# Towards a Uniform Version Model for Software Configuration Management

Reidar Conradi[1] and Bernhard Westfechtel[2]

[1] Norwegian University of Science and Technology (NTNU)
N-7034 Trondheim, Norway
conradi@idt.ntnu.no
[2] Lehrstuhl für Informatik III, RWTH Aachen
Ahornstrasse 55, D-52074 Aachen
bernhard@i3.informatik.rwth-aachen.de

**Abstract.** A rich variety of version models for software configuration management (SCM) has been proposed over the years, and understanding of the basic concepts and their interrelations has been growing accordingly. In this paper, we propose a uniform version model as a common base, and discuss this in view of current SCM systems.

## 1 Introduction

Version models for software configuration management have been studied for a long time. A *version model* defines the data (usually objects) to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions.

A wide variety of version models has been proposed. Among others, the seminal papers of Katz [12] and Feiler [8] have surveyed the state-of-the art as of the late 80's and beginning 90's. Only recently, we have conducted a comprehensive and updated survey of version models realized both in commercial systems and research prototypes [5]. In Section 2, we summarize the most essential definitions and the taxonomy used to describe and compare existing version models.

Based on this work, the current paper goes a step further and proposes a uniform version model which may serve as a common base (Section 3). A uniform model promises many benefits. First, it aids in understanding the commonalities and differences between version models incorporated in different SCM system. Second, it may serve as a basis for providing a customizable and reusable base layer on top of which different version models may be implemented according to the needs of a specific application domain. Third, interoperability between SCM systems can be enhanced if there is a common "reference model".

## 2 Basic Definitions and Taxonomy

**Product space and version space** We may view a database for SCM as a combination of product space and version space. The *product space* contains

the software objects and their relationships. Typical examples of software objects are requirements definitions, software architectures, programs, etc.; relationships are often classified into composition relationships and dependencies. The *version space* contains the versions of product elements (called items below) and arranges them e.g. in version graphs. Furthermore, the relations between product space and version space need to be described (e.g., which items are versioned and which ones are not, how are versions of different items interrelated, etc.).

**Items, versions, version identifiers** 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. An item covers, e.g., files (usually textual) and file directories, objects in object-oriented databases, entities, relationships and attributes in EER databases, etc.

A version of an item may or may not be *immutable*. In case of immutability, each update creates a new version. Immutability may also be controlled selectively. For example, the contents of a C file may be immutable, while detected implementation errors may be stored in a mutable attribute [7].

A *versioned item* is an item put under version control in a SCM *repository* serving as a versioned database. In contrast, only one state at a time is maintained for an *unversioned item*, i.e., changes are done by overwriting. Versioning requires a *sameness criterion*, i.e., there must be some way to decide whether two versions belong to ("are") 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 or offer additional symbolic and user-defined names serving as primary keys. However, a version can also be identified by an expression (a query) over local or global *selection-attributes*, being general attributes or variables. Each selection-attribute characterizes some logical or functional change.

**Revisions and variants** According to the type of evolution, versions are traditionally classified into revisions and variants. Sequential versions that evolve along the time dimension are called *revisions*, and are created to fix bugs in older versions or to otherwise enhance these. Parallel or alternative versions co-existing at a given time are denoted *variants*.

**Deltas, fragments, visibilities** All versions of an item share some *common properties*, which can be represented by unversioned attributes or relationships. Some of these common properties are *constant*, e.g., OIDs, and contribute to the above sameness criteria. Which (common) properties are shared between versions, depend on the specific version model and the actual data schema.

Versions differ with respect to *specific properties*, e.g., represented by non-shared or versioned attributes. The difference between two versions is denoted a *delta*, and serves to identify the changes and to save space.

A *directed delta* records the effect of a sequence of modify operations, which, when applied to a version $v_1$, yields another version $v_2$. In case of *embedded deltas*, fragments (deltas) are stored in an interleaved manner. We also have *selective deltas*, where each linearly-ordered fragment is tagged by a boolean expression over selection-attributes, regulating the *visibility* of the fragment. The latter characterizes the reasons and nature of a change.

**State-based and change-based versioning** Version models which focus on the explicit states, such as revisions and variants, of versioned items are called *state-based (S-B)*. That is, a delta or change is *the difference between two versions*: $\Delta_{ji} = v_j - v_i$.

Some archetypical SCM systems with S-B models are SCCS [20], RCS [26] and Adele [6], although with different delta implementations.

*Change-based (C-B)* models are generalizations of conditional compilation. All variability of an item is expressed by an ordered set $D$ of selective deltas $\Delta$, where each delta is prefixed by a visibility $vis(\Delta)$. Each delta describes certain changes applied to some base version $v_0$, possibly empty. That is, a version is implicit and created as a *set of selected changes*: $v_j = v_0 + \sum_{\{\Delta \in D | vis(\Delta)\}} \Delta$. The visibilities will contain global selection-attributes or an encoding of such, and correspond to *change identifiers*.

Typical SCM systems with C-B are EPOS with its change-oriented versioning (COV) model [15], and NORA with its ICE [27] version model. COV and ICE differ mainly in the logic used to express visibility expressions, but are otherwise similar. SCCS also allows selective deltas, but does not record possible extra "merge-changes" when combining two changes.

A variation of the C-B model is the *change-set* version model, where deltas from different items are grouped into more global deltas to record the implementation of a change request. The different change-sets may be rather freely combined. A typical SCM system offering change-sets is Aide-de-Camp [22].

We can also mention *Operation-Based (O-B)* version models, where a version is characterized by a specific sequence of change-operations. We will not pursue this further.

Note: Even for S-B models, there can be a rather free composition of changes on the configuration level, by treating each versioned item as a "delta"! On the individual item level, we have to rely on merges, possibly tool-supported.

**Version sets, extensional or intensional versioning** A versioned item is a container, often called a *version group*, for the set $V$ of all 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 only supports retrieval of previously constructed versions. New and combined versions must be created by explicit merging. Each version

$v_i$ is identified by a unique number or VID, e.g., V1 or V2.3.4. The external user retrieves some version $v_i$, performs changes to the retrieved version, and finally submits the changed version as a new immutable version $v_{i+1}$. Note that any version $v_i$ can be retrieved at any time.

*Intensional versioning* supports flexible, automatic construction of consistent versions in a large version space. The term "potential versions" [17] emphasizes that a certain version may not have been explicitly constructed before. In intensional versioning, the version set is defined by a composite *predicate*:

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

This means that all versions belong the version set which meet the constraint $c$. A specific version $v$ is then described by another predicate $vd$ (user-defined version description) such that the following condition holds:

$$vd(v) \wedge c(v)$$

Both $vd$ (a user-defined version description) and $c$ (additional internal constraints) are boolean expressions over the space of selection-attributes $A_i$. Examples of such attributes are an `os`-attribute to determine the operating system, or a boolean `Fix`-attribute to indicate whether a certain bug fix should be included.

$vd$ is typically a query over a tuple of attribute values (attribute bindings), i.e. a VID. An example is a $vd$ = `(Unix,X11)` that describes a `Unix` version supporting the `X11` window system.

$c$ defines a (set of) *constraint(s)* which have to be satisfied by all members of $V$. An example is mutual exclusion of a `SUN` and `VAX` variant.

A given version $v$ is thus the "sum" (on-the-spot-merge) of visible fragments, namely the ones matching $q = vd \wedge c$.

A comparison: The difference between extensional and intensional versioning may be illustrated by comparing SCCS or RCS to conditional compilation as supported by the C programming language [13]. SCCS and RCS both store and reconstruct whole versions of text files (extensional versioning). On the other hand, the C-preprocessor for conditional compilation will construct any text file, typically a source program, based on the values of preprocessor variables (intensional versioning). It will only include those text fragments whose conditions (tagged expressions) evaluate to `true`.

Note: Intensional versioning can work against both S-B and C-B, while extensional versioning can only work meaningfully against S-B. See discussion in Section 3.

**Version rules** Intensional versioning is driven by *version rules* which may either be stored in the versioned database or submitted as part of a query. 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 a weak preference and 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. Both preferences and defaults may thus further supplement a user *vd* with extra attribute bindings, going from an external and partially bound vd (a *evd*) to an internal and fully bound one (a *ivd*). Thus the rule part of a versioned database will check and possibly elaborate upon an initial *evd*, thus serving as a *deductive database*:
(evd, version rules) ⇒ (ivd, 'messages').

The number of possible change combinations is exponential, so resolution of version rules may be NP-complete.

The rule base may itself be an evolving entity, e.g., to characterize new changes, to allow merges (new combinations) of old changes, or to prohibit previous combinations due to bad experiences. We must also support new selection-attributes and revise or extend their domains, i.e. a dynamic schema. Simple revisions(!) of the rule base may suffice, allowing re-generation of old versions/configurations by using the corresponding rule base.

Among SCM systems with a stored rule base, we can mention Adele that allows incremental expansion of high-level *vd*s. High-level *evd*s are also applied by HICOV in EPOS [11] [18], and partly in CVS [2] in using RCS.

**Version space** To represent the *version space*, the version set is often organized in a *version graph*, whose nodes and edges correspond to (groups of) versions and their relationships, respectively. Each versioned item has its local version graph which may vary between the items, as there is no uniform global version graph. For example, in RCS and ClearCase [14] a version graph is organized into branches, each of which consists of a sequence of revisions. Variants can be represented by branches only to a limited extent.

Alternatively, a *grid* may be used to arrange versions in an $n$-dimensional space, where each dimension corresponds e.g. to a (variant) selection-attribute, typically being of boolean or enumerated type.

For example, multi-dimensional variation may be represented this way, e.g., varying the window system, the operating system, the DBMS, etc.

Note, that version graphs and grids are rather closely related. In an unconstrained grid, each point corresponds to a version. By adding constraints, we may exclude inconsistent or unreal combinations. For example, consider an $n$-dimensional grid where each dimension corresponds to a change which may either be applied or omitted [18]. Revision chains can be built from constraints of the form $c_2 \Rightarrow c_1$. That is, if $c_2$ is applied, the previous change $c_1$ must be applied as well. Variants correspond to mutually exclusive changes, as expressed by values of an enumerated type like `(c1, c2)`. E.g., $c_1 \otimes c_2$ means that either $c_1$ or $c_2$ may be applied.

**Granularity, component and product versioning** The user normally selects versions of *software objects*, which are rather coarse-grained items but treated as atomic objects by the SCM system. On the other hand, a SCM system

may operate at a much finer granularity – text lines or even syntactical tokens – to effectively store, compare, and construct versions of software objects. From the user's point of view, we distinguish between component and product versioning:

In case of *component versioning*, versions of single components are maintained and assembled into composite configurations. This is called the composition model in [8]. The relations between version spaces of different components are defined by version rules.

In contrast, *product versioning* establishes a total view of a software product or even an entire database. This is done by arranging versions of all items in a uniform, global version space – see e.g. COV [19] against the EPOSDB.

In addition to a version model for general items, we must also have a product model and an underlying repository:

**Product model** This is used to select which items constitute a complete and consistent configuration, whose structure may itself be versioned. AND/OR graphs [25] satisfy this need for the S-B model. C-B models require a uni-version view of the database (see below), where any relationship can connect any object.

**SCM repository, workspace** In such a versioned repository (database), the requested versions must be explicitly checked-in and -out towards a uni-versioned, external and usually file-based *workspace*. A *transaction* embeds a traditional "check-out – modify – check-in" cycle.

Some SCM systems offer a *virtual file system*, where a *vd* dynamically serves as an access filter upon *all* repository accesses, so-called *transparent versioning*. This eliminates the need for an explicit workspace, although the repository now must offer a more advanced transaction mechanism.

Examples of virtual file systems are DSEE[16] and its ClearCase [14] successor, and some research SCM systems. In this paper, we will not consider virtual file systems further.

**Summary** Most of the definitions given above are summarized in the taxonomy shown in Figure 1. Note that in most cases, the displayed alternatives are not mutually exclusive. In particular, a version model may support both revisions and variants, both state- and change-based versioning, both extensional and intensional versioning, as well as different classes of version rules.

## 3 Towards a Uniform Version Model

### 3.1 A summary and some problems with existing version models

– The **S-B model** is the archetypical version model.
Its main advantage is naturalness. The version space is most often a set of local version graphs, with revisions and variants. All this is simple to understand.

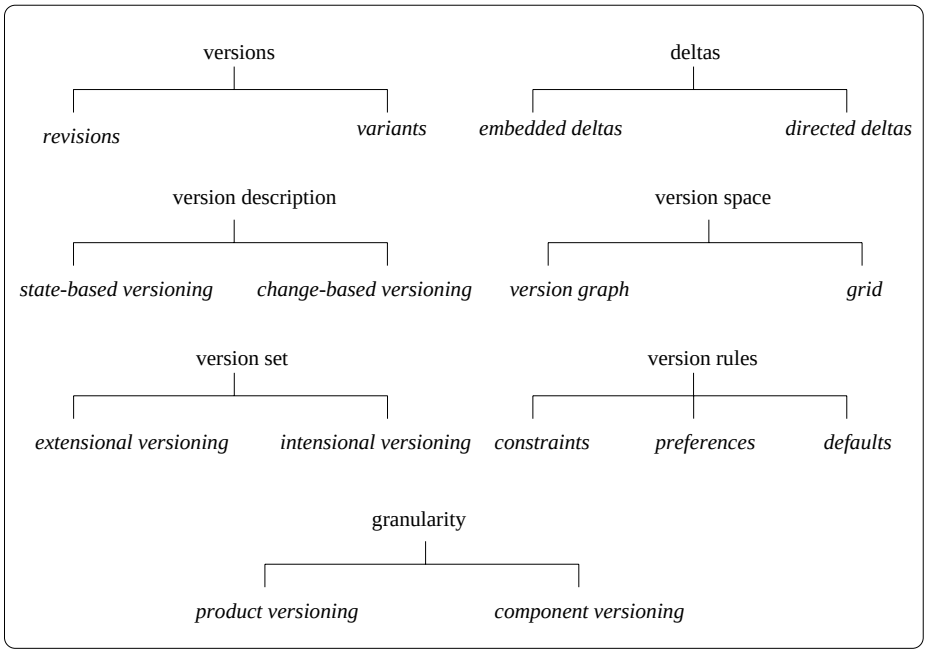



**Fig. 1.** Taxonomy

Its main disadvantage is an explicit explosion of variants to represent combinations of changes, although automatic merging may alleviate this. It is also difficult to consistently combine versions from different objects with structurally different version graphs, although version rules may and do help.

– The **C-B model** is more advanced.

  Its main advantage is compactness (one underlying delta mechanism) and generality (it applies on all data), and combinability of changes also across items.

  Its main disadvantage is complexity and "weirdness". How to get a comprehensive view of the possible versions and their combinations, including how to represent time? We must also find a way to effectively limit the combinatorial explosion of possible versions that now are available, through above version rules. So the need for a high-level (graphical user) interface is high, e.g. how to translate constraints into a visible version graph and how to also display a global *transaction graph* to capture history?

  Immutability must be secured by extra constraints, saying that no more changes are allowed with a certain *vd* selection, i.e., a new selection-attribute is needed.

– **In both models**, it is not clear whether a version possesses individual object existence. E.g., in S-B, are versions explicit and first-class entities, that can play roles in relationships? In C-B, are versions first-class objects or merely

implicit "ghost" objects?
Further, what about (versioned) relationships between different versions, either within the same item or across different items, e.g. how to couple a change request to such a transition?
All we can say, is that most of the above problems will disappear with a uni-version view of a versioned database, see Section 2.

Figure 2 illustrates some proposed versioning models wrt. freedom of combining changes. Below the "(SCCS)" are C-B models, while the other SCM systems represent S-B models. Dimension expressions mean visibilities that are expressions over selection-attributes of enumerated type.

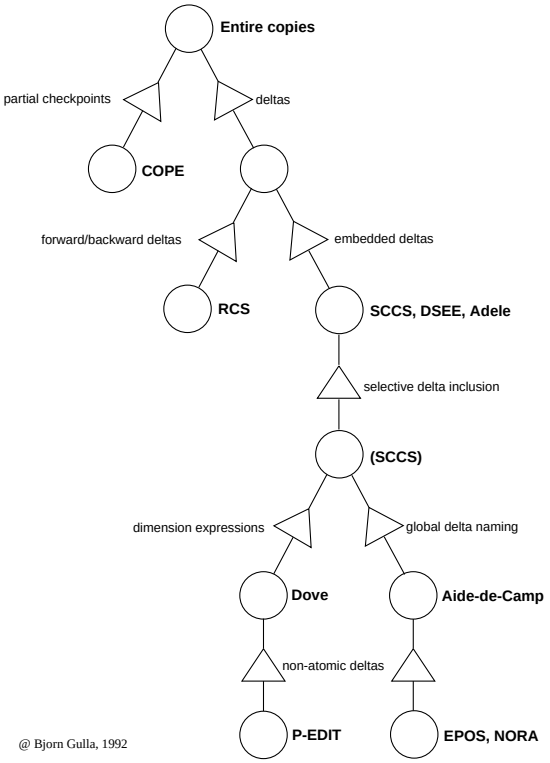



**Fig. 2.** Version models for text versioning, after [10].

As we see, the degree of combinability of changes increases towards the bottom, as summarized in Figure 3. The version models used in P-EDIT [21], DOVE [4] and COPE [1] will not be commented further.

| Feature | Partial check-points | Deltas | Embedded deltas | Selective deltas | Dimension. expres-sions | Global delta naming |
|---|---|---|---|---|---|---|
| COPE | yes | | | | | |
| RCS | | yes | | | | |
| SCCS | | yes | yes | (yes) | | |
| Dove | | yes | yes | yes | yes | |
| P-EDIT | | yes | yes | yes | yes | |
| Aide-de-Camp | yes | yes | yes | yes | | yes |
| COV/EPOS | | yes | | yes | yes | yes |
| ICE/NORA | | yes | | yes | yes | yes |

**Fig. 3.** A characterization of different version models, adapted after [10].

### 3.2 A unified version model: prelude

Understanding of the basic concepts underlying version models has been growing over the years. General agreement upon "the" version model for SCM is unlikely to emerge. However, it seems feasible to come up with a basic version mechanism, and an associated tool kit for defining customized version models. This means provision of a common *base layer*, plus "knobs" to adjust the basic version model. We could conceive of having a basic version model at the bottom, and let either C-B or S-B be visible or accessible on the top.

Further, how can the two main version models, S-B and C-B, be combined? E.g. one pragmatical and compact solution for general text versioning is to use conditional compilation (C-B) for variants and RCS (S-B) for revisions of each "variant file". Such a solution can be applied to all kinds of text files, e.g. requirements, code files, test data, documentation, and even makefiles (the product model) and project plans (the process model).

In an intuitive sense, C-B is the most powerful. So is S-B expressible by C-B, and in case how? For instance, we can mention the more high-level CVS on top of the simpler RCS.

### 3.3 A unified version model: the overall architecture

We are therefore looking for an instrumentable *version engine*, offering a basic delta mechanism and support for version rules/graphs. This version engine must be able to support both a S-B or a C-B version model. On top of this, there should be a product/data model for the specific domain, and a flexible transaction model. On top of that, we need check-out/in facilities for uni-version workspaces (or subdatabases), possibly through a virtal file system. The totality of all this constitutes a *versioned SCM repository*. This is used by external tools such as editors and compilers, and by SCM support tools, e.g. to enter, display and edit general version and product information.

| | |
|---|---|
| Applications | General software tools |
| SCM reposi-itory | Uni-version workspace / Virtual file system |
| | Transaction support & model |
| | Product model / data model |
| Instrumentable version engine | C-B / S-B version model |
| | Version rules and/or version graphs |
| | Basic delta storage |

**Fig. 4.** Architecture for an instrumentable version engine, as part of a SCM repository.

The total architecture is shown in Figure 4. Below, we will discuss three of these layers: the basic delta storage, the version rule/graph, and the product model layer.

Then, which *version model* is the most basic? All version models are founded on *deltas* at the lowest level. Both directed and embedded deltas are suitable candidates, which do not predetermine the high-level layers. For example, directed deltas have been used for both S-B and C-B models, respectively for RCS and PIE [9]. And it could be intriguing to explore if the classic SCCS could be used to provide basic deltas for textual data.

In the sequel, we assume *selective deltas* tagged with general *visibility expressions* as the basic version layer, although such could be rephrased in terms of directed deltas. The visibilities can be compactly coded by internal visibility numbers, one per unique visibility expression. Experience from EPOS [19] indicates that the space and time demands for selective deltas vs. RCS-like directed deltas are similar: In both cases the space demand is twice that of an unversioned text file, while check-out/in time is anyhow dominated by I/O time.

We also assume user-defined and global versioning-attributes, with associated *version rules* to structure and constrain the version space. Such attributes should be of enumerated type, cf. features in ICE. EPOS uses extended boolean options with domain {`false, true, unset`}, but enumerated types can be simulated on top of this. Such version rules offer a continuum of combinations, ranging from "wild merges" to classicly constrained version graphs. NP-complete evaluation of version rules may be a problem, but experience from EPOS indicate a linear growth in expression lengths with the number of relevant versions.

As a product model, we recommend global product versioning as in Section 2. This contrasts with the composition model, which suffers from the complexities of version selections from more or less unrelated version graphs of different components. Global version selection is achieved through selection-attributes with a global scope, e.g., an attribute `os` denotes the operating system throughout the whole software product or a change `fix` may affect multiple components. On the other hand, we should try to apply scoping rules as in block-structured languages, cf. product-specific options in [18].

### 3.4 A unified version model: some details

Thus we propose:

1. **One common representation of versioned items, as follows**:
   Unversioned attributes:
   `OID=...`
   `GAttr1=... , GAttr2=...` etc.
   `VAttr1=... , VAttr2=...` etc.
   Versioned attributes:
   *ivd*: `Status = ...`
   `Attr1` = A set of alternative $vis_{a1} : \Delta_{a1}$
   `Attr2` = A set of alternative $vis_{a2} : \Delta_{a2}$
   etc.
   `Contents` = Sequence of $vis_i : \Delta_i$, where $\Delta_i \in D$.
   `OID` is unversioned and has obvious meaning.
   `GAttr1`, etc. are system-defined and unversioned item attributes, and will normally include a `NAME`.
   `VAttr1`, `VAttr2`, etc. are user-defined and unversioned attributes.
   *ivd* is an internal version description, regulating the `Status` attribute. This attribute has domain `(Illegal, Raw, Edited, Approved, Compiled, Tested, ..., Released, Frozen)`. Thus, a not-recorded-before but legal *ivd* (cf. below constraints) implies that `Status` is set to `Raw`, saying that this version choice (delta combination) has never occurred before.
   $vis_{a1}$, $vis_{a2}$, and $vis_i$ are visibility expressions over global selection-attributes (see below). They also contain a transaction identifier ($TID$, serving as a global change identifier and used for traceability). These visibilities are, repectively, used to tag versioned attribute values and textual deltas.
   `Attr1`, `Attr2` etc. are user-defined and versioned attributes, and all $\Delta_{a1}$ etc. are different.
   `Contents` is a special user-defined and versioned attribute of textual type, containing a sequence of selective deltas.
2. **A version schema** of global selection-attributes each of enumerated types, i.e., meta-data.
   There is a linear revision chain of such, each marked with a `Timestamp`, to reflect new attributes and revised domains.
   Both *vis*, external *evd* and internal *ivd* are expressions over these attributes.
3. **Version rule base**, i.e., also meta-data.
   There will be a corresponding revision chain of such rule bases. Each rule base revision will contain a `TimeStamp` and rules from the following categories over selection-attributes $A_i$, as shown in Figure 5.

4. **Transaction support**.
   A transaction will be recorded in an unversioned(!) *Transaction log* with the following information:
   `Timestamp`,
   Previous $TID$, this $TID$, *eid*, *ivd*, *ambition*,

| Rule type | Example | Comment |
|---|---|---|
| **Constraints**: | | |
| Revision-rule | $A1 \Rightarrow A2$ | (Combination of these two |
| Variant-rule | $A1 \otimes A2$ | gives the version graph). |
| Inconsistency | *logexpr* | Marks an illegal combination. |
| Immutability | *ivd* | I.e., *ivd* cannot be used to write back changes later, see below. |
| **Preferences**: | $A3 \Rightarrow A4$ | |
| **Defaults**: | *"last" changes* | Using the $TID$. |
| **Aggregates**: | $An = logexpr$ | Used to express macros; also product-specific options? |

**Fig. 5.** Version rule base

Description: place, person, general comments,
Change Request Id.

The previous $TID$ is the starting point (if any) of this transaction, and serves to define the *transaction graph*.

As mentioned, the rule base will map a more high-level and incomplete external *evd* to a low-level and complete internal *ivd*, where all relevant selection-attributes are bound.

The given `Timestamp` (its default is the present time) will select a corresponding and possibly previous version schema and rule base. All fragments having a $TID$ *younger* than this `Timestamp` will be discarded upon read-accesses.

The current *ivd*-"filter" is applied on the repository for all read-accesses, causing a subset of the visibilities *vis* to become `true`.

For write-accesses, we apply an *ambition* to denote the change-scope of the updates, and where $ivd \in ambition$ and *ambition* is not marked as immutable – cf. COV. When an operation on a fragment (delta) with visibility *vis* is performed under an ambition $a$, the following update rules apply, see Figure 6.

| Change operation | Old visibility | New visibility |
|---|---|---|
| deletion | *vis* | $vis \wedge \neg a$ |
| insertion | — | $a$ |
| modification | *vis* | $vis \wedge a$ (new), $vis \wedge \neg a$ (old) |

**Fig. 6.** COV rules for update of visibilities on serial deltas.

**Digression**: How to keep track of what changes from one version may be included into other versions, is one of the most difficult problems wrt. visibilities and version rules. I.e. that the current change is done for a given *point* in the version space, but propagated into an *entire plane* in this space. In conditional compilation, "all" versions are edited at the same time.

In P-EDIT, the user can insert-and-merge the current changes into a set of other versions, one at a time. Some concurrent editors allow the same functionality.
In COV, the combinatorial change-constraints are expressed partly by the prescribed update rules on local visibilities, and partly by more global version-rules. Indeed, if we did not update the visibilities as shown above, we have to resort to using $TID$s to express the more detailed "merge"-rules on local changes.
In ICE, the *ambition* for write is equal to the *ivd* for read. Thus further propagation must be done by pairwise merges for the set of relevant combinations, as in most S-B models. This may work fine to realize the desired merges, but is anyhow a "patch" on the underlying version model and global version constraints are needed anyhow. In case of wide propagation (unbound features), the user is faced with similar problems as in conditional compilation (no single-version read view).

5. **External Workspace**.
   A workspace will be established by explicit check-out, or through a virtual file system. All this this is done in a transaction context, see above point.
6. **Versioning assistant**.
   This is needed to generally maintain meta-information: selection-attributes and rules. It will also set *evd*s and *ambition*s, and do check-out and check-in. It should display product models, version rules/graphs, and also transaction graphs.

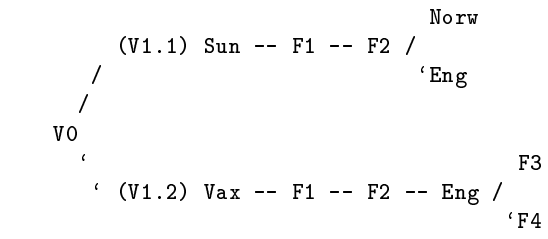
### 3.5 Scenarios

**Example 1: Version graph**

Let us assume six global selection-attributes:

```
OS = (NoOS, Vax, Sun);
Lang = (NoLang, Norw, Eng);
E1 = (NoF1, F1);
E2 = (NoF2, F2);
E3 = (NoF3, F3);
E4 = (NoF4, F4);
```

Example version graph:



Below the corresponding version rules are shown, mostly constraints (simplified). Variant-rules, which can be generated atomatically:

$NoOS \otimes Vax \otimes Sun$
$NoLang \otimes Norw \otimes Eng$
$NoF1 \otimes F1$
$NoF2 \otimes F2$
$NoF3 \otimes F3$
$NoF4 \otimes F4$

Revision-rules:

$\neq NoLang \Rightarrow F2$
$\neg(F3 \wedge F4)$
$F3 \vee F4 \Rightarrow Eng$
$F2 \Rightarrow F1$
$F1 \Rightarrow \neq NoOS$
$NoOS \Rightarrow$ "$V0''$

Possible further evolutions might be to reconcile (merge) $F3$ and $F4$, to let $F1$ and $F2$ be independent and not sequential changes, to allow $Norw$ also for $Vax$ etc.

**Example 2: Visibility and impact of changes**

Consider the following COV-inspired example, with three selection-attributes, $A_{1-3}$. Consider that we have two sets of changes, respectively characterized by $A_1 = true$ and $A_2 = true$.

We now want to perform a merge transaction, allowing the result to be visible for all settings of $A_3$. We then set $ivd = \{A_1 = true, A_2 = true, A_3 = true\}$ and the $ambition = a = \{A_1 = true, A_2 = true, A_3 = unset\}$. Omitting an attribute from a $vd$ or $vis$ means the attribute is left $unset$.

| Comment | Old visibility | New delta | New visibility | New delta |
|---|---|---|---|---|
| either | $\neg A_1$: | `int a;` | (same) | (same) |
| or | $A_1$: | `real a;` | $A_1 \wedge A_2$: | `real a;` |
| | ... | ... | ... | |
| old | $A_2$: | `int i:=a;` | $A_2 \wedge \neg A_1$: | `int i:=a;` |
| merge | | | $A_1 \wedge A_2$: | `int i:=check(a);` |

**Fig. 7.** COV-style update.

## 4 Conclusion

A uniform version model has to support extensional and intensional versioning, revisions and variants, state- and change-based versioning, and different classes of version rules (constraints, preferences, defaults). We have proposed a uniform model featuring: one common representation of versioned items, a dynamic schema, dynamic version rules, a versioned repository with workspaces and transactions, and a versioning assistant.

The approach proposed in this paper has been realized to some extent in the EPOS and NORA research prototypes of C-B SCM systems. Both systems rely on a very flexible base layer which stores deltas tagged with visibility expressions. Above the base layer, a wide range of version models may be implemented.

Then, what about letting the version engine architecture from the last section be incorporated in a standard DBMS? Presently, many SCM systems use a commercial DBMS as a basic repository, like Clearcase using an OODBMS or PCMS using a RDBMS. Further, in a RDBMS the version filter over visibilities can be expressed as a normal read-view, as in the SIO prototype SCM system [3]. However, the data model and the version model are rather entangled in many OODBMSes or CAD systems with versioning support. We rather propose to let the version model be orthogonal to both the actual data model and transaction model (as in EPOSDB), e.g. by letting tagged deltas serve as basic storage fragments in some index-sequential file-system.

Thus, our proposal is a step towards a uniform version model, but we still see unresolved issues and expect further, potentially controversial discussions. Anyhow, further implementation and experimentation is needed to validate the ideas presented in this paper.

**Acknowledgements** Many thanks go to the EPOS group in Trondheim and the IPSEN group in Aachen.

## References

1. J. E. Archer, R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, Jan. 1984.
2. B. Berliner. CVS II: Parallelizing software development. In *Proceedings of 1990 Winter USENIX Conference*, Washington, D.C., Winter 1990.
3. 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.
4. D. M. Brown. *Editing Techniques For Multi-Version Objects.* PhD thesis, University of Southern California, Computer Science Department, Los Angeles, CA 90089-0782, Aug. 1983. TR-83-214.
5. R. Conradi and B. Westfechtel. Version models for software configuration management. Technical Report AIB 96–10, RWTH Aachen, Aachen, Germany, 1996. submitted for publication.

6. J. Estublier and R. Casallas. The Adele configuration manager. In Tichy [24], pages 99–134.
7. J. Estublier and R. Casallas. Three dimensional versioning. In J. Estublier, editor, *Software Configuration Management: Selected Papers SCM-4 and SCM-5*, LNCS 1005, pages 118–135. Springer-Verlag, 1995.
8. 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.
9. I. P. Goldstein and D. G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, XEROX PARC, Palo Alto, California, 1980.
10. B. Gulla. *User Support Facilities for Software Configuration Management.* PhD thesis, NTNU, Trondheim, 1997 (forthcoming).
11. B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version descriptions in EPOS. *Software Engineering Journal*, 6(6):378–386, Nov. 1991.
12. R. H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, Dec. 1990.
13. B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1978.
14. D. Leblang. The CM challenge: Configuration management that works. In Tichy [24], pages 1–38.
15. 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.
16. D. Lubkin. Heterogeneous configuration management with DSEE. In P. H. Feiler, editor, *Proceedings of the 3rd International Workshop on Software Configuration Management*, pages 153–160, Trondheim, Norway, June 1991.
17. B. P. Munch. *Versioning in a Software Engineering Database — the Change Oriented Way.* PhD thesis, NTH, Trondheim, Norway, Aug. 1993. 265 p. (PhD thesis NTH 1993:78).
18. B. P. Munch. HiCOV: Managing the Version Space. In Sommerville [23], pages 110–126.
19. B. P. Munch, J.-O. Larsen, B. Gulla, R. Conradi, and E.-A. Karlsson. Uniform versioning: The change-oriented model. In S. Feldman, editor, *Proceedings of the 4th International Workshop on Software Configuration Management (Preprint)*, pages 188–196, Baltimore, Maryland, May 1993.
20. M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, Dec. 1975.
21. N. Sarnak, R. Bernstein, and V. Kruskal. Creation and maintenance of multiple versions. In J. F. H. Winkler, editor, *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 264–275, Grassau, Germany, 1988. Teubner Verlag.
22. Software Maintenance and Development Systems, Concord, Massachusetts. *Aide-de-Camp Product Overview*, 1990.
23. I. Sommerville, editor. *Proceedings of the 6th International Workshop on Software Configuration Management*, LNCS 1167. Springer-Verlag, 1996.
24. W. Tichy, editor. *Configuration Management.* John Wiley and Sons, New York, 1994.
25. 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.
26. W. F. Tichy. RCS – A system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.
27. 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.

This article was processed using the LaTeX macro package with LLNCS style