Version Models for Software Configuration Management

Reidar Conradi, Trondheim*
Bernhard Westfechtel, Aachen†

Technical Report AIB 96-10, RWTH Aachen
October 25, 1996

Abstract

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 paper focuses on the version models underlying both commercial systems and research prototypes. It provides an overview and classification of different versioning paradigms and discusses their strengths and weaknesses. Furthermore, it defines and relates fundamental concepts such as revisions, variants, configurations, and changes. In particular, we focus on intensional versioning, i.e., construction of versions based on version rules. Finally, we provide an overview of systems which 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 (CASE)*;
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: Versions, Configurations

Additional Key Words and Phrases: Revisions, variants, changes, version rules

Notice. This paper has been submitted for publication elsewhere. It has been published as a technical report for early dissemination of its contents. As a courtesy to the intended publisher, it should not be widely distributed after the date of outside publication.

*Detailed address: Norwegian University of Science and Technology (NTNU), N-7034 Trondheim, Norway. Phone: +47 73 59 34 44, Fax: +47 73 59 44 66, Email: conradi@idt.unit.no.

†Detailed address: Lehrstuhl für Informatik III, RWTH Aachen, D-52056 Aachen. Phone: +49 241 80 21310, Fax: +49 241 8888 218, Email: bernhard@i3.informatik.rwth-aachen.de. This work was partially carried out during a research stay at NTNU. Support from NTNU is gratefully acknowledged.

Contents

1 INTRODUCTION **3**

2 PRODUCT SPACE **4**
2.1 Basic Notions 4
2.2 Examples 6

3 VERSION SPACE **6**
3.1 Basic Notions 8
3.2 Examples 10

4 INTEGRATION OF PRODUCT SPACE AND VERSION SPACE **14**
4.1 Basic Notions 14
4.2 Examples 15

5 INTENSIONAL VERSIONING **21**
5.1 BasicNotions 21
5.2 Construction of Source Versions 23
5.3 Construction of Derived Versions 29

6 VERSION MODELS IN SCM SYSTEMS **31**
6.1 Overview 32
6.2 System Descriptions 32

7 RELATED WORK **44**
7.1 Related Work on Version Models 44
7.2 Related Disciplines 45

8 CONCLUSION **47**

REFERENCES **48**

A COMPARISON OF SCM SYSTEMS **57**

1 INTRODUCTION

Software configuration management (SCM) is the discipline of managing the evolution of large and complex software systems [124]. SCM serves different needs [41]:

- As a *management support discipline*, SCM is concerned with controlling changes to software products. It is this view of SCM which is addressed in the classical textbook by Bersoff [12] and the IEEE standard [58, 59]. 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 which assist developers in performing coordinated changes to software products. This view of SCM is described e.g. in the textbook by Babich [7]. 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 from intensional descriptions based on version rules.

In this paper, 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 database* combines product and version space. A specific version model is characterized by the way the version space is structured, by the decision which objects are versioned both externally (from the user's point of view) and internally (within the versioned database), by the relationships between version spaces of different objects, by the way reconstruction of old and construction of new versions are supported, and several other issues to be discussed throughout the paper (e.g., the interplay with workspace management and distribution).

SCM systems express version models in considerably varying ways. Many systems, including most commercial ones [103], are *file-based* and apply versioning to files and directories [104, 123, 46]. Various *language-based* approaches have been developed as well, based on modular programming languages [71], module interconnection languages [97], or system modeling languages [85]. These languages are typically used to represent versions of modules and relationships such as imports, include dependencies, etc. 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 [32, 93], object-oriented [39], and deductive [136] ones.

The variety of formalisms makes it difficult to compare the version models of different SCM systems with each other. Furthermore, 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 paper. 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 database, even if the corresponding system is not graph-based. For example, SCCS [104] and RCS [123] 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, e.g., textual languages for expressing version rules.

This paper 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 with respect to the product space (Section 3). After having introduced product and version space separately, their interplay is addressed in Section 4. The next section (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 which 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 Basic Notions

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. This includes requirements specifications, designs, documentations, program code, test plans, test cases, user manuals, project plans, etc. Each software object carries an *object identifier* (OID) which serves to identify it uniquely within a certain context. An external OID is a name assigned by the user, while a system-generated, unique OID may be used internally.

Software objects are rather *coarse-grained* units which 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, while many object-oriented data models represent coarse- and fine-grained units in a uniform way. Below, 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 tool-kit environments, software objects are stored as text files [104]. In contrast, syntax trees [53] or graphs [90] are used in structure-oriented environments. As to be 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* do not make any assumptions about the types of software objects to be maintained. All software objects produced througout the whole software life cycle project are subject to version control [123]. *Domain-specific models* are tailored towards specific types of software objects, e.g., abstract data types in algebraic specifications [34].

Software objects are connected by various types of relationships. *Composition relationships* are used to organize software objects vertically. For example, a software product may be composed of subsystems which in turn consist of modules. Objects which are decomposed are called *composite objects*. Objects residing at the leaves of the composition hierarchy are denoted as *atomic objects*. The root of a composition hierarchy is called *(software) product*.

The semantics of composite objects vary 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 which represents a subgraph of the product graph. All objects which 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 only be attached 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 establish horizontal, directed connections between objects. They include e.g. life cycle 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 relationship 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, e.g., text editors or graphical editors. A derived object is created automatically by a tool, e.g., a compiler or linker. Note that the classification of a software object as a source or derived object depends on the tool support which is available. Furthermore, software objects may partially be derived and partially be constructed manually. For example, the skeleton of a module body might be created automatically from its interface and subsequently be 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, i.e., *build dependencies* must be taken into account. In contrast, *source dependencies* represent relationships between source objects.

2.2 Examples

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:

- In Part 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 [44]).

- In Part c), we assume a data model which supports typed objects and relationships (e.g., an EER model as used in PCTE [93]). 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 relationships reflecting include dependencies. Build dependencies are not given explicitly and can be computed automatically from source dependencies and composition relationships.

- In Part 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 in POEM [78], corresponds directly to the logical structure displayed in Part a).

The reader may easily imagine other representations of `foo`. For example, in a file system representation source and derived objects may be stored in different directories.

3 VERSION SPACE

The last section described the product space without taking versioning into account. The current section focuses on the organization of the *version space*. We introduce basic notions such as versions, versioned objects, and deltas, and discuss several approaches to structuring the version space. After having introduced the product space and the version space separately from each other, we will investigate their interplay in the next section.

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

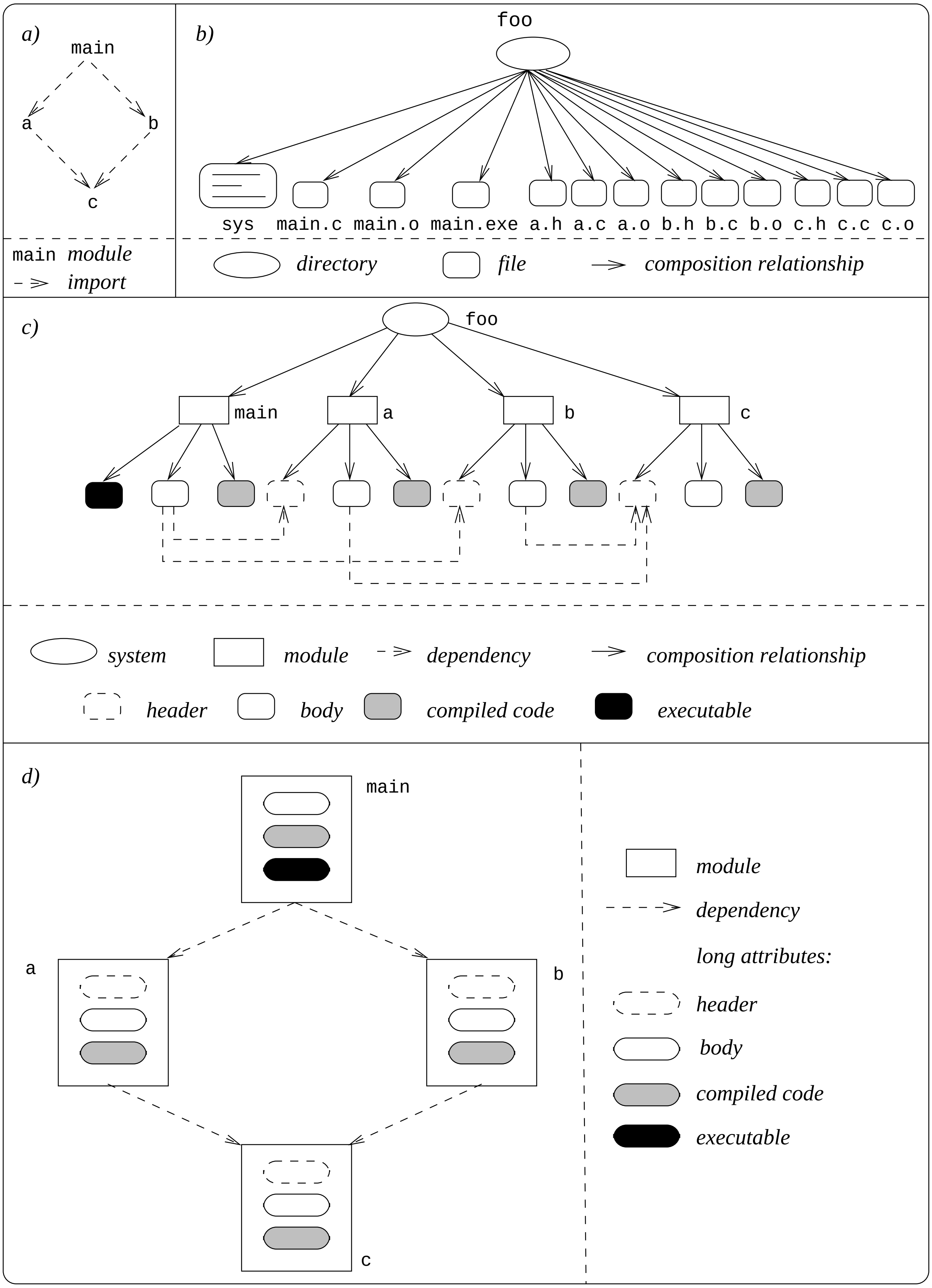

Figure 1: Different representations of a software product

3.1 Basic Notions

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 which may be put under version control. This includes e.g. files and directories in file-based systems, objects stored in object-oriented databases, entities, relationships, and attributes in EER databases, etc. Versioning can be applied at any level of granularity, ranging from a software product down to text lines.

A *versioned item* is an item which is put under version control. In contrast, only one state is maintained for an *unversioned item*. Versioning requires a *sameness criterion*, i.e., 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*, e.g., an OID in the case of software objects.

Within a versioned item, each version must uniquely be 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.

According to the type of evolution, versions are classified into revisions and variants. Versions evolving along the time dimension are called *revisions*. Revisions are maintained to recover from erroneous updates or to fix bugs in old versions having been delivered to customers. Alternative versions coexisting at a given time are denoted as *variants*. For example, variants of data structures may differ with respect to storage consumption, runtime efficiency, and access operations. Furthermore, a software product may support multiple operating systems or window systems.

All versions of an item share *common properties* which are denoted as *invariants*. These invariants can e.g. be represented 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, systems such as SCCS and RCS do not make any assumptions concerning the contents of versioned files. At the other end of the spectrum, versions must share semantic properties. For example, version control in algebraic specification [34] enforces that all versions of a module realize the shared interface.

Versions differ with respect to *specific properties* (e.g., represented by versioned attributes). The difference between two versions is denoted as 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 both v_1- and v_2-specific properties ($v_1 \setminus v_2$ and $v_2 \setminus v_1$, respectively).
- A *directed delta*, also called a change, is a sequence of (elementary) change operations which, when applied to one version v_1, yields another version v_2.

At the beginning of this subsection, a version has been defined as a state of an evolving item. Version models which focus on the states of versioned items are called *state-oriented*. Changes provide an alternative way of characterizing versions. In *change-oriented* models, a version is described in terms of changes applied to some base version. To this

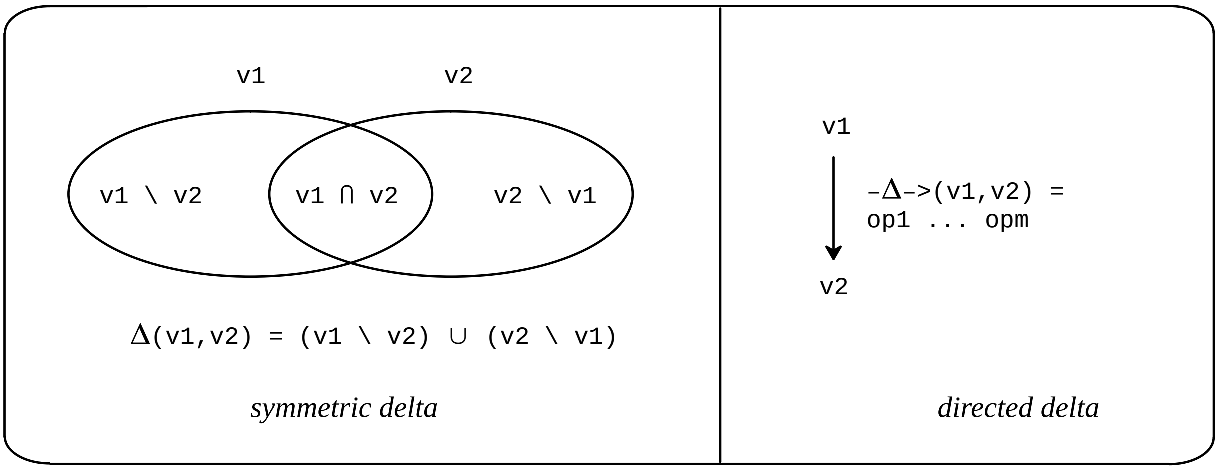

Figure 2: Deltas

end, changes are assigned *change identifiers* (CID) and potentially further attributes to characterize the reasons and the nature of a change.

A change is a partial function $c : V \longrightarrow V$, 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 base version b:

$$v = c_1 \ldots c_n(b) = c_n(\ldots c_1(b) \ldots)$$

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

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

Extensional versioning supports retrieval of previously constructed versions. Each version is typically identified by a unique number. The user interacting with the SCM system retrieves some version v_i, performs changes to 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 database [104]; in others, explicit operations are provided to freeze mutable versions [132]. Furthermore, immutability may be enforced selectively, e.g., by distinguishing between mutable and immutable attributes [40].

Intensional versioning is applied when flexible, automatic construction of consistent versions in a large version space needs to be supported. The term “potential versions” [87] emphasizes that it is not important whether a certain version has been constructed before or is constructed anew. In case of intensional versioning, the version set is defined by a predicate:

$$V = \{v | c(v)\}$$

c defines the *constraints* which have to 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

$$q : A_1 \times ... \times A_n \longrightarrow V$$

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 [104] and RCS [123] to conditional compilation, as e.g. supported with the C programming language [65]. 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 which 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 instead by combined into a single SCM system.

To represent the version space, versions may be organized in a *version graph* whose nodes and edges correspond to (groups of) versions and their relationships, respectively. Alternatively, a *grid* may be used to arrange versions in an n-dimensional space, where each dimension corresponds e.g. to a version attribute.

3.2 Examples

After having introduced a conceptual framework for versioning, we provide an overview of modeling approaches which have been realized in different SCM systems. These approaches are described using the terminology introduced above, and their merits and shortcomings are discussed as well.

Many SCM systems use *version graphs* for representing the version spaces of items. Since version graphs focus on states rather than changes and assume an explicitly given set of versions, this approach may be classified as state-oriented, extensional versioning. Although a rich variety of version graphs is conceivable, the mainstream of SCM systems are based on a small number of graph types which are discussed below.

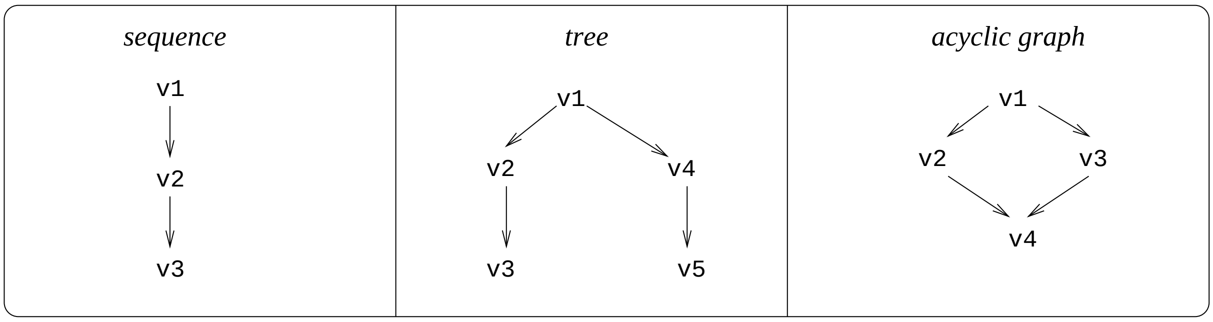

Figure 3: Version graphs (one-level organization)

In the simplest case (*one-level organization*), versions are connected by relationships of a single type, denoted as *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, e.g. 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 non-leaf versions may be created, e.g., in order to maintain old versions which have already been delivered to a customer. In an acyclic graph, a version may have multiple predecessors, e.g., in order to merge a bug fix in an old version with the currently developed version. Several SCM systems use one of these different kinds of one-level organization, e.g., sequences in NSE [2] and acyclic graphs in PCTE [93]. DAMOKLES [32] 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 DAG.

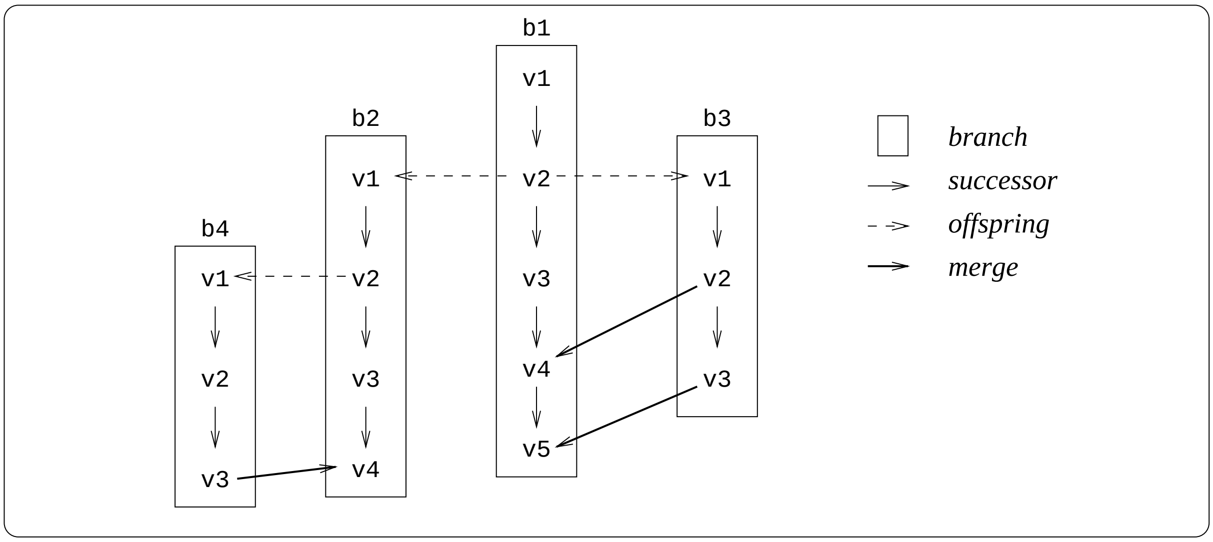

Figure 4: Version graphs (two-level organization)

Some SCM systems use a more sophisticated organization of the version graph which distinguishes between multiple relationship types. The organization employed in ClearCase [73] is illustrated in Figure 4. A version graph is composed of *branches* each of which consists of a sequence of revisions (*two-level organization*). Branches may not be joined, but merging may be performed to propagate changes performed on one branch to another branch. The application of merge operations is recorded in the version graph.

Version graphs as presented above suffer from several limitations:

1. Branches are used for representing variants. However, only one dimension of variation can be modeled in this way. Otherwise, this approach breaks down because the number of branches explodes combinatorially.

2. Version graphs are prone to the *multiple maintenance problem* [7]. It may happen that a certain change, e.g., a bug fix, must be applied to multiple branches so that the same work needs to be repeated over and over again.

3. There is no direct way to determine the changes which have gone into a certain version. Due to the state-oriented approach versions are named, while changes are

anonymous. The changes included in some version are inferred from the topology of the version graph. This may become difficult when merging is applied extensively and in complicated ways.

To investigate the first problem, 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`). To model the version space, version graphs may be generalized in order to represent multi-dimensional variation. For example, in Figure 5a) versions are organized into *clusters* which are used to construct classification hierarchies [33]. Alternatively, versions may be arranged in a *grid* (Figure 5b).

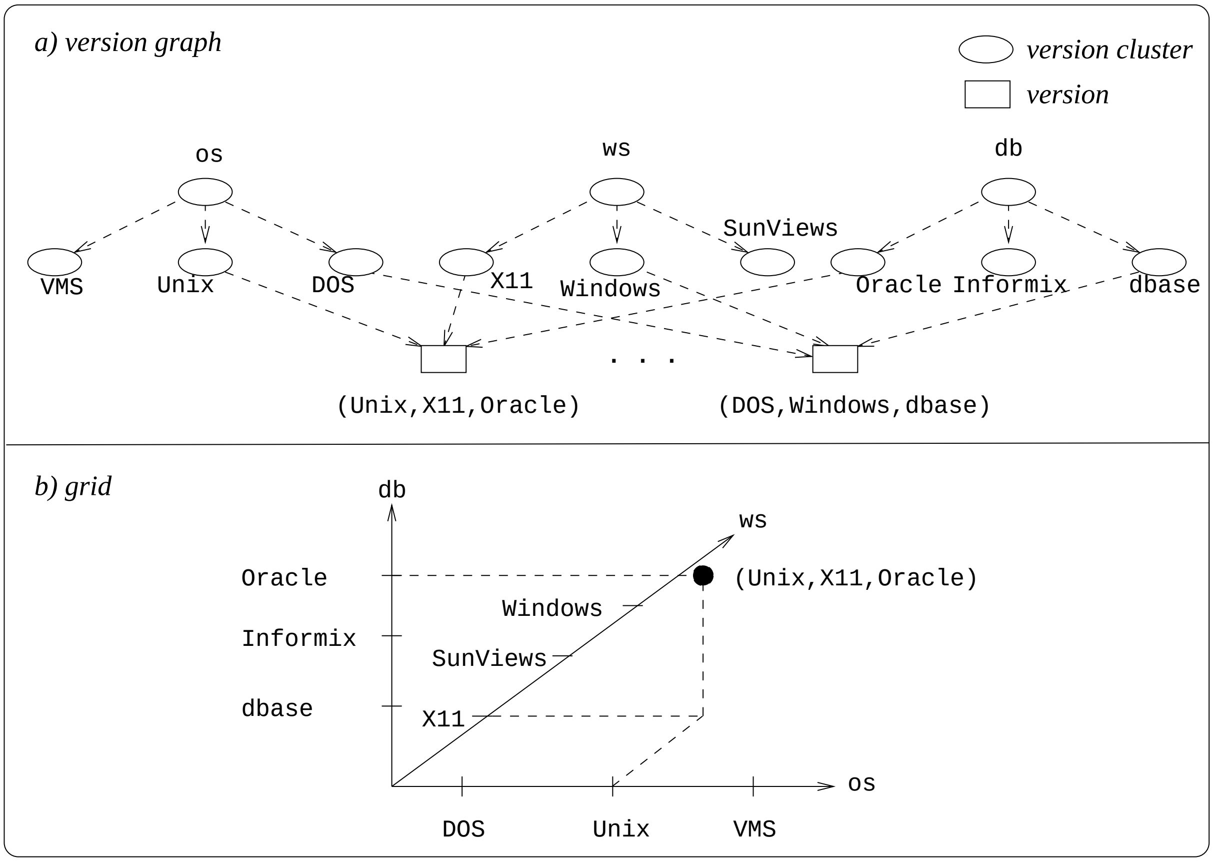

Figure 5: n-dimensional variant space

Figure 5 only illustrates 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* [100]). In case of the version graph (Figure 5a), one level may be added at the bottom, and successor relationships may be introduced to represent histories.

The second problem can be addressed by leaving the shape of the version graph unchanged and providing *merge tools* [17] which automate redundant work as far as possible. Still, shared fragments occur in multiple copies, at least at the logical level[2]. Keeping redundant copies in multiple versions is called *variant segregation* [83]. Alternatively, all

[2]Shared fragments may be stored only once at the physical level.

variants can be stored in a single source object (*single-source variation*), e.g., by applying conditional compilation. Then, each fragment exists in only one copy, and a change affects all variants. Since variants need no longer be represented in the version graph, the version graph ideally degenerates to a sequence of revisions each of which corresponds to a multi-variant software object.

Change-oriented versioning solves the third problem by introducing CIDs and using these to describe versions in terms of changes. Furthermore, the multiple maintenance problem is addressed by supporting flexible combination of changes[3]. Change-oriented versioning has been realized e.g. in COV [52], PIE [50], and Aide-de-Camp [112, 25], which are based on different notions of change.

In PIE, any subset of changes can be applied in any order. In Aide-de-Camp, changes are totally ordered with respect to their creation time: if c_1 was created before c_2 and both changes are applied to some version v, then c_1 is always applied before c_2. Under this assumption, the version space may be visualized by a matrix whose lines and columns correspond to versions and changes, respectively (Figure 6a). 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.

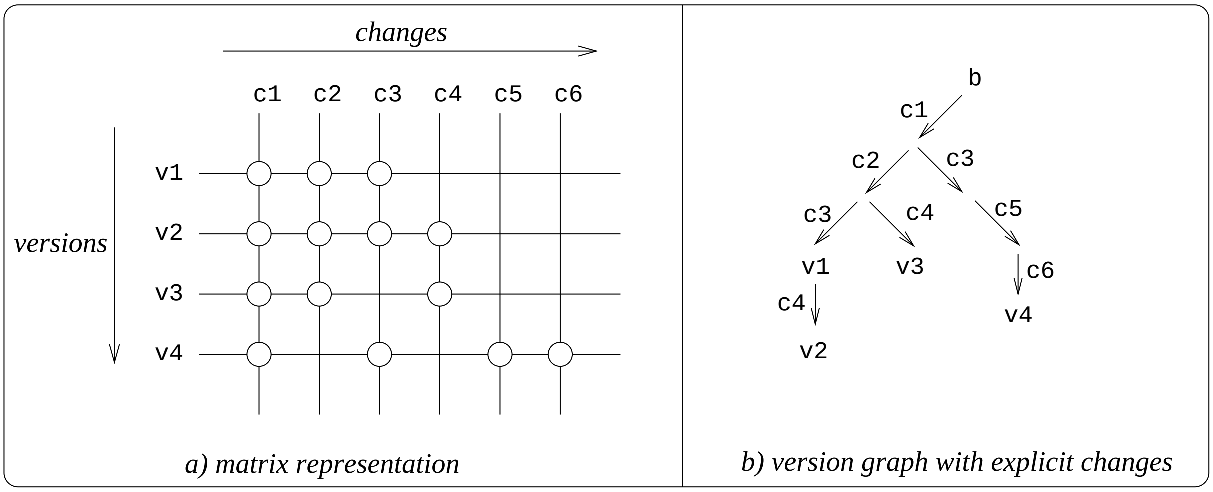

Figure 6: Change space

Figure 6b) illustrates the relations to the version graphs as introduced in Figures 3 and 4, respectively. Part b) arranges the versions shown in Part a) in a graph, where nodes and edges correspond to versions and changes, respectively. `b` denotes the base version, and intermediate 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, respectively. In particular, it makes explicit when certain changes were applied to multiple versions. For example, `c4` was used to construct both `v2` and `v3`.

All change-oriented SCM systems developed so far are based on a single concept, namely the change. Both revisions and variants are represented by changes. In COV, versions are arranged in an *option space*, where each option denotes a property which

[3]The relation to merge tools based on version graphs will be discussed in Section 5.

may be either included or omitted. For example, `Fix` and `Unix` may denote a bug fix and the Unix variant, respectively. A version is specified by all options to be included. In contrast to Aide-de-Camp and PIE, there is no order defined on the options. Thus, the option space is modeled as an n-dimensional grid, where each dimension corresponds to a boolean option.

4 INTEGRATION OF PRODUCT SPACE AND VERSION SPACE

In Section 2, the product space was described under the assumption that only one version of each object 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 database*. In particular, we investigate which objects of the product space are put under version control, how versions of different objects are interrelated, and in which way versions are represented in the database.

4.1 Basic Notions

In the last section, we introduced the term "item" to denote anything which can be versioned, including entities, relationships, and attributes. In general, versioning can be applied at any level of granularity. However, from the user's point of view versioned items have a minimal granularity. The user normally selects versions of *software objects*, which are rather coarse-grained items. On the other hand, an SCM system may operate at a much finer granularity in order to store, compare, and construct versions of software objects.

To emphasize this point, we distinguish between *external* and *internal versioning*. Items whose versions can be identified and selected at the user level are called externally versioned. Internally, more fine-grained items may be versioned as well, e.g., declarations and statements in program modules. These items are not selected *per se* by the user; rather, they are automatically selected as part of a version of some software object.

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, e.g. modeled 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.

A version of a composite object is called a *configuration*. To construct a configuration, object references have to be *bound* to specific versions. The act of selecting a version of some referenced object is called binding. In general, this is performed with the help of *version rules* to be discussed later (see Section 5). Binding can be performed either statically or dynamically. *Static binding* means that a bound configuration is constructed before any object is accessed in the versioned database [76]. *Dynamic binding* is performed only when an object is actually accessed [73].

To represent versions in the database, *deltas* are used both at the coarse-grained and the fine-grained level. The selection of an appropriate delta representation is driven by different requirements to be considered at the physical and logical level, respectively. Storage and runtime efficiency have to be achieved at the physical level. At the logical level, deltas are used to compare versions in such a way that a 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 which has to meet the requirements stated in its intensional description. Accordingly, we distinguish between *physical* and *logical deltas*. While many SCM systems use a single delta representation for both purposes [87], physical and logical deltas may also be separated (e.g., syntactic merging of program modules stored as text files [17]).

The distinction between directed and symmetric deltas, as illustrated in Figure 2 on page 9, can readily be transferred to the implementation level. Using *directed deltas* [122], a version is constructed by applying a sequence of changes to some base version. In 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 [47], or the fragments are decorated with *control expressions* for determining the versions in which they are visible (*interleaved deltas* [104, 76]).

4.2 Examples

AND/OR graphs [121] 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 on page 12 can be regarded as an AND/OR graph as well: Version clusters and versions correspond to OR nodes and AND nodes, respectively. However, below 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 nodes 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 *unbound*. A *configuration* is represented by a subgraph consisting of all nodes which are transitively reachable from the root node of the configuration. If this subgraph only contains AND nodes, the configuration is called *bound*. Otherwise, it is *generic* and represents a set of bound configurations which can be constructed by eliminating the OR nodes (for each OR node, one of its successors is selected). The number of potential bound configurations is obtained by multiplying the fan-outs of all OR nodes.

In the following, we take the examples of product graphs given in Figure 1 on page 7 up again and compare several approaches to versioning the product graph. Figure 7 shows AND/OR graphs which are all based on Part b) of Figure 1[4]. In Figure 7a), only atomic objects are versioned. `foo` represents a generic configuration. In Figure 7b), `foo` is

[4]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).

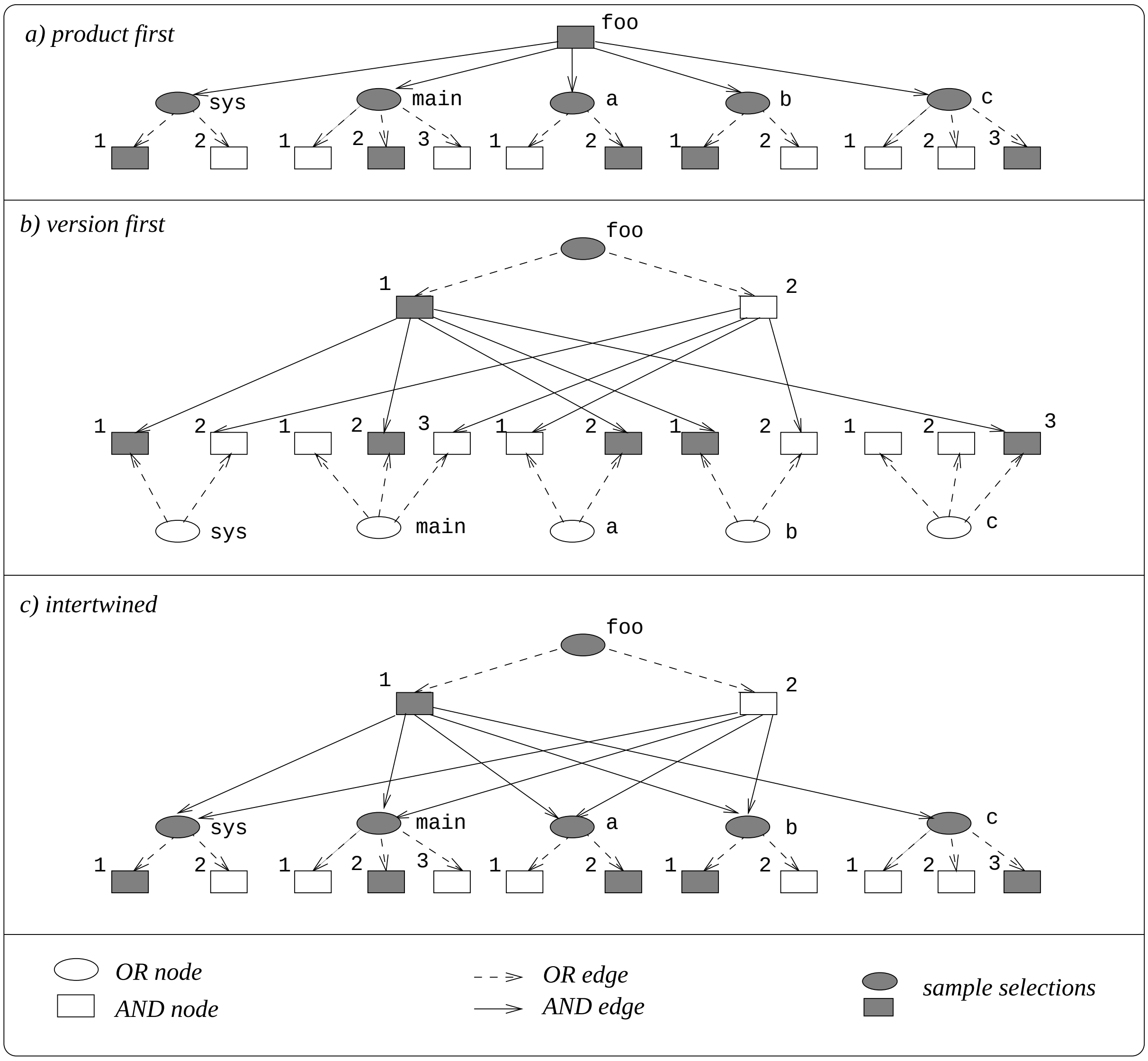

Figure 7: Different kinds of AND/OR graphs and selection orders (I)

versioned as well. Each version of `foo` corresponds to a bound configuration. In Figure 7c), references to components are unbound, 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, which starts at the root node of the configuration and performs a transitive closure over outgoing edges:

- *Product first* (Part a) means that the product structure is selected first; subsequently, versions of components are selected. This approach is followed e.g. 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* (Part 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 [93] serves as an example of an SCM system using this organization.

- *Intertwined* (Part c) means that AND and OR selections are performed in alternating order. The intertwined organization is used e.g. by ClearCase [73], which versions both files and directories. Again, this selection scheme supports structural versioning.

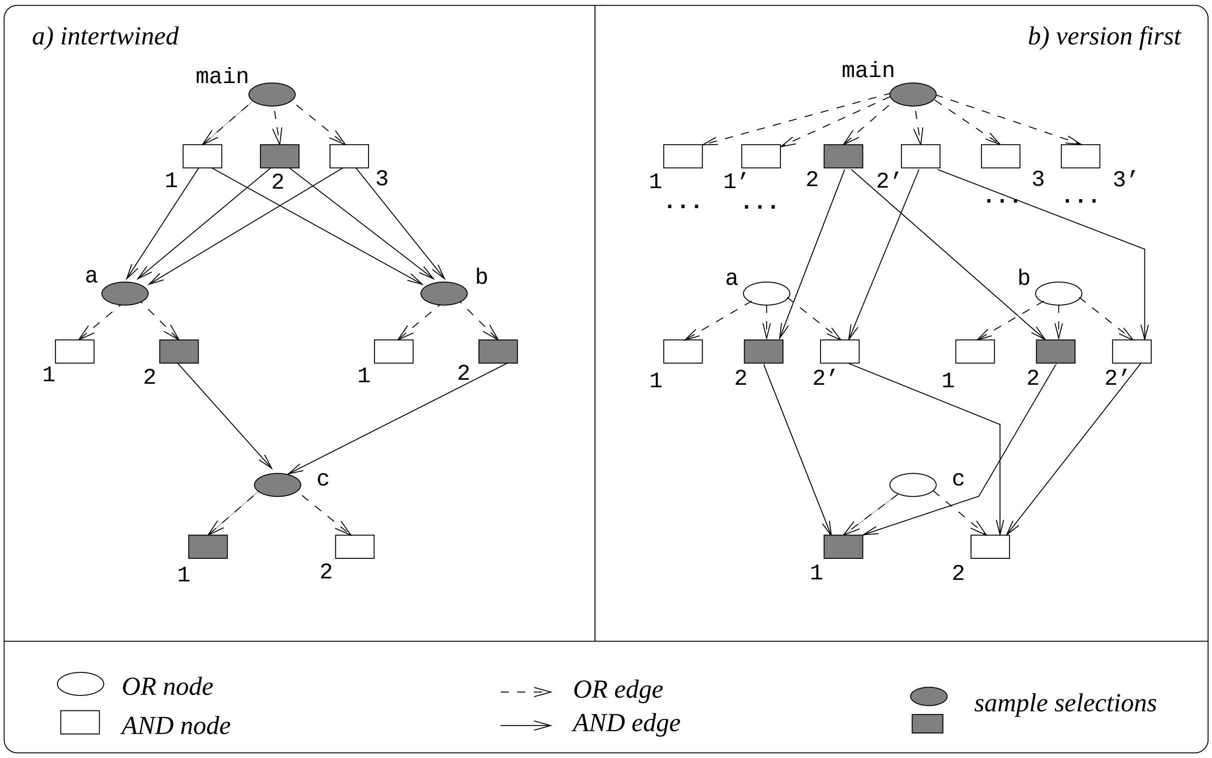

Figure 8: Different kinds of AND/OR graphs and selection orders (II)

So far, we have applied versioning only to the product graph organization shown in Part b) of Figure 1. In Figure 8, the alternatives "intertwined" and "version first" are illustrated for the product graph of Figure 1d)[5]. In contrast to the previous figure, AND edges represent dependencies instead of composition relationships. Intertwined selection is performed e.g. in Adele [35], while "version first" is realized e.g. in POEM [78].

To illustrate the differences between a) and b), let us assume that a bug is fixed in module `c` which does not affect its interface. A new version `c.2` is created which is to be included into the new release of our sample product `foo`. In a), a new configuration is constructed in which `c.2` is selected instead of `c.1`[6]. 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

[5]Again, each module version is represented as a single node.

[6]Note that bound configurations constructed from generic configurations can be represented in the AND/OR graph as well (as AND nodes whose outgoing edges point to the component versions).

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 configurations, respectively (in b), the versions of `main` play both roles simultaneously).

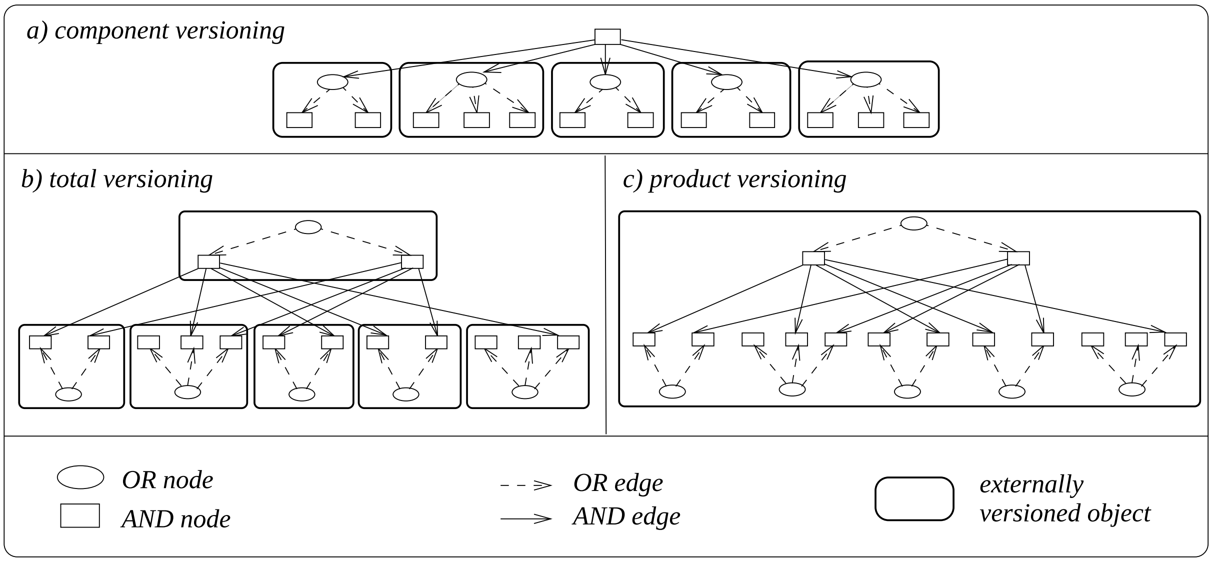

Figure 9: External versioning

In the last subsection, we introduced a distinction between *external* and *internal versioning* and classified the former into component, total, and product versioning, respectively. This is illustrated in Figure 9, where externally versioned objects are surrounded by boxes. The AND/OR graph of Part a) was taken over from Figure 7a), while the graphs shown in Parts b) and c) were both copied from Figure 7b). Note that the topology of an AND/OR graph only 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 case of product versioning the user may select versions of non-root objects, but this is done in the version space attached to the whole product.

In case of *component versioning*, the product structure is selected first. Typically, extensional versioning is applied at the component level, and versions of components are organized into version graphs (see e.g. RCS [123]). A configuration is constructed by assembling versions of components; this is called *composition model* in [41]. Construction of a consistent configuration is difficult if the version graphs of different components are not or only weakly related to each other. Figure 10 shows a sample configuration whose components are located at different places in the respective version graphs[7]. Construction of consistent configurations is supported by version rules, i.e., intensional versioning is applied at the configuration level. Note that the granularity of composition is an "atomic" software object (no merging at the fine-grained level).

Total versioning generalizes component versioning in that all objects are versioned rather than only the leaves of the composition hierarchy. Total versioning offers several advantages over component versioning. In addition to supporting structural versioning, composite and atomic objects are versioned uniformly, and versioning of composite objects

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

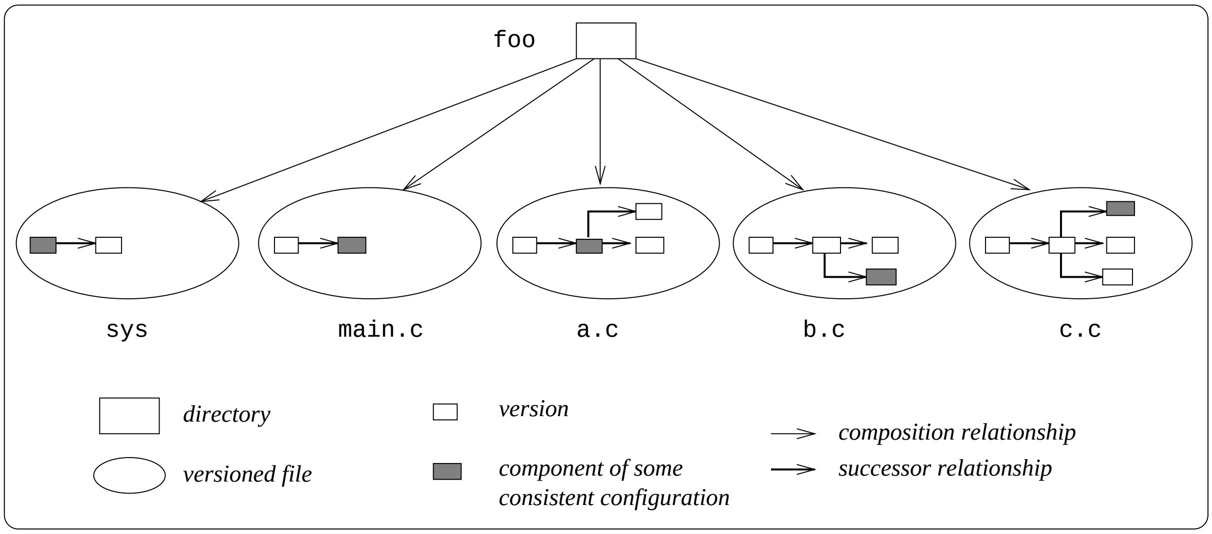

Figure 10: AND/OR graph augmented with successor relationships

can be used to ease the construction of consistent configurations. Total versioning may be combined with both the "version first" and the "intertwined" selection order. For example, PCTE [93] supports extensional versioning of composite objects ("version first"). In contrast, in ClearCase [73] 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 version rules (intensional versioning with dynamic binding).

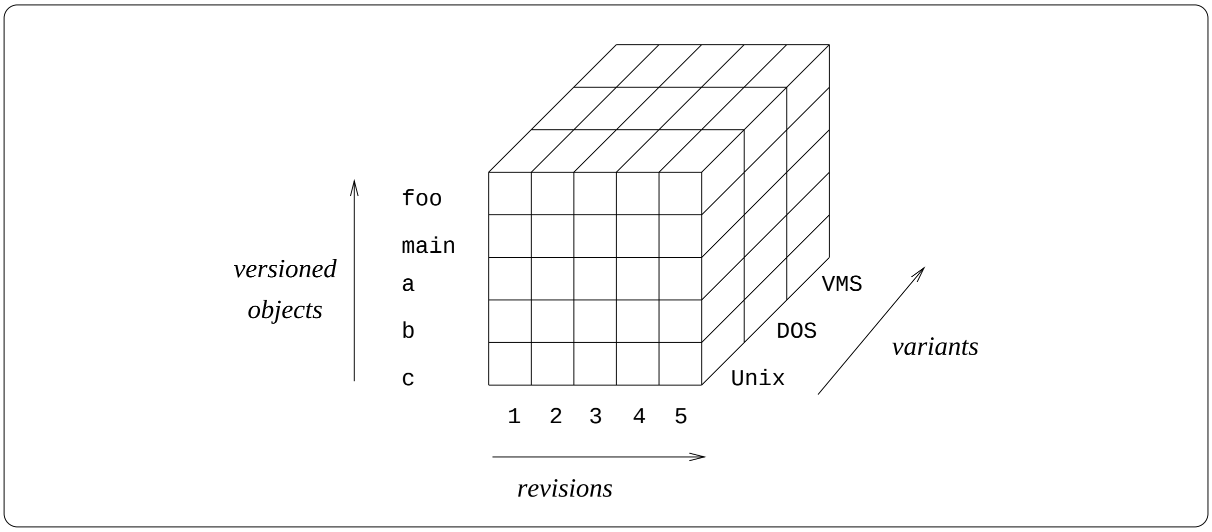

Figure 11: Product versioning

Product versioning differs from total versioning by arranging versions of all objects in a uniform, global version space. 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-oriented SCM system Vodoo [100] is illustrated in Figure 11, where versioned objects, revisions, and variants are organized into ortogonal dimensions.

All *change-oriented systems*, e.g., COV [52], Aide-de-Camp [112], and DaSC [82], 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-oriented 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 in the *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 COV) is described in [86].

Deltas are used to present versioned objects in the database. A given version model may be realized in multiple ways, taking requirements such as efficiency or flexible intensional construction of consistent configurations into account. In the following, several examples of delta representations are presented without elaborating on efficiency considerations. Intensional versioning with deltas will be taken up again in Section 5.

RCS [123], 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. While RCS deltas are fine-grained, change-oriented SCM systems such as PIE [50] and DaSC [82] 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 which are stacked on top of each other[8].

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* (version rules) stating e.g. that a certain version can be selected only when configuring a `Unix` variant.

Adding control expressions results in an *interleaved delta* scheme which is realized e.g. in SCCS [104] and DSEE [76]. Note that both systems implement a version model similar to RCS, which is based on directed deltas. In SCCS and DSEE, fragments of text lines are tagged with the set of versions in which they are included.

While SCCS and DSEE support extensional versioning (of text files), *conditional compilation* employs interleaved deltas for intensional versioning. For example, a C source file may contain preprocessor directives controlling which fragments are passed to the compiler [65]. This is illustrated in Figure 12a) which shows preprocessor directives in a source file varying with respect to the operating system. Figure 12b) 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 [125]). 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 in C (single-source variation), different versions of this component are stored in separate files (variant segregation).

[8]For further details, see the description of PIE in Section 6, in particular Figure 22 on page 34.

```
/* 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

Single-source variation solves (or at least reduces) the multiple maintenance problem, but editing of source files cluttered with control expressions may confuse the user. *Multi-version editors* overcome this problem by hiding control expressions (see e.g. P-Edit [68], MVPE [107] and COV [52]). 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 such that all changes are specific to the `Unix` version.

5 INTENSIONAL VERSIONING

When complex software products have to be maintained which evolve into many versions, it is crucial that an SCM system supports automatic construction of versions from an intensional specification [24]. Intensional versioning can be applied to both source and derived objects. After introducing a common conceptual framework, we mainly focus at the construction of source versions. Construction of derived versions (system building) is only discussed briefly.

5.1 BasicNotions

Intensional versioning is driven by *version rules*. In case of source versions, version rules are used to select versions of items stored in the database. In case of derived versions, version rules state how a derived version is constructed from the versions on which it depends (both source and derived ones).

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

The *scope* of a rule defines the items to which it is applied. *Type-level rules* are applied to all instances of a certain type. For example, a type-level (derivation) rule may state that each `.o` file is compiled from the corresponding `.c` file and all included `.h` files. An *instance-level rule* refers to (one or more) specific instances of a certain type (e.g., in our sample software product `foo` the compiled code `a.o` of module `a` is derived from `a.h` and `a.c`; see Figure 1 on page 7).

Rules may be ordered with respect to their *strictness*. A *constraint* is a mandatory rule which must be satisfied. Any violation of a constraint indicates an inconsistency (e.g., the `SUN` and the `VAX` variant must not be selected simultaneously). A *preference* is an optional rule which is only applied when it may be satisfied (e.g., released module versions are preferred). Finally, a *default* is also an optional rule, but it is weaker than a preference. A default rule is only applied when otherwise no unique selection could be performed (e.g., the latest version of the main branch in a version graph may be selected by default).

Rules may also be given *priorities*. Rules with high priorities are considered before low-priority rules. A priority may be assigned explicitly, or it may be defined implicitly by textual ordering. Priorities may be combined with strictness classes, e.g., by assigning priorities to preferences [72].

A *configurator* constructs a version by evaluating version rules against a versioned database. 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 $a_1 \ldots a_n$, i.e., 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; nondeterminism may still be involved in the evaluation process (e.g., with respect to the order in which subexpressions are evaluated).

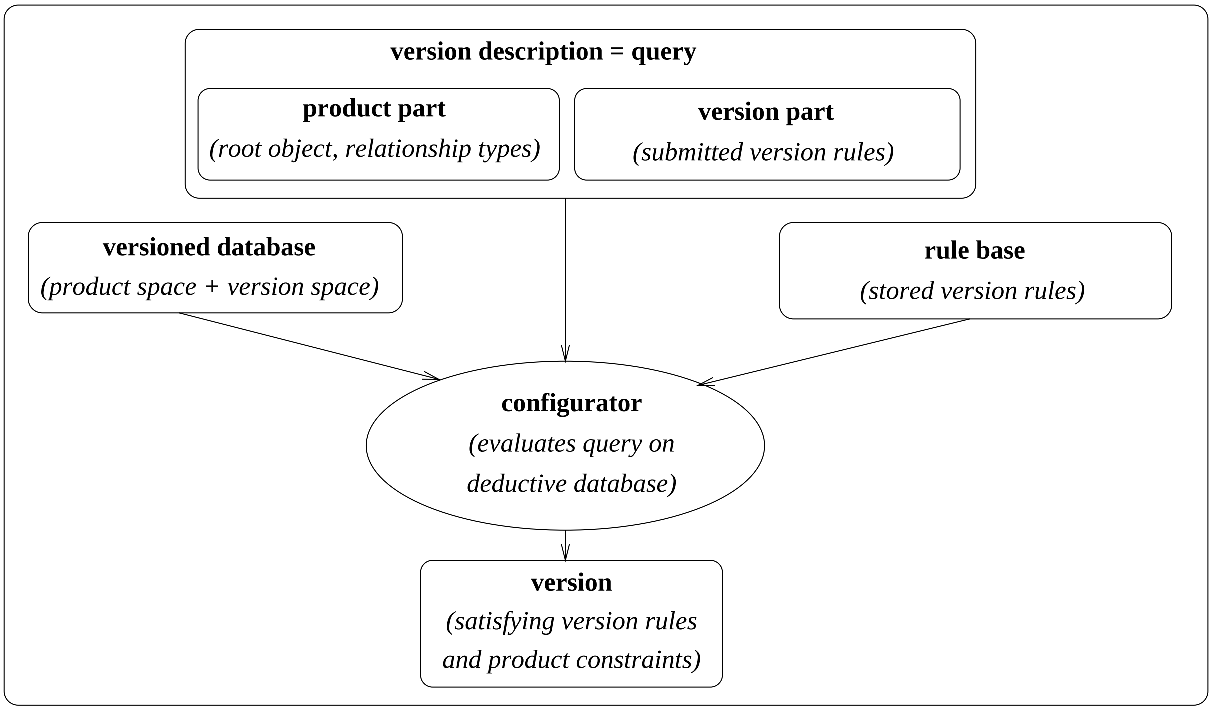

Figure 13: Rule-based version construction

In a *rule-based framework*, version construction is modeled as the evaluation of a query against a *deductive database* [99, 98] (Figure 13). The deductive database consists of a versioned database — which in turn is composed of product space and version space — and a rule base which contains stored version rules. The product part of a query describes the software objects to be configured, e.g., by supplying a root object and a set of relationship types. The version part of a query consists of submitted version rules which are considered in addition to the stored version rules. The configurator constructs a version which has to satisfy both version rules and product constraints (syntactic or semantic consistency).

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. In the simplest case, the configurator performs *random choices* without exploring multiple alternatives. More sophisticated configurators search the solution space in *depth first* order and backtrack from wrong decisions. Alternatively, multiple alternatives can be considered in parallel (*breadth first* search).

5.2 Construction of Source Versions

Although construction of source and derived versions can be handled in a uniform framework, the problems to be solved still differ considerably, and many SCM systems support intensional versioning by distinct mechanisms at the level of source and derived objects, respetively. Therefore, both the current and the next section start with a problem statement before discussing version rules and configurators.

5.2.1 Problem

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

For example, *component versioning* in SCCS or RCS is very flexible since any combination of component versions can be composed into a configuration (see also Figure 10). 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-oriented versioning* reduces the combinability problem by grouping logically related modifications of multiple components. Thus, we do not have worry about which versions of components actually fit together. On the other hand, the selection problem has not disappeared. Rather, it has been moved from the product space to the version space [86]. In case of unconstrained combination of changes (each change may either be applied or skipped), there are 2^c potential configurations for c changes, i.e. the number of potential configurations even grows exponentially in c.

The challenge of intensional versioning of source objects consists in 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, version 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 the 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 configurator is employed, the user must be warned if a new version is constructed that has never been configured before. While old versions can be assigned levels of "confidentiality" (e.g., tested or released), a new version cannot be trusted blindly. Therefore, the version constructed by the configurator is subject to *quality assurance* (e.g., testing). Potentially, changes to the constructed version need to be performed (*correction delta*).

5.2.2 Version Rules

Version rules take on different forms, depending on the way the version space is structured (see Section 3). Figure 14 provides some typical examples which refer to the revision, variant, and change space, respectively. Version rules are stated as logical formulas in some fancy notation, abstracting form the actual syntax as implemented in different SCM systems.

revision space
(1) $\texttt{t = max} \wedge \texttt{branch = main}$
(2) `name = alpha_release`
(3) `merge(v1,v2)`
variant space
(4) $\texttt{os = Unix} \wedge \texttt{ws = X11} \wedge \texttt{db = Oracle}$
(5) $\neg\ \texttt{(os = DOS} \wedge \texttt{ws = X11)}$
change space
(6) `c1 c2 c4`
(7) $\texttt{c2} \Rightarrow \texttt{c1}$
(8) $\texttt{c1} \otimes \texttt{c2} \otimes \texttt{c3}$

Figure 14: Examples of version rules

In the *revision space* category, version rules are listed which are based on version graphs[9]. Rules (1) and (2) select some already existing version from the version graph in order to compose a configuration from components. Rule (1) selects the latest revision on the main branch. Rule (2) uses a symbolic name to retrieve the version belonging to the alpha release. Rule (3) is used to construct a new version by merging two versions `v1` and `v2`.

In the *variant space*, version rules refer to values of variant attributes. Rule (4) identifies a variant by specifying operating system, window system, and database system, respectively. Rule (5) expresses a constraint on the combination of attribute values: the `X11` window system is not available under the `DOS` operating system.

[9]Recall from Section 3 that version graphs can also represent variants to a limited extent. Thus, the categorization used in Figure 14 is a bit simplified.

In the *change space*, Rule (6) specifies a version in terms of the changes to be applied (see also Section 3). Rules (7) and (8) specify further relationships which describe consistent change combinations. Rule (7) states that inclusion of change `c2` implies that `c1` be included as well (`c2` is based on `c1`). Rule (8) states that changes `c1`, `c2`, and `c3` are mutually exclusive, 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 15: Scoping of version rules

So far, we have only discussed the *version parts* of version rules (see Figure 13). The *product part* describes the scope of a version rule in the product space. In Figure 15, a version 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 such that the same variant is selected throughout the whole product. The product part of Rule (3) denotes all modules which are reachable from `b` by a reflective and transitive closure over relationships of type `DependsOn`, i.e., the rule applies to `b` and to all modules on which `b` depends transitively.

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

Figure 16: Ordering of version rules

Version rules can be ordered in strictness classes. The *constraint* (1) in Figure 16 ensures that all selected versions belong to a given variant. By means of the *preference* (2), a checked out version is selected provided it is available. Otherwise, the *default* (3) selects the most recent version.

Finally, we may distinguish between *stored* and *submitted* version rules. The rules given in Figure 16 would typically be submitted in a query in order to specify properties requested by the user. Rule (5) in Figure 14 serves as an example of a stored version rule expressing a constraint which has to be fulfilled by all configured versions.

5.2.3 Configurators

Configurators may be classified with respect to the underlying version model (state- or change-oriented) and with respect to the computational paradigm (functional, rule-based). This results in four combinations which are described in turn below.

A *state-oriented functional* configurator operates on a versioned database represented by interleaved deltas. Conditional compilation in C [65] and configuration construction at the coarse-grained level in PCL [125] serve as typical examples. The versioned database is usually traversed in a "context-free" manner (e.g., sequences of text lines in conditional compilation and trees of components in PCL). Version rules take the form of control expressions as shown e.g. in Figure 12 on page 21. The configurator is supplied with a tuple of attribute values $(a_1, ..., a_n)$ and constructs some version v. Control expressions are evaluated against this tuple to select the relevant constituents of v. When applied to a composite object co, expressions may also be used to transform $(a_1, ..., a_n)$ into a new tuple $(a'_1, ..., a'_m)$ which is then recursively employed 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.

A *state-oriented 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 which is traversed along "context-sensitive" relationships (e.g., dependencies between modules in Adele [35]). For each OR node reached in the course of the configuration process, a successor is selected with the help of version rules. As version selections are performed, constraints are added which 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 (see Figure 8a, where node `c` may be reached via multiple paths from the root). 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 17. Part a) shows an AND/OR graph whose AND nodes (versions) are annotated with version rules[10]. For each version, a triple of attribute values denotes the variants to which it belongs; `*` denotes 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. Part b) shows a query which 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.

A *change-oriented functional* configurator is supplied with a sequence of changes. Internally, the database may be stored using either interleaved or directed deltas (e.g., Aide-de-Camp [112] and PIE [50], respectively). Conceptually, the changes are applied in order to some base version. Inconsistencies are detected when a change operation fails, e.g., insertion of a text line at an undefined location.

A *change-oriented rule-based* configurator differs from its functional counterpart by exploiting version rules in the configuration process. Version 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 e.g. in COV [86]. For example, consider the mutual exclusion rule (8) in Figure 14 on page 24. 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.

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

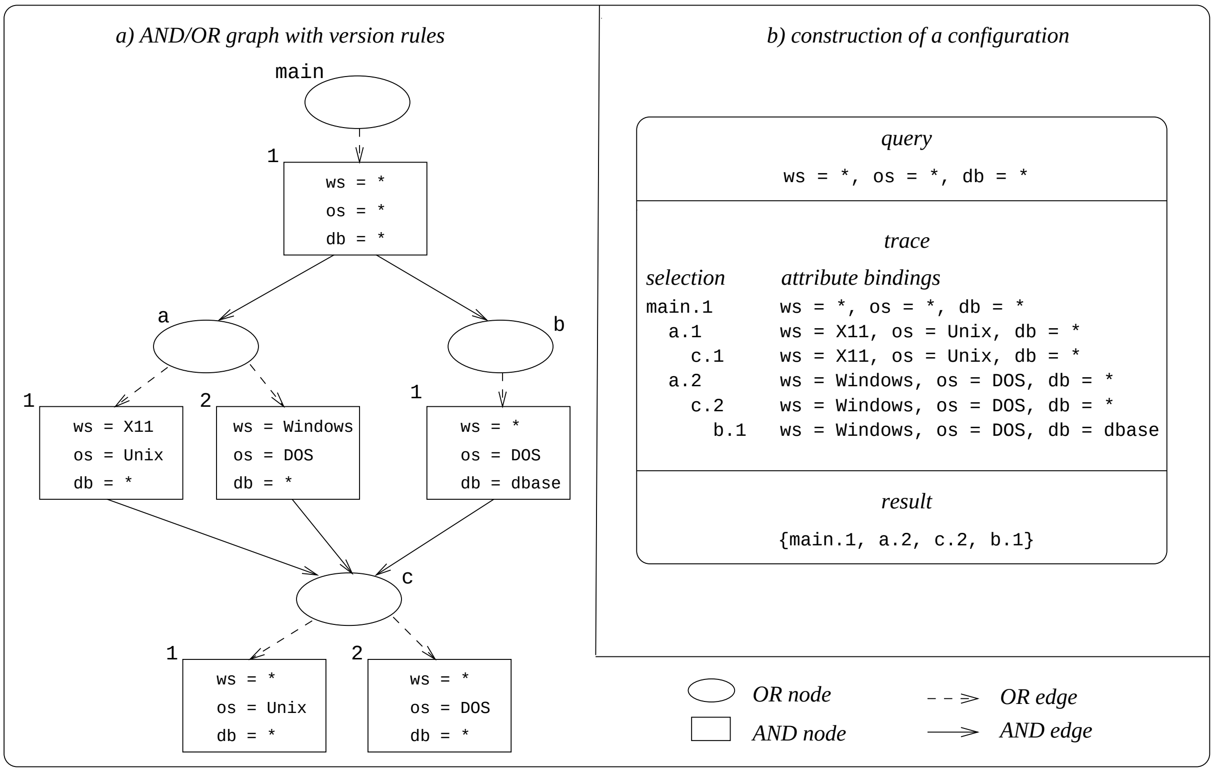

Figure 17: Rule-based construction of a configuration

So far, we have distinguished between state-oriented and change-oriented configurators. Although state- and change-oriented SCM systems may differ considerably at first glance, the borderline is in fact rather fuzzy. This was already demonstrated by discussing the relationships between version graphs and change-oriented versioning in Section 3 (see Figure 6 on page 13). In particular, state-oriented SCM systems often offer *merge tools* [17, 80] to combine versions/changes. Note that merge tools have been used in both state- and change-oriented SCM systems.

Merge tools combine versions or changes. They may be classified as follows (Figure 18):

- *Raw merging* simply applies a change in a different context. For example, in Figure 18a) change `c2` was originally performed independently of change `c1` and is later combined with `c1` to produce version `v4`. Raw merging is supported e.g. by SCCS [104]. Later, it has been generalized in change-oriented SCM systems such as Aide-de-Camp [112].

- A *2-way merge tool* (Figure 18b) 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 2-way merge tool can merely detect differences, but it cannot resolve them automatically.

- To reduce the number of decisions to be performed by the user, a *3-way merge tool* (Figure 18c) consults a common base version `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 which can be resolved either manually or automatically (the latter of which is not recommended).

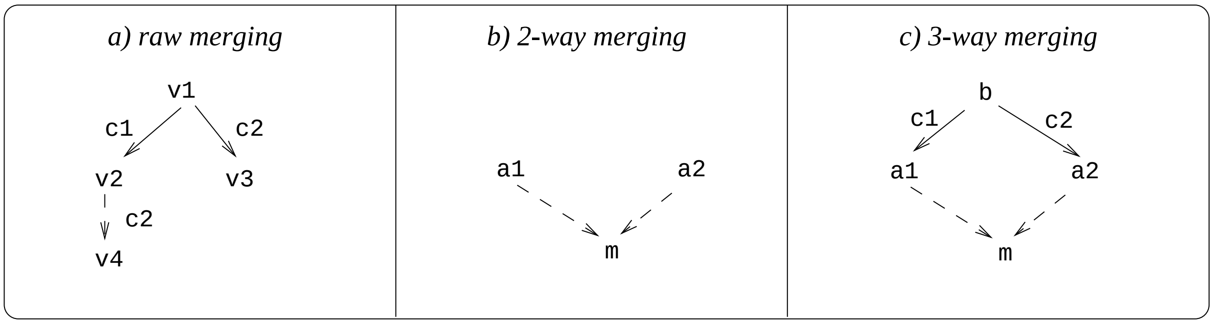

Figure 18: Types of merging

Conflict detection is an important advantage of 3-way merging. For example, let us assume that conflicting changes to the name of a procedure were carried out in the changes `c1` and `c2` in Figure 18 (Parts a) and c), respectively). In case of raw merging, the change performed in `c1` is simply overridden. In contrast, 3-way merging reports a conflict to the user.

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

- *Textual merging* is applied to text files [1]. Almost all commercial SCM systems support textual merging [103]. 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 [73].

- *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 discussions. However, syntactic merging has merely been realized in a few research prototypes [17, 130].

- Finally, *semantic merging* takes the semantics of programs into account (see [13] for a collection of the most important papers on semantic merging). 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 which is neither too strong nor too weak (and is decidable). Furthermore, the merge algorithms which have been developed so far are only applicable to simple programming languages (e.g. not C or C++). For these reasons, semantic merge tools have not (yet?) made their way into practice.

Note that merge tools try to detect conflicts in the *product space*. Alternatively, conflict detection can be performed in the *version space*. This approach is followed in COV [86], where version rules are used to constrain change combinations. In the example discussed above, changes `c1` and `c2` could be declared as mutually exclusive by the rule `c1` $\otimes$ `c2`. However, this should only be done if there is no way to apply `c1` and `c2` simultaneously. Otherwise, the conflict would be resolved in the automatically merged version `c1 c2` (*correction delta*).

5.3 Construction of Derived Versions

In Subsection 2.1, the process of creating derived objects was denoted as system building. A configurator for constructing derived versions is called a *builder*. The builder executes a build plan which is composed of build rules. Each build rule describes a build step, i.e., an action to be performed such as running a formatter, compiler or linker.

5.3.1 Problem

System building is concerned with keeping derived data (derived objects) consistent with intrinsic data (source objects). To this end, the following requirements must be met:

1. Build steps have to be executed in a correct order.
2. In case of changes, all affected build steps must be re-executed.
3. (Re-)building must be performed in minimal time.
4. Building must rely on correct dependency information.
5. The storage space for holding derived objects must be minimized.

These requirements differ from those to be considered at the source level:

- In case of source version construction, we have a *selection problem* in both product and version space. The selection must be performed such that the outcome of the configuration process obeys all version rules and product constraints. Here, non-determinism 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. Non-determinism only deals with the order in which build steps are executed (the build plan only imposes a partial order on the build steps). Some authors advertise the advantages of non-deterministic build rules (e.g. "build some `sort` program no matter what algorithm and which compiler is used" [102]). However, in many situations it is crucial to control the details of a build without leaving the freedom for non-deterministic choices (e.g., even the functional behaviour of a program may depend on whether it is compiled with or without optimization).

5.3.2 Version Rules

A *build rule* describes a build step to be performed. This is done by specifying (versions of) input objects, output objects, and actions. An action typically invokes some tool on the input objects in order to produce output objects.

Figure 19 illustrates build rules at *instance* and *type level*, respectively. Part a) shows two rules of a make file [44][11]. The first line of each rule describes the target (before the colon) and the inputs of a build step, the second line contains a tool invocation. The first rule states that program `foo` is linked from the object codes of all constituents. In the

[11] Note that Make also supports type-level rules to a limited extent.

a) instance-level rules	*b) type-level rules*

`foo : main.o a.o b.o c.o`
`   cc main.o a.o b.o c.o -o foo # link`
`main.o : main.c main.h a.h b.h`
`   cc -c main.c # compile`
	`compile [m : MODULE]:`
	`   m.invalid # precondition`
	`   {cc -c m.c} # tool activation`
	`   not m.invalid; # postcondition success`
	`   m.invalid; # postconditon failure`

Figure 19: Build rules on instance and type level

second rule, the compiler is called on `main.c` which includes further header files. In Part b), a Marvel-like rule [61] is given which generalizes the compilation rule of the make file. The rule is applicable to any C module. The `invalid` flag ensures that no compilation is performed if the compiled code is still up-to-date[12]. There are two postconditions for successful and failing compilations, respectively.

5.3.3 Configurators

A *functional builder* [71, 55, 19] considers construction of derived objects as application of functions which should be free of sideeffects (otherwise, the result may depend on the order in which parallel subexpressions are evaluated). There are close connections to incremental attribute evaluation [57, 66, 132]. Evaluation may be performed either in a lazy or in a busy fashion. Stored derived objects are used to accelerate builds.

A *rule-based builder* is driven by build rules stored in some rule base [44, 61, 3, 15]. In analogy to lazy and busy evaluation of functional expressions, backward and forward chaining are used to trigger execution of build steps. Non-determinism is taken into account by considering alternative build plans.

Let us now return to the problems listed in Subsection 5.3.1. The first problem is solved by constructing a *build plan* which is composed of build steps and data dependencies. Such a build plan is usually derived from a system model [71, 76]. The system can either be stored in a text file or be represented by dependencies in the database (see also Figure 1 on page 7).

Borison [14] proposes a solution to the second and the third problem. A complete record of a build is represented by a *derivation graph* (Figure 20a). Each parameter which might influence the outputs of a build step is recorded as an input, including source and derived objects, tool options, tool versions, operating system, and host machine.

Rebuilds are handled as follows: In case of changed inputs, a revised derivation graph is constructed by copying the old derivation graph and replacing all outputs of build steps with variables. Subsequently, these variables are replaced with values, either by executing a build step or by skipping it and reusing the old outputs. For each build step, this decision is based on a *difference predicate* which compares old and new inputs (see Figure 20b, where the switch is moved up (down) if the old output can (cannot) be reused).

A variety of difference predicates has been used for system building. Make [44] relies on time stamps and can guarantee neither minimal nor necessary rebuilds. DSEE [76] and ClearCase [73] maintain a complete build record for each derived object. A rebuild may be avoided safely when old and new build records are identical. However, this does

[12] Further rules are required to maintain the `invalid` flag.

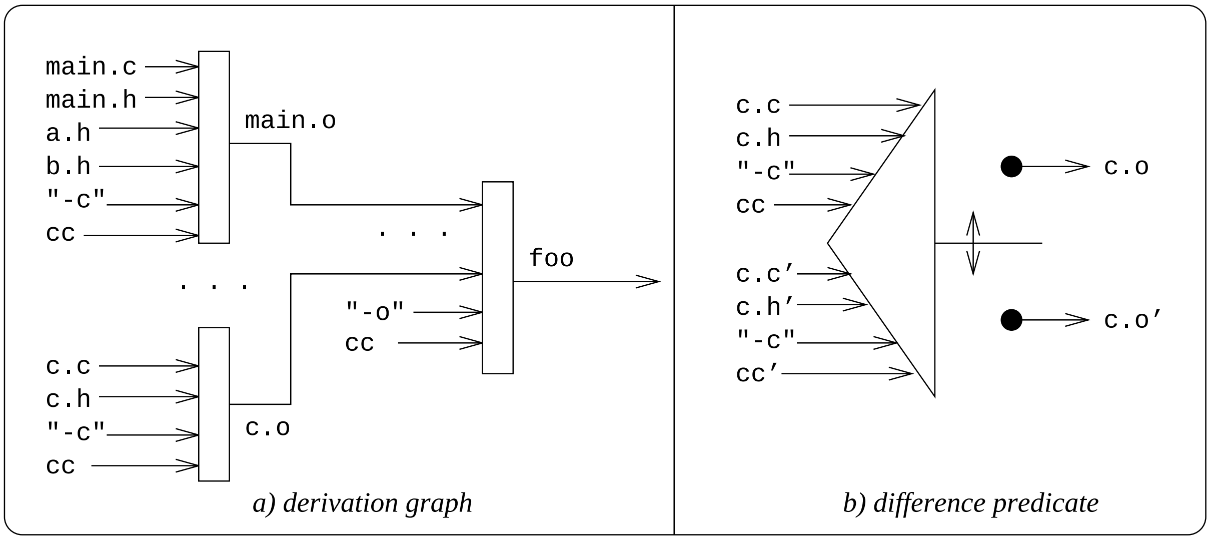

Figure 20: Borison's model for system building

not guarantee minimal rebuilds. Various selective recompilation strategies have been developed which inspect the nature of changes [4]. To select an appropriate strategy, the costs of evaluating difference predicates must be balanced against the benefits of saved recompilations.

The fourth problem concerns the correctness of the build plan[13]. This problem can be solved by automatic generation of system models from the sources [127]. A more elegant solution avoids generation of dependencies. For example, in ClearCase [73] and Odin [19, 18] no header file dependencies are declared in the system model. Rather, these dependencies are caught during system building and are used later on for triggering recompilations.

The last problem concerns storage of derived objects. Here, runtime effort must be balanced against storage consumption. A *derived object cache* [76, 55, 69] may store derived objects which are likely to be used in future build steps. Each derived object is tagged with a build record which serves as a primary key into the object cache. Policies such as LRU control which derived objects are held in the cache.

6 VERSION MODELS IN SCM SYSTEMS

After having characterized version models in general, we now take a look at concrete systems. We intend to draw a picture of the SCM landscape by describing the contributions of a representative selection of influential SCM systems. These descriptions are primarily based on the published scientific literature. Although the survey given below does include commercial systems in addition to research prototypes, we do not intend to provide a comprehensive overview of SCM systems available in the commercial marketplace (see [103]). Furthermore, our main goal consists in illustrating different version models rather than in evaluating the functionalities provided by SCM systems.

[13]Note that correct dependency information is also a prerequisite for successful construction of source configurations.

6.1 Overview

We have selected more than 25 SCM systems which vary widely with respect to their underlying version models. The *version graph* in Figure 21 illustrates the evolution of SCM since the early 70's. Each node corresponds to a specific system and briefly describes its original contribution(s). Incoming edges express the most important influences of previous systems.

The version graph is traversed in chronological order in the next subsection. Instead of reading this subsection sequentially from its beginning to its end, the reader may concentrate on some systems in which (s)he is interested. Furthermore, the reader may prefer a contents-based order (e.g., read the descriptions of all approaches which were developed from conditional compilation). The tables in Appendix A classify SCM systems according to a taxonomy derived from the previous sections and supplement the descriptions given below.

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

- Initially, SCM was supported through *isolated tools* covering specific aspects (e.g., Make for derived objects, SCCS for versioning of source objects). Later on, *integrated systems* were developed which combined the functionalities of individual tools into a coherent environment (e.g., DSEE or ClearCase).

- The pioneers addressed many problems in an *ad hoc manner*. Subsequent systems addressed topics such as intensional versioning of source and derived objects in a more *systematic* and *general way* (e.g., compare version selection by `checkout` options to full-fledged logic-based approaches such as ICE).

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

- While early systems/tools were implemented on top of the *file system*, more recent systems are using *database technology* to an increasing extent. This is demonstrated e.g. by the Adele system, which evolved from a file-based system with an ad hoc, hard-wired data model into an active, object-oriented database system supporting historical versioning (revisions), logical versioning (variants), and cooperative versioning (workspaces).

6.2 System Descriptions

Conditional compilation [65] supports intensional versioning of *C programs* at the fine-grained level. Although it has originally been developed for a specific domain, the underlying idea is fairly general. The source text is interspersed with *preprocessor directives* which refer to the values of preprocessor variables (see Figure 12 on page 21). 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 non-determinism/backtracking). Note that conditional compilation

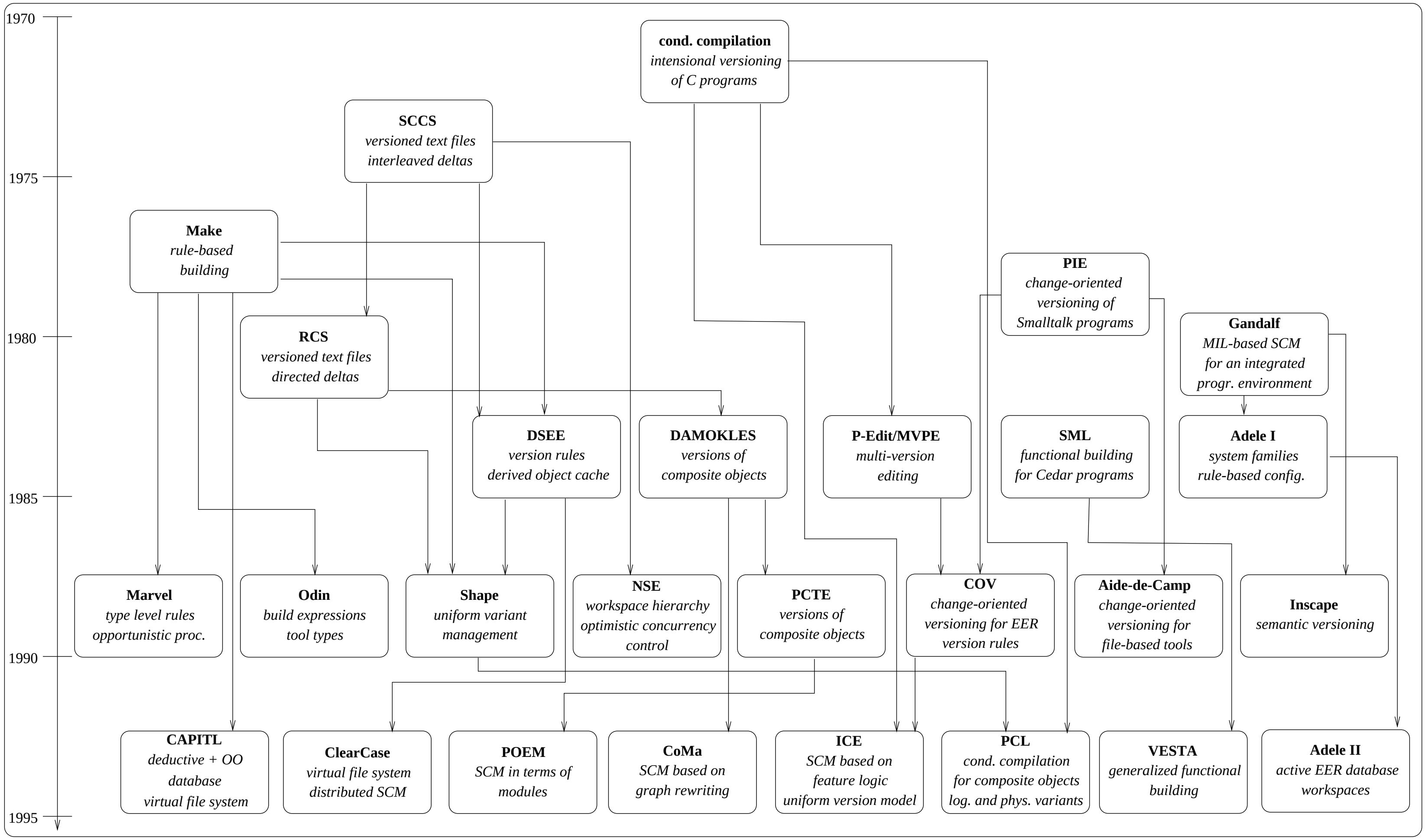

Figure 21: Version graph of SCM systems

is a low-level and general mechanism on top of which different version models can be implemented (both state- and change-oriented ones).

SCCS [104] manages *versions* of *text files*[14]. Versions are arranged in a tree. To work on a version, it is physically copied into a *workspace* (directory). In case of write accesses, the modified version is later 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 extensional versioning, it does provide some commands for controlling delta applications (and even fixing deltas) in a change-oriented fashion.

Make [44] supports rule-based construction of derived objects. A *make file* consists of a collection of build rules, composed of a set of targets, a set of sources, and a sequence of actions (see Figure 19 on page 30). Both sources and targets are files. Build rules primarily refer to specific instances, but Make also provides limited support for defining type-level rules. The necessity of executing a build step is decided with the help of time stamps. Builds are performed on demand by calling Make on a make file and a target. The order in which build steps are executed is determined by topological sorting. Since manual creation of make files is cumbersome and error-prone, many generators have been developed which create make files by analyzing dependencies between source objects [127].

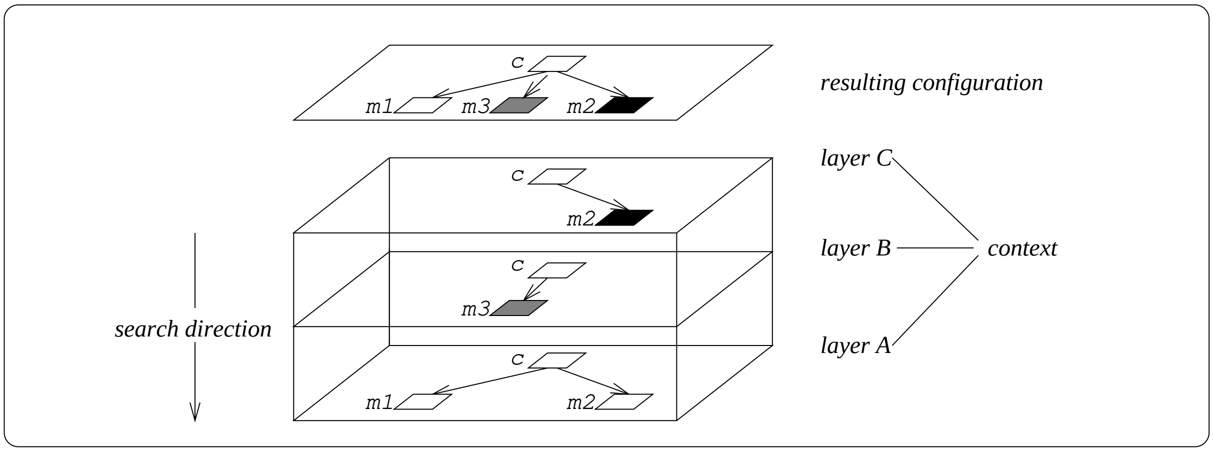

Figure 22: Layers and contexts in PIE

An early approach to change-oriented versioning has been developed at XEROX PARC [50]. **PIE** manages configurations of *Smalltalk programs* which 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*. Layers are aggregated into contexts which act as search paths through the layers[15]. This is illustrated in Figure 22, 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: First, each layer may either be included or omitted; second, the included layers can be arranged in any se-

[14]Binaries can also be stored, but delta storage is not supported.
[15]DaSC [82] is based on a similar approach.

quential 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.

RCS [122, 123] 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), while all other revisions are reconstructed by applying backward and forward deltas. Furthermore, RCS provides an *adjustable locking scheme* for coordinating concurrent work. Locks are optional and can be broken in order to perform urgent fixes. Finally, RCS provides a set of built-in *attributes*, including e.g. 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 on page 19). 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.

The **Gandalf** project [53] was dedicated to the generation of structure-oriented software development environments which are based on *abstract syntax trees*. Gandalf C is an environment instance which supports C at the programming-in-the-small level. Programming-in-the-large and version control are handled by the SVCE subsystem [63] which is based on the Intercol *module interconnection language* [120]. Each module has a unique interface and potentially multiple realization variants each of which evolves into a sequence of revisions. A realization of an interface can either be provided by writing C code, or it can be composed of other modules which jointly provide the resources listed in the export interface (compositions). Gandalf enforces *syntactic consistency* of all interfaces and realizations which are 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 which performs intertwined AND/OR selections. The configurator is driven by simple *version rules* which are classified into constraints, preferences, and defaults. Constraints and preferences are attached to imports or compositions and select specific variants/revisions. If no other rules are applicable, the standard variant/revision is selected by default.

In contrast to Gandalf, **Adele I** [35, 10] was developed in the mid 80's for *tool-kit environments* (a posteriori integration of file-based tools). Adele generalizes Gandalf's approach to configuring modular programs in several ways. While each Gandalf module has a unique interface, an Adele *family* may have multiple versions of its interface (Figure 23[16]). 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 version rules are more sophisticated and allow for attribute-based version selection. Using these rules, short intensional descriptions can be given for large and complex configurations. For example, the following query selects a `tested` configuration running under `Unix` and supporting both the `SunViews` window system and the `Informix` database system:

[16]For the sake of readability, hybrid AND/OR nodes are used to represent interfaces and variants by single nodes.

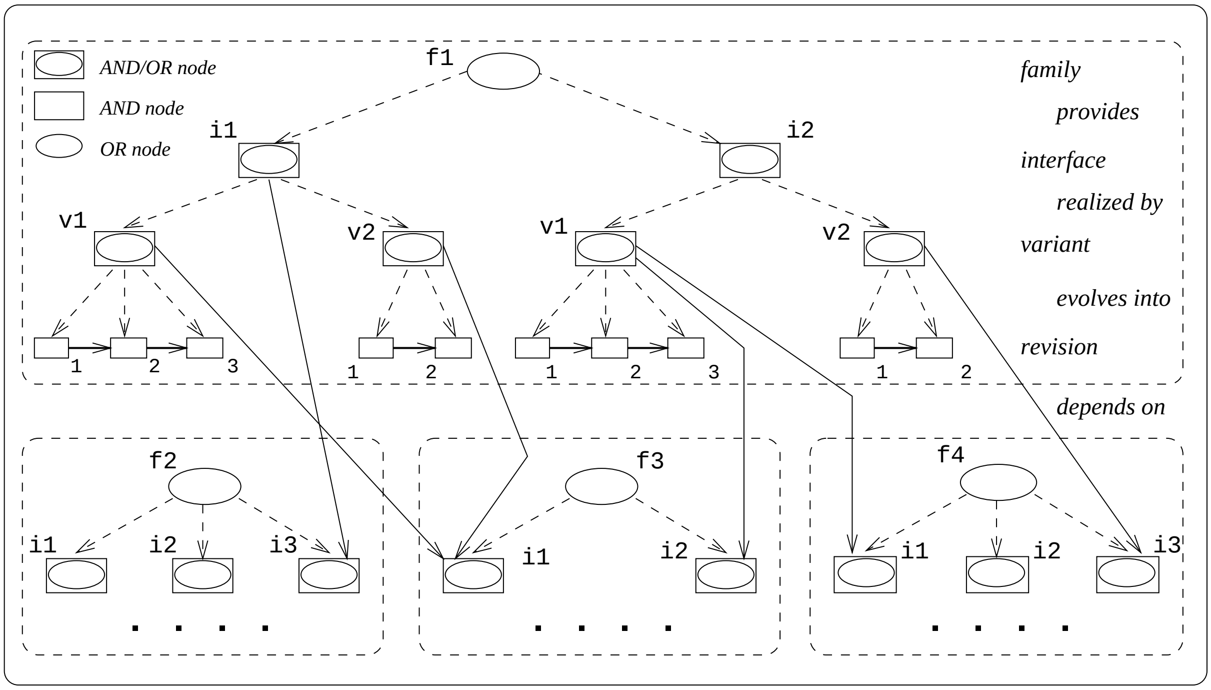

Figure 23: Families in Adele

```
([os = Unix] and [ws = SunViews]
 and [db = Informix] and [state = tested])
```

Given a configuration description for an interface, the configurator traverses the AND/OR graph of families, selecting one revision for each family (*intertwined selection*). The configurator maintains a set of version rules R initialized with the configuration description. R is extended gradually by adding the stored rules attached to selected versions. To avoid wrong choices, version selection for some family f is performed only after all constraints are known. This is the case when versions of all families above f have been selected. However, once a selection has been made (possibly with the help of preferences and defaults), it cannot be retracted (no backtracking).

DSEE [76, 74, 75] integrates functions which 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 on page 11. A configuration is specified by a system model and a thread through the version graphs of components (Figure 24). 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 version rules in the *configuration thread*. In contrast to Adele, DSEE does not distinguish between constraints, preferences, and defaults. Rather, rules are ordered sequentially according to their priorities. The DSEE configurator selects the product structure first, which is described in the system model, and adds version bindings, resulting in a *bound configuration thread* (BCT)[17]. The BCT is used for system building. The

[17]However, it should be noted that the system model itself is version controlled. Thus, a version of the

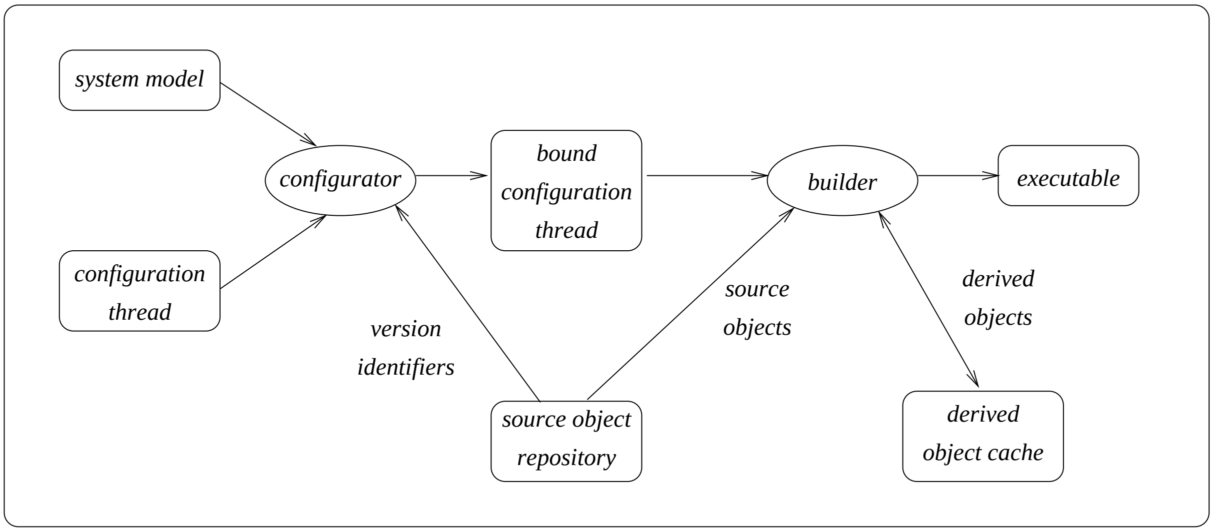

Figure 24: DSEE

builder considers all information which might influence the result of a build step, including e.g. tool options and tool versions. A build step is executed only when the derived object to be constructed is not contained in the cache of derived objects.

The Cedar environment [117] supports programming in Cedar, a modular language which has been developed at XEROX PARC. The Cedar System Modeler [70] takes care of compiling and linking Cedar programs and is based on the *functional language* **SML** [71]. In SML, tools such as compilers and linkers are handled as functions whose applications deliver compiled and linked code, respectively. A system model is an expression from which the modeler derives the compilation and link steps for building a program. Like in DSEE, derived objects are stored in a cache which is used to accelerate building. Unlike DSEE system models which represent generic (i.e., unbound) configurations, Cedar system models describe specific configuration versions in terms of immutable source files [48]. In Cedar, each editing session creates a new source version. Immutability of source files and accurate recording of tool options ensure that a configuration can safely be rebuilt from a system model. Thus, a system model in Cedar plays a similar role as a bound configuration thread in DSEE (extensional versioning, "version first" selection).

P-Edit [68] and its successor **MVPE** [107] support *simultaneous editing* of *multiple versions* of a text file. A text file is composed of fragments (sequences of words). Like 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 25, 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 Figure 25 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).

system model is selected in an initial step not shown in Figure 24.

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 25: MVPE

DAMOKLES [32, 51] is a database system dedicated to (software) engineering applications. 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 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 on page 10). 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). Finally, DAMOKLES supports an acyclic hierarchy of databases, long transactions, and checkout/checkin operations with locking.

Marvel [61] is usually classified as a process management system. However, from the perspective of SCM we may consider it a generalization of Make. Marvel is based on a structurally object-oriented data model. Attributes are either simple values or represent (bidirectional) relationships to other objects. A process model consists of a collection of rules each which is composed of a *precondition*, an *activity*, and a *postcondition* (see also Figure 19 on page 30). In contrast to Make, all rules are type-level ones. Marvel generalizes the backward chaining approach of Make to *opportunistic processing*, a mixture of backward and forward chaining. Backward chaining is triggered when a user issues a command whose precondition is not fulfilled. Forward chaining utilizes otherwise idle computing resources, resulting in smaller response times to user commands (shorter backward chains). The application of forward and backward chaining is controlled through user-defined strategies.

While Marvel has been designed with the general goal of automating software processes, **Odin** [19, 18] is rather a classical SCM tool which focuses on *management* of *derived objects*. Odin introduces several improvements over Make. Like DSEE, Odin records all parameters influencing the result of a build step. Furthermore, rederivations are cut off in case of unchanged tool outputs. Odin differs from Make, DSEE, and SML (and agrees with Marvel) in that the user never specifies instance-level build rules. Rather, derivation graphs are constructed from type-level rules describing inputs, outputs, and invocation of tools. The user interacts with Odin using build expressions. A *build expression* is essentially composed of a source object and a list of derived object types. From such an expression, Odin deduces the tools to be applied. Like in DSEE, building is accelerated by a cache of derived objects.

Shape [69, 84] is an SCM system which combines ideas drawn from Make, DSEE, and RCS. An attributed file system stores versioned files using directed deltas [91]. 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 *uniform variant management* [83]. 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 e.g. 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 which were previously handled separately from each other.

PCTE [126] is a standard for open repositories which provides an interface for implementing software engineering tools. PCTE is based on a data model which combines concepts from the *EER model* and the Unix file system. Each object may have at most one long attribute for which Unix-like file operations are provided. Links between objects are classified into predefined categories, including composition links for representing composite objects. PCTE offers basic versioning facilities [93]. *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).

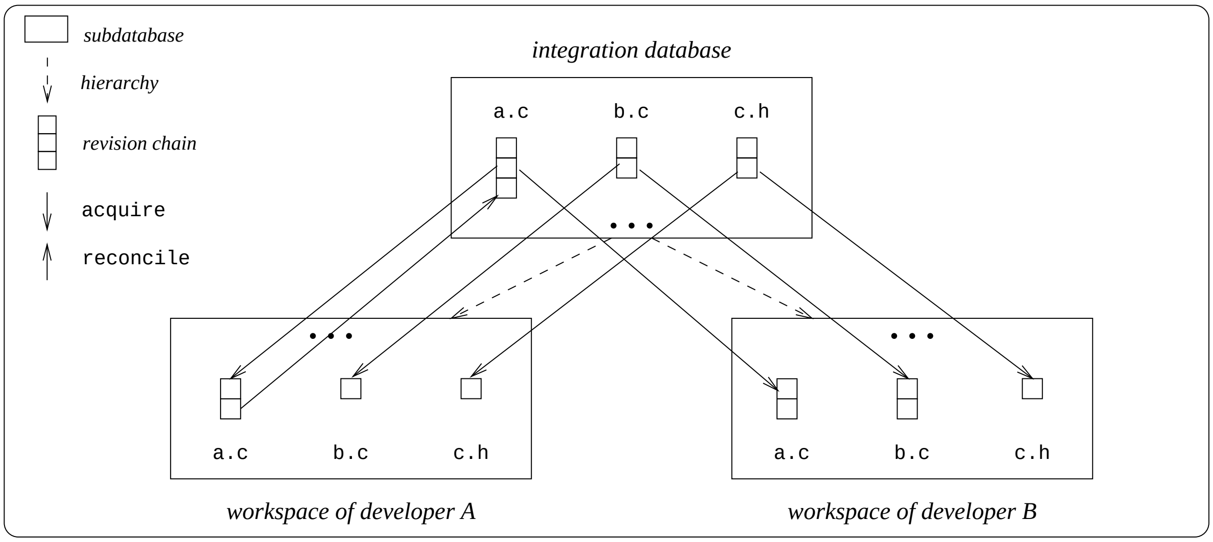

Figure 26: NSE

Like PCTE, **NSE** [2] supports versioning of composite objects. In contrast to PCTE, evolution histories are linear. To support cooperation, NSE provides *subdatabases* which unify repositories and workspaces. Subdatabases are organized into strict *hierarchies*, i.e., each subdatabase may have at most one parent (Figure 26). Subdatabases may be nested to arbitrary depth. Data may only be exchanged along the hierarchy. In contrast to RCS, *optimistic concurrency control* is used to coordinate the activities of multiple developers. `acquire` copies a version of a composite object from a parent to a child database without setting up any locks. After having finished the changes in the child, `reconcile` copies the changed objects back to the parent. A *merge conflict* occurs when both a new version is present in the parent and a local modication has been performed in the child. For example, in Figure 26 developer A has already commited his (her) changes, and `reconcile` will fail for developer B due to a merge conflict on `a.c`. Then, B resolves the conflict with the help of a (textual) merge tool and invokes `reconcile` again.

Aide-de-Camp [25, 112] describes versions of products in terms of change sets relative to a base version. A *change set* describes logically related, global changes which 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 case of overlaps, c_2 overrides the changes in c_1. Each change set may be viewed as a switch which can either be turned on or off (see Figure 6 on page 13). Aide-de-Camp detects inconsistencies when reconstructing a product version from a base and a sequence of change sets (e.g., modifications to non-existing text lines). Furthermore, Aide-de-Camp provides a 3-way merge tool for conflict detection. Unlike PIE, Aide-de-Camp does not support relationships which can be used to detect inconsistent combinations of change sets.

COV [77] denotes the version model underlying ECM [88], the SCM subsystem of the EPOS software engineering environment [21]. EPOS is based on an *EER model*, where files are represented by entities with long attributes. COV supports *change-oriented versioning*, where each change is considered global and is described at the granularity of text lines and attribute values. The version space is structured by global options. Each *option* defines a dimension of the version space. Options may denote variants, e.g., options `Unix`, `DOS`, and `VMS` may define different variants of the underlying operating system. Options may also be used to represent changes, e.g., options `Fix1` and `Fix2` may represent certain bug fixes. Like MVPE, COV distinguishes between read and write filter which are called *choice* and *ambition*, respectively. A choice is specified by a complete list of option bindings. An ambition contains a subset of the option bindings of the choice and correspond to a region of the n-dimensional option space.

Unlike other change-oriented SCM systems such as PIE or Aide-de-Camp, ECM pays much attention to the definition of *version rules* for consistent change combinations [52]. Some examples have already been given in Figure 14 on page 24. An aggregate is used to ensure that certain changes are applied together (see Rule (6)). The operator $\otimes$ indicates mutually exclusive changes/variants (see Rule (8)). With the help of implication, dependencies between changes may be expressed (see Rule (7)). In addition to the standard logic implication, there are further dedicated operators for describing other kinds of dependencies [86]. An option dependency means that an option can only be given a value when another option is set. For example, `Fix3` $\hookrightarrow$ `Unix` says that `Fix3` is specific to the `Unix` variant. Furthermore, a product dependent option is only valid for a certain subproduct. In [86], a version selection tool is proposed which supports the user in setting up consistent ambitions and choices by automatically deriving option bindings with the help of version rules.

Cooperation is supported by *nested, long transactions* [23, 22]. The transaction hierarchy forms a tree. Transactions work on subproducts and affect the versions specified in their ambition. To each transaction, a workspace is attached which contains locally modified objects (copy-on-write). Cooperation support is more flexible than e.g. in NSE. *Horizontal data exchange* between sibling transactions supplements the hierarchical data flow. Furthermore, *cooperation policies* may be negotiated between pairs of transactions. To this end, a set of primitives are provided which determine when changes are propagated out of or into a workspace, in which way update conflicts are resolved or prevented (e.g., locking or optimistic concurrency control), when notifications are sent to whom, etc. Planning of transaction schedules based on estimated read and write sets is also supported.

Inscape [94] goes beyond Gandalf and Adele by 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 [95] 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) which 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.

CAPITL [3] integrates persistence, object-oriented and logic programming. POL (*persistent objects* with *logic*) allows C++ programs to access persistent objects as well as to invoke the logic-based *planner* and the *builder*. Conversely, C++ functions may be called during planning and building. The planner makes use of tool descriptions (type-level build rules). Planning is separated from building. Multiple plans for a build can be constructed due to the backtracking capabilities of the planner. With respect to tool integration, CAPITL provides a *virtual file system* which relies on a mount capability offered by NFS. This allows for reusing existing tools without modifications (neither to the tools nor to the file system). With respect to versioning, CAPITL follows a product-oriented approach. Product versions are arranged in an RCS-like version graph.

ClearCase [73] differs from its predecessor DSEE in several respects. While DSEE only versions files, 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) which is defined by a configuration description (Figure 27). The view is a filter which provides tools with the illusion of working in a single-version environment (virtual file system). The view excludes components which are not used in the specified configuration (`c.h` in Figure 27). In contrast to DSEE, generic references to elements are only bound dynamically when an element is accessed (no pre-computed version map). Furthermore, in ClearCase the system model is accessed in the same way as any other element (while in DSEE it is used for constructing a BCT, see also Figure 24 on page 37).

Version rules are similar to those offered by DSEE. They 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. For example, the following sequence of rules selects private versions with the highest priority, the most recent ver-

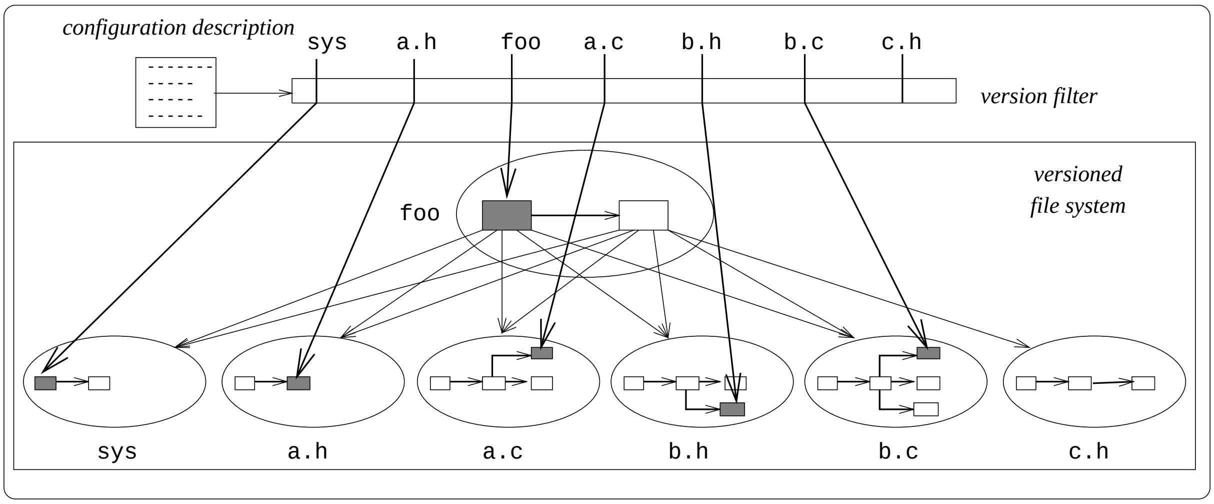

Figure 27: ClearCase

sions on the `Unix` branch (for elements which vary according to the supported operating system), and finally the versions belonging to release `R1`:

```
element /foo/... CHECKEDOUT
element /foo/... /Unix/LATEST
element /foo/... R1
```

To support *distributed SCM*, ClearCase assigns ownerships to branches in version graphs [5]. Each site appends revisions to its allocated branch. After having propagated new revisions to another site, 3-way merging is used to combine local and remote changes.

POEM [78, 79] is an environment for programming in C/C++ which strives for simplifying SCM at the user interface. To this end, SCM is provided in terms of *modules*, i.e., 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 denoted as a *software unit*. Software units are connected by relationships which represent *dependencies* between source objects. For each software unit, a set of operations is provided for editing, building, and version control. A version of a software unit uniquely determines all versions of units on which it depends ("version first" selection, see Figure 8b) on page 17). Versions of one software unit are arranged in a version tree. Building is performed recursively by calling build operations on subunits. Users of POEM are not at all concerned with build rules. However, they specifiy tool parameters which are stored as attributes in the software units.

CoMa [131, 132] manages configurations of *engineering design documents* and has been applied both to 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 has been defined by a programmed graph rewriting system using the PROGRES specification language [108]. The formal specification provides a precise description which aids in both understanding the model and implementing a corresponding SCM system. At the implementation level, the database system GRAS [66] is used which offers primitives for version control (*graph deltas*) and maintenance of derived information (*derived attributes*, which can be employed e.g. for system building). Finally, for software engineering applications a *structure-oriented merge tool* [130] provides for 3-way merging of versions of software documents (e.g., requirements definitions, software architectures, module implementations) which are internally represented as abstract syntax graphs.

ICE [136], the SCM subsystem of the software engineering environment NORA, applies *deductive database technology* to SCM. There are some close relations to COV. Like in COV, a versioned database consists of components which are tagged with control expressions. These control expressions (and potentially additional version rules) are evaluated when a product version is configured from an intensional description. While COV is based on a boolean logic, ICE applies *feature logic* for expressing and combining version rules. In contrast to Prolog terms, arguments of feature terms are identified by names rather than by position. A wide variety of operators is provided, some of which have specifically been motivated by applying feature logic to SCM [135].

Versions of some object are represented by *feature terms* containing a special `object` feature whose value is an object name. For example, the versions of `a` and `b` shown in Figure 17 on page 27 may be described by the following feature terms[18]:

```
a.1 = [object : a, ws : X11, os : Unix]
a.2 = [object : a, ws : Windows, os : DOS]
b.1 = [object : b, os : DOS, db : dbase]
```

The feature term of a configuration is constructed by an intersection operator (*unification*) which handles the `object` feature in a special way (union of the values instead of intersection). An intersection fails if corresponding features cannot be unified (mutually inconsistent constraints). For example, `b.1` may be combined with `a.2`, but not with `a.1`:

```
a.2 ⊓ b.1 =
  [object : {a, b}, ws : Windows, os : DOS, db : dbase]
a.1 ⊓ b.1 = ⊥
```

In contrast to COV which is committed to a specific versioning paradigm (change-oriented versioning), feature logic is employed as a base mechanism on top of which different version models may be realized[19]. In [134, 137], 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 ([41], see also Subsection 7.1). The common base of these models are interleaved deltas which are represented with the help of feature terms.

PCL [125] is the configuration language developed in the PROTEUS project. PCL is influenced by *conditional compilation* and *module interconnection languages* (in particular SySL [115]). 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 Figure 12b) on page 21). 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.

[18] A missing feature may take any value.

[19] That could have been done in COV, as well. However, COV always comes with multi-version editing and hidden control expressions.

All kinds of variation are controlled with a single mechanism, namely *attributes*[20]. 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). Attribute values are propagated through the composition hierarchy. When all physical components have been determined, a make file is generated as a final step. Tool descriptions are used to derive the build steps, and the options passed to tools are fixed by attributes, again.

Vesta [55] picks up many ideas underlying the Cedar System Modeler. The Vesta language is a *functional language* for describing *system builds*. Like in the Cedar System Modeler, tools are handled as functions without side effects, files are immutable objects, and derived objects are stored in a cache. While the System Modeler has only been used in the Cedar programming environment, Vesta is designed for integration with tools for any programming language. Further improvements include a more readable syntax of the system modeling language and more sophisticated caching of derived objects.

Adele II, the current version of Adele [39, 40, 38], differs considerably from the initial version which was described earlier in this subsection. 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 [40]. *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. 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 co-existing at a given time). *Cooperative versioning* is realized with the help of typed workspaces [38]. 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).

7 RELATED WORK

We have given a comprehensive description of the current state-of-the-art of version models for SCM. We have mainly focused on the organization of the version space and the flexible construction of consistent configurations from intensional specifications. Furthermore, we described and classified a significant number of representative SCM systems. In Subsection 7.1, we discuss related surveys and attempts to define a unified version model. 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 [133], which has launched a (still ongoing) series of follow-ups [118, 42, 43, 37, 114]. Tichy's paper [124] introduces basic notions such as software object, source and derived object, etc., and discusses version

[20]Note the similarity to Shape.

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 [36] 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 recognized papers which review and survey the state-of-the-art in SCM [16, 27, 29]. Perhaps, the most influential of these was written in 1991 by Feiler [41], who classified the models underlying SCM systems into four categories. These categories correspond 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 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 no 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 [64] surveyed version models for *engineering databases*. Katz primarily considers electrical and mechanical engineering (CAD) and only mentions a few software engineering approaches. Although both domains have evolved nearly independently for a long time, many parallels do exist [28]. 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 vs. definitions (instances of component versions may have properties which depend on the respective contexts in configurations), change notification and propagation (when and where to propagate changes), and object sharing mechanisms (workspaces).

In several points, our view of version models goes beyond the work presented by Katz. His paper focuses on approaches which are based on version graphs and support only extensional versioning at the component level. Intensional versioning is only provided at the configuration level, resulting in the composition model introduced by Feiler. As we have shown, there are radically different approaches such as e.g. conditional compilation which are not based on version graphs at all. Furthermore, Katz primarily discusses state-oriented versioning. In contrast, this paper covers change-oriented versioning, as well, and it investigates the relations between these complementary approaches. Finally, version rules and the consistency problems of intensional versioning are only touched upon 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 in clarifying the relations between different disciplines such that an overall architecture can be designed with well-defined interfaces between different components (see also Section 8).

Temporal databases [116, 111] 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-oriented 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 accomodate changes to the database schema. In 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* [105] makes it possible to view the data under different versions of the schema. In SCM systems, versioning of the schema (and other meta data such as version rules) is rarely considered seriously. On the other hand, schema versioning often does not take the versioning of instance data into account.

Deductive databases [30, 99, 98] provide for persistent storage of facts and rules 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 [134, 11, 72]. Rather, many SCM systems incorporate home-grown deductive components which 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* [9, 62, 41]. Rather, pre-commit 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 [9]. This is a severe restriction since versions play a crucial role in cooperation control [40]. So far, only a few SCM systems support long transactions [23, 49]. Many others merely provide workspaces and mechanisms for controlling change propagation between them [38].

Software process management [45, 26, 106] is concerned with the definition, analysis, and enactment of high-level models of software processes. Many different formalisms have been developed to specify process models, including Petri nets [31, 8], task nets [60, 56], rules [61, 96], blackboards [6], etc. In order to integrate SCM and process management, functional overlap has to be considered (e.g., between build tools and rule-based process engines such as Marvel [61]). Furthermore, the definition of "product space" has to 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 [81]).

Tool integration [129] is provided by SCM systems through workspaces, which hide versioning from the tools. Workspaces can be separated physically from the versioned database [104], or they are realized as updatable database views (virtual workspaces, e.g., virtual file systems [73, 46]). Current SCM systems focus on integration of file-based tools and offer poor support for integrating tools operating on databases (e.g., CASE tools [128]). The problem of integrating heterogeneous database systems without sacrificing their autonomy is addressed by *federated database systems* [110] and *data warehouses* [54]. Furthermore, different kinds of platforms or frameworks offer plug-in interfaces for tool integration, e.g., *broadcast message servers* [101] and *object request brokers* [92, 113].

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 between 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. This problem can be addressed by applying *hypertext systems* [20] to the software engineering domain.

The emerging discipline of *software architectures* [109] 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 labour, understanding the interfaces between different components, etc. [89]. SCM may benefit from software architectures in two ways. First, SCM systems are primarily source-oriented and represent the product structure by rather low-level system models which 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 between 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 and compare their merits and shortcomings. Initial attempts in developing a *uniform model* have been undertaken [134, 137]. Based on the material presented in this paper, we believe that a version model can be developed which 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. 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. Similar trends can already be observed with respect to workspace management and long transactions.

REFERENCES

[1] E. Adams, W. Gramlich, S. Muchnick, and S. Tirfing. SunPro: Engineering a practical program development environment. In *Proceedings of the International Workshop on Advanced Programming Environments*, LNCS 244, pages 86–96, Trondheim, June 1986. Springer-Verlag.

[2] E. W. Adams, M. Honda, and T. C. Miller. Object management in a CASE environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 154–163, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[3] P. Adams and M. Solomon. An overview of the CAPITL software development environment. In Estublier [37], pages 1–34.

[4] R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, Jan. 1994.

[5] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In Estublier [37], pages 194–214.

[6] V. Ambriola, P. Ciancarini, and C. Montangero. Software process enactment in Oikos. *Software Engineering Notes*, 15(6):183–192, Dec. 1990.

[7] W. A. Babich. *Software Configuration Management.* Addison-Wesley, Reading, Massachusetts, 1986.

[8] S. C. Bandinelli, A. Fuggetta, and C. Ghezzi. Software process model evolution in the SPADE environment. *Transactions on Software Engineering*, 19(12):1128–1144, Dec. 1993.

[9] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[10] N. Belkhatir and J. Estublier. Experience with a data base of programs. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices 22(1), pages 84–91, Palo Alto, California, Dec. 1986.

[11] Y. Bernard, M. Lacroix, P. Lavency, and M. Vanhoedenaghe. Configuration management in an open environment. In G. Goos and J. Hartmanis, editors, *Proceedings of the 1st European Software Engineering Conference*, LNCS 289, pages 35–43, Straßburg, Sept. 1987. Springer-Verlag.

[12] E. H. Bersoff, V. D. Henderson, and S. G. Siegel. *Software Configuration Management: An Investment in Product Integrity.* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[13] V. Berzins, editor. *Software Merging and Slicing.* IEEE Computer Society Press, Los Alamitos, California, 1995.

[14] E. A. Borison. Program changes and the cost of selective recompilation. Technical Report CMU-CS-89-205, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1989.

[15] P. Brereton and P. Singleton. Deductive software building. In Estublier [37], pages 81–87.

[16] A. Brown, S. Dart, P. Feiler, and K. Wallnau. The state of automated configuration management. Technical Report ATR92, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, Sept. 1991.

[17] J. Buffenbarger. Syntactic software merging. In Estublier [37], pages 153–172.

[18] G. Clemm. The Odin system. In Estublier [37], pages 241–262.

[19] G. Clemm and L. Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, Jan. 1990.

[20] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, Sept. 1987.

[21] R. Conradi et al. EPOS: Object-oriented and cooperative process modelling. In Finkelstein et al. [45], pages 33–70.

[22] R. Conradi, C. Liu, and M. Hagaseth. Planning support for cooperating transactions in EPOS. *Information Systems*, 20(4):317–335, 1995.

[23] R. Conradi and C. Malm. Cooperating transactions and workspaces in EPOS: Design and preliminary implementation. In *Proceedings of the Third International Conference on Advanced Information Systems Engineering (CAISE '91)*, LNCS 498, pages 375–392, Trondheim, May 1991. Springer-Verlag.

[24] R. Conradi and B. Westfechtel. Configuring versioned software products. In Sommerville [114].

[25] R. D. Cronk. Tributaries and deltas. *BYTE*, 17(1):177–186, Jan. 1992.

[26] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Communications of the ACM*, 35(9):75–90, Sept. 1992.

[27] S. Dart. Concepts in configuration management systems. In Feiler [42], pages 1–18.

[28] S. A. Dart. Parallels in computer-aided design frameworks and software development environments efforts. *IFIP Transactions A*, 16:175–189, 1992.

[29] S. A. Dart. The past, present, and future of configuration management. Technical Report CMU/SEI-92-TR-8, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, July 1992.

[30] S. K. Das. *Deductive Databases and Logic Programming.* Addison-Wesley, Reading, Massachusetts, 1992.

[31] W. Deiters and V. Gruhn. Managing software processes in the environment MELMAC. *Software Engineering Notes*, 15(6):193–205, Dec. 1990.

[32] K. Dittrich, W. Gotthard, and P. Lockemann. DAMOKLES, a database system for software engineering environments. In *Proceedings on the International Workshop on Advanced Programming Environments*, LNCS 244, pages 353–371, Trondheim, June 1986. Springer-Verlag.

[33] K. R. Dittrich and R. A. Lorie. Version support for engineering database systems. *IEEE Transactions on Software Engineering*, 14(4):429–437, Apr. 1988.

[34] H. Ehrig, W. Fey, H. Hansen, M. Löwe, and D. Jacobs. Algebraic software development concepts for module and configuration families. In *Proceedings of the 9th Conference on Foundation of Software Technology and Theoretical Computer Science*, LNCS 405, pages 181–192, Bangalore, India, Dec. 1989. Springer-Verlag.

[35] J. Estublier. A configuration manager: The Adele data base of programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, Harwichport, Massachusetts, June 1985.

[36] J. Estublier. Configuration management: The notion and the tools. In Winkler [133], pages 38–61.

[37] J. Estublier, editor. *Software Configuration Management: Selected Papers SCM-4 and SCM-5*, LNCS 1005. Springer-Verlag, 1995.

[38] J. Estublier. Workspace management in software engineering environments. In Sommerville [114].

[39] J. Estublier and R. Casallas. The Adele configuration manager. In Tichy [119], pages 99–134.

[40] J. Estublier and R. Casallas. Three dimensional versioning. In Estublier [37], pages 118–135.

[41] P. H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Mar. 1991.

[42] P. H. Feiler, editor. *Proceedings of the 3rd International Workshop on Software Configuration Management*, Trondheim, Norway, June 1991. ACM Press.

[43] S. Feldman, editor. *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint)*, Baltimore, Maryland, May 1993.

[44] S. I. Feldman. Make — A program for maintaining computer programs. *Software–Practice and Experience*, 9:255–265, Apr. 1979.

[45] A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Advanced Software Development Series. Research Studies Press (John Wiley), Chichester, UK, 1994.

[46] G. Fowler, D. Korn, and H. Rao. n-dfs: The multiple dimensional file system. In Tichy [119], pages 135–154.

[47] C. Fraser and E. Myers. An editor for revision control. *ACM Transactions on Programming Languages and Systems*, 9(2):277–295, Apr. 1986.

[48] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, Mar. 1988.

[49] C. Godart, G. Canals, F. Charoy, and P. Molli. About some relationships between configuration management, software process, and cooperative work: The COO environment. In Estublier [37], pages 173–178.

[50] I. P. Goldstein and D. G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, XEROX PARC, Palo Alto, California, 1980.

[51] W. Gotthard. *Datenbanksysteme für Software-Produktionsumgebungen.* IFB 193. Springer Verlag, Berlin, 1988.

[52] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386, Nov. 1991.

[53] N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, Dec. 1986.

[54] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford data warehousing project. *Data Engineering Bulletin*, 18(2):41–48, 1995.

[55] C. B. Hanna and R. Levin. The Vesta language for configuration management. Technical Report 107, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, June 1993.

[56] P. Heimann, G. Joeris, C.-A. Krapp, and B. Westfechtel. DYNAMITE: Dynamic task nets for software process management. In *Proceedings of 18th International Conference on Software Engineering*, pages 331–341, Berlin, Mar. 1996. IEEE Computer Society Press.

[57] S. E. Hudson and R. King. The Cactis project: Database support for software environments. *IEEE Transactions on Software Engineering*, 14(6):709–719, June 1988.

[58] The Institute of Electrical and Electronics Engineers, Inc., New York, New York. *IEEE Standard for Software Configuration Management Plans: ANSI/IEEE Std 828-1983*, June 1983.

[59] The Institute of Electrical and Electronics Engineers, Inc., New York, New York. *IEEE Guide to Software Configuration Management: ANSI/IEEE Std 1042-1987*, Mar. 1988.

[60] M. L. Jaccheri and R. Conradi. Techniques for process model evolution in EPOS. *Transactions on Software Engineering*, 19(12):1145–1156, Dec. 1993.

[61] G. Kaiser, P. Feiler, and S. Popovich. Intelligent assistance for software development and maintenance. *IEEE Software*, 5(3):40–49, May 1988.

[62] G. E. Kaiser. Cooperative transactions for multiuser environments. In Kim [67], pages 409–433.

[63] G. E. Kaiser and A. N. Habermann. An environment for system version control. In *Digest of Papers of Spring CompCon '83*, pages 415–420. IEEE Computer Society Press, Los Alamitos, Feb. 1983.

[64] R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, Dec. 1990.

[65] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[66] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS, a graph-oriented software engineering database system. *Information Systems*, 20(1):21–51, Jan. 1995.

[67] W. Kim, editor. *Modern Database Systems.* Addison Wesley, Reading, Massachusetts, 1995.

[68] V. Kruskal. Managing multi-version programs with an editor. *IBM Journal of Research and Development*, 28(1):74–81, 1984.

[69] A. Lampen and A. Mahler. shape – A software configuration management tool. In Winkler [133], pages 228–243.

[70] B. W. Lampson and E. E. Schmidt. Organizing software in a distributed environment. *ACM SIGPLAN Notices*, 18(6):1–13, June 1983.

[71] B. W. Lampson and E. E. Schmidt. Practical use of a practical polymorphic applicative language. In *Tenth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices 18(6), pages 237–255, Austin, Texas, Jan. 1983.

[72] P. Lavency and M. Vanhoedenaghe. Knowledge based configuration management. In B. Shriver, editor, *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 83–92. IEEE Computer Society Press, 1988.

[73] D. Leblang. The CM challenge: Configuration management that works. In Tichy [119], pages 1–38.

[74] D. B. Leblang and R. P. Chase, Jr. 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 Notices 19(5), pages 104–112, May 1984.

[75] D. B. Leblang, R. P. Chase Jr., and H. Spilke. Increasing productivity with a parallel configuration manager. In Winkler [133], pages 21–37.

[76] D. B. Leblang and G. D. McLean, Jr. Configuration management for large-scale software development efforts. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 122–127, Harwichport, Massachusetts, June 1985.

[77] A. Lie, R. Conradi, T. Didriksen, E. Karlsson, S. O. Hallsteinsen, and P. Holager. Change oriented versioning. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, LNCS 387, pages 191–202, Coventry, UK, Sept. 1989. Springer-Verlag.

[78] Y.-J. Lin and S. P. Reiss. Configuration management in terms of modules. In Estublier [37], pages 101–117.

[79] Y.-J. Lin and S. P. Reiss. Configuration management with logical structures. In *Proceedings of the 18th International Conference on Software Engineering*, pages 298–307, Berlin, Mar. 1996. IEEE Computer Society Press.

[80] E. Lippe and N. van Oosterom. Operation-based merging. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments (SDE5)*, ACM SIGSOFT Software Engineering Notes 17(5), pages 78–87, Tyson's Corner, Virginia, Dec. 1992.

[81] C. Liu and R. Conradi. Automatic replanning of task networks for support process model evolution in EPOS. In I. Sommerville and M. Paul, editors, *Proceedings of the European Software Engineering Conference '93*, LNCS 717, pages 434–450, Garmisch-Partenkirchen, Sept. 1993. Springer-Verlag.

[82] S. A. MacKay. The state-of-the-art in concurrent, distributed configuration management. In Estublier [37], pages 180–194.

[83] A. Mahler. Variants: Keeping things together and telling them apart. In Tichy [119], pages 73–98.

[84] A. Mahler and A. Lampen. An integrated toolset for engineering software configurations. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM Software Engineering Notes 13(5), pages 191–200, Boston, Massachusetts, Nov. 1988.

[85] K. Marzullo and D. Wiebe. Jasmine: A software system modelling facility. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, ACM SIGPLAN Notices 22(1), pages 121–130, Palo Alto, California, Dec. 1986.

[86] B. Munch. HiCOV: Managing the version space. In Sommerville [114].

[87] B. P. Munch. *Versioning in a software engineering database — the change oriented way.* PhD thesis, NTNU Trondheim, Norway, 1993.

[88] B. P. Munch, J.-O. Larsen, B. Gulla, R. Conradi, and E.-A. Karlsson. Uniform versioning: The change-oriented model. In Feldman [43], pages 188–196.

[89] M. Nagl. *Software Technik: Methodisches Programmieren-im-Großen.* Springer-Verlag, Berlin, 1990.

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

[91] W. Obst. Delta technique and string-to-string correction. In G. Goos and J. Hartmanis, editors, *Proceedings of the 1st European Software Engineering Conference*, LNCS 289, pages 64–68, Straßburg, Sept. 1987. Springer-Verlag.

[92] OMG. *The Common Object Request Broker*, Dec. 1993.

[93] F. Oquendo, K. Berrado, F. Gallo, R. Minot, and I. Thomas. Version management in the PACT integrated software engineering environment. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, LNCS 387, pages 222–242, Coventry, UK, Sept. 1989. Springer-Verlag.

[94] D. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[95] D. E. Perry. Version control in the Inscape environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 142–149, Monterey, CA, Mar. 1987. IEEE Computer Society Press.

[96] B. Peuschel and W. Schäfer. Concepts and implementation of a rule-based process engine. In *Proceedings of the 14th International Conference on Software Engineering*, pages 262–279, Melbourne, Australia, May 1992. IEEE Computer Society Press.

[97] R. Prieto-Diaz and J. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, Nov. 1986.

[98] R. Ramakrishnan and J. D. Ullman. A survey of deductive database systems. *The Journal of Logic Programming*, 23(2):125–149, May 1995.

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

[100] C. Reichenberger. Concepts and techniques for software version control. *Software — Concepts and Tools*, 15:97–104, 1994.

[101] S. Reiss. Interacting with the FIELD environment. *Software Practice and Experience*, 20(S1):89–115, June 1990.

[102] A. Rich and M. Solomon. A logic-based approach to system modelling. In Feiler [42], pages 84–93.

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

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

[105] J. F. Roddick. A survey of schema versioning issues for database systems. *Information Software and Technology*, 37(7):383–393, July 1995.

[106] H. D. Rombach and M. Verlage. Directions in software process research. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 41, pages 1–63. Academic Press, San Diego, California, 1995.

[107] N. Sarnak, R. Bernstein, and V. Kruskal. Creation and maintenance of multiple versions. In Winkler [133], pages 264–275.

[108] A. Schürr, A. Winter, and A. Zündorf. Graph grammar engineering with progres. In *Proceedings of the European Software Engineering Conference (ESEC '95)*, LNCS 989, pages 219–234, Barcelona, Spain, Sept. 1995. Springer Verlag.

[109] M. Shaw and D. Garlan. *Software Architecture — Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[110] A. P. Sheth and J. A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, Sept. 1990.

[111] R. T. Snodgrass. Temporal databases. In A. Frank, I.Campari, and U. Formentini, editors, *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, volume LNCS 639, pages 22–64, Pisa, Italy, Sept. 1992. Springer Verlag.

[112] Software Maintenance and Development Systems, Concord, Massachusetts. *Aide-de-Camp Product Overview*, 1990.

[113] R. M. Soley and W. Kent. The OMG object model. In Kim [67], pages 18–41.

[114] I. Sommerville, editor. *Proceedings of the 6th International Workshop on Software Configuration Management*, LNCS. Springer-Verlag, 1996.

[115] I. Sommerville and R. Thomson. An approach to the support of software evolution. *The Computer Journal*, 32(5):386–398, 1989.

[116] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases — Theory, Design, and Implementation*. Benjamin/Cummings, Redwood City, California, 1993.

[117] W. Teitelman. A tour through Cedar. In *Proceedings of the 7th International Conference on Software Engineering*, pages 181–195, Orlando, Florida, Mar. 1984. IEEE Computer Society Press.

[118] W. Tichy, editor. *Proceedings of the 2nd International Workshop on Software Configuration Management*, ACM Software Engineering Notes 14(7), Princeton, New Jersey, Nov. 1989.

[119] W. Tichy, editor. *Configuration Management.* John Wiley and Sons, New York, 1994.

[120] W. F. Tichy. Software development control based on module interconnection. In *Proceedings of the IEEE 4th International Conference on Software Engineering*, pages 29–41, Pittsburgh, PA, Sept. 1979. IEEE Computer Society Press.

[121] W. F. Tichy. 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*, pages 31–48, New Orleans, Louisiana, Jan. 1982. North-Holland.

[122] W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, pages 58–67, Tokyo, Japan, Sept. 1982. IEEE Computer Society Press.

[123] W. F. Tichy. RCS – A system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.

[124] W. F. Tichy. Tools for software configuration management. In Winkler [133], pages 1–20.

[125] E. Tryggeseth, B. Gulla, and R. Conradi. Modelling systems with variability using the PROTEUS configuration language. In Estublier [37], pages 216–240.

[126] L. Wakeman and J. Jowett. *PCTE — The Standard for Open Repositories.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[127] K. Walden. Automatic generation of make dependencies. *Software–Practice and Experience*, 14(6):575–585, June 1984.

[128] K. C. Wallnau. Issues and techniques of CASE integration with configuration management. Technical Report CMU/SEI-92-TR-5, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, Mar. 1992.

[129] A. Wasserman. Tool integration in software engineering environments. In *Proceedings of the 2nd International Workshop on Software Engineering Environments*, LNCS 467, pages 137–149, Chinon, France, Sept. 1990. Springer-Verlag.

[130] B. Westfechtel. Structure-oriented merging of revisions of software documents. In Feiler [42], pages 68–79.

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

[132] B. Westfechtel. A graph-based system for managing configurations of engineering design documents. *International Journal of Software Engineering and Knowledge Engineering*, 6(4), Dec. 1996.

[133] J. F. H. Winkler, editor. *Proceedings of the International Workshop on Software Version and Configuration Control*, Grassau, Germany, 1988. Teubner Verlag.

[134] A. Zeller. A unified version model for configuration management. In *Proceedings of the ACM SIGSOFT '95 Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes 20(4), pages 151–160, Washington, Oct. 1995.

[135] A. Zeller. Smooth operations with square operators — the version set model in ICE. In Sommerville [114].

[136] A. Zeller and G. Snelting. Handling version sets through feature logic. In *Proceedings 5th European Software Engineering Conference*, LNCS 989, pages 191–204, Barcelona, Spain, Sept. 1995. Springer-Verlag.

[137] A. Zeller and G. Snelting. Unified versioning through feature logic. Technical Report 96-01, Technische Universität Braunschweig, Germany, 1996.

A COMPARISON OF SCM SYSTEMS

In the following, we classify the SCM systems described in Section 6. Each column of the tables given below corresponds to a certain property. + and - indicate that the property is fulfilled or not fulfilled, respectively. Hatched entries are used if the respective SCM system does not cover the area to which the classification applies (e.g., Make is not concerned with the version space). The tables below supplement the descriptions of SCM systems and may be cross-referenced while reading Section 6.

	general					*product space*					
	environment			*object management*		*domain*		*granularity*		*(coarse–gr.) relationships*	
	tool kit	*lang. based*	*struct. oriented*	*file system*	*data–base system*	*specific*	*general*	*coarse*	*fine*	*composition*	*dependencies*
cond. comp.	+	–	–	+	–	+	–	–	+		
SCCS	+	–	–	+	–	–	+	+	+		
Make	+	–	–	+	–	–	+	+	–	–	+
PIE	–	+	–	+	–	+	–	+	+	+	+
RCS	+	–	–	+	–	–	+	+	+		
Gandalf	–	–	+	–	+	+	–	+	+	+	+
Adele I	+	–	–	+	–	+	–	+	–	+	+
DSEE	+	–	–	+	–	–	+	+	+	+	+
SML	–	+	–	+	–	+	–	+	–	+	+
P–Edit/MVPE	+	–	–	+	–	–	+	–	+		
DAMOKLES	–	–	+	–	+	–	+	+	+	+	+
Marvel	+	–	–	–	+	–	+	+	–	+	+
Odin	+	–	–	+	–	–	+	+	–	+	+
Shape	+	–	–	+	–	–	+	+	+	+	+
PCTE	+	–	–	–	+	–	+	+	–	+	+
NSE	+	–	–	+	–	–	+	+	+	+	+
Aide–de–Camp	+	–	–	+	–	–	+	+	+	+	+
COV	+	–	–	–	+	–	+	+	+	+	+
Inscape	–	+	–	+	–	+	–	+	+	+	+
CAPITL	+	–	–	–	+	–	+	+	+	+	+
ClearCase	+	–	–	+	–	–	+	+	+	+	+
POEM	–	+	–	–	+	+	–	+	–	–	+
CoMa	–	–	+	–	+	–	+	+	+	+	+
ICE	+	–	–	–	+	–	+	+	+	+	–
PCL	+	–	–	+	–	–	+	+	–	+	–
Vesta	+	–	–	+	–	–	+	+	–	–	+
Adele II	+	–	–	–	+	–	+	+	+	+	+

Table 1: Classification of SCM systems (I)

	intensional construction of source versions								*intensional construction of derived versions (building)*						
	computational paradigm		*classes of version rules*			*mode of configuration*		*back-tracking*	*computational paradigm*		*level of build rules*		*evaluation strategies*		*derived object cache*
	functio-nal	*rule–based*	*con-straint*	*prefe-rence*	*default*	*automa-tic*	*interac-tive*		*functio-nal*	*rule–based*	*type*	*instance*	*lazy*	*busy*	
cond. comp.	+	–	–	+	–	+	–	–							
SCCS															
Make									–	+	–	+	+	–	–
PIE	+	–	–	+	–	+	–	–							
RCS	–	+	–	+	+	+	–	–							
Gandalf	–	+	+	+	+	+	–	–	(incremental compiler)						
Adele I	–	+	+	+	+	+	+	–							
DSEE	–	+	–	+	–	+	–	–	–	+	–	+	+	–	+
SML									+	–	–	+	+	–	+
P–Edit/MVPE	+	–	+	–	–	+	–	–							
DAMOKLES															
Marvel									–	+	+	–	+	+	–
Odin									–	+	+	–	+	+	+
shape	–	+	–	+	+	+	–	–	–	+	–	+	+	–	+
PCTE															
NSE									–	+	–	+	+	–	–
Aide–de–Camp	+	–	+	–	–	+	–	–	(generates make files)						
COV	–	+	+	+	+	+	–	+							
Inscape															
CAPITL									–	+	+	–	+	–	+
ClearCase	–	+	–	+	–	+	–	–	–	+	–	+	+	–	+
POEM									+	–			+	–	–
CoMa									+	–	+	–	+	–	–
ICE	–	+	+	–	–	+	+	+							
PCL	+	–	–	+	–	+	+	–	(generates make files)						
Vesta									+	–	+	+	+	–	+
Adele II	see Adele I								definable (e.g., using triggers)						

Table 2: Classification of SCM systems (II)

	version space								***interplay of product and version space***									
	structure		*version set*		*evolution*		*version specification*		*selection order in AND/OR graphs*			*external granularity of versioning*			*deltas*		*control expressions*	
	version graph	*grid*	*extensional*	*intensional*	*revi–sions*	*variants*	*state oriented*	*change oriented*	*product first*	*version first*	*inter–twined*	*product*	*component*	*total*	*embedded*	*directed*	*visible*	*hidden*
cond. comp.	–	+	–	+	+	+	+	+				–	+	–	+	–	+	–
SCCS	+	–	+	–	+	+	+	–	+	–	–	–	+	–	+	–	–	+
Make																		
PIE	–	+	–	+	+	+	–	+				+	–	–	–	+		
RCS	+	–	+	+	+	+	+	–	+	–	–	–	+	–	–	+		
Gandalf	+	–	+	+	+	+	+	–	–	–	+	–	–	+				
Adele I	+	–	+	+	+	+	+	–	–	–	+	–	–	+				
DSEE	+	–	+	+	+	+	+	–	+	–	–	–	–	+	+	–	–	+
SML																		
P–Edit/MVPE	–	+	–	+	+	+	+	+				–	+	–	+	–	–	+
DAMOKLES	+	–	+	–	+	+	+	–	+	+	+	–	–	+				
Marvel																		
Odin																		
Shape	+	–	+	+	+	+	+	–	–	–	+	–	–	+	–	+		
PCTE	+	–	+	–	+	+	+	–	–	+	–	–	–	+				
NSE	+	–	+	–	+	–	+	–	–	+	–	–	–	+	+	–	–	+
Aide–de–Camp	–	+	–	+	+	+	–	–				+	–	–	+	–	–	+
COV	–	+	–	+	–	+	–	+				+	–	–	+	–	–	+
Inscape	+	–	+	–	+	+	+	+	–	–	+	–	+	–				
CAPITL	+	–	+	–	+	+	+	–	–	+	–	+	–	–	–	+		
ClearCase	+	–	+	+	+	+	+	–	–	–	+	–	–	+	+	–	–	+
POEM	+	–	+	–	+	+	+	–	–	+	–	–	–	+	+	–		
CoMa	+	–	+	–	+	+	+	–	–	+	–	–	–	+	–	+		
ICE	definable																	
PCL	–	+	–	+	+	+	+	–				–	–	+	+	–	+	–
Vesta																		
Adele II	+	–	+	+	+	+	+	–	definable			–	–	+	–	+		

Table 3: Classification of SCM systems (III)