

The process of creating derived objects from source and other derived objects is called *system building.* The actions to be performed are specified by build rules. The build tool has to ensure that build steps corresponding to these rules are executed in the correct order; that is, *build dependencies* must be taken into account. In contrast, *source dependencies* represent relationships between source objects (e.g., lifecycle dependencies as mentioned previously).

2.3 Representations of the Product Space

Figure 1 illustrates different representations of a sample software product foo which is implemented in the programming language C. Part (a) shows the modules of foo and their import dependencies. The top-level module main imports from a and b, which both import from c. foo may be represented in different ways. Some examples are given in (b), (c), and (d):

—In (b), foo is stored in the file system. Each module is represented by multiple files. The suffixes .h, .c, .o, and .exe denote header files, body files, compiled code, and executables, respectively. Dependencies and build steps are stored in a text file (the system model sys, e.g., a make file [Feldman 1979]).

—In (c), we assume a data model that supports typed objects and relationships (e.g., an EER model as used in PCTE [Oquendo et al. 1989]). As in the file system representation, there is still a composition tree whose leaves correspond to single files. However, dependencies are not represented in a separate text file. Rather, the tree is augmented with relation-

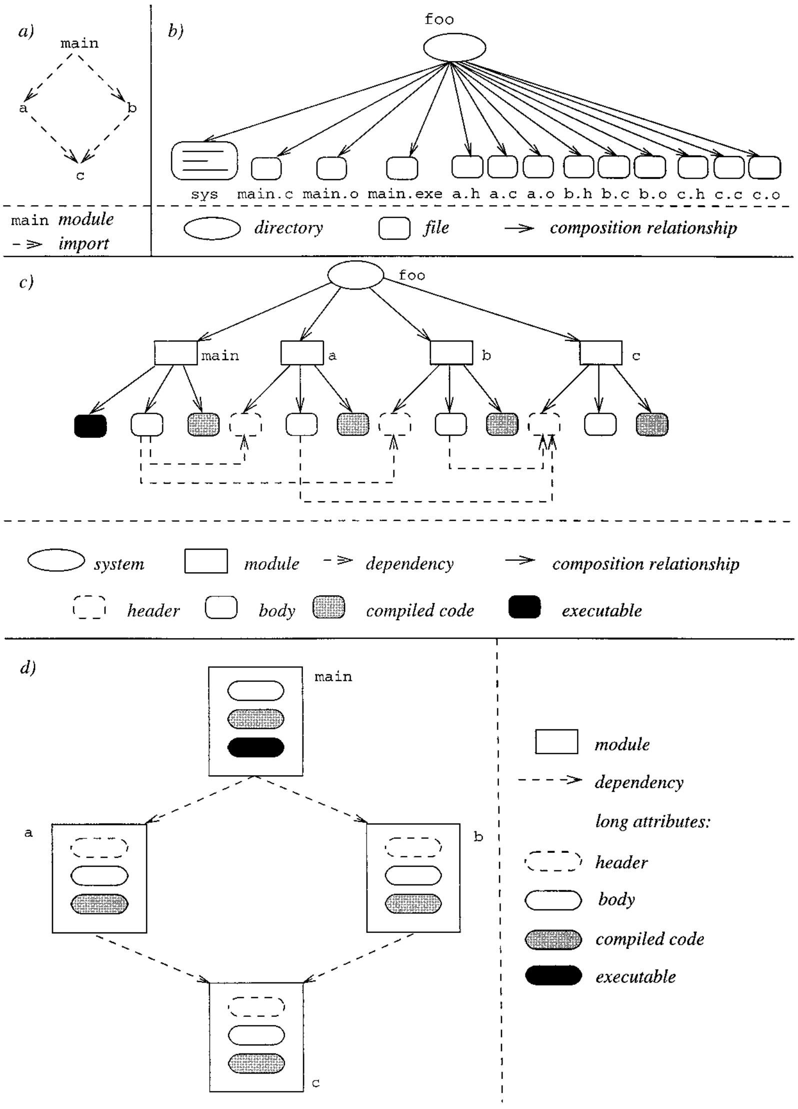


Figure 1. Different representations of a software product.

ships reflecting include dependencies. Build dependencies are not given explicitly because they can be computed automatically from source dependencies and composition relationships.

—In (d), there is no longer a spanning tree, and all files making up a module are summarized in one object. Furthermore, only a single type of relationship is used, which represents source dependencies between modules.[1] This organization, which has been realized, for example, in POEM [Lin and Reiss 1995], corresponds directly to the logical structure displayed in (a).

[1] This relationship may be annotated by an attribute that distinguishes between include dependencies emanating from header and body.

3. VERSION SPACE

A *version model* defines the items to be versioned, the common properties shared by all versions of an item, and the deltas, that is, the differences between them. Furthermore, it determines the way version sets are organized. To this end, it introduces dimensions of evolution such as revisions and variants, it defines whether a version is characterized in terms of the state it represents or in terms of some changes relative to some baseline, it selects a suitable representation for the version set (e.g., version graphs), and it also provides operations for retrieving old versions and constructing new versions.

The characterization of version models given in this section is still incomplete. Although the previous section described the product space without taking versioning into account, the current section conversely focuses on the version space, abstracting from the product space. Thus we are not concerned with the kinds of items put under version control, and we also consider versions of a single item only. However, a version model needs to address the interplay between product space and version space as well (Section 4).

3.1 Versions, Versioned Items, and Deltas

A *version* v represents a state of an evolving item i. 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 term *item* covers anything that may be put under version control, including, for example, files and directories in file-based systems, objects stored in object-oriented databases, entities, relationships, and attributes in EER databases, and so on. Versioning can be applied at any level of granularity, ranging from a software product down to text lines.

A *versioned item* is an item that is put under version control. In contrast, only one state is maintained for an unversioned item; that is, changes are performed by overwriting. Versioning requires a *sameness criterion;* that is, there must be some way to decide whether two versions belong to the same item. This decision can be performed with the help of a *unique identifier,* for example, an OID in the case of software objects.

Within a versioned item, each version must be uniquely identifiable through a *version identifier* (VID). Many SCM systems automatically generate unique version numbers and offer additional symbolic (user-defined) names serving as primary keys. However, a version can also be identified by an expression, which is the identification scheme used by intensional versioning.

All versions of an item share common properties called *invariants.* These invariants can be represented, for example, by unversioned attributes or relationships. Which invariants are shared by versions depends on the specific version model or the way it is customized to a certain application. At one end of the spectrum, versions virtually share only a common OID. For example, in systems such as SCCS and RCS versions of a text file may differ in arbitrary ways. At the other end of the spectrum, versions must share semantic properties. For example, version control in algebraic specification [Ehrig et al. 1989] enforces that all versions of a module body realize the shared interface.

Versions differ with respect to specific properties (e.g., represented by versioned attributes). The difference between two versions is called a *delta.* This term suggests that differences should be small compared to invariants. Delta can be defined in two ways (Figure 2):

—a *symmetric delta* between two versions v_1 and v_2 consists of properties specific to both v_1 and v_2 ($v_1 \setminus v_2$ and

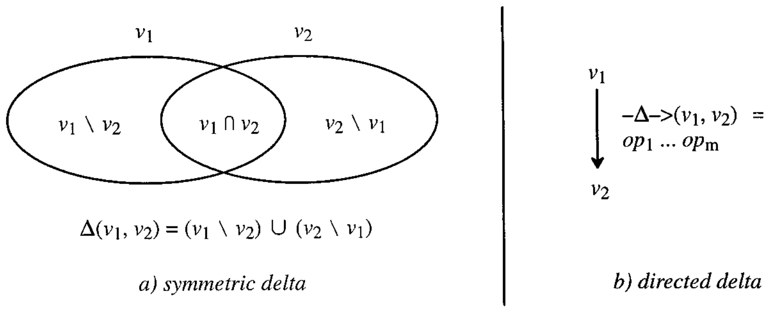

v_1
v_2
$v_1 \setminus v_2$
$v_1 \cap v_2$
$v_2 \setminus v_1$
$\Delta(v_1, v_2) = (v_1 \setminus v_2) \cup (v_2 \setminus v_1)$
a) symmetric delta
v_1
$-\Delta\text{->}(v_1, v_2) = op_1 \ldots op_m$
v_2
b) directed delta

Figure 2. Deltas: (a) symmetric; (b) directed.

$v_2 \setminus v_1$, respectively, where \ denotes set difference); or

—a *directed delta,* also called a *change,* is a sequence of (elementary) change operations $op_1 \ldots op_m$ which, when applied to one version v_1, yields another version v_2 (note the correspondence to transaction logs in databases).

In practice, deltas are not necessarily small. In the worst case, the common part of v_1 and v_2 may even be empty. In fact, items may undergo major changes, and the common properties may become smaller and smaller the more versions are created. For example, it is usually unrealistic to assume that all versions of module bodies realize the same interface (this assumption is made, e.g., in the algebraic approach already cited [Ehrig et al. 1989] and in the Gandalf system [Kaiser and Habermann 1983]). On the other hand, common properties do have to be asserted because otherwise it does not make sense to group versions at all.

A way out of this dilemma is *multi-level versioning;* that is, a version may have versions themselves. For example, in Adele [Estublier 1985] a module has multiple versions of interfaces each of which is realized by a set of body versions. DAMOKLES [Dittrich et al. 1986] generalizes this idea and supports recursive versioning; that is, any version may be versioned in turn.

3.2 Extensional and Intensional Versioning

A versioned item is a container for a set V of versions. The functionality of version control is heavily influenced by the way V is defined. *Extensional versioning* means that V is defined by enumerating its members:

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

Extensional versioning supports retrieval of previously constructed versions (which is a necessary requirement to any version model). All versions are explicit and have been checked in once before. Each version is typically identified by a unique number. The user interacting with the SCM system retrieves some version v_i, performs changes on the retrieved version, and finally submits the changed version as a new version v_{i+1}. To ensure safe retrieval of previously constructed versions, versions can be made immutable. In many systems, all versions are made immutable when they are checked into the object base [Rochkind 1975]; in others, explicit operations are provided to freeze mutable versions [Westfechtel 1996]. Furthermore, immutability may be enforced selectively (e.g., by distinguishing between mutable and immutable attributes [Estublier and Casallas 1995]).

Intensional versioning is applied when flexible automatic construction of consistent versions in a large version space needs to be supported. Instead of enumerating its members, the version set is defined by a predicate:

$$V = \{v \mid c(v)\}.$$

In this case, versions are implicit and many new combinations are constructed on demand. The predicate c defines the constraints that must be satisfied by all members of V. A specific version v is described intensionally by its properties (e.g., the Unix version supporting the X11 window system). A version is constructed in response to some query. For example, such a query may simply consist of a tuple of attribute values (which may be considered the VID of the version). In this case, a query corresponds to a (partial or total) function q that creates versions in V based on attributes ranging over the domains $A_1 . . A_n$:

$$q : A_1 \, x \, . . . \, x \, A_n \rightarrow V.$$

Here the term "attribute" is used in a general way; for example, attributes may identify variants (e.g., an os attribute determining the operating system) or changes (e.g., a Boolean attribute Fix to indicate whether a certain bug fix should be included or omitted).

The difference between extensional and intensional versioning may be illustrated by comparing SCCS [Rochkind 1975] and RCS [Tichy 1985] to conditional compilation as, for example, supported with the C programming language [Kernighan and Ritchie 1978]. SCCS and RCS store and reconstruct versions of text files (extensional versioning). The preprocessor used for conditional compilation constructs any source file based on the values of preprocessor variables (intensional versioning). All fragments of the source file are excluded whose conditions evaluate to false.

From SCCS/RCS and conditional compilation, SCM systems have been developed that differ significantly in their versioning capabilities. On the other hand, it must be emphasized that extensional and intensional versioning are by no means mutually exclusive, but can (and should) be combined into a single SCM system.

3.3 Intents of Evolution: Revisions, Variants, and Cooperation

Versioning is performed with different intents. A version intended to supersede its predecessor is called a *revision*. Revisions evolve along the time dimension and may be created for various reasons, such as fixing bugs, enhancing or extending functionality, adapting to changes in base libraries, and the like. Instead of performing modifications by overwriting, old revisions are preserved to support maintenance of software delivered to customers, to recover from erroneous updates, and so on.

Versions intended to coexist are called *variants*. For example, variants of data structures may differ with respect to storage consumption, run-time efficiency, and access operations. Furthermore, a software product may support multiple operating systems or window systems.

Finally, versions may also be maintained to support *cooperation*. In this case, multiple developers work in parallel on different versions. Each developer operates in a *workspace* [Estublier 1996] that contains the versions created and used. Cooperation policies regulate when versions are exported from or imported into a workspace. These issues are closely related to software process management and in the following are only discussed in passing (see also Section 7.2).

3.4 Representations of the Version Space: Version Graphs and Grids

Many SCM systems use *version graphs* for representing version spaces. A version graph consists of nodes and edges corresponding to (groups of) versions and their relationships, respectively. Since a version graph assumes an explicitly given set of versions, it is applied in conjunction with extensional versioning. Despite the limitations discussed in the following, they are widely used in practice.

Although a rich variety of version

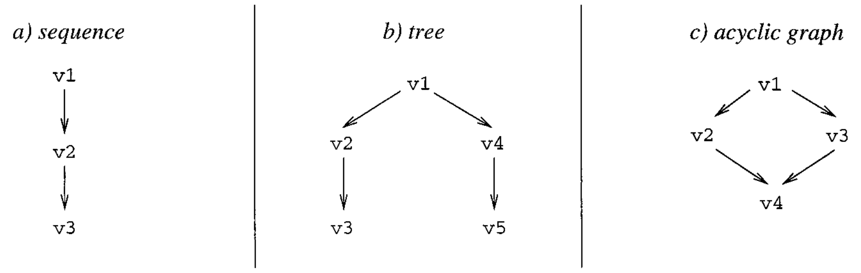


Figure 3. Version graphs (one-level organization): (a) sequence; (b) tree; (c) acyclic graph.

graphs is conceivable, the mainstream of SCM systems is based on a small number of graph types.

In the simplest case (*one-level organization*), a version graph consists of a set of versions connected by relationships of a single type, called *successor relationships.* A version graph of this type primarily represents the evolution history of a versioned item. "v_2 is a successor of v_1" means that v_2 has been derived from v_1, for example, by modifying a copy of v_1.

Version graphs may have different shapes (Figure 3). In the most restrictive case, versions are organized into a sequence of revisions. In a version tree, successors of nonleaf versions may be created, for example, in order to maintain old versions that have already been delivered to a customer. In an acyclic graph, a version may have multiple predecessors, for example, in order to express that a bug fix in an old version is merged with the currently developed version.

Several SCM systems use one of these different kinds of one-level organization, for example, sequences in NSE [Adams et al. 1989] and acyclic graphs in PCTE [Oquendo et al. 1989]. DAMOKLES [Dittrich et al. 1986] supports user-defined structural constraints. The structure of a version graph may be defined in the database schema as a sequence, a tree, or a directed acyclic graph.

In *two-level organization,* a version graph is composed of *branches,* each of which consists of a sequence of revisions. Here at least two relationship types are required, called successor (within a branch) and offspring (between branches) in Figure 4. This organization is applied, for example, in RCS. ClearCase [Leblang 1994] goes beyond the RCS organization by recording *merges* in the version graph. By means of merging, changes performed on one branch can be propagated to another branch. Essentially, this results in a directed acyclic graph. However, the branches are not joined; rather, each of them continues to exist.

Version graphs as presented previously support management of variants only to a limited extent. Variants can be represented by branches as long as their number is small. In the case of *multidimensional variation,* this approach breaks down because the number of branches explodes combinatorially. Let us assume that each dimension is modeled by an attribute with domain A_i. Then the number of branches b is dominated by the product of the domain cardinalities:

$$b \leq |A_1|\ldots|A_n|.$$

To illustrate this, let us assume that our sample product foo varies with respect to the operating system (DOS, Unix, VMS), the database system (Oracle, Informix, dbase), and the window system (X11, SunViews, Windows). In this case, up to 27 branches would be required.

This problem can be solved in the following ways.

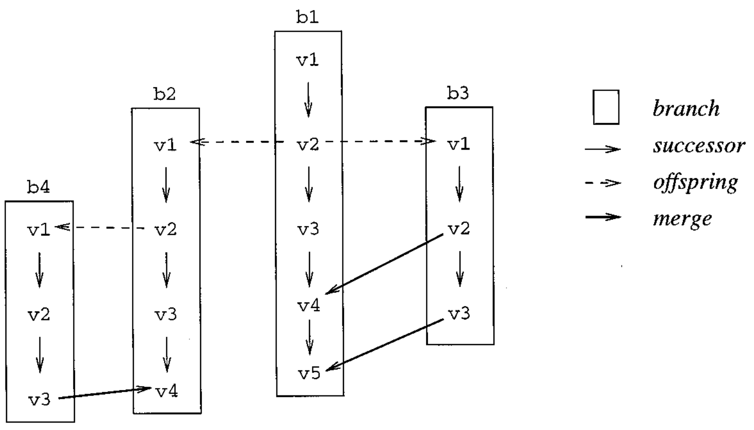


Figure 4. Version graphs (two-level organization).

—Version graphs may be generalized in order to support multidimensional variation. In Figure 5(a), versions are organized into *clusters* that are used to construct classification hierarchies [Dittrich and Lorie 1988].

—Alternatively, versions may be arranged in a *grid* (Figure 5b), that is, an n-dimensional space whose dimensions correspond to variant attributes [Sciore 1994].

Figure 5 illustrates only the variant space, assuming that there is no evolution along the time axis. Revisions can be represented in the grid by adding a time dimension (*orthogonal version management* [Reichenberger 1994]). In the case of the version graph, one level may be added at the bottom and successor relationships may be introduced to represent histories.

3.5 State-Based and Change-Based Versioning

In Section 3.1, a version has been defined as a state of an evolving item. Version models that focus on the states of versioned items are called *state-based.* In state-based versioning, versions are described in terms of revisions and variants.

Changes provide an alternative way of characterizing versions. In *change-based* models, a version is described in terms of changes applied to some baseline. To this end, changes are assigned *change identifiers* (CID) and potentially further attributes to characterize the reasons and the nature of a change. Change-based versioning provides a nice link to change requests: a change request is implemented by a (possibly composite) change. Thus a version may be described in terms of the change requests it implements.

"State- versus change-based" is orthogonal to "extensional versus intensional." *State-based extensional versioning* is provided, for example, by SCCS and RCS and *state-based intensional versioning* can be realized, for example, by conditional compilation: preprocessor variables can be used to represent variants, and a state may be specified by tuples of values for these variables. However, we emphasize that in general conditional compilation can be used for both state- and change-based versioning (the latter is done, e.g., in the COV system [Gulla et al. 1991]).

Change-based versioning comes in two forms. In the case of *change-based extensional versioning,* the version set is defined explicitly by enumerating its members and each version is described by the changes relative to some baseline. Thus changes are used only for

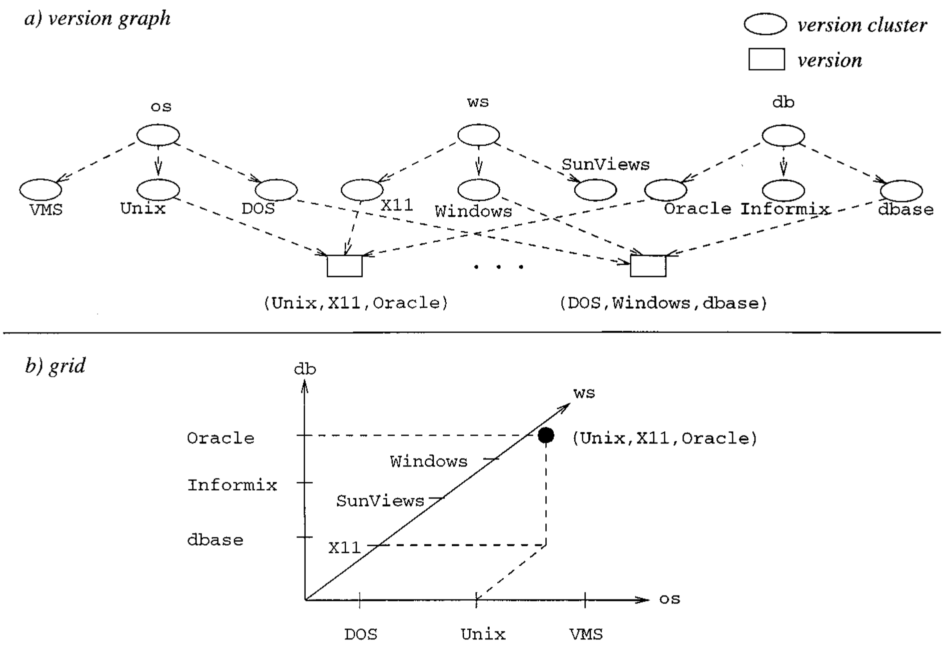


Figure 5. n-dimensional variant space: (a) version graph; (b) grid.

documentation. The OVUM report about SCM systems [Rigg et al. 1995] uses the term *change package* to denote this form of change-based versioning. Several SCM systems support change packages by annotating versions in version graphs with change identifiers (e.g., ClearCase and PCMS).

In *change-based intensional versioning,* changes are combined freely to construct new versions as required. Therefore a *change* is considered a partial function $c : V \rightarrow V$, where V denotes the set of all potential versions of some item. A version v is constructed by applying a sequence of changes $c_1 \ldots c_n$ to a baseline b:

$$v = c_1 \, o \ldots o \, c_n(b) = c_n(\ldots c_1(b) \ldots).$$

The OVUM report (and also the survey by Feiler [1991a]) adopts the terminology coined by the Aide-de-Camp system [Software Maintenance and Development Systems 1990] and calls this form of versioning the *change set* model. Further examples of systems supporting change-based intensional versioning are the COV system [Gulla et al. 1991], PIE [Goldstein and Bobrow 1980], DaSC [MacKay 1995], and Asgard [Micallef and Clemm 1996].

The *change space* (version space structured in terms of changes) can be represented in different ways. The COV system arranges versions in an n-dimensional grid called option space. Each change corresponds to a Boolean option that is set to true (false) if the change is applied (omitted). The Aide-de-Camp documentation introduces a matrix representation as shown in Figure 6(a). Lines and columns correspond to versions and changes, respectively. The application of a change is indicated by a circle at an intersection point. For example, v1 is constructed by applying c1, c2, and c3 in order.

Figure 6(b) illustrates the relations to the version graphs introduced in Figures 3 and 4, respectively. The versions shown in (a) are arranged in a graph whose nodes and edges correspond to versions and changes, respectively. b denotes the baseline, and intermediate

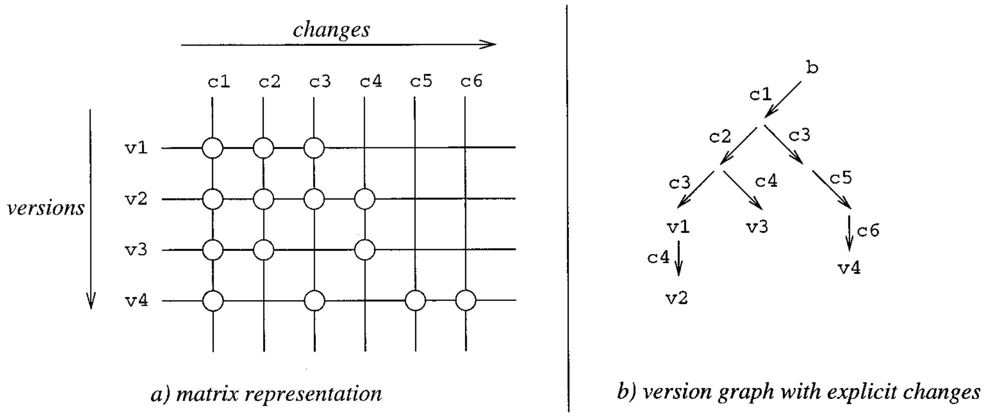


Figure 6. Change space: (a) matrix representation; (b) version graph with explicit changes.

versions are anonymous. For example, the path from b to v1 again contains the changes c1, c2, and c3. This version graph explicitly expresses all changes included in a certain version, and therefore provides more information than the version graphs shown in Figures 3 and 4. In particular, it makes explicit when certain changes were applied to multiple versions. For example, c4 was used to construct both v2 and v3. In contrast, state-based versioning does not name the changes; that is, the changes are anonymous. As a consequence, the changes applied to a version must be deduced from the topology of the version graph. This may become difficult if merging is applied in extensive and complicated ways (or even impossible if merges are not recorded at all).

4. INTERPLAY OF PRODUCT SPACE AND VERSION SPACE

In Section 2, the product space was described under the assumption that only one version of each item is maintained. In Section 3, basic definitions for versioning were given without considering the product space. The current section combines product space and version space into a *versioned object base.*

So far, we have considered versioning of a single item only, and we have made no assumptions concerning the kinds of items put under version control. As mentioned earlier, a version model needs to address the interplay between product space and version space as well. In the following, we discuss those aspects of version models that are concerned with this interplay. In particular, we investigate which items are put under version control, at what granularity versioning is applied both externally and internally, how versions of different items are interrelated, what models are used for representing versioned object bases, and how the version model may be related to the data model.

4.1 AND/OR Graphs

AND/OR graphs [Tichy 1982a] provide a general model for integrating product space and version space. An AND/OR graph contains two types of nodes, namely, AND nodes and OR nodes. Analogously, a distinction is made between AND and OR edges, which emanate from AND and OR nodes, respectively. An unversioned product graph can be represented by an AND/OR graph consisting exclusively of AND nodes/edges. Versioning of the product graph is modeled by introducing OR nodes. Versioned objects and their versions are represented by OR nodes and AND nodes, respectively.

Note that the version graph illustrated in Figure 5(a) can be regarded as an AND/OR graph as well: version clusters and versions correspond to OR nodes and AND nodes, respectively. However, in the following we are not concerned with the capabilities of AND/OR graphs to model the version space of a single object. Rather, we are interested in the relations between versions of different objects, abstracting for the moment from the ways in which their version spaces are structured. Therefore in the following a versioned object is simply represented by an OR node whose outgoing edges point to its versions.

AND edges are used to represent both composition and dependency relationships. A relationship is *bound* to a specific version if the corresponding AND edge ends at an AND node; otherwise it is called *generic*. A *configuration* is represented by a subgraph spanned by all nodes that are transitively reachable from the root node of the configuration. If all AND edges belonging to this subgraph are bound, the configuration is called bound as well; otherwise it is called generic. Furthermore, we may distinguish between *partially* and *totally generic* configurations. In the first case, there are both bound and generic AND edges; in the latter case, all AND edges are generic. A bound configuration can be constructed from a generic configuration by eliminating the OR nodes, that is, by selecting one successor of each OR node reached during traversal from the root node.

In the following, we return to the examples of product graphs in Figure 1 and compare several approaches to versioning the product graph. Figure 7 shows AND/OR graphs that are all based on Figure 1(b).[2] In Figure 7(a) only atomic software objects are versioned. foo represents a generic configuration. In Figure 7(b) foo is versioned as well. Each version of foo corresponds to a bound configuration. In Figure 7(c) references to components are generic, and the versions of foo therefore represent generic configurations.

Using this figure, we may classify version models according to the *selection order* during the configuration process:

—*Product first* (Figure 7(a)) means that the product structure is selected first; subsequently, versions of components are selected. This approach is followed, for example, by SCCS and RCS. It suffers from the restriction that structural versioning cannot be expressed (the product structure is the same for all configurations).

—*Version first* (Figure 7(b)) inverts this approach: the product version is selected first and uniquely determines the component versions. Different product versions may be structured in different ways. For example, a version of component c is contained only in foo.1 (version 1 of product foo). PCTE [Oquendo et al. 1989] is an example of an SCM system using this organization.

—*Intertwined* (Figure 7(c)) means that AND and OR selections are performed in alternating order. The intertwined organization is used, for example, by ClearCase [Leblang 1994], which versions both files and directories. Again, this selection scheme supports structural versioning.

Thus, "version first" and "intertwined" both take into account that different versions of an object may vary with respect to their relationships to (versions of) other objects. This means that in addition to objects, relationships are versioned as well.

So far, we have applied versioning only to the product graph organization shown in Figure 1(b). In Figure 8, the alternatives "intertwined" and "version first" are illustrated for the product graph of Figure 1(d) (To save space, "product first" has been omitted). In contrast to the previous figure, AND

[2] Module versions are identified by numbers. For the sake of simplicity, each module version is represented by a single node (e.g., no distinction between header files and bodies).

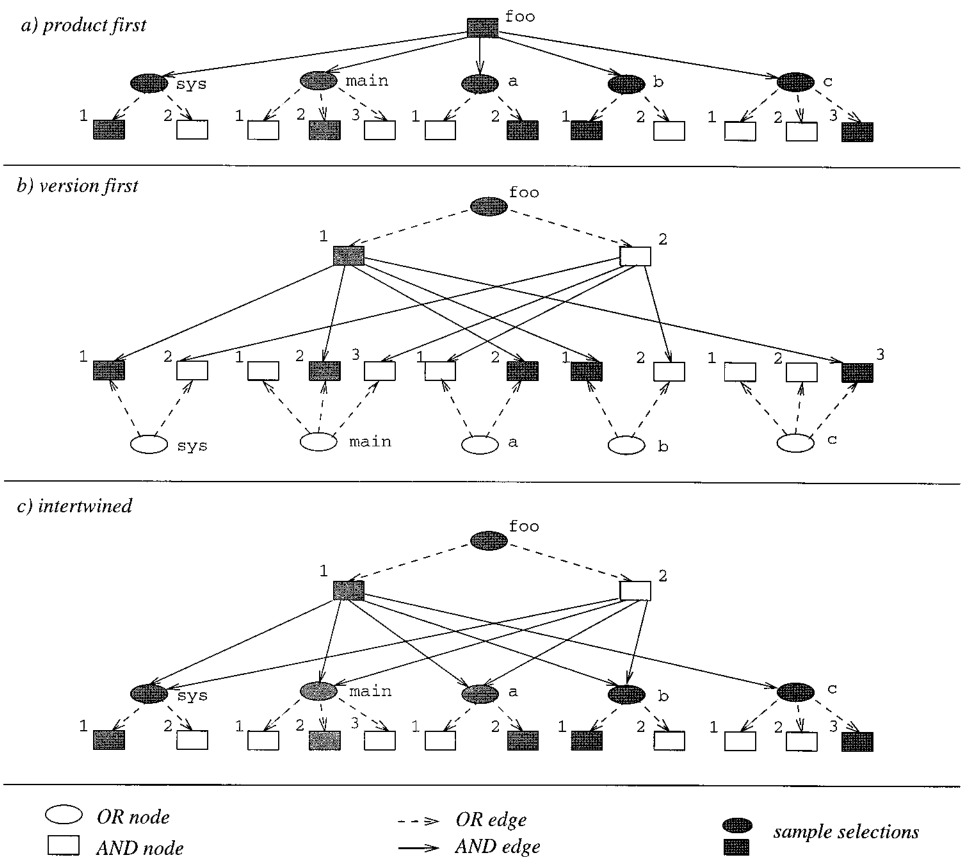


Figure 7. Different kinds of AND/OR graphs and selection orders (I): (a) product first; (b) version first; (c) intertwined.

edges represent dependencies instead of composition relationships. Intertwined selection is performed, for example, in Adele [Estublier 1985], whereas "version first" is realized, for example, in POEM [Lin and Reiss 1995].

To illustrate the differences between Figures 8(a) and (b), let us assume that a bug is fixed in module c that does not affect its interface. A new version c.2 is created that is to be included in the new release of our sample product foo.[3] In (a), a new configuration is constructed in which c.2 is selected instead of c.1. However, in (b) new versions of all modules above c have to be created (versions 2′ of a, b, and main, respectively). This effect is called *version proliferation.* Note that version proliferation need not involve physical copying (e.g., multiple versions may share the same source file through pointers). However, this does not solve the problem at the logical level (the user is confronted with a combinatorially exploding number of versions). To get rid of version proliferation at the logical level, we have to distinguish between versions of modules and versions of configurations (in (b), the versions of main play both roles simultaneously).

[3] The current release (the starting point for the bug fix) is represented in Figure 8 by filled nodes.

4.2 Granularity of Versioning

In the previous examples we considered versioning only at the coarse-grained

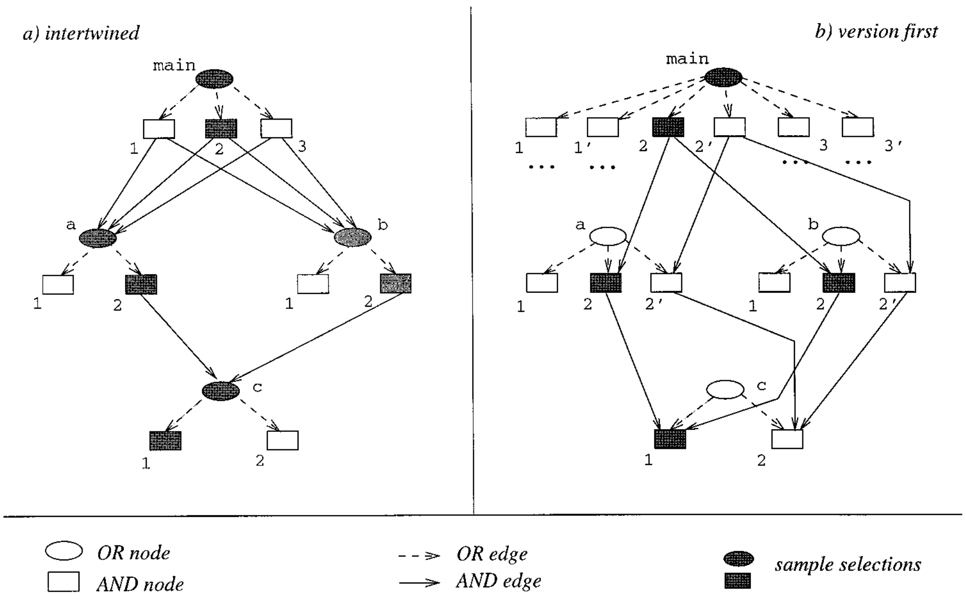


Figure 8. Different kinds of AND/OR graphs and selection orders (II): (a) intertwined; (b) version first.

level. This means that we have applied versioning to product graphs as introduced in Section 2. As explained earlier, product graphs detail the product structure only down to the level of software objects such as interfaces and bodies of modules; the fine-grained contents are represented as long attributes.

However, in Section 3 we introduced the term "item" in a more general way to denote anything that can be versioned, including entities, relationships, and attributes. In particular, versioning can be applied at any level of granularity.

To clarify this issue, let us further elaborate the notion of granularity. First, *version granularity* refers to the size of a version and second, *delta granularity* refers to the size of those units in terms of which deltas are recorded: in RCS version and delta granularity are at the level of text files and text lines, respectively. In this case, the delta granularity is much finer than the version granularity.

With respect to version granularity, we need to distinguish further between *external versioning* and *internal versioning*. An SCM system provides an external interface to the versioned object base that offers versioned items as well as identification and selection of versions to its users. At the external interface, software objects are the items subject to version control. The internal granularity may be much smaller (see the following).

External versioning may be applied in different ways to the composition hierarchy of software objects. *Component versioning* means that only atomic objects are put under version control. Each object has its own version space, modeled, for example, by a version graph. *Total versioning* applies to all levels of the composition hierarchy. *Product versioning* differs from total versioning by arranging versions of all objects in a uniform, global version space.

This classification is illustrated in Figure 9, where externally versioned objects are surrounded by boxes. The AND/OR graph in Figure 9(a) was taken from Figure 7(a), and the graphs shown in (b) and (c) were both copied from Figure 7(b). Note that the topology of an

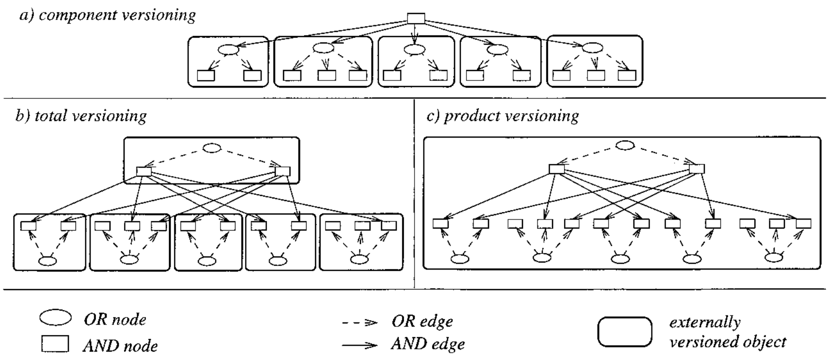


Figure 9. External versioning: (a) component versioning; (b) total versioning.

AND/OR graph shows which objects are versioned, but externally and internally versioned objects are not distinguished. Thus a given AND/OR graph can be accessed in different ways, depending on the version model presented to the user. In particular, in the case of product versioning the user may select versions of nonroot objects, but this is done in the version space attached to the whole product.

Component Versioning. In component versioning, the product structure is selected first. Typically, versions of components are organized into version graphs (see, e.g., RCS [Tichy 1985]). A configuration is constructed by assembling versions of components; this is called *composition model* in Feiler [1991a]. Note that the granularity of composition is an "atomic" software object (rather than fine-grained units such as statements or text lines).

Frequently, the version graphs of different components are related only weakly to each other. To illustrate this, Figure 10 shows a sample configuration whose components are located at different places in the respective version graphs.[4] The selection problem can be alleviated by tagging all components of some consistent configuration with the same symbolic name (or more generally through configuration rules referring to revisions and variants). Thus a configuration is represented implicitly through an attribute value rather than as a first-class entity.

Total Versioning. Total versioning generalizes component versioning in that all objects are versioned rather than only the leaves of the composition hierarchy. In contrast to component versioning, versions of composite objects are represented explicitly as entities; atomic and composite objects are versioned uniformly. Whereas component versioning implies that the product structure is selected first, total versioning may be combined with both the "version first" and the "intertwined" selection order and can therefore express structural versioning. For example, PCTE [Oquendo et al. 1989] supports extensional versioning of composite objects ("version first"). In contrast, in ClearCase [Leblang 1994] AND/OR selections are intertwined: a version of a directory references a set of versioned files (or directories) rather than specific versions of these. A single-version view on the versioned file system is provided through dynamically evaluated configuration rules (intensional versioning with dynamic binding).

Product Versioning. Product versioning differs from total versioning by arranging versions of all objects in a

[4] AND nodes representing versions are placed inside the OR node representing the versioned component. Furthermore, the AND/OR graph is augmented with successor relationships.

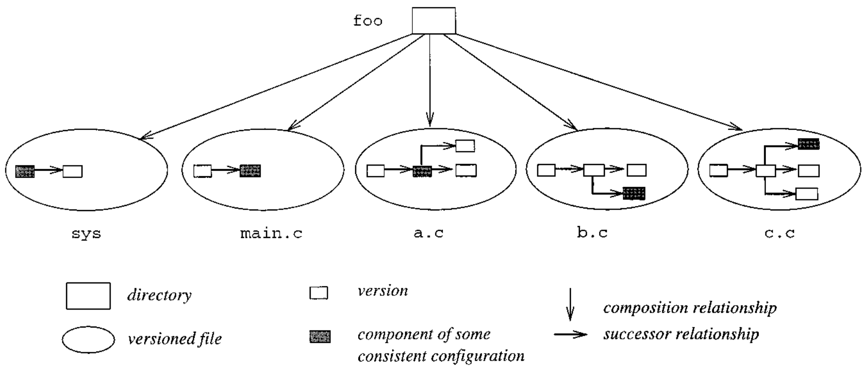


Figure 10. AND/OR graph augmented with successor relationships.

uniform, global version space. It may be regarded as a layer that simplifies selection against a versioned object base. Product versioning sets up a transparent single-version view that hides internally maintained versions of objects and relationships from users and tools. Product space and version space are orthogonal to each other: a given model of the product space can be combined with different models of the version space and vice versa. The integration of product space and version space in the state-based SCM system VOODOO [Reichenberger 1994] is illustrated in Figure 11, where versioned objects, revisions, and variants are organized into orthogonal dimensions.

All change-based systems (e.g., the COV system [Gulla et al. 1991], Aide-de-Camp [Software Maintenance and Development Systems 1990], and DaSC [MacKay 1995]) support product versioning to overcome the limitations of component versioning. In general, a change to a software product may affect multiple components. Since the composition model falls short of recording these cross-dependencies, it is difficult to incorporate a change into a product version by selecting the respective component versions. Therefore in change-based approaches product versions are described in terms of global changes. Note that combination of changes operates at a finer granularity (e.g., text lines) than composition of components.

Product versioning has become more and more popular because of its global view on the software product. A potential problem consists of lacking modularity of the version space. All changes, revisions, and variants are global. For example, if there are two implementation variants of a procedure Sort (e.g., QuickSort and HeapSort), we must introduce an attribute at the global level in order to enable choices between these variants. An approach to structure the version space (in the COV system) is described in Munch [1996].

4.3 Deltas

To represent versions in the object base, deltas are used both at the coarse-grained and the fine-grained levels. The selection of an appropriate delta representation is driven by different requirements to be considered at the physical and logical level, respectively. Storage and run-time efficiency have to be achieved at the physical level. At the logical level, deltas are used to compare versions in such a way that the user is provided with a high-level description of differences. Furthermore, intensional versioning relies on deltas as well. By processing the deltas, a version is constructed that must meet the requirements stated in its intensional description. Accordingly, we distinguish between *physical* and *logical deltas.* Although many SCM systems use a single delta

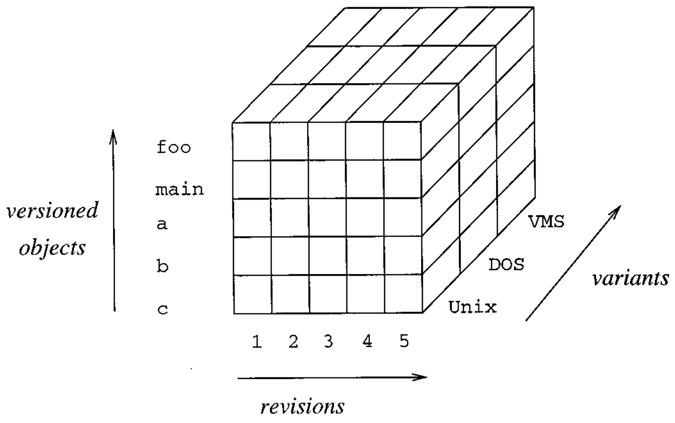


Figure 11. Product versioning.

representation for both purposes [Munch 1993], physical and logical deltas may also be separated (e.g., syntactic merging of program modules stored as text files [Buffenbarger 1995]).

The distinction between directed and symmetric deltas, as illustrated in Figure 2, can be transferred directly to the implementation level. Using *directed deltas* [Tichy 1982b], a version is constructed by applying a sequence of changes to some base version. In the case of *embedded deltas,* all versions are stored in an overlapping manner so that common fragments are shared. Either each version points to its fragments [Fraser and Myers 1986], or the fragments are decorated with *control expressions* for determining the versions in which they are visible (*interleaved deltas* [Rochkind 1975; Leblang and McLean 1985]).

Please note that the internal delta representation is orthogonal to the external version model. For example, interleaved deltas may be used to realize both extensional and intensional versioning as well as both state- and change-based versioning. In the following, the different types of delta representations are discussed in turn.

RCS [Tichy 1985], which is based on directed deltas, reconstructs versions of text files from the most recent version on the main trunk of the version graph by applying backward deltas on the main trunk and forward deltas on the branches. RCS deltas are fine-grained, whereas change-based SCM systems such as PIE [Goldstein and Bobrow 1980] and DaSC [MacKay 1995] employ directed deltas at both the coarse-grained and the fine-grained level. In both systems, the modifications performed in a change are stored in a layer. A product version is composed of a sequence of layers that are stacked on top of each other. (For further details, see the description of PIE in Section 6, in particular, Figure 21.)

Since versions are stored in an overlapping manner, AND/OR graphs can be classified as embedded deltas. As described so far, versions can be combined freely to derive bound configurations from some generic configuration. This "combinability" can be constrained by control expressions (configuration rules) stating, for instance, that a certain version can be selected only when configuring a Unix variant.

Adding control expressions results in an interleaved delta scheme that is realized in SCCS [Rochkind 1975] and DSEE [Leblang and McLean 1985] among others. Note that both systems implement a version model similar to RCS, which is based on directed deltas. In SCCS and DSEE, fragments of text

```
/* common parts */
...
/* dependent on operating system */
#if (OS == Unix)
...
#elif (OS == VMS)
...
#else
...
#endif
...
```

a) fine–grained level

```
family foo
   attributes os : (Unix,VMS,DOS);
   ...
   end
   parts
      root => main;
      files =>
         if os = Unix then unix_fs
         elsif os = VMS then vms_fs
         else dos_fs;
      ...
   end
end
```

b) coarse–grained level

Figure 12. Interleaved deltas: (a) fine-grained level; (b) coarse-grained level.

lines are tagged with the set of versions in which they are included.

SCCS and DSEE support extensional versioning (of text files) but conditional compilation employs interleaved deltas for intensional versioning. For example, Figure 12(a) illustrates conditional compilation in a source file varying with respect to the operating system. Figure 12(b) demonstrates that conditional compilation can be applied at the coarse-grained level as well. The figure contains a cutout of a product description written in the Proteus Configuration Language (PCL [Tryggeseth et al. 1995]). The selected version of the files component depends on the value of the os attribute, which refers to the operating system. In contrast to conditional compilation (*single-source versioning*), different versions of this component are stored in separate files (*version segregation*) [Mahler 1994].

Version segregation is vulnerable to the *multiple maintenance problem* [Babich 1986]: a change to a common fragment must be applied to all versions in turn. This is even the case when the fragment is shared at the physical level (e.g., interleaved deltas in SCCS). Merge tools may reduce the multiple maintenance problem by partly automating change propagation. However, single-source versioning is a more elegant solution since the change needs to be performed only once for all versions, although it may still have to be tested in multiple versions.

On the other hand, editing of source files cluttered with control expressions may confuse the user. *Multiversion editors* overcome this problem by hiding control expressions (see, e.g., P-Edit [Kruskal 1984], MVPE [Sarnak et al. 1988], and COV [Gulla et al. 1991]). A *read filter* selects a single version, removing all control expressions; a *write filter* constrains the set of versions to which the change is applied. For example, a general change in common parts of the source file may be performed by setting up a universal write filter and selecting some arbitrary version by the read filter (e.g., the Unix version). If Unix is also selected for writing, the control expressions of all changed parts are set up so that all changes are specific to the Unix version.

4.4 Relations Between Version Model and Data Model

To conclude this section, we discuss the interplay between product space and version space in terms of its implications for database management. When designing a database management system for software engineering, the designer must decide how to support versioning. In particular, there are several alternatives concerning the relations

```
TYPE Interface IS Object;
   COMMON (* shared by all revisions *)
      system : {unix, hp, vms};
      graphics : {X11, Open_win};
   MODIFIABLE (* revision-specific *)
      belong-to : Conf;
      bug-report : set_of Document;
   IMMUTABLE (* update creates new revision *)
      header : File;
      realization : VERSIONED Realization; (* realization variants *);
END;
```

Figure 13. Definition of a versioned object type.

between the data model and the version model.

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 has been adopted, for example, by PCTE [Oquendo et al. 1989] and CoMa [Westfechtel 1996], which are based on an EER and a graph data model, respectively. Its main advantage is that the data model is kept simple and potentially application-specific extensions are avoided (a widely accepted uniform version model does not yet exist).

However, implementing version management completely on top of a database management system has a number of limitations. For example, there is no support for storing versions efficiently, the transaction manager does not take versioning into account, and so on. Therefore several database management systems provide *predefined classes* for version management (e.g., O_2 [GOODSTEP 1995]). In this case, the data model is still not aware of versioning, but components of the database management system such as storage and transaction manager are modified to support version management efficiently.

Version Model Built into the Data Model. If the data model is extended with versioning, applications can be supported through a data-definition language that provides customized constructs for defining versioned object types. In addition, the query language is modified so that queries against a versioned database can be written in a convenient and natural way. DAMOKLES [Dittrich et al. 1986], EXTRA-V [Sciore 1994], and Adele [Estublier and Casallas 1994] follow this approach.

If the version model is defined on top of the data model, different types are required for versioned objects and their versions. As argued in Estublier and Casallas [1994] and Sciore [1994], this distinction is awkward and complicates both schemata and queries. Figure 13 illustrates how this problem is solved in Adele. In this example, only one object type is defined for a versioned interface. Adele distinguishes between common attributes shared by all revisions, modifiable attributes whose values are revision-specific, and immutable attributes, where each update triggers the creation of a new revision. Furthermore, the attribute realization is declared as versioned, meaning that an interface revision may have multiple realization variants.

Data Model on Top of the Version Model. So far, the version model depends heavily on the data model, being either defined on top of or built into the data model. In a few SCM systems, such as ICE [Zeller and Snelting 1995] and COV [Munch 1996], the version model is completely orthogonal to the data model. This may be achieved through an *instrumentable version engine* that provides a basic delta storage and configuration rules by means of which specific version models may be expressed [Conradi and Westfechtel 1997]. The version engine is not aware of the data model and can thus be combined with any data model (e.g., EER, object-oriented, or simply files).

Both ICE and COV are based on interleaved deltas, but use different logics for control expressions. In ICE, versioning is applied to file-based data. The COV system applies versioning to an EER data model. To this end, a layer that takes care of the consistency constraints inherent in the data model is placed on top of the version engine (e.g., a relationship is visible only when both ends belong to the currently selected version). Thus this architecture differs considerably from the architecture of conventional database management systems where version model and data model are rather entangled.

5. INTENSIONAL VERSIONING

In Section 3.2 we distinguished between extensional and intensional versioning. Extensional versioning is concerned with the reconstruction of previously created versions and requires version identification, immutability, and efficient storage. On the other hand, intensional versioning deals with the construction of new versions from property-based descriptions. Intensional versioning is very important for large version spaces, where a software product evolves into many revisions and variants and many changes have to be combined.

In order to support intensional versioning, an SCM system must provide for both combinability—any version has to be constructed on demand—and consistency control—a constructed version must meet certain constraints. The construction of a version may be viewed as a selection against a versioned object base. The selection is directed by *configuration rules,* which constitute an essential part of a version model, and is performed both in the product space and the version space. Having discussed the interplay between product space and version space in Section 4, we are now ready to elaborate on rule-based version construction, a topic that could have been addressed only partially in Section 3.

5.1 Problem: Combinability Versus Consistency Control and Manageability

Configuration rules are used to configure consistent versions from a versioned object base. Rules are required to address the *combinability* problem. The number of potential versions explodes combinatorially; only a few are actually consistent or relevant. The combinability problem has to be solved in any version model.

For example, the (state-based) *composition model* [Feiler 1991a] applies extensional versioning at the component level; that is, previously constructed component versions are reused. Rule-based construction of configurations realizes intensional versioning at the configuration level; that is, new combinations of component versions are assembled into configurations. Without any constraints, the number of potential configurations is very large. For a product consisting of m modules existing in v versions, there exist v^m potential configurations (i.e., the number of potential configurations grows polynomially in v).

On the one hand, change-based versioning reduces the combinability problem by grouping logically related modifications of multiple components. Thus we do not have to worry about which versions of components actually fit together. However, the selection problem has not disappeared. Rather, it has been moved from the product space to the version space [Munch 1996]. In the case of unconstrained combination of changes (each change may either be applied or skipped), there are 2^c potential configurations for c changes; that is, the number of potential configurations grows exponentially in c.

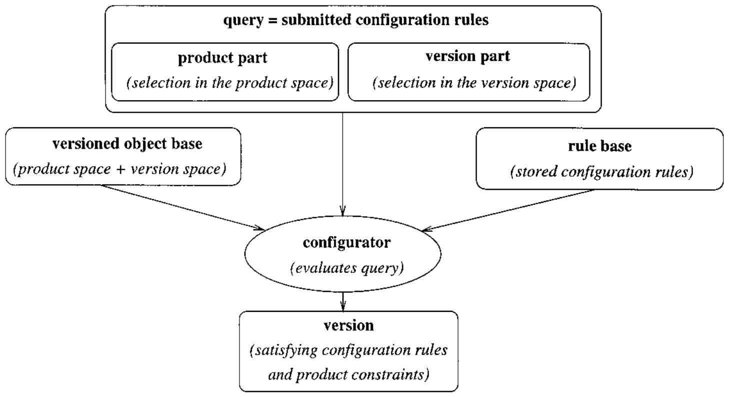

query = submitted configuration rules
product part
(selection in the product space)
version part
(selection in the version space)
versioned object base
(product space + version space)
rule base
(stored configuration rules)
configurator
(evaluates query)
version
(satisfying configuration rules and product constraints)

Figure 14. Intensional versioning.

The challenge of intensional versioning consists of providing for *consistency control* while still supporting combinability. The space of all potential versions is much larger than the space of consistent ones. The problem of consistency control can be addressed both in the version space and in the product space. In the version space, configuration rules are used to eliminate inconsistent combinations; in the product space, the knowledge about software objects, their contents, and their relationships is enriched in order to check and ensure product constraints. SCM systems tend to solve the problem in the version space because they frequently only have limited knowledge of the product space (typically, software objects are represented as text files).

Even if a sophisticated tool for constructing a version is employed, the user must be warned if a new version is created that has never before been configured. Although old versions can be assigned levels of "confidentiality" (e.g., tested or released), a new version cannot be trusted blindly. Therefore the configured version is subject to quality assurance (e.g., testing). Potentially, changes to the constructed version need to be performed (*correction delta*).

5.2 Conceptual Framework for Intensional Versioning

Figure 14 illustrates our conceptual framework for intensional versioning. First let us define the central notion of *configuration rule:* a configuration rule guides or constrains version selection for a certain part of a software product. Thus a configuration rule consists of a *product part,* which determines its scope in the product space, and a *version part,* which performs a selection in the version space (see Section 5.3 for further details).

A *versioned object base* combines product space and version space and stores all versions of a software product, relying, for example, on interleaved deltas. The versioned object base is augmented with a *rule base* of stored configuration rules (e.g., control expressions as shown in Figure 12).

A *query* consists of a set of submitted configuration rules, each composed of a product part and a version part. A *configurator* is a tool that constructs a version by evaluating a query against a versioned object base and a rule base. The constructed version has to satisfy both *version constraints* (e.g., consistent selection of the Unix version) and *product constraints* (e.g., syntactic or semantic consistency). Configurators are discussed further in Sections 5.4 and 5.5.

The configuration process is concerned with the binding of generic references. Often, its result is a bound configuration, but it may also deliver a configuration that is partially generic.

The latter results in a *multistage configuration process.*

Binding can be performed at different points in time. *Static binding* means that the configurator resolves all unbound references before any component is accessed. To this end, the configurator constructs a table that maps each component name to a specific version. In *dynamic binding,* each reference is evaluated on demand only (e.g., when a source file is read by a compiler).

Binding may even be deferred until runtime. In this case, a program is configured dynamically without stopping its execution. A popular example is Java applets, which are loaded dynamically when they are activated through a Web browser. *Dynamic configuration* has been studied in the context of distributed systems (see Kramer [1993] for an overview). Up to now, the relations to SCM have not been investigated thoroughly. Only recently have a few approaches been proposed that apply version-selection techniques from SCM to dynamic configuration [Warren and Sommerville 1995; Schmerl and Marlin 1995]. However, elaborating on this topic goes beyond the scope of this article.

To sharpen the focus, we also refrain from discussing *system building* [Borison 1989]. Although the conceptual framework illustrated in Figure 14 can be applied to both source and derived objects, the problems to be considered are different in the following respects.

—In source version construction, we have a selection problem in both product and version space. The selection must be performed so that the outcome of the configuration process obeys all configuration rules and product constraints. Here nondeterminism may have to be taken into account and the configurator may have to backtrack from wrong selections.

—When constructing derived versions, we primarily have to consider the efficiency and accuracy of the build. Mostly, build rules are deterministic with respect to the result of building. Nondeterminism deals only with the order in which build steps are executed (the build plan imposes only a partial order on the build steps). Some authors advertise the advantages of nondeterministic build rules (e.g., "build some sort program no matter what algorithm and which compiler is used" [Rich and Solomon 1991]). However in many situations it is crucial to control the details of a build without leaving the freedom for nondeterministic choices (e.g., even the functional behavior of a program may depend on whether it is compiled with or without optimization).

Finally, Figure 14 suggests that the rule base is not put under version control. On the other hand, versioning of the rule base is desirable as well because the configuration rules may evolve along with the software product. In the case of our sample software product foo the set of supported operating systems, window systems, and database systems may evolve, as may the constraints on combining these dimensions of variation. Currently, versioning of the rule base is handled at best in a rudimentary way, such as by storing configuration rules in text files that are subject to version control. As argued in Conradi and Westfechtel [1997], simple time-stamped revisions of the rule base may suffice to reconstruct not only old product versions, but also the configuration rules valid at that time. Some further remarks on this topic follow in Section 7.2.

5.3 Configuration Rules

Intensional versioning is driven by configuration rules that are classified in the following. First, *built-in rules* are hardwired into the respective SCM system and cannot be changed by the user. For example, a built-in rule may enforce that at most one version of a software object is contained in any constructed

revision space
(1) `t = max`
(2) `no = 1.1.1.1`
variant space
(3) `os = Unix` $\wedge$ `ws = X11` $\wedge$ `db = Oracle`
(4) $\neg$ `(os = DOS` $\wedge$ `ws = X11)`
change space
(5) `c1 c2 c4`
(6) `c2` $\Rightarrow$ `c1`
(7) `c1` $\otimes$ `c2` $\otimes$ `c3`

Figure 15. Version parts of configuration rules.

configuration. *User-defined rules* are supplied by the user (e.g., "select the latest version before May 22nd").

Version Parts of Configuration Rules. Configuration rules take on different forms depending on how the version space is structured. Figure 15 provides some typical examples that refer to the revision, variant, and change space, respectively. Configuration rules are stated as logical formulas, abstracting from the actual syntax as implemented in different SCM systems.

In the *revision-space* category, configuration rules refer to the time dimension. Rule (1) selects the latest revision by the maximal time stamp. Rule (2) refers to a revision by its number.

In the *variant space,* configuration rules refer to values of variant attributes. Rule (3) identifies a variant by specifying operating system, window system, and database system. Rule (4) expresses a constraint on the combination of attribute values: the X11 window system is not available under the DOS operating system.

In the *change space,* Rule (5) specifies a version in terms of the changes to be applied. Rules (6) and (7) specify further relationships that describe consistent change combinations. Rule (6) states that inclusion of change c2 implies that c1 is included as well (c2 is based on c1). Rule (7) states that changes c1, c2, and c3 are mutually exclusive (operator $\otimes$) (i.e., at most one of these changes may be applied).

(1) `a :  status = checked_out`
(2) `* :  os = Unix` $\wedge$ `ws = X11` $\wedge$ `db = Oracle`
(3) `b.DependsOn* :  name = alpha_release`

Figure 16. Scoping of configuration rules.

Product Parts of Configuration Rules. So far we have discussed only the version parts of configuration rules. The product part describes the scope of a configuration rule in the product space. In Figure 16, a configuration rule is written in the form $p : v$, where p and v denote product part and version part, respectively. Rule (1) applies to a single module a (local rule) and selects the version checked out by the current user. The star in Rule (2) indicates global application to all modules so that the same variant is selected throughout the whole product. The product part of Rule (3) denotes all modules reachable from b by a reflective and transitive closure over relationships of type DependsOn; that is, the rule applies to b and to all modules on which b depends transitively.

Ordering of Configuration Rules. Configuration rules can be ordered in *strictness classes.* A *constraint* is a mandatory rule that must be satisfied. Any violation of a constraint indicates an inconsistency. For example, Rule (1) in Figure 17 ensures that all selected versions belong to a given variant. A *preference* is an optional rule that is applied only when it can be satisfied. For example, Rule (2) states that a checked-out version is selected provided it is available. Finally, a *default* is also an optional rule, but is weaker than a preference: a default rule is applied only when no unique selection could be performed otherwise. For example, Rule (3) selects the most recent version as the default.

Strictness classes determine the order in which configuration rules are evaluated (constraints $\rightarrow$ preferences $\rightarrow$ defaults). In addition, rules may be given *priorities*. Rules with high priorities are considered before low-priority rules. A priority may be assigned explicitly or

constraint
(1) * : $\texttt{os = Unix} \wedge \texttt{ws = X11} \wedge \texttt{db = Oracle}$
preference
(2) * : `status = checked_out`
default
(3) * : `t = max`

Figure 17. Strictness classes.

may be defined implicitly by textual ordering. Priorities may be combined with strictness classes such as by assigning priorities to preferences [Lavency and Vanhoedenaghe 1988]. However they can also be used without strictness classes. For example, in ClearCase [Leblang 1994] rules are evaluated according to their textual order until a (unique) match is found.

Finally, we may distinguish between *stored* and *submitted* configuration rules. The rules given in Figure 17 would typically be submitted in a query in order to specify properties requested by the user. Rule (4) in Figure 15 is an example of a stored configuration rule expressing a constraint that must be fulfilled by all configured versions.

Relations Between Version Graphs and Configuration Rules. To conclude this section, we discuss briefly how version graphs and configuration rules are related in different SCM systems:

—*Configuration rules on top of version graphs.* This is the classical solution, called *composition model* in Feiler [1991a]. Version graphs are maintained for components (extensional versioning); configurations are constructed by selecting component versions with the help of configuration rules (intensional versioning). The composition model is realized, for example, in DSEE [Leblang and McLean 1985] and ClearCase [Leblang 1994].

—*Version graphs on top of configuration rules.* This inverted approach is followed by a few research prototypes such as ICE [Zeller 1995] and COV [Munch 1996]. Both systems rely on a flexible and powerful base layer providing for delta storage (interleaved deltas) and configuration rules. Any desired version model may be realized on top of configuration rules. In the case of version graphs, revision chains and branches may be expressed by implication and mutual exclusion, respectively (see Rules (6) and (7) in Figure 15).

5.4 Configurators: Tools for Evaluating Configuration Rules

A configurator constructs a version by evaluating configuration rules against a versioned object base. Construction can be performed in different computational frameworks. In a *functional framework,* intensional versioning is modeled by applying a (potentially partial) function q (query) to its arguments $a_1 \ldots a_n$; that is, a version v is constructed by evaluating the expression $q(a_1, \ldots, a_n)$. This approach assumes that version construction delivers a deterministic result (i.e., version selection is unique).

In a *rule-based framework,* version construction is modeled as the evaluation of a query against a *deductive database* [Ramamohanarao and Harland 1994; Ramakrishnan and Ullman 1995]. The deductive database consists of a versioned object base and a rule base containing stored configuration rules. Since a query may not specify the version to be constructed in a unique way, a rule-based configurator has to cope with nondeterminism. Ambiguous choices can be resolved either automatically or interactively. In any case, the configurator needs to explore a search space of potential solutions. To this end, it may use different search strategies such as "depth-first" or "breadth-first."

Configurators may be classified not only with respect to the computational paradigm (functional, rule-based) but also with respect to the underlying version model (state- or change-based). This results in four combinations, as described in the following.

State-Based Functional Configurators. A state-based functional configurator operates on a versioned object base typically represented by interleaved deltas. Conditional compilation in C [Kernighan and Ritchie 1978] and configuration construction at the coarse-grained level in PCL [Tryggeseth et al. 1995] may be cited as examples.[5] The versioned object base is usually traversed in a "context-free" manner (e.g., sequences of text lines in conditional compilation and trees of components in PCL). Configuration rules take the form of control expressions as shown, for example, in Figure 12. The configurator is supplied with a tuple of attribute values ($a_1, \ldots, a_n$) and constructs some version v. Control expressions are evaluated against this tuple to select the relevant components of v. When applied to a composite object co, expressions may also be used to transform ($a_1, \ldots, a_n$) into a new tuple ($a'_1, \ldots, a'_m$), which is then employed recursively to configure co. This means that attribute values are transformed and propagated through the composition hierarchy. In this way, it is possible to enforce certain constraints (e.g., selection of the same variant throughout the whole configuration process). However rule-based configurators are in general more powerful in enforcing constraints or detecting inconsistencies.

State- and Rule-Based Configurators. A state- and rule-based configurator supports nondeterminism through an appropriate search strategy (e.g., depth-first search with backtracking). It typically operates on some AND/OR graph that is traversed along "context-sensitive" relationships (e.g., dependencies between modules in Adele [Estublier 1985]; see also Figure 8). For each OR node reached in the course of the configuration process, a successor is selected with the help of configuration rules. As version selections are performed, constraints are added that narrow down further choices. For example, if version uniqueness is required, we may not select a different successor each time a certain OR node is visited. Variant selection serves as another example: after having selected a certain variant of an operating system, we must ensure that this selection is performed consistently for all further nodes reached in the course of the configuration process. If this turns out to be impossible, the configurator has to backtrack from the wrong selection.

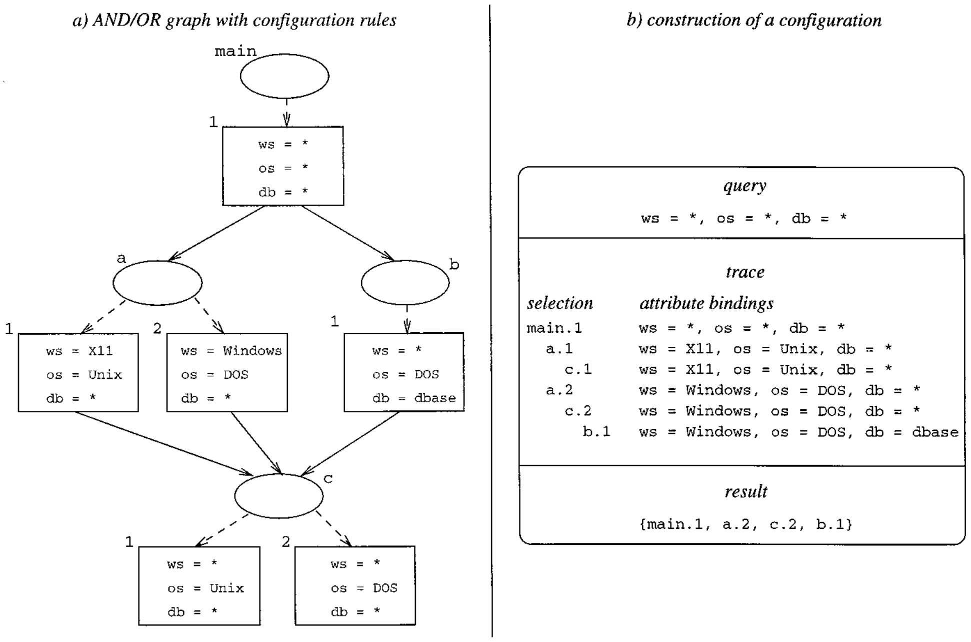
A simple example is given in Figure 18(a) which shows an AND/OR graph whose AND nodes (versions) are annotated with configuration rules.[6] For each version, a triple of attribute values denotes the variants to which it belongs; * stands for any value. a depends on the window system, b on the database system, and c on the file system; main can be used with any variant. Figure 18(b) shows a query that leaves all attribute values unspecified. Below, a sample trace of the configuration process is given. After selection of a.1, the os attribute is bound to Unix. Thus the Unix version of c is selected. However, selection of b fails and triggers backtracking.

Change-Based Functional Configurators. A change-based functional configurator is supplied with a sequence of changes. Internally, the object base may be stored using either interleaved or directed deltas (e.g., Aide-de-Camp [Software Maintenance and Development Systems 1990] and PIE [Goldstein and Bobrow 1980], respectively). Conceptually, the changes are applied in order to some baseline. Inconsistencies are detected when a change operation fails, for example, insertion of a text line at an undefined location.

Change- and Rule-based Configurators. A change- and rule-based con-

[5] Please recall that conditional compilation can be used for change-based versioning as well (see Section 3.5).

[6] Note that only variants are considered in this example (no revisions).



query

ws = *, os = *, db = *

trace

selection	*attribute bindings*
main.1	ws = *, os = *, db = *
a.1	ws = X11, os = Unix, db = *
c.1	ws = X11, os = Unix, db = *
a.2	ws = Windows, os = DOS, db = *
c.2	ws = Windows, os = DOS, db = *
b.1	ws = Windows, os = DOS, db = dbase

result

{main.1, a.2, c.2, b.1}

Figure 18. Rule-based construction of a configuration: (a) AND/OR graph with configuration rules; (b) construction of a configuration.

figurator differs from its functional counterpart by considering stored configuration rules constraining change combinations. Configuration rules can be used to detect inconsistent change combinations; furthermore, they can be employed in a constructive manner to complete a partial specification. This is done, for example, in the COV system [Munch 1996]. Consider the mutual exclusion rule (7) in Figure 15. If the user insists on applying both c1 and c2, the configurator reports an inconsistency. Alternatively, the configurator may automatically disable change c2 (and c3) after c1 has been selected.

5.5 Merge Tools

So far, we have distinguished between state-based and change-based configurators. Although state- and change-based SCM systems may differ considerably at first glance, the borderline is in fact rather fuzzy.[7] In particular, state-based SCM systems often offer merge tools [Buffenbarger 1995] to combine versions/changes. Although mostly used in state-based systems, they have been applied in change-based systems as well.

Merge tools combine versions or changes. They may be classified as follows (Figure 19).

—*Raw merging* simply applies a change in a different context. For example, in Figure 19(a) change c2 was originally performed independently of change c1 and is later combined with c1 to produce version v4. Raw merging is supported by SCCS [Rochkind 1975] among others. It was later generalized in change-based SCM systems such as Aide-de-Camp [Software Maintenance and Development Sys-

[7] This was already demonstrated in the discussion of the relationships between version graphs and change-based versioning (Section 3; see Figure 6).

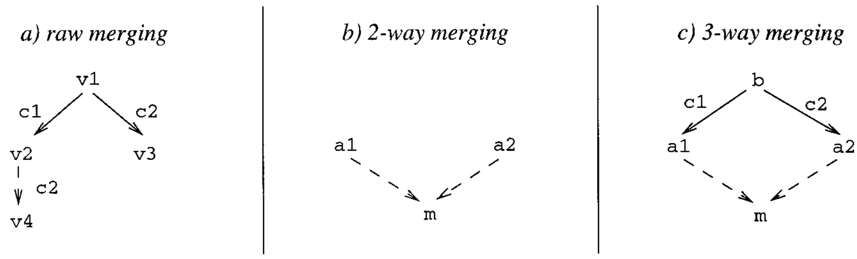


Figure 19. Types of merging: (a) raw merging; (b) two-way merging; (c) three-way merging.

tems 1990] and the COV system [Munch et al. 1993].

—A *two-way merge tool* (Figure 19(b)) compares two alternative versions a1 and a2 and merges them into a single version m. To this end, it displays the differences to the user who has to select the appropriate alternative. A two-way merge tool can merely detect differences, and cannot resolve them automatically.

—To reduce the number of decisions to be performed by the user, a *three-way merge tool* (Figure 19(c)) consults a common baseline b if a difference is detected. If a change has been applied in only one version, this change is incorporated automatically. Otherwise, a conflict is detected that can be resolved either manually or automatically (the latter is not recommended). Notably, the change-based system Aide-de-Camp offers a three-way merge tool in addition to raw merging.

In comparing raw merging and three-way merging, we have to distinguish between inconsistencies and conflicts:

—In raw merging, a change *c* is applied in a different context. A change is a sequence of change operations, say $op_1 \ldots op_m$. If any op_i fails (e.g., because it is applied to a nonexistent object), there is an *inconsistency*.

—In addition to inconsistencies, three-way merging can detect *conflicts,* that is contradictory changes. Three-way merging attempts to combine two sequences of change operations. A conflict arises if two operations do not commute (e.g., contradictory changes to the name of a procedure). In raw merging, one change wins and the other is overridden.

Merge tools can be characterized by the semantic level at which merging is performed (i.e., their knowledge about the product space):

—*Textual merging* is applied to text files [Adams et al. 1986]. Almost all commercial SCM systems support textual merging [Rigg et al. 1995]. Although we can expect only an arbitrary text file as the result of the merge (and, e.g., not a legal C program) and only physical conflicts can be detected, textual merging seems to yield good results in practice [Leblang 1994]. In particular, it works well when small, local changes to large well-structured programs are combined and changes have been coordinated beforehand so that semantic conflicts are unlikely to occur.

—*Syntactic merging* exploits the context-free (or even context-sensitive) syntax of the versions to be merged. Therefore, it can guarantee a syntactically correct result and can perform more intelligent merge decisions. However, syntactic merging has been realized only in a few research prototypes [Buffenbarger 1995; Westfechtel 1991].

—*Semantic merging* takes the semantics of programs into account [Berzins 1994, 1995; Binkley et al. 1995; Horwitz et al. 1989]. Semantic merge tools perform sophisticated analyses in order to detect conflicts between changes. However, it is a hard problem to come up with a definition of semantic conflict that is neither too strong nor too weak (and is decidable). Furthermore, the merge algorithms developed so far are applicable only to simple programming languages (not C or C++). For these reasons, semantic merge tools have not (yet?) made their way into practice.

Finally, *operation-based merging* [Lippe and van Oosterom 1992] comes up with a general algorithm that makes no assumptions about the product space. The algorithm takes two sequences of change operations and combines them into a single sequence, detecting both inconsistencies and conflicts. However, the application of this algorithm is complex because a huge search space of potential merged operation sequences must be considered. Because of its generality, the merge tool cannot rely on any hints as to which operation should be appended next and where conflicting operations are positioned in the input sequences.

Note that merge tools try to detect conflicts in the product space (by detecting noncommuting operations). Alternatively, conflict detection can be performed in the version space to some extent. This approach is followed in the COV system [Munch 1996], where configuration rules are used to constrain change combinations. Conflicting merges can be excluded by constraints of the form $c_1 \otimes c_2$ (mutual exclusion of c_1 and c_2). Once such a constraint has been set up, attempts to combine c_1 and c_2 can be detected as erroneous before performing the actual merge. However, before c_1 and c_2 are known to conflict, it is frequently necessary to combine them (by raw merging) and test the result.

6. VERSION MODELS IN SCM SYSTEMS

After having characterized version models in general, we now take a look at concrete systems. We draw a picture of the SCM landscape by describing the contributions of a representative selection of influential SCM systems. These descriptions are based primarily on the published scientific literature. Although the following survey includes commercial systems in addition to research prototypes, we do not provide a comprehensive overview of SCM systems available in the commercial marketplace (see Rigg et al. [1995] for such an overview). Furthermore, our main goal is to illustrate different version models rather than to evaluate the functionalities provided by SCM systems.

The survey focuses on the core issues of versioning discussed in Sections 2 through 5. Further topics such as system building, cooperation support, workspace management, and distribution are mentioned briefly at best. As a consequence, influential systems whose main merits lie in these fields are not included. For example, Make [Feldman 1979], Odin [Clemm 1995], and CAPITL [Adams and Solomon 1995] are concerned with system building, and NSE [Adams et al. 1989] has contributed to workspace management (hierarchy of workspaces) and cooperation support (optimistic concurrency control).

6.1 Overview

We have selected more than 20 SCM systems that vary widely in their underlying version models. The evolution graph in Figure 20 illustrates the evolution of SCM since the early '70s. Each node corresponds to a specific system and briefly describes its original contribution(s). Incoming edges express the most important influences of previous systems.

Before delving into the details, let us make some global remarks.

—Initially, SCM was supported through isolated tools covering specific aspects

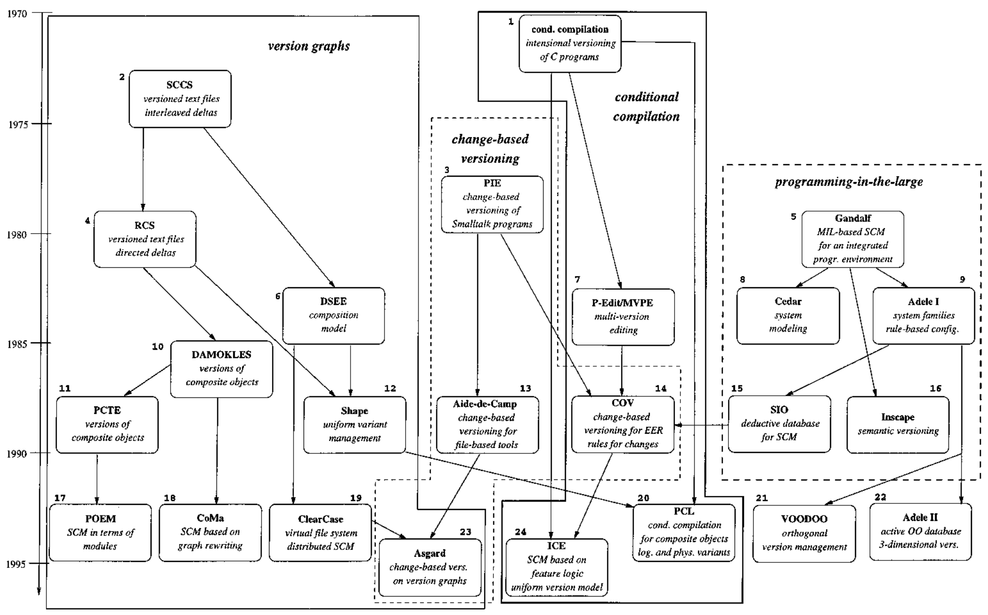

1970
1975
1980
1985
1990
1995
version graphs
conditional compilation
change-based versioning
programming-in-the-large
1 cond. compilation intensional versioning of C programs
2 SCCS versioned text files interleaved deltas
3 PIE change-based versioning of Smalltalk programs
4 RCS versioned text files directed deltas
5 Gandalf MIL-based SCM for an integrated progr. environment
6 DSEE composition model
7 P-Edit/MVPE multi-version editing
8 Cedar system modeling
9 Adele I system families rule-based config.
10 DAMOKLES versions of composite objects
11 PCTE versions of composite objects
12 Shape uniform variant management
13 Aide-de-Camp change-based versioning for file-based tools
14 COV change-based versioning for EER rules for changes
15 SIO deductive database for SCM
16 Inscape semantic versioning
17 POEM SCM in terms of modules
18 CoMa SCM based on graph rewriting
19 ClearCase virtual file system distributed SCM
20 PCL cond. compilation for composite objects log. and phys. variants
21 VOODOO orthogonal version management
22 Adele II active OO database 3-dimensional vers.
23 Asgard change-based vers. on version graphs
24 ICE SCM based on feature logic uniform version model

Figure 20. Evolution graph of SCM systems.

(e.g., SCCS [Rochkind 1975] for versioning of source objects). Later on, integrated systems were developed that combined the functionalities of individual tools into a coherent environment (e.g., DSEE [Leblang and McLean 1985] or ClearCase [Leblang 1994]).

—The pioneers addressed many problems in an ad hoc manner. Subsequent systems addressed topics such as intensional versioning in a more systematic and general way (e.g., compare version selection by checkout options to full-fledged logic-based approaches such as ICE [Zeller 1995]).

—Many fundamental ideas are rather old. For example, although change-based versioning has been attracting significant attention only recently, the idea can be traced back at least to the PIE system (change-based versioning of Smalltalk programs [Goldstein and Bobrow 1980]), which was already developed in the late '70s.

—Although early systems/tools were implemented on top of the file system, more recent systems increasingly use database technology. This is demonstrated by the Adele system, for example, which evolved from a file-based system with an ad hoc hardwired data model [Estublier 1985] into an active object-oriented database system supporting historical, logical, and cooperative versioning (revisions, variants, and workspaces, respectively) [Estublier and Casallas 1994].

The evolution graph is traversed in chronological order in Section 6.3. Instead of reading all system descriptions sequentially, the reader may focus on specific systems. Furthermore, the edges of the evolution graph may serve as "hypertext links."

To ease orientation, we have clustered SCM systems into families such as "version graphs," "conditional compilation," "change-based versioning," and "programming-in-the-large" (systems for configuring modular programs). These families should be viewed just as examples. They are neither disjoint (e.g., Asgard supports change-based versioning

Table I. Classification of SCM Systems (I)

		general					*product space*						*version space*					
		environment ⊗			*object management ⊗*		*domain ⊗*		*granularity **		*(coarse gr.) relationships **		*structure **		*version set **		*version specification **	
		tool kit	*lang. based*	*struct. oriented*	*file system*	*data base system*	*specific*	*general*	*coarse*	*fine*	*composition*	*dependencies*	*version graph*	*grid*	*extensional*	*intensional*	*state-based*	*change-based*
1	**cond. comp.**	x			x		x			+	▨	▨		+		+	+	+
2	**SCCS**	x			x			x	+	+	+		+		+		+	
3	**PIE**		x		x		x		+	+	+	+		+	+	+		+
4	**RCS**	x			x			x	+	+	+		+		+	+	+	
5	**Gandalf**			x		x	x		+	+	+	+	+	+	+	+	+	
6	**DSEE**	x			x			x	+	+	+	+	+		+	+	+	
7	**P Edit/MVPE**	x			x			x		+	▨	▨		+		+	+	+
8	**Cedar**		x		x		x		+		+	+		+	+		+	
9	**Adele I**	x			x		x		+		+	+	+	+	+	+	+	
10	**DAMOKLES**			x		x		x	+	+	+	+	+		+		+	
11	**PCTE**	x				x		x	+		+	+	+		+		+	
12	**Shape**	x			x			x	+	+	+	+	+		+	+	+	
13	**Aide de Camp**	x			x			x	+	+	+	+		+	+	+		+
14	**COV**	x				x		x	+	+	+	+		+	+	+	+	+
15	**SIO**	x				x	x		+		+	+		+	+	+	+	
16	**Inscape**		x		x		x		+	+	+	+		+	+		+	
17	**POEM**		x			x	x		+		+	+	+		+		+	
18	**CoMa**			x		x		x	+	+	+	+	+		+		+	
19	**ClearCase**	x			x			x	+	+	+	+	+		+	+	+	
20	**PCL**	x			x			x	+		+			+	+	+	+	
21	**VOODOO**	x			x			x	+	+	+			+	+		+	
22	**Adele II**	x				x		x	+	+	+	+	+	+	+	+	+	
23	**Asgard**	x			x			x	+		+		+	+	+	+	+	+
24	**ICE**	x			x			x	+	+	+	+		+	+	+	+	+

on top of version graphs), nor do they cover all SCM systems.

6.2 Taxonomy-Based Classification

In this section, we classify SCM systems according to a taxonomy derived from Sections 2 through 5. This taxonomy provides a much more detailed and systematic classification than the categories introduced in the previous section. Furthermore, we claim that this classification is orthogonal. Indeed, an important contribution of this article is to distinguish among different aspects that used to be mingled together.

For example, according to our definition, change-based versioning just means that versions are described in terms of changes relative to some baseline. On the other hand, the term has other connotations, resulting from the models actually realized in change-based systems:

—Intensional versioning: as explained in Section 3.5, change-based versioning is not necessarily intensional (see the discussion of change packages). The converse is also not true (see, e.g., the classical composition model, which is state-based).

—Product versioning: although all change-based systems happen to support global changes, there is no inherent reason why change-based versioning could not be applied at the component level. Conversely, there are state-based systems (e.g., VOODOO [Reichenberger 1994]) that maintain versions at the product level.

SCM systems are classified in Tables I and II. These tables may be crossreferenced while reading the system descriptions in the next section. For each system, there is a corresponding row in the table. The columns are organized in a three-level hierarchy. The first level defines *categories* ("general," "product space," etc.). Each category serves to group multiple *features* each of which corresponds to one dimension of the classification scheme ("environment," "object management," etc.).

A feature may have *values* from some enumeration type (e.g., "tool-kit," "language-based," "structure-oriented"). Each feature is either *single-* or *multivalued.* In the first case, the feature is annotated with ⊗ and its value is indicated

Table II. Classification of SCM Systems (II)

		interplay of product and version space								*intensional versioning*						
		*selection order in AND/OR graphs **			*external granularity of versioning ⊗*			*(fine–grained) deltas ⊗*		*computational paradigm ⊗*		*classes of configuration rules **			*configurator **	
		product first	*version first*	*inter-twined*	*product*	*compo-nent*	*total*	*embed-ded*	*directed*	*functio-nal*	*rule-based*	*con-straint*	*prefe-rence*	*default*	*automa-tic*	*interac-tive*
1	**cond. comp.**	▨	▨	▨		x		x		x		▨	▨	▨	+	
2	**SCCS**	+				x		x		▨	▨	▨	▨	▨	▨	▨
3	**PIE**	▨	▨	▨	x				x	x		▨	▨	▨	+	
4	**RCS**	+				x			x		x	▨	▨	▨	+	
5	**Gandalf**			+			x	▨	▨		x	+	+	+	+	
6	**DSEE**	+					x	x			x	▨	▨	▨	+	
7	**P Edit/MVPE**	▨	▨	▨		x		x		x		▨	▨	▨	+	
8	**Cedar**		+				x	▨	▨	▨	▨	▨	▨	▨	▨	▨
9	**Adele I**			+			x	▨	▨		x	+	+	+	+	+
10	**DAMOKLES**	+	+	+			x	x		▨	▨	▨	▨	▨	▨	▨
11	**PCTE**		+				x	▨	▨	▨	▨	▨	▨	▨	▨	▨
12	**Shape**			+			x		x		x		+	+	+	
13	**Aide de Camp**	▨	▨	▨	x			x		x		▨	▨	▨	+	
14	**COV**	▨	▨	▨	x			x			x	+	+	+	+	+
15	**SIO**	+				x		▨	▨		x	+	+		+	
16	**Inscape**			+		x		▨	▨	▨	▨	▨	▨	▨	▨	▨
17	**POEM**		+				x	▨	▨	▨	▨	▨	▨	▨	▨	▨
18	**CoMa**		+				x		x	▨	▨	▨	▨	▨	▨	▨
19	**ClearCase**			+			x	x			x	▨	▨	▨	+	
20	**PCL**	▨	▨	▨			x	▨	▨	x		▨	▨	▨	+	+
21	**VOODOO**	▨	▨	▨	x				x	▨	▨	▨	▨	▨	▨	▨
22	**Adele II**	+	+	+			x		x		x	+	+	+	+	+
23	**Asgard**			+			x	▨	▨		x	▨	▨	▨	+	
24	**ICE**	▨	▨	▨			x	x			x	+	+		+	+

by an x sign in the respective column. In the second case, we use * to annotate the feature and a + for each of its values.

Some features are defined only *partially*; they are not applicable to all SCM systems. For example, "selection order" is only meaningful for systems based on AND/OR graphs, and not all systems support intensional versioning. Undefined values are indicated by hatched entries.

The tables are explained briefly in the following, ordered by categories.

General. Following Dart et al. [1987], we distinguish among tool-kit, language-based, and structure-oriented *environments*; most SCM systems belong to the first class. For *object management* of the versioned object base, a file system or a database system may be used.

Product Space. The *domain* is specific if the SCM system deals with specific types of software objects; otherwise, it is general. All systems classified as specific deal with certain kinds of programs; however the underlying concepts may still be fairly general (e.g., conditional compilation). Concerning *granularity*, there are some systems that do not deal with the fine-grained level (e.g., PCTE), and a few that do not consider the coarse-grained level (e.g., the multiversion text editor MVPE). The latter have undefined entries for *coarse-grained relationships*. All other systems support composition relationships, but some of them do not take dependencies into account.

Version Space. The *structure* feature refers to the way the version space is modeled. In some cases (e.g., Gandalf), both version graphs and grids are suitable for representing the version space. Similarly, extensional and intensional versioning are nonexclusive (*version set*). Virtually all systems support extensional versioning (i.e., reconstruction of old versions, conditional compilation being a remarkable exception), but some do not consider intensional versioning. Finally, most SCM systems are state- rather than change-based (*version specification*). A few systems provide base mechanisms for both paradigms (e.g., ICE and COV).

Interplay of Product and Version Space. The *selection order* (during the configuration process) can be applied only to SCM systems based on AND/OR graphs; in particular, it is not applied to systems derived from conditional compilation. Note that the selection order follows from the topology of AND/OR graphs, which can be defined freely in some systems (e.g., DAMOKLES). The *external granularity* is classified into product, component, and total versioning. Finally, the *deltas* feature refers to the efficient representation of atomic software objects only, that is, sharing at the coarse-grained level is not considered.

Intensional Versioning.[8] With respect to the underlying computational paradigm, we distinguish between *functional* configurators (e.g., conditional compilation) and *rule-based* configurators (e.g., DSEE or ClearCase). *Classes of configuration rules* refer to the strictness classes constraints, preferences, and defaults and are applied only if the SCM system distinguishes among different strictness classes. Finally, all SCM systems supporting intensional versioning (must) provide *automatic* configurators. In some systems, the configurator may also be used in *interactive* mode (e.g., ICE or Adele).

6.3 Descriptions of SCM Systems

In the following, each system is described briefly in turn. The system descriptions are not organized according to the tables presented in the previous section (proceeding through the table entries would be rather boring). Instead, we focus on the specific contributions of each system and its relations to other systems.

6.3.1 *Conditional Compilation.* Conditional compilation supports intensional versioning at the fine-grained level and has become popular particularly in conjunction with the C programming language [Kernighan and Ritchie 1978]. Although it was originally developed for a specific domain, the underlying idea is fairly general. The source text is interspersed with *preprocessor directives* that refer to the values of preprocessor variables (see Figure 12). Thus conditional compilation uses interleaved deltas with visible control expressions. A specific source version is constructed by the preprocessor by filtering out all fragments whose control expressions evaluate to false. Source version construction follows a functional paradigm (no nondeterminism/backtracking). Note that conditional compilation is a low-level and general mechanism on top of which different version models can be implemented (both state- and change-based ones).

6.3.2 *SCCS.* SCCS [Rochkind 1975] manages *versions* of *text files*. (Binaries can also be stored, but delta storage is not supported.) Versions are arranged in a tree. To work on a version, it is physically copied into a workspace (directory). When the user is done with his or her changes, the modified version is checked back into the SCCS repository. SCCS uses *interleaved deltas* to store versions in a space-efficient manner. However, in contrast to conditional compilation, this data structure is hidden. Thus there are interleaved deltas at two levels when a C file with preprocessor directives is stored under SCCS control. Although SCCS primarily supports state-based versioning, it does provide some low-level commands for controlling delta applications (and even fixing deltas) in a change-based fashion.

6.3.3 *PIE.* An early approach to change-based versioning has been developed at XEROX PARC [Goldstein and Bobrow 1980]. PIE manages configurations of Smalltalk programs that are internally represented by graph-like data structures. Changes are considered global and may affect multiple components of a software product (product versioning). Each change is placed in a *layer*, and layers are aggregated into *contexts* that act as search paths through the layers. (DaSC [MacKay 1995] is based on a similar approach.)

[8] Note that merge tools (Section 5.5) were not taken into account here.

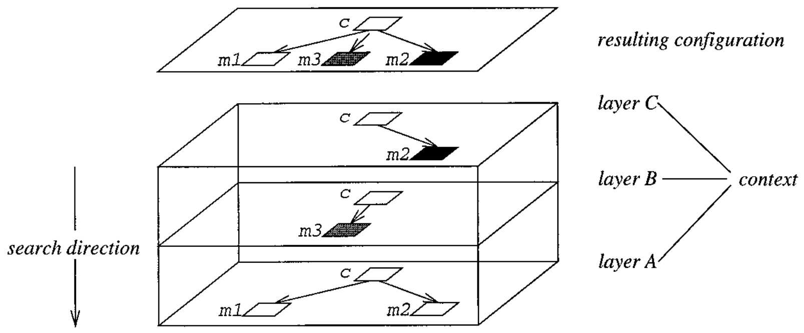


Figure 21. Layers and contexts in PIE.

This is illustrated in Figure 21, where layers contain different implementations of methods m1, m2, m3 exported by some class c.

When constructing a context, there are two degrees of freedom: each layer may either be included or omitted; and the included layers can be arranged in any sequential order. PIE provides relationships to document conditions for the combination of layers. For example, A depends_on B implies that each context containing A should include B as well. However PIE does not enforce any constraints. Rather, the documented relationships are merely used to warn the user of possible inconsistencies.

6.3.4 *RCS.* RCS [Tichy 1982b, 1985] differs from its predecessor SCCS in several ways. RCS stores versions of text files using *directed deltas*. The latest revision on the main trunk can be accessed directly (and therefore efficiently), whereas all other revisions are reconstructed by applying backward and forward deltas. Furthermore, RCS provides a set of built-in attributes such as version status and symbolic name. In particular, symbolic names may be attached to all components belonging to a consistent configuration. In this way, symbolic names define threads through the version graphs and make reconstruction of configurations easier (see Figure 10). However, support for intensional versioning is rather limited (options of checkout commands referring to version attributes); in particular, the product structure is selected first so that structural versioning cannot be modeled.

6.3.5 *Gandalf.* The Gandalf project [Habermann and Notkin 1986] was dedicated to the generation of structure-oriented software development environments based on *abstract syntax trees*. Gandalf C is an environment instance that supports C at the programming-in-the-small level. Programming-in-the-large and version control are handled by the SVCE subsystem [Kaiser and Habermann 1983], which is based on the Intercol *module interconnection language* [Tichy 1979]. Each module has a unique and immutable interface and potentially multiple realization variants, each of which evolves into a sequence of revisions. Thus revisions and variants are separated rather than intermixed (SCCS, RCS). A realization of an interface can be provided either by writing C code, or can be composed of other modules that jointly provide the resources listed in the export interface (compositions). Gandalf enforces syntactic consistency of all interfaces and realizations deposited into the public database. Therefore it can guarantee syntactic consistency of all constructed configurations.

Gandalf supports intensional construction of source configurations through a configurator that performs intertwined AND/OR selections. The

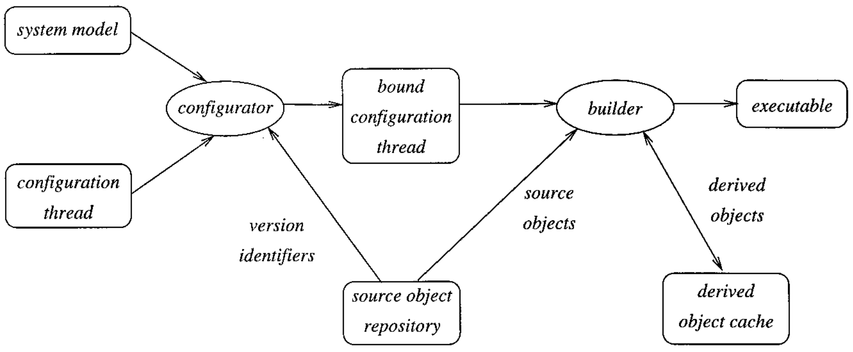


Figure 22. DSEE.

configurator is driven by simple *configuration rules* that are classified into constraints, preferences, and defaults. Constraints and preferences are attached to imports or compositions and select specific variants/revisions. If no selection rules are given, the standard variant/revision is selected by default.

6.3.6 *DSEE.* DSEE [Leblang and McLean 1985; Leblang and Chase 1984; Leblang et al. 1988] integrates functions that were previously provided independently by tools such as Make and SCCS/RCS. Furthermore, DSEE supports rule-based construction of source configurations and improves system building by maintaining a cache of derived objects, using more accurate difference predicates than Make and parallelizing builds over a network of workstations.

Source code versions are arranged in version graphs as shown in Figure 4. A configuration is specified by a system model and a thread through the version graphs of components (Figure 22). The *system model* describes a software system in terms of source objects and their relationships; furthermore, it also contains build rules. Versions are not referenced in the system model. Rather, they are selected by configuration rules in the *configuration thread*. Rules are ordered sequentially according to their priorities. The DSEE configurator selects the product structure first,[9] which is described in the system model, and adds version bindings, resulting in a *bound configuration thread* that is used for system building.

6.3.7 *P-Edit/MVPE.* P-Edit [Kruskal 1984] and its successor MVPE [Sarnak et al. 1988] support *simultaneous editing* of *multiple versions* of a text file. A text file is composed of fragments (sequences of words). As in conditional compilation, control expressions are used to determine the visibilities of fragments. Unlike conditional compilation, P-Edit and MVPE hide these control expressions when the text file is displayed. Furthermore, control expressions are maintained automatically. A write filter (*edit set*) controls which versions will be affected by a change, a read filter (*view*) selects a single version to be displayed to the user. The version space is modeled as a grid.

In Figure 23, the table on the left-hand side lists all versions determined by the attributes os, ws, and db. A view corresponds to a single row of the table, and an edit set is specified in a query-by-example style with the help of regular expressions (e.g., the edit set in Fig-

[9] However, it should be noted that the system model itself is version-controlled: thus, a version of the system model is selected in an initial step not shown in Figure 22.

versions

os	ws	db
DOS	Windows	dbase
Unix	X11	Oracle
Unix	SunViews	Informix
Unix	X11	Informix
...	...	...

edit set

os	ws	db
Unix	*	*

view

os	ws	db
Unix	X11	Informix

Figure 23. MVPE.

ure 23 denotes all Unix versions). When displaying a view, the editor highlights fragments belonging to all versions in the edit set (a change should be confined to these fragments).

6.3.8 *Cedar.* The Cedar environment [Teitelman 1984] supports programming in Cedar, a modular language developed at XEROX PARC. Modules of Cedar programs contain generic references to other modules. Notably, a client module may use multiple realization versions of an imported interface simultaneously. The Cedar System Modeler [Lampson and Schmidt 1983b, 1983a] takes care of compiling and linking Cedar programs. A *system model* in Cedar differs considerably from what is also called system model in DSEE. In particular, a Cedar system model binds generic references to specific versions. System models describe configuration versions in terms of immutable source files and therefore roughly correspond to bound configuration threads in DSEE (extensional versioning, "version-first" selection).

6.3.9 *Adele I.* In contrast to Gandalf, Adele I [Estublier 1985; Belkhatir and Estublier 1986] was developed in the mid-'80s for *tool-kit environments* (a posteriori integration of file-based tools). Adele generalizes Gandalf's approach to configuring modular programs in several ways. Whereas each Gandalf module has a unique interface, an Adele *family* may have multiple versions of its interface. Each interface version and its realization variants and revisions correspond to a module in Gandalf. Like Gandalf, Adele explicitly distinguishes three different classes of rules, namely, *constraints*, *preferences*, and *defaults*. However, the configuration rules are more sophisticated and allow attribute-based version selection. Using these rules, short intensional descriptions can be given for large and complex configurations.

6.3.10 *DAMOKLES.* DAMOKLES [Dittrich et al. 1986; Gotthard 1988] was among the first systems applying database technology to SCM. DAMOKLES is based on an *EER data model* featuring composite objects, structural inheritance at both type and instance level, versioning, and database hierarchies. *Composite objects* may overlap at the instance level (acyclic graphs), and they may be defined recursively at the type level. Objects may carry both short and long attributes, and the smallest granularity (leaves of the composition hierarchy) may be chosen as desired (e.g., coarse-grained objects such as modules or fine-grained objects such as statements).

The version model, which is built into the data model rather than defined on top of it, is partly influenced by SCCS/RCS. Versions of one object are arranged in a *version graph,* which may be defined as a sequence, tree, or DAG in the database schema (see also Figure 3). Any object type may be defined as versioned (i.e., total versioning of objects at all levels of the composition hierarchy). Versions may even have versions themselves (recursive versioning). Any structure of an AND/OR graph may be represented (product first, version first, intertwined). A version inherits from its generic object all attributes and components (delegation).

6.3.11 *PCTE.* PCTE [Wakeman and Jowett 1993] is a standard for open repositories that provides an interface for implementing software engineering tools. PCTE is based on a data model combining 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 composite objects.

PCTE offers basic versioning facilities [Oquendo et al. 1989]. Unlike DAMOKLES, the version model is defined on top of the data model. *Versions* of *composite objects* represent bound configurations ("version-first" selection). A new version is created by recursively copying the whole composition hierarchy and establishing successor relationships between all components. Incoming and outgoing links are copied selectively (depending on link categories and cardinalities).

6.3.12 *Shape.* Shape [Lampen and Mahler 1988; Mahler and Lampen 1988] is an SCM system that combines ideas drawn from Make, DSEE, and RCS. An attributed file system stores versioned files using directed deltas [Obst 1987]. A Shape file consists of both version selection rules and build rules and roughly corresponds to a Make file plus a DSEE configuration thread. Derived objects are stored in a cache.

The original contribution of Shape lies in its *integrated variant management* [Mahler 1994]. Variant attributes are used in version selection rules. Furthermore, a set of bindings is attached to each variant. Each binding associates the name of an attribute with a value. Bindings control in which order directories are searched for source objects (variant segregation), which set of files is passed to a compiler or linker (structural variation), which options are passed to a preprocessor (single-source variation, conditional compilation), and which flags are passed to a compiler or linker (derivation variations). In this way, Shape integrates a blend of heterogeneous mechanisms previously handled separately from each other.

6.3.13 *Aide-de-Camp.* Aide-de-Camp [Cronk 1992; Software Maintenance and Development Systems 1990] describes versions of products in terms of change sets relative to a baseline. A *change set* describes logically related, global changes that may affect multiple files. The finest grain of change is a text line. In contrast to layers in PIE, change sets are totally ordered according to their creation times. If change set c_1 was created before c_2, c_1 will be applied before c_2 when both change sets are included in some product version. In the case of overlaps, c_2 overrides the changes in c_1.

Each change set may be viewed as a switch that can be turned either on or off (see Figure 6). Aide-de-Camp detects inconsistencies when reconstructing a product version from a baseline and a sequence of change sets (e.g., modifications to nonexisting text lines). Furthermore, Aide-de-Camp provides a three-way merge tool for conflict detection. Unlike PIE, Aide-de-Camp does not support relationships that can be used to detect inconsistent combinations of change sets.

6.3.14 *COV.* COV [Lie et al. 1989; Munch et al. 1993] denotes the version model underlying the SCM subsystem of the EPOS software engineering environment [Conradi et al. 1994]. EPOS is based on an *EER data model*, where files are represented by entities with long attributes. A versioned database consists of fragments corresponding, for example, to groups of attributes or sequences of text lines. Similarly to conditional compilation, each fragment is annotated with a control expression called *visibility*. Visibilities refer to global Boolean options that constitute an n-dimensional version grid. Like MVPE, COV distinguishes between read and write filters, called *ambition* and *choice,* respectively. A choice is a complete set of option bindings (single version). An ambition contains a subset of the option bindings of the choice and corresponds to some region of the version space. The ambition defines the versions to be affected by a change, and the choice determines the version presented to the user.

COV stands for *change-oriented versioning,* which suggests a specific interpretation of options: each option corresponds to a global change that can be either included or omitted when configuring a product version. In fact, both state- and change-based version models can be expressed with options [Conradi and Westfechtel 1997]. For example, version graphs may be represented with the help of *configuration rules* [Gulla et al. 1991] that constrain delta application (e.g., implication for revision chains and mutual exclusion for branches). In particular, these configuration rules distinguish COV from change-based systems, such as Aide-de-Camp or PIE, which provide little support for excluding inconsistent change combinations [Conradi and Westfechtel 1996].

In COV, configuration rules are not only used passively for detecting inconsistent selections. In Munch [1996], a version-selection tool is presented that actively supports the user in setting up consistent ambitions and choices by automatically deducing option bindings from configuration rules.

6.3.15 *SIO*. SIO [Bernard et al. 1987; Lavency and Vanhoedenaghe 1988] extends *relational database technology* with *deductive rules* for version selection. In SIO, a software product consists of a set of modules each of which is represented by a relation. Each tuple of a relation corresponds to a single version characterized by a set of attributes used for version selection.

Configurations are described in an SQL-like manner. Configuration rules in queries are classified into *constraints* and *preferences*, the latter of which can be ordered sequentially. Preferences act as filters that are applied only if the resulting set of versions is not empty.

In addition, the rule base contains constraints specified by *compatibility rules.* A compatibility rule is an assertion in a restricted first-order predicate calculus. The conditions under which two versions from different modules are compatible are stated in terms of version attributes. Due to the restricted form of constraints, SIO can efficiently check for contradictions between them.

6.3.16 *Inscape.* Inscape [Perry 1989] goes beyond Gandalf and Adele in addressing *semantic* rather than syntactic *consistency*. Operations exported by a module are annotated with Hoare-like *pre-* and *postconditions*. The Inscape environment assists the user in constructing semantically consistent programs in various ways (inference of pre- and postconditions from the implementation of an operation, detection of unsatisfied preconditions at call sites of operations, etc.).

Version control [Perry 1987] is performed at a semantic level and addresses *substitutability* of operation versions. To this end, Inscape defines (and checks) various compatibility predicates (between versions v_1 and v_2) that ensure either global or local substitutability. In the global case, v_1 can be substituted for v_2 without having to inspect the call sites. In contrast, local substitutability refers to specific call sites.

6.3.17 *POEM.* POEM [Lin and Reiss 1995, 1996] is an environment for programming in C/C++ that strives to simplify SCM at the user interface. To this end, SCM is provided in terms of *modules*; that is, SCM matches the logical abstractions made by programmers. All components of a module (including source code, object code, and documentation) are aggregated into a single object called a *software unit.* Software units are connected by relationships that represent dependencies between source objects (see Figure 1(d)).

For each software unit, a set of operations is provided for editing, building, and version control. Versions of one software unit are arranged in a version tree. A version of a software unit uniquely determines all versions of units on which it depends ("version-first" selection, see Figure 8(b)).

6.3.18 *CoMa.* CoMa [Westfechtel 1994, 1996] manages configurations of *engineering design documents* and has been applied to both software engineering and mechanical engineering. The CoMa model integrates composition hierarchies, dependencies, and versions into a coherent framework based on a small set of concepts. Configurations are versioned objects whose components are connected by dependencies. Version graphs are maintained for both documents and configurations.

CoMa is based on *attributed graphs*. The underlying model was defined by a programmed graph rewriting system using the PROGRES specification language [Schürr et al. 1995]. At the implementation level, the database system GRAS [Kiesel et al. 1995] is used, which offers primitives for version control (*graph deltas*) but does not introduce a version model (this is done on top of the data model).

Finally, for software engineering applications, a *structure-oriented merge tool* [Westfechtel 1991] provides for three-way merging of versions of software documents (e.g., requirements definitions, software architectures, module implementations) that are internally represented as abstract syntax graphs. The merge tool preserves context-free correctness and detects certain kinds of context-sensitive conflicts by analyzing bindings of identifiers to their declarations.

6.3.19 *ClearCase.* ClearCase [Leblang 1994] differs from its predecessor DSEE in several aspects. DSEE versions only files; however, ClearCase manages versions of directories as well. All kinds of versioned objects are uniformly denoted as *elements*. The versioned file system may be accessed through a single-version view (configuration thread) defined by a configuration description. The view is a filter that provides tools with the illusion of working in a single-version environment (*virtual file system*).

In contrast to DSEE, generic references to elements are bound dynamically only when an element is accessed (there is no precomputed version map). Furthermore, in ClearCase the system model is accessed in the same way as any other element (whereas in DSEE it is used for constructing a bound configuration thread; see also Figure 22).

Configuration rules are similar to those offered by DSEE and are used to establish workspaces for developers and to control change propagation between these workspaces. A stable work environment may be set up by *static rules* (referring to specific versions). *Dynamic rules* are used to see recent changes performed by other developers.

To support *distributed SCM*, ClearCase assigns ownerships to branches in version graphs [Allen et al. 1995]. Each site appends revisions to its allocated branch. After having imported new revisions from another site, three-way merging is used to combine local and remote changes.

6.3.20 *PCL.* PCL [Tryggeseth et al. 1995], the configuration language developed in the PROTEUS project, is influenced by *conditional compilation* and *module interconnection languages* (in particular, SySL [Sommerville and Thomson 1989]). PCL is designed to manage different types of variants at the coarse-grained level. Variation of the logical structure refers to the decomposition of a system into logical components (an example was given in part (b) of Figure 12). Variation of the physical structure means that a logical component may be mapped in different ways into physical components (files residing in directories). Finally, variation of system building occurs when a source object is compiled in different ways with different compilation switches.

All kinds of variations are controlled with a single mechanism, namely, *attributes*. In a configuration description, some of these attributes are assigned specific values. The configuration process then proceeds top-down, resolving logical and physical variation by means of attributes (functional configurator). When all physical components have been determined, a Make file is generated as a final step.

6.3.21 *VOODOO.* VOODOO [Reichenberger 1994, 1995] is a file-based SCM system that supports *orthogonal version management*. Similarly to Gandalf and Adele, revisions and variants are separated from each other rather than intermixed in version graphs. However, VOODOO inverts the selection order and selects the revision first. Furthermore, versioning is applied at the product level rather than the component level. Through a carefully designed user interface, VOODOO tries to make version management as simple as possible.

For a given revision, software objects are organized hierarchically in a *project tree* whose leaves represent versioned components. A set of globally defined variants is associated with each version. Based on the three-dimensional model shown in Figure 11, VOODOO provides different views on a software product. For example, when the user first selects a product revision and then a variant, a project tree is displayed that is purged from all components not belonging to this variant.

6.3.22 *Adele II.* Adele II, the current version of Adele [Estublier and Casallas 1994, 1995; Estublier 1996], differs considerably from the initial version described earlier in this section. In particular, the data-modeling capabilities have been improved and generalized. Now Adele may be viewed as an *active object-oriented database* system with general facilities for composite objects, versioning, workspaces, and process management. On top of these, dedicated SCM tools may be built (e.g., the Adele configurator described earlier).

Adele distinguishes between three orthogonal dimensions of versioning [Estublier and Casallas 1995]. *Historical versioning* refers to the time dimension and introduces temporal database support. A versioned object evolves linearly along the time axis. Attributes are divided into three classes: common attributes shared by all versions, version-specific mutable attributes, and version-specific immutable attributes (see also Figure 13). Updates to immutable attributes result in creating a new version. *Logical versioning* (variants) is supported through set-valued attributes (e.g., a module may have multiple realization variants coexisting at a given time). *Cooperative versioning* is realized with the help of typed workspaces [Estublier 1996]. A workspace type defines the types of objects and relationships it contains, as well as propagation policies for exchange of versions between neighbors (both vertically and horizontally).

6.3.23 *Asgard.* Asgard [Micallef and Clemm 1996], which has been realized on top of ClearCase, provides *change-based versioning* on top of *version graphs*. Thus it inverts the approach followed by COV, where version graphs may be introduced on top of change-based versioning by defining constraints on the combination of changes. In Asgard, these constraints are derived from the version graphs of components.

Each component version is tagged with the name of the *activity* (Asgard's term for change) that created it. A workspace is defined by a baseline and a set of activities A, one of which is designated as the current activity. If a version was created by some activity $a \in A$, all versions on the path from the baseline must have been created by some other activity $a' \in A$ (complete selection). Furthermore, there must be a unique maximal element (unique selection). In the case of a selection error, the user must either add further activities to *A* or apply a merge tool to resolve an ambiguity.

6.3.24 *ICE.* Like COV, ICE [Zeller and Snelting 1995] is derived from conditional compilation and represents a versioned object base as a set of fragments that are tagged with control expressions. COV and ICE differ with respect to their underlying logic calculus. ICE is based on *feature logic:* a feature corresponds to an attribute whose value is defined by a feature term. For example, [ws : X11] means that the ws feature has the value X11. In general, a feature term denotes a set of potential values and may be composed by a wide range of operators such as unification, subsumption, negation, and the like [Zeller 1996]. Probably the most important of these is unification, which is used to compose configurations (the feature terms of component versions are unified). A configuration is inconsistent if unification fails (empty intersection of value sets as, e.g., in [ws : X11] $\sqcap$ [ws : Windows]).

Feature logic may be employed as a base mechanism on top of which different version models may be realized (*uniform version model*). In Zeller [1995] and Zeller and Snelting [1997], feature logic is used to realize the checkout/checkin model, the composition model, the long transaction model, and the change set model as introduced by Feiler [1991a] (see also Section 7.1).

Like COV, ICE supports multiversion editing. However, there is no distinction between read and write filters. Rather, ICE presents all versions to be edited to the user, using the syntax of conditional compilation. To this end, feature terms are mapped onto preprocessor directives [Zeller 1996]. *Partial evaluation* is used to remove all fragments whose feature terms cannot be unified with the submitted query. Like ClearCase, ICE supports a *virtual file system* to enable smooth tool integration.

7. RELATED WORK

We have given a comprehensive description of the current state of the art of version models for SCM. We have focused mainly on the organization of the version space and the flexible construction of consistent configurations from intensional specifications. Furthermore, we have described and classified a significant number of representative SCM systems. In Section 7.1, we discuss related surveys conducted earlier (1988 through 1991). Subsequently, we point out how the work presented here is related to other disciplines.

7.1 Related Work on Version Models

Two overviews were presented in 1988 at the first SCM workshop [Winkler 1988], which has launched a (still ongoing) series of follow-ups.[10] Tichy's [1988] paper introduces basic notions such as software object, source and derived object, and the like, and discusses version graphs, system building, and version selection based on AND/OR graphs. Furthermore, the paper attempts to unify the terminology in the SCM field by means of a glossary. Estublier [1988] complements Tichy's presentation by focusing on the construction of consistent configurations. In combination, these papers reflect the state of the art as of 1988.

The Software Engineering Institute (SEI) has published several papers that review and survey the state of the art in SCM [Brown et al. 1991; Dart 1991, 1992b]. Perhaps the most influential of these was written in 1991 by Feiler [1991a], who classified the models underlying SCM systems into four categories corresponding to different ways in which a user interacts with an SCM system. In the *checkout/checkin model,* component versions are transferred individually between repository and workspace. The *composition model* supports

[10] Please see Tichy [1989], Feiler [1991b], Feldman [1993], Estublier [1995], Sommerville [1996], and Conradi [1997].

version selection through rules and assists the user in selecting consistent combinations of component versions ("product first"). In the *long transaction model,* a user connects to a long transaction and operates on a configuration version ("version first"). Finally, in the *change set model* a configuration is described in terms of change sets each of which aggregates all modifications performed in response to some change request.

The classification proposed by Feiler helps in understanding different paradigms underlying SCM systems. Furthermore, the models are evaluated by discussing their merits and shortcomings. Unfortunately, the classification is not orthogonal. The composition model and the checkout/checkin model are not alternatives; rather, the former is built on top of the latter. Furthermore, long transactions can be used in combination with any approach to specifying a configuration.

In 1990, Katz [1990] surveyed version models for *engineering databases*. Katz primarily considers electrical engineering (CAD) and only mentions a few software engineering approaches. Although both domains have evolved almost independently for a long time, many parallels do exist [Dart 1992a]. Katz introduces the following concepts: organizing the version set (histories), dynamic configuration mechanisms (binding of generic references), hierarchical compositions (versions of composite objects), version-grouping mechanisms (to represent variants), instances versus definitions (instances of component versions may have properties that depend on the respective contexts in configurations), change notification and propagation (when and where to propagate changes), and object sharing mechanisms (workspaces).

In several aspects, our view of version models goes beyond the work presented by Katz. His paper focuses on approaches based on version graphs and restricted to extensional versioning at the component level. Intensional versioning is provided only at the configuration level, resulting in the composition model introduced by Feiler. As we have shown, there are radically different approaches such as conditional compilation that are not based on version graphs at all. Furthermore, Katz primarily discusses state-based versioning. In contrast, this article covers change-based versioning as well and investigates the relations among these complementary approaches. Finally, configuration rules and the consistency problems of intensional versioning are discussed only briefly by Katz.

7.2 Related Disciplines

Version management is related to many other disciplines of computer science. In the following, these relations are described briefly. A major challenge of future research consists of clarifying the relations to these disciplines.

Temporal databases [Tansel et al. 1993; Snodgrass 1992] record the evolution history of data such that previous states can be retrieved in addition to the current state. Temporal databases focus solely on the time dimension and cover neither variants nor change-based versioning. Furthermore, they often distinguish between valid time (time in the real world) and transaction time (time of recording a fact in the database). This distinction is not relevant for SCM because software objects do not represent real-world objects existing independently of the database.

Different approaches have been developed to accommodate changes to the database schema. In the case of *schema evolution,* only the current version of the schema is valid, and all data must be converted (in lazy or eager mode) in order to maintain the consistency of the database. In contrast, *schema versioning* [Roddick 1995] makes it possible to view the data under different versions of the schema. In SCM systems, versioning of the schema (and other metadata such as configuration rules) is rarely considered seriously. On the other hand, schema versioning often does not take the versioning of instance data into account.

Deductive databases [Das 1992; Ramamohanarao and Harland 1994; Ramakrishnan and Ullman 1995] provide for persistent storage of facts and rules and are usually based on a Prolog-like data model. Deductive capabilities are urgently needed for intensional versioning. On the other hand, deductive databases have been employed only rarely in SCM [Zeller 1995; Bernard et al. 1987; Lavency and Vanhoedenaghe 1988]. Rather, many SCM systems incorporate home-grown deductive components that have been developed in an ad hoc manner.

It has been recognized for a long time that the ACID principle cannot be transferred from short to *long transactions* [Barghouti and Kaiser 1991; Kaiser 1995; Feiler 1991a]. Rather, precommit cooperation is required in order to coordinate long-lasting development and maintenance tasks. Customizable policies have been developed to control cooperation. Many approaches to long transactions do not take versioning into account [Barghouti and Kaiser 1991]. This is a severe restriction since versions play a crucial role in cooperation control [Estublier and Casallas 1995]. So far, only a few SCM systems support long transactions [Conradi and Malm 1991; Godart et al. 1995]. Many others merely provide workspaces and mechanisms for controlling change propagation between them [Estublier 1996].

Software process modeling [Finkelstein et al. 1994; Curtis et al. 1992; Rombach and Verlage 1995] is concerned with the definition, analysis, and enactment of models of real-world software processes. Many different paradigms have been applied to process modeling [Conradi et al. 1991], including active databases [Estublier and Casallas 1994], rules [Kaiser et al. 1988; Peuschel and Schäfer 1992], nets [Deiters and Gruhn 1990; Bandinelli et al. 1993; Jaccheri and Conradi 1993; Heimann et al. 1996], imperative programming [Sutton et al. 1995], and hybrids of these. In order to integrate SCM and process modeling, functional overlap has to be considered (e.g., between build tools and rule-based process engines such as Marvel [Kaiser et al. 1988]). Furthermore, the definition of "product space" must be widened and must cover process models as well. Finally, dynamic interactions between product and process need to be taken into account (e.g., replanning of task nets after changes to the product structure [Liu and Conradi 1993]).

Tool integration [Wasserman 1990] is provided by SCM systems through workspaces that hide versioning from the tools. Workspaces can be separated physically from the versioned database [Rochkind 1975], or be realized as updatable database views (virtual workspaces, e.g., virtual file systems [Leblang 1994; Fowler et al. 1994]). Current SCM systems focus on integration of file-based tools and offer poor support for integrating tools operating on databases (e.g., CASE tools [Wallnau 1992]). The problem of integrating heterogeneous database systems without sacrificing their autonomy is addressed by federated database systems [Sheth and Larson 1990] and data warehouses [Hammer et al. 1995]. Furthermore, different kinds of platforms or frameworks offer plug-in interfaces for tool integration, for example, broadcast message servers [Reiss 1990] and object request brokers [Soley and Kent 1995]. Future generations of SCM systems need to interface with these frameworks.

Current SCM systems are severely limited with respect to managing dependencies between software objects. First, they are mainly concerned with dependencies between source code modules rather than with dependencies among any kinds of software objects produced in the software lifecycle. Second, they are not capable of representing fine-grained dependencies, which is crucial to provide for detailed traceability throughout the whole lifecycle. These problems can partly be addressed by applying the concepts of *hypertext systems* [Conklin 1987] to the software engineering domain.

The emerging discipline of *software architectures* [Shaw and Garlan 1996] stresses the importance of a high-level description of software products above the source code level. The software architecture acts as a central document for impact analysis, planning of development and maintenance activities, division of labor, understanding the interfaces between different components, and so on [Nagl 1990]. SCM may benefit from software architectures in some ways. First, SCM systems focus primarily on source code and represent the product structure by rather low-level system models that are mainly used to drive system building. Architecture-oriented SCM will improve the software process through a high-level product description. Second, the architecture of the SCM system is a major research challenge as well. In order to design an appropriate architecture, a clear understanding of the relations among an SCM system and other software components (e.g., process management systems, broadcast message servers, object request brokers) needs to be developed.

8. CONCLUSION

Over the past 20 years, many approaches to versioning have been developed. Now we have gained a sufficient level of understanding to classify these approaches. Initial attempts to develop a uniform model have been undertaken [Zeller 1995; Zeller and Snelting 1997]. Furthermore, the recent evolution of SCM systems shows that their underlying version models are converging to an increasing extent. For example, change-based versioning has been realized on top of version graphs and vice versa.

Based on the material presented in this article, we believe that a version model can be developed that integrates extensional and intensional versioning, state-based and change-based versioning, revisions and variants, construction of source and derived versions, as well as workspaces and long transactions into a coherent framework [Conradi and Westfechtel 1997]. This framework is not expected to provide "the" model; rather, it must be customizable to suit the needs of a specific application.

In the future, we expect that more and more SCM systems will be built with the help of database technology. Having gained a better understanding of version models, versioning can be pulled out of SCM systems and moved into database systems. In particular, intensional versioning will benefit from the powerful facilities of an underlying deductive database system.

ACKNOWLEDGMENTS

Many thanks go to the EPOS team in Trondheim and to the IPSEN team in Aachen. Furthermore, we would like to acknowledge the competent and comprehensive comments of the unknown reviewers. Finally, the references were prepared partly using the bibliography of Hal Render, University of Colorado at Boulder.

REFERENCES

ADAMS E., GRAMLICH, W., MUCHNICK, S., AND TIRFING, S. 1986. SunPro: Engineering a practical program development environment. In *Proceedings of the International Workshop on Advanced Programming Environments* (Trondheim, June) R. Conradi, T. M. Didriksen, and D. H. Wanvik, Eds., LNCS 244, Springer-Verlag, 86–96.

ADAMS, E. W., HONDA, M., AND MILLER, T. C. 1989. Object management in a CASE environment. In *Proceedings of the Eleventh International Conference on Software Engineering* (Pittsburgh, PA, May), IEEE Computer Society Press, Los Alamitos, CA, 154–163.

ADAMS, P. AND SOLOMON, M. 1995. An overview of the CAPITL software development environment. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 1–34.

ALLEN, L., FERNANDEZ, G., KANE, K., LEBLANG, D., MINARD, D., AND POSNER, J. 1995. ClearCase MultiSite: Supporting geographically-distributed software development. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5,* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 194–214.

BABICH, W. A. 1986. *Software Configuration Management.* Addison-Wesley, Reading, MA.

BANDINELLI, S. C., FUGGETTA, A., AND GHEZZI, C. 1993. Software process model evolution in the SPADE environment. *Trans. Softw. Eng. 19,* 12 (Dec.), 1128–1144.

BARGHOUTI, N. S. AND KAISER, G. E. 1991. Concurrency control in advanced database applications. *ACM Comput. Surv. 23,* 3 (Sept.), 269–317.

BELKHATIR, N., AND ESTUBLIER, J. 1986. Experience with a data base of programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Palo Alto, CA, Dec.) *ACM SIGPLAN Not. 22,* 1, 84–91.

BERNARD, Y., LACROIX, M., LAVENCY, P., AND VANHOEDENAGHE, M. 1987. Configuration management in an open environment. In *Proceedings of the 1st European Software Engineering Conference* (Straßburg, Sept.), G. Goos and J. Hartmanis, Eds., LNCS 289, Springer-Verlag, 35–43.

BERSOFF, E. H., HENDERSON, V. D., AND SIEGEL, S. G. 1980. *Software Configuration Management: An Investment in Product Integrity.* Prentice-Hall, Englewood Cliffs, NJ.

BERZINS, V. 1994. Software merge: Semantics of combining changes to programs. *ACM Trans. Program. Lang. Syst. 16,* 6 (Nov.), 1875–1903.

BERZINS, V., ED. 1995. *Software Merging and Slicing.* IEEE Computer Society Press, Los Alamitos, CA.

BINKLEY, D., HORWITZ, S., AND REPS, T. 1995. Program integration for languages with procedure calls. *ACM Trans. Softw. Eng. Methodol. 4,* 1 (Jan.), 3–35.

BORISON, E. A. 1989. Program changes and the cost of selective recompilation. Tech. Rep. CMU-CS-89-205 (July), Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

BROWN, A., DART, S., FEILER, P., AND WALLNAU, K. 1991. The state of automated configuration management. Tech. Rep. ATR92 (Sept.), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

BUFFENBARGER, J. 1995. Syntactic software merging. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LCNS 1005, Springer-Verlag, 153–172.

CLEMM, G. 1995. The Odin system. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 241–262.

CONKLIN, J. 1987. Hypertext: An introduction and survey. *IEEE Comput. 20,* 9 (Sept.), 17–41.

CONRADI, R. ED. 1997. *Software Configuration Management: ICSE'97 SCM-7 Workshop* (Boston, MA, May), LNCS 1235, Springer-Verlag.

CONRADI, R., ET AL. 1994. EPOS: Object-oriented and cooperative process modelling. In *Software Process Modelling and Technology,* Advanced Software Development Series. A. Finkelstein, J. Kramer, and B. Nuseibeh, Eds., Research Studies Press (John Wiley), Chichester, UK, 33–70.

CONRADI, R. AND MALM, C. 1991. Cooperating transactions and workspaces in EPOS: Design and preliminary implementation. In *Proceedings of the Third International Conference on Advanced Information Systems Engineering (CAISE '91)* R. Andersen, J. A. Bubenko, and A. Solvberg, Eds. (Trondheim, May), LNCS 498, Springer-Verlag, 375–392.

CONRADI, R., AND WESTFECHTEL, B. 1996. Configuring versioned software products. In *Software Configuration Management: ICSE'96 SCM-6 Workshop* (Berlin, March), I. Sommerville, Ed., LNCS 1167, Springer-Verlag, 88–109.

CONRADI, R. AND WESTFECHTEL, B. 1997. Towards a uniform version model for software configuration management. In *Software Configuration Management: ICSE'97 SCM-7 Workshop* (Boston, MA, May), R. Conradi, Ed., LNCS 1235, Springer-Verlag, 1–17.

CONRADI, R., LIU, C., AND JACCHERI, M. L. 1991. Process modeling paradigms: An evaluation. In *Proceedings of the Seventh International Software Process Workshop* (Yountville, CA, Oct.), IEEE Computer Society Press, Los Alamitos, CA, 51–54.

CRONK, R. D. 1992. Tributaries and deltas. *BYTE 17,* 1 (Jan.), 177–186.

CURTIS, B., KELLNER, M. I., AND OVER, J. 1992. Process modeling. *Commun. ACM 35,* 9 (Sept.), 75–90.

DART, S. 1991. Concepts in configuration management systems. In *Proceedings of the Third International Workshop on Software Configuration Management* (Trondheim, Norway, June), P. H. Feiler, Ed., ACM Press, New York, 1–18.

DART, S. A. 1992a. Parallels in computer-aided design frameworks and software development environments efforts. *IFIP Trans. A 16,* 175–189.

DART, S. A. 1992b. The past, present, and future of configuration management. Tech. Rep. CMU/SEI-92-TR-8 (July), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

DART, S. A., ELLISON, R. J., FEILER, P. H., AND HABERMANN, A. N. 1987. Software development environments. *IEEE Computer 20,* 11 (Nov.), 18–28.

DAS, S. K. 1992. *Deductive Databases and Logic Programming.* Addison-Wesley, Reading, MA.

DEITERS, W. AND GRUHN, V. 1990. Managing software processes in the environment MELMAC. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments* (Irvine, CA, Dec.), *ACM SIGSOFT Softw. Eng. Not. 15,* (6) R. Taylor, Ed., 193–205.

DITTRICH, K., GOTTHARD, W., AND LOCKEMANN, P. 1986. DAMOKLES, a database system for software engineering environments. In *Proceedings of the International Workshop on Advanced Programming Environments* (Trondheim, June), R. Conradi, T. M. Didriksen, and D. H. Wanvik, Eds., LNCS 244, Springer-Verlag, 353–371.

DITTRICH, K. R. AND LORIE, R. A. 1988. Version support for engineering database systems. *IEEE Trans. Softw. Eng. 14,* 4 (April), 429–437.

EHRIG, H., FEY, W., HANSEN, H., LOWE, M., AND JACOBS, D. 1989. Algebraic software development concepts for module and configuration families. In *Proceedings of the Ninth Conference on Foundation of Software Technology and Theoretical Computer Science* (Bangalore, Dec.), V. Madhavan, Ed., LNCS 405, Springer-Verlag, 181–192.

ESTUBLIER, J. 1985. A configuration manager: The Adele data base of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large* (Harwichport, MA, June), 140–147.

ESTUBLIER, J. 1988. Configuration management: The notion and the tools. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany), J. F. H. Winkler, Ed., Teubner Verlag, 38–61.

ESTUBLIER, J. 1996. Workspace management in software engineering environments. In *Software Configuration Management: ICSE'96 SCM-6 Workshop* (Berlin, March) I. Sommerville, Ed., LCNS 1167, Springer-Verlag.

ESTUBLIER, J. ED. 1995. *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), LNCS 1005, Springer-Verlag.

ESTUBLIER, J. AND CASALLAS, R. 1994. The Adele configuration manager. In *Configuration Management,* W. F. Tichy, Ed., Vol. 2 of *Trends in Software,* Wiley, New York, 99–134.

ESTUBLIER, J. AND CASALLAS, R. 1995. Three dimensional versioning. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 118–135.

FEILER, P. H. 1991a. Configuration management models in commercial environments. Tech. Rep. CMU/SEI-91-TR-7 (March), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

FEILER, P. H., ED. 1991b. *Proceedings of the Third International Workshop on Software Configuration Management* (Trondheim, Norway, June), ACM Press, New York.

FELDMAN, S., ED. 1993. *Proceedings of the Fourth International Workshop on Software Configuration Management (Preprint)* (Baltimore, MD, May).

FELDMAN, S. I. 1979. Make—A program for maintaining computer programs. *Softw. Pract. Exper. 9,* 4 (April), 255–265.

FINKELSTEIN, A., KRAMER, J., AND NUSEIBEH, B., EDS. 1994. *Software Process Modelling and Technology.* Advanced Software Development Series. Research Studies Press (Wiley), Chichester, UK.

FOWLER, G., KORN, D., AND RAO, H. 1994. n-DFS: The multiple dimensional file system. In *Configuration Management,* W. F. Tichy, Ed., Vol. 2 of *Trends in Software,* Wiley, New York, 135–154.

FRASER, C. AND MYERS, E. 1986. An editor for revision control. *ACM Trans. Program. Lang. Syst. 9,* 2 (April), 277–295.

GODART, C., CANALS, G., CHAROY, F., AND MOLLI, P. 1995. About some relationships between configuration management, software process, and cooperative work: The COO environment. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 173–178.

GOLDSTEIN, I. P. AND BOBROW, D. G. 1980. A layered approach to software design. Tech. Rep. CSL-80-5, XEROX PARC, Palo Alto, CA.

GOODSTEP 1995. The GOODSTEP Project—Final Report. GOODSTEP ESPRIT Project 6115.

GOTTHARD, W. 1988. *Datenbanksysteme für Software-Produktionsumgebungen.* IFB 193, Springer-Verlag, Berlin.

GULLA, B., KARLSSON, E.-A., AND YEH, D. 1991. Change-oriented version descriptions in EPOS. *Softw. Eng. J. 6,* 6 (Nov.), 378–386.

HABERMANN, N. AND NOTKIN, D. 1986. Gandalf: Software development environments. *IEEE Trans. Softw. Eng. 12,* 12 (Dec.), 1117–1127.

HAMMER, J., GARCIA-MOLINA, H., WIDOM, J., LABIO, W., AND ZHUGE, Y. 1995. The Stanford data warehousing project. *Data Eng. Bull. 18,* 2, 41–48.

HEIMANN, P., JOERIS, G., KRAPP, C.-A., AND WESTFECHTEL, B. 1996. DYNAMITE: Dynamic task nets for software process management. In *Proceedings of the Eighteenth International Conference on Software Engineering* (Berlin, March), IEEE Computer Society Press, Los Alamitos, CA, 331–341.

HORWITZ, S., PRINS, J., AND REPS, T. 1989. Integrating non-interfering versions of programs. *ACM Trans. Program. Lang. Syst. 11,* 3 (July), 345–387.

HUMPHREY, W. S. 1989. *Managing the Software Process.* SEI Series in Software Engineering, Addison-Wesley, Reading, MA.

IEEE 1983. *IEEE Standard for Software Configuration Management Plans: ANSI/IEEE Std 828-1983.* IEEE, New York.

IEEE 1988. *IEEE Guide to Software Configuration Management: ANSI/IEEE Std 1042-1987.* IEEE, New York.

JACCHERI, M. L. AND CONRADI, R. 1993. Techniques for process model evolution in EPOS. *IEEE Trans. Softw. Eng. 19,* 12 (Dec.), 1145–1156.

KAISER, G., FEILER, P., AND POPOVICH, S. 1988. Intelligent assistance for software development and maintenance. *IEEE Softw. 5,* 3 (May), 40–49.

KAISER, G. E. 1995. Cooperative transactions for multiuser environments. In *Modern Database Systems* (Reading, MA), W. Kim, Ed., Addison Wesley, 409–433.

KAISER, G. E. AND HABERMANN, A. N. 1983. An environment for system version control. In *Digest of Papers of Spring CompCon '83,* IEEE Computer Society Press, Los Alamitos, CA, 415–420.

KATZ, R. H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Comput. Surv. 22,* 4 (Dec.), 375–408.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1978. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ.

KIESEL, N., SCHÜRR, A., AND WESTFECHTEL, B. 1995. GRAS, a graph-oriented software engineering database system. *Inf. Syst. 20,* 1 (Jan.), 21–51.

KIM, W., ED. 1995. *Modern Database Systems.* Addison-Wesley, Reading, MA.

KRAMER, J. 1993. Special issue on configurable distributed systems. *Softw. Eng. J. 8,* 2 (March), 51–52.

KRUSKAL, V. 1984. Managing multi-version programs with an editor. *IBM J. Res. Dev. 28,* 1, 74–81.

LAMPEN, A., AND MAHLER, A. 1988. Shape—A software configuration management tool. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany), J. F. H. Winkler, Ed., Teubner Verlag, 228–243.

LAMPSON, B. W. AND SCHMIDT, E. E. 1983a. Organizing software in a distributed environment. In *ACM Symposium on Programming Language Issues in Software Systems, ACM SIGPLAN Not. 18,* 6 (June), 1–13.

LAMPSON, B. W. AND SCHMIDT, E. E. 1983b. Practical use of a practical polymorphic applicative language. In *Tenth Annual ACM Symposium on Principles of Programming Languages* (Austin, TX, Jan.), *ACM SIGPLAN Not. 18,* 1, 237–255.

LAVENCY, P. AND VANHOEDENAGHE, M. 1988. Knowledge based configuration management. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences* (Hawaii, Jan.), B. Shriver, Ed., IEEE Computer Society Press, Los Alamitos, CA, 83–92.

LEBLANG, D. 1994. The CM challenge: Configuration management that works. In *Configuration Management,* W. F. Tichy, Ed., Vol. 2 of *Trends in Software*, Wiley, New York, 1–38.

LEBLANG, D. B. AND CHASE, R. P. 1984. Computer-aided software engineering in a distributed workstation environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Not. 19,* 5 (May), 104–112.

LEBLANG, D. B. AND MCLEAN, G. D. 1985. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large* (Harwichport, MA, June), 122–127.

LEBLANG, D. B., CHASE, R. P., JR., AND SPILKE, H. 1988. Increasing productivity with a parallel configuration manager. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany), J. F. H. Winkler, Ed., Teubner Verlag, 21–37.

LIE, A., CONRADI, R., DIDRIKSEN, T., KARLSSON, E., HALLSTEINSEN, S. O., AND HOLAGER, P. 1989. Change oriented versioning. In *Proceedings of the Second European Software Engineering Conference* (Coventry, UK, Sept.), C. Ghezzi and J. A. McDermid, Eds., LNCS 387, Springer-Verlag, 191–202.

LIN, Y.-J. AND REISS, S. P. 1995. Configuration management in terms of modules. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 101–117.

LIN, Y.-J. AND REISS, S. P. 1996. Configuration management with logical structures. In *Proceedings of the Eighteenth International Conference on Software Engineering* (Berlin, March), IEEE Computer Society Press, Los Alamitos, CA, 298–307.

LIPPE, E., AND VAN OOSTEROM, N. 1992. Operation-based merging. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments (SDE5),* (Tyson's Corner, VA, Dec.) *ACM SIGSOFT Softw. Eng. Not. 17* 5, 78–87.

LIU, C., AND CONRADI, R. 1993. Automatic replanning of task networks for supporting process model evolution in EPOS. In *Proceedings of the European Software Engineering Conference '93* (Garmisch-Partenkirchen, Germany, Sept.), I. Sommerville and M. Paul, Eds., Springer-Verlag, 434–450.

MacKay, S. A. 1995. The state-of-the-art in concurrent, distributed configuration management. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 180–194.

Mahler, A. 1994. Variants: Keeping things together and telling them apart. In *Configuration Management,* W. F. Tichy, Ed., Vol. 2 of *Trends in Software*, Wiley, New York, 73–98.

Mahler, A. and Lampen, A. 1988. An integrated toolset for engineering software configurations. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Boston, MA, Nov.), *ACM Softw. Eng. Not. 13,* 5, 191–200.

Marzullo, K. and Wiebe, D. 1986. Jasmine: A software system modelling facility. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Palo Alto, CA, Dec.), *ACM SIGPLAN Not. 22,* 1, 121–130.

Micallef, J., and Clemm, G. 1996. The Asgard system: Activity-based configuration management. In *Software Configuration Management: ICSE'96 SCM-6 Workshop* (Berlin, March), I. Sommerville, Ed., LNCS 1167, Springer-Verlag, 175–186.

Munch, B. 1996. HiCOV: Managing the version space. In *Software Configuration Management: ICSE'96 SCM-6 Workshop* (Berlin, March), I. Sommerville, Ed., LNCS 1167, Springer-Verlag, 110–126.

Munch, B. P. 1993. Versioning in a software engineering database—the change oriented way. Ph.D. Thesis, NTNU Trondheim, Norway.

Munch, B. P., Larsen, J.-O., Gulla, B., Conradi, R., and Karlsson, E.-A. 1993. Uniform versioning: The change-oriented model. In *Proceedings of the Fourth International Workshop on Software Configuration Management* (Baltimore, MD, May), S. Feldman, Ed., (Preprint) 188–196.

Nagl, M. 1990. *Softwaretechnik: Methodisches Programmieren-im-Großen.* Springer-Verlag, Berlin.

Nagl, M. Ed. 1996. *Building Tightly-Integrated Software Development Environments: The IPSEN Approach.* LNCS 1170, Springer-Verlag, Berlin.

Obst, W. 1987. Delta technique and string-to-string correction. In *Proceedings of the First European Software Engineering Conference* (Straßburg, Sept., 1987), J. Goos and J. Hartmanis, Eds., LNCS 289, Springer-Verlag, 64–68.

Oquendo, F., Berrado, K., Gallo, F., Minot, R., and Thomas, I. 1989. Version management in the PACT integrated software engineering environment. In *Proceedings of the Second European Software Engineering Conference,* (Coventry, UK, Sept.), C. Ghezzi and J. A. McDermid, Eds., LNCS 387, Springer-Verlag, 222–242.

Paulk, M. C., Weber, C. V., Curtis, B., and Chrissis, M. B. 1997. *The Capability Maturity Model—Guidelines for Improving the Software Process.* Addison-Wesley, Reading, MA.

Perry, D. 1989. The Inscape environment. In *Proceedings of the Eleventh International Conference on Software Engineering* (Pittsburgh, PA, May), IEEE Computer Society Press, Los Alamitos, CA, 2–12.

Perry, D. E. 1987. Version control in the Inscape environment. In *Proceedings of the Ninth International Conference on Software Engineering* (Monterey, CA, March), IEEE Computer Society Press, Los Alamitos, CA, 142–149.

Peuschel, B., and Schäfer, W. 1992. Concepts and implementation of a rule-based process engine. In *Proceedings of the Fourteenth International Conference on Software Engineering* (Melbourne, Australia, May), IEEE Computer Society Press, Los Alamitos, CA, 262–279.

Prieto-Diaz, R., and Neighbors, J. 1986. Module interconnection languages. *J. Syst. Softw. 6,* 4 (Nov.), 307–334.

Ramakrishnan, R., and Ullman, J. D. 1995. A survey of deductive database systems. *J. Logic Program. 23,* 2 (May), 125–149.

Ramamohanarao, K., and Harland, J. 1994. An introduction to deductive database languages and systems. *VLDB J. 3,* 2 (April), 107–122.

Reichenberger, C. 1994. Concepts and techniques for software version control. *Softw. Concepts Tools 15,* 3 (July), 97–104.

Reichenberger, C. 1995. VOODOO—a tool for orthogonal version management. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 61–79.

Reiss, S. 1990. Interacting with the FIELD environment. *Softw. Pract. Exper. 20,* S1 (June), 89–115.

Rich, A., and Solomon, M. 1991. A logic-based approach to system modelling. In *Proceedings of the Third International Workshop on Software Configuration Management* (Trondheim, Norway, June), P. H. Feiler, Ed., ACM Press, New York, 84–93.

Rigg, W., Burrows, C., and Ingram, P. 1995. *Configuration Management Tools.* Ovum Ltd., London.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Trans. Softw. Eng. 1,* 4 (Dec.), 364–370.

RODDICK, J. F. 1995. A survey of schema versioning issues for database systems. *Inf. Softw. Technol. 37,* 7 (July), 383–393.

ROMBACH, H. D. AND VERLAGE, M. 1995.- Directions in software process research. In *Advances in Computers,* M. V. Zelkowitz, Ed., Vol. 41, Academic Press, San Diego, 1–63.

SARNAK, N., BERNSTEIN, R., AND KRUSKAL, V. 1988. Creation and maintenance of multiple versions. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany), J. F. H. Winkler, Ed., Teubner Verlag, 264–275.

SCHMERL, B. R., AND MARLIN, C. D. 1995. Designing configuration management facilities for dynamically bound systems. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 88–100.

SCHÜRR, A., WINTER, A., AND ZÜNDORF, A. 1995. Graph grammar engineering with PROGRES. In *Proceedings of the European Software Engineering Conference (ESEC '95)* (Barcelona, Sept.), W. Schäfer and P. Botella, Eds., LNCS 989, Springer-Verlag, 219–234.

SCIORE, E. 1994. Version and configuration management in an object-oriented data model. *VLDB J. 3,* 1 (Jan.), 77–106.

SHAW, M., AND GARLAN, D. 1996. *Software Architecture—Perspectives on an Emerging Discipline.* Prentice-Hall, Englewood Cliffs, NJ.

SHETH, A. P. AND LARSON, J. A. 1990. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Comput. Surv. 22,* 3 (Sept.), 183–236.

SNODGRASS, R. T. 1992. Temporal databases. In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space* (Pisa, Sept.), A. Frank, I. Campari, and U. Formentini, Eds., LNCS 639, Springer-Verlag, 22–64.

SOFTWARE MAINTENANCE AND DEVELOPMENT SYSTEMS. 1990. *Aide-de-Camp Product Overview.* Software Maintenance and Development Systems, Concord, MA.

SOLEY, R. M. AND KENT, W. 1995. The OMG object model. In *Modern Database Systems,* (Reading, MA), W. Kim, Ed., Addison-Wesley, Reading, MA, 18–41.

SOMMERVILLE, I., ED. 1996. *Software Configuration Management: ICSE'96 SCM-6 Workshop* (Berlin, March) LNCS 1167, Springer-Verlag.

SOMMERVILLE, I., AND THOMSON, R. 1989. An approach to the support of software evolution. *Comput. J. 32,* 5 (Dec.), 386–398.

SUTTON, S. M., HEIMBIGNER, D., AND OSTERWEIL, L. J. 1995. APPL/A: A language for software process programming. *ACM Trans. Softw. Eng. Methodol. 4,* 3 (July), 221–286.

TANSEL, A. U., CLIFFORD, J., GADIA, S., JAJODIA, S., SEGEV, A., AND SNODGRASS, R. 1993. *Temporal Databases—Theory, Design, and Implementation.* Benjamin/Cummings, Redwood City, CA.

TEITELMAN, W. 1984. A tour through Cedar. In *Proceedings of the Seventh International Conference on Software Engineering* (Orlando, FL, March), IEEE Computer Society Press, Los Alamitos, CA, 181–195.

TICHY, W. F. 1979. Software development control based on module interconnection. In *Proceedings of the IEEE Fourth International Conference on Software Engineering* (Pittsburgh, Sept.), IEEE Computer Society Press, Los Alamitos, CA, 29–41.

TICHY, W. F. 1982a. A data model for programming support environments. In *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Information System Design and Development* (New Orleans, Jan.), North-Holland, 31–48.

TICHY, W. F. 1982b. Design, implementation, and evaluation of a revision control system. In *Proceedings of the Sixth International Conference on Software Engineering* (Tokyo, Sept.), IEEE Computer Society Press, Los Alamitos, CA, 58–67.

TICHY, W. F. 1985. RCS—A system for version control. *Softw. Pract. Exper. 15,* 7 (July), 637–654.

TICHY, W. F. 1988. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany), J. F. H. Winkler, Ed., Teubner Verlag, 1–20.

TICHY, W. F., ED. 1989. *Proceedings of the Second International Workshop on Software Configuration Management* (Princeton, NJ, Nov.), *ACM Softw. Eng. Not. 14,* 7.

TICHY, W. F., ED. 1994. *Configuration Management,* Vol. 2 of *Trends in Software.* Wiley, New York.

TRYGGESETH, E., GULLA, B., AND CONRADI, R. 1995. Modelling systems with variability using the PROTEUS configuration language. In *Software Configuration Management: Selected Papers SCM-4 and SCM-5* (Seattle, WA, April), J. Estublier, Ed., LNCS 1005, Springer-Verlag, 216–240.

WAKEMAN, L., AND JOWETT, J. 1993. *PCTE—The Standard for Open Repositories.* Prentice-Hall, Englewood Cliffs, NJ.

WALLNAU, K. C. 1992. Issues and techniques of CASE integration with configuration management. Tech. Rep. CMU/SEI-92-TR-5 (March), Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

WARREN, I., AND SOMMERVILLE, I. 1995. Dynamic configuration abstraction. In *Proceedings of the European Software Engineering Conference (ESEC '95)* (Barcelona, Sept.), W. Schäfer and P. Botella, Eds., LNCS 989, Springer-Verlag, 173–190.

WASSERMAN, A. 1990. Tool integration in software engineering environments. In *Proceedings of the Second International Workshop on Software Engineering Environ.* (Chinon, France, Sept.), F. Long, Ed., LNCS 467, Springer-Verlag, 137–149.

WESTFECHTEL, B. 1991. Structure-oriented merging of revisions of software documents. In *Proceedings of the Third International Workshop on Software Configuration Management* (Trondheim, Norway, June), P. H. Feiler, Ed., ACM Press, New York, 68–79.

WESTFECHTEL, B. 1994. Using programmed graph rewriting for the formal specification of a configuration management system. In *Proceedings WG '94 Workshop on Graph-Theoretic Concepts in Computer Science* (Herrsching, Germany, June), E. Mayr, G. Schmidt, and G. Tinhofer, Eds., LNCS 903, Springer-Verlag, 164–179.

WESTFECHTEL, B. 1996. A graph-based system for managing configurations of engineering design documents. *Int. J. Softw. Eng. Knowledge Eng. 6,* 4 (Dec.), 549–583.

WINKLER, J. F. H., ED. 1988. *Proceedings of the International Workshop on Software Version and Configuration Control* (Grassau, Germany), Teubner Verlag.

ZELLER, A. 1995. A unified version model for configuration management. In *Proceedings of the ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering* (Washington, Oct), *ACM Softw. Eng. Not. 20,* 4, 151–160.

ZELLER, A. 1996. Smooth operations with square operators—the version set model in ICE. In *Software Configuration Management: ICSE'96 SCM-6 Workshop* (Berlin, March), I. Sommerville, Ed., LNCS 1167, Springer-Verlag.

ZELLER, A. AND SNELTING, G. 1995. Handling version sets through feature logic. In *Proceedings of the Fifth European Software Engineering Conference* (Barcelona, Sept.), W. Schäfer and P. Botella, Eds., LNCS 989, Springer-Verlag, 191–204.

ZELLER, A., AND SNELTING, G. 1997. Unified versioning through feature logic. *ACM Trans. Softw. Eng. Methodol. 6,* 4 (Oct.), 397–440.

Received October 1996; revised July 1997; accepted August 1997