A Unified Version Model for Configuration Management

Andreas Zeller*

Abteilung Softwaretechnologie
Technische Universität Braunschweig, Germany

Abstract

Integration of configuration management (CM) tools into software development environments raises the need for CM models to interoperate through a unified CM model. A possible foundation is the *version set model*, a unified model for specifying versions and version operations, where versions, components, and aggregates are grouped into sets according to their features, using *feature logic* as a formal base to denote sets and operations and deduce consistency. Version sets generalize well-known CM concepts such as components, repositories, workspaces, aggregates, or configurations and allow for unprecedented flexibility in combining these concepts. Arbitrary revision/variant combinations of components and aggregates are modeled in a uniform and orthogonal way.

We show how the concepts of four central configuration management models—the checkin/checkout model, the change set model, the composition model, and the long transaction model—are encompassed and combined by the version set model, making it a unified basis for modeling, realizing and integrating configuration management tasks. Finally, some conditions for efficient realization are identified, based on our practical experience with the configuration management tool ICE. Although the described operations generally result in exponential time complexity, it turns out that the discussed CM concepts can be realized and combined without loss of efficiency.

Key words: Software configuration management, Version control, Deduction and theorem proving, Knowledge representation formalisms and methods, Feature logic

*This work is funded by the Deutsche Forschungsgemeinschaft, grants Sn11/1-2 and Sn11/2-2. Author's current address: Technische Universität Braunschweig, Abteilung Softwaretechnologie, Gaußstr. 17, D-38092 Braunschweig/Germany. E-mail: zeller@acm.org.

SIGSOFT '95 Washington, D.C., USA
© 1995 ACM 0-89791-716-2/95/0010...$3.50

1 Introduction

In his recent survey on configuration management models in commercial environments [6], Peter H. Feiler concludes:

> Configuration management (CM) capabilities can be found not only in CM tools and environment frameworks, but also in development tools. Integration of such tools into environments raises the need for different CM models to interoperate. Therefore, it is desirable to evolve to a unified CM model that encompasses the full range of CM concepts and can be adapted to different software process needs.

In this paper, we present the *version set model*, a unified versioning model that encompasses and integrates the models discussed by Feiler and constitutes a possible base of a fully unified CM model. Version sets are sets of objects (typically software components), characterized by a *feature term*—a boolean expression over (*feature*: *value*)-attributions denoting common and individual version properties, following the CM convention to characterize objects by their attributes. Version sets generalize well-known CM concepts such as components, repositories, work spaces, variant sets, or revision histories and allow for unprecedented flexibility in combining these CM concepts. Using *feature logic*, intersection, union, and complement operations on version sets are realized in order to express and generalize the semantics of CM models.

We have implemented the version set model in a CM tool, named ICE for *Incremental Configuration Engine*. ICE represents version sets as files and realizes version operations through file manipulations. Through *feature unification*, a constraint-solving technique, we can determine whether version sets exist, ensuring consistency of configurations and inferring necessary steps for their construction. Although feature unification is NP-complete, our experiences show that the discussed CM models can be simulated and combined without loss of efficiency.

2 Basic Notions of Feature Logic

We begin with a short overview of feature logic. Feature terms and feature logic have originally been developed for semantic analysis of natural language [14, 30]. Later, they were used as a general mechanism for knowledge representation [3, 25] and as a basis for logic programming [12, 32]. Throughout this paper, we will concentrate on an intuitive understanding, based on the formal semantics described by Smolka [31].

In feature logic, *feature terms* denote sets of objects characterized by certain features. In their simplest form, feature terms consist of a conjunction of (*feature*: *value*)-pairs, called *slots*, where each feature represents an attribute of an object. Feature values may be literals, variables, and (nested) feature terms; PROLOG-like first-order terms may also be used. As an example, consider the following feature term T, which expresses linguistic properties of a piece of natural language:

$$T = \left[\begin{array}{l} \textit{tense}: \textit{present}, \\ \textit{predicate}: [\textit{verb}: \textit{sing}, \textit{agent}: X, \textit{what}: Y], \\ \textit{subject}: [X, \textit{num}: \textit{sg}, \textit{person}: \textit{third}], \\ \textit{object}: Y \end{array}\right]$$

This term says that the language fragment is in present tense, third person singular, that the agent of the predicate is equal to the subject etc.: T is a representation of the sentence template "X sings Y".

The syntax of feature terms is given in table 1, where a denotes a literal (e.g. numbers, strings, and atomic constants), V denotes a variable, f and g denote features, and S and T denote feature terms. Feature terms are constructed using intersection, union, and complement operations. If $S = [f: X]$ and $T = [g: Y]$, then $S \sqcap T = [f: X, g: Y]$, which is read as "$f$ has value X and g has value Y". Similarly, $S \sqcup T = \{f: X, g: Y\}$, which is read as "$f$ has value X or g has value Y". Feature values may be feature terms as well; in this paper, we restrict ourselves to simple equivalences like $f: \{X, Y\} = \{f: X, f: Y\}$, which is read "$f$ has value X or Y". Feature terms form a boolean algebra, hence boolean transformations like distribution, de Morgan's law etc. hold for feature terms as well.

Notation	Name	Interpretation
a	Literal	
V	Variable	
$\top$	Universe	Ignorance
$\bot$	Empty set	Inconsistence
$f: S$	Selection	The value of f is S
$f: \top$	Existence	f is defined
$f\uparrow$	Divergence	f is undefined
$f \downarrow g$	Agreement	f and g have the same value
$f \uparrow g$	Disagreement	f and g have different values
$S \sqcap T$ or $[S, T]$	Intersection	Both S and T hold
$S \sqcup T$ or $\{S, T\}$	Union	S or T holds
$\sim S$	Complement	S does not hold
$S \sqsupseteq T$	Subsumption	S subsumes T; T implies S
$S \boxplus_I T$	Aggregation	(See text)

Table 1: The syntax of feature terms

Sometimes it is necessary to specify that a feature exists (i.e. is defined, but without giving any value), or that a feature does not exist in a feature term. This is written $f: \top$ resp. $\sim f: \top$ (abbreviated as $f\uparrow$). The possibility to specify complements greatly increases the expressive power of the logic. For example, the term $\sim$[*compiler*: *gcc*] denotes all objects whose feature *compiler* is either undefined or has another value than *gcc*. The term [*compiler*: $\sim$*gcc*] denotes all objects whose feature *compiler* is defined, but with a value other than *gcc*.

A feature term can be interpreted as a representation of the infinite set of all ground terms T' which are *subsumed* by the original term T (that is, $T \sqsupseteq T'$). Subsumed terms are obtained by substituting variables or adding more features. Hence, feature terms always allow for further specialization, like classes in object-oriented models. For instance, $\top \sqsupseteq$ [*fruit*: *apple*] $\sqsupseteq$ [*fruit*: *apple*, *color*: *green*] $\sqsupseteq$ [*fruit*: *apple*, *color*: *green*, *wormy*: *no*], and so on.

Feature logic also provides a simple consistency notion. As feature logic assumes that each feature can have only one value, the term [*os*: *dos*, *os*: *unix*] is equivalent to $\bot$, the empty set; such terms are called *inconsistent*. Through *feature unification* [31], a constraint-solving technique, one can determine consistency of arbitrary feature terms. For terms without unions and complements, feature unification works similar to classical unification of first-order terms; the only difference is that subterms are not identified by position (as in PROLOG), but by feature name. Adding unions forces unification to compute a (finite) union of unifiers as well, whereas complements are usually handled by constraint solving (similar to negation as failure).

Feature unification implies that the intersection of feature terms is non-empty only if all common features have same values. This is not always desirable. A car, for instance, inherits its features from all its components, but enforcing that every component has the same feature *manufacturer* would be unrealistic. Thus, we define a special *aggregation operator* "$\boxplus_I$" similar to $\sqcap$, but uniting the values of the *independent features* in the set I instead of unifying them:

$$([f: a] \sqcap S) \boxplus_I ([f: b] \sqcap T) = \begin{cases} [f: a \sqcup b] \sqcap S \sqcap T & \text{if } f \in I, \\ [f: a \sqcap b] \sqcap S \sqcap T & \text{if } f \notin I. \end{cases}$$

Obviously, $S \boxplus_{\{\}} T = S \sqcap T$ holds.

3 Versions and Version Sets

To model version sets, each version of a component c is tagged with a term $T' = [object: c] \sqcap F$, where c is some unique component identifier, and F is a feature term describing the features of the specific version. F and T' may be interpreted as boolean expressions denoting the features of c; formally, each T' is a singleton set containing the version F of the object c. The component itself is identified by T, where T is the union of all sets T' (or, using boolean interpretation, the logical "or" of all version features). So, if we have a component c in n versions $T_1, T_2, \ldots, T_n$, the version set T is determined as

$$T = T_1 \sqcup T_2 \sqcup \cdots \sqcup T_n = \bigsqcup_{i=1}^{n} T_i \ .$$

Features F of the component itself (as $[object: c]$) are the same across all versions, and hence can be factored out through

$$(F \sqcap T_1) \sqcup (F \sqcap T_2) = F \sqcap (T_1 \sqcup T_2) \ .$$

As an example, consider the version set shown in figure 1, modeling a simple portable CD-ROM player. It contains two components *screen* and *drive*, each present in two versions. The *screen* and *drive* version sets are identified by $screen = screen_1 \sqcup screen_2$ and $drive = drive_1 \sqcup drive_2$, where

$$\begin{aligned} screen_1 &= [object: screen, resolution: high, \\ &\qquad speed: high] \\ screen_2 &= [object: screen, resolution: medium, \\ &\qquad speed: \{medium, low\}] \\ drive_1 &= [object: drive, speed: high] \\ drive_2 &= [object: drive, speed: medium] \ . \end{aligned}$$

Hence, the version set denoting *drive* is

$$\begin{aligned} drive &= drive_1 \sqcup drive_2 \\ &= [object: drive, speed: \{high, medium\}] \ . \end{aligned}$$

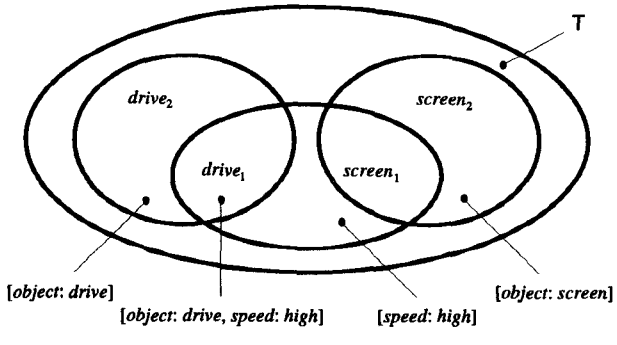

Figure 1: Selection in a version set

To access a specific version in a version set T, we intersect T with a *selection term* S specifying the desired features. For any selection term S and a set of versions T, we can identify the versions satisfying S by determining $T' = T \sqcap S$—that is, the set of versions that are in S as well as in T. If $T' = \bot$, or if T' does not denote any existing version, selection fails. In our example, selecting $S = [speed: high]$ from the version set *drive* returns $drive \sqcap S = (drive_1 \sqcup drive_2) \sqcap S = (drive_1 \sqcap S) \sqcup (drive_2 \sqcap S) = drive_1 \sqcup \bot = drive_1$. The component $drive_2$ is excluded, because the *speed* feature may have only one value and therefore $drive_2 \sqcap S = \bot$ holds.

4 Aggregates and System Structures

We shall now turn to *systems*, or, generally, aggregates of components. In the version set model, aggregates inherit the features from their components; the features of an aggregate composed of $screen_1$ and $drive_1$ are $[resolution: high, speed: high]$. Obviously, aggregate features are obtained by intersection of the component features. However, not all component features need be common: features like *author* or *status* might be different across components, *object* features differ by definition.

This is where the aggregation operator can be used. Using "$\boxplus_I$" with $I \supseteq \{object\}$ containing the independent features, $screen_1 \boxplus_I drive_1$ is

$$[object: \{screen, drive\}, resolution: high, speed: high] \ ,$$

a version set containing the components $screen_1$ and $drive_1$, excluding other possible components, and correctly describing the features of the aggregate. Formally, if T is an aggregate from n components identified by $T_1, T_2, \ldots, T_n$, T is defined as

$$T = T_1 \boxplus_I T_2 \boxplus_I \cdots \boxplus_I T_n = \boxplus_{i=1}^{n}{}_I T_i \ ,$$

where $object \in I$ holds.[1]

As each T_i is a version set, it need not be singleton. Consequently, the version set T need not be singleton, but can denote a whole set of possible aggregates, or *configurations*. For instance, the version set $T = screen \boxplus_I drive$ contains all possible configurations from the *screen* and *drive* components. A simple evaluation to

$$\begin{aligned} T = [object: \{screen, drive\}] \\ \sqcap([resolution: high, speed: high] \\ \sqcup[resolution: medium, speed: high] \\ \sqcup[resolution: medium, speed: medium]) \end{aligned}$$

[1] In [42], we used simple intersection for aggregation, ignoring independent features.

shows that T contains exactly three configurations; the high-resolution/medium-speed and low-speed configurations are excluded by the features of the components.

As the components of an aggregate may be aggregates again, we can describe a full system structure by compositions ($\boxplus_I$) and alternatives ($\sqcup$), in a manner similar to Tichy's AND/OR graphs [36]. Through transformations of T according to the rules of feature logic, arbitrary interchanged selection and composition stages are possible. Complements may also be used to express that a specific version set *not* be included in an aggregate—for instance, $T' = T \sqcap \sim[os\colon unix]$ contains all non-UNIX configurations of T. As versions, components, aggregates and configurations are all modeled by version sets, all version set operations can be applied equally, making aggregates and configurations first-class objects as in [15].

5 Revisions and Repositories

We now show how classical CM models are embedded into the version set model. In the *checkout/checkin* model, as exemplified by the well-known SCCS [29] or RCS [37] tools, revisions[2] of files are stored in a *repository*. Users must *check out* (retrieve) a specific revision from the repository to access the revision, possibly *checking in* (storing) derived new revisions in the repository. Individual revisions can be *locked* against concurrent editing. Obviously, a repository can be regarded as a version set; however, special care must be taken to represent the revision history.

We assume that revisions are created by applying a change (or *delta*) to an existing version set. Hence, we distinguish the old and the new revision by checking whether the change has been applied, or not—which is represented by existence or non-existence of a *delta feature* standing for the change application. In the following, we shall denote the individual revisions of a component by $\Delta_0, \Delta_1, \Delta_2, \ldots$ and so on. The change leading up to a revision Δ_i is denoted by δ_i. To every Δ_i revision except Δ_0, we assign a delta feature $[\delta_i\colon \top]$. Each Δ_i revision also inherits the delta features of all revisions it is based upon, such that a Δ_2 revision based on the Δ_1 revision has the features $[\delta_1\colon \top, \delta_2\colon \top]$—that is, Δ_2 is the result of the δ_1 and δ_2 changes applied to Δ_0.

Each time a new revision δ_i of a version set T is created, T is *split* into two parts $T \sqcap [\delta_i\colon \top]$ (the versions where the δ_i change has been applied), and $T \sqcap [\delta_i\!\uparrow]$ (the versions where the δ_i change has not been applied). As both $T \sqcap [\delta_i\colon \top]$ and $T \sqcap [\delta_i\!\uparrow]$ are subsets of T, the resulting version sets form a *subsumption lattice* that is isomorph to the evolution diagram, as shown in figure 2. Arbitrary version sets can be selected by specifying a list of included or excluded changes, denoting paths in the lattice. For instance, the version set $[\delta_4\colon \top]$ denotes Δ_4 and all its descendants; selecting $[\delta_3\colon \top, \delta_4\!\uparrow]$ from the repository returns the single version Δ_3.

[2] According with Winkler [40], our version concept encompasses both *revisions* and *variants*; revisions supersede an existing version set, while variants do not.

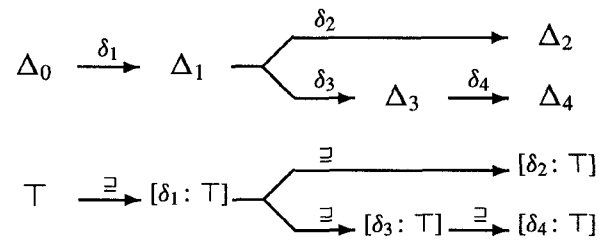

Figure 2: An evolution diagram and its version sets

In figure 2, there is no version set $[\delta_2\colon \top, \delta_4\colon \top]$, since $[\delta_3\colon \top] \sqsupseteq [\delta_4\colon \top]$ holds and the sets $[\delta_2\colon \top]$ and $[\delta_3\colon \top]$ are disjoint (formally, $[\delta_3\!\uparrow] \sqsupseteq [\delta_2\colon \top]$ and $[\delta_2\!\uparrow] \sqsupseteq [\delta_3\colon \top]$). To integrate the changes δ_2 and δ_4, one would have to create a integrated delta δ_5 derived from both $[\delta_2\colon \top]$ and $[\delta_4\colon \top]$. In the resulting subsumption lattice, we have $[\delta_2\colon \top] \sqsupseteq [\delta_5\colon \top]$ and $[\delta_4\colon \top] \sqsupseteq [\delta_5\colon \top]$; thus, since $[\delta_2\colon \top] \sqcap [\delta_4\colon \top] \sqsupseteq [\delta_5\colon \top]$, the selection $[\delta_2\colon \top, \delta_4\colon \top]$ correctly returns the new Δ_5 revision.

A repository relying on delta features only is rather uncomfortable, as it provides no support for currency or protection from concurrent changes. In order to realize "current" and "locked" revisions, we introduce *current* and *locked* features applying to certain revisions only.

As an example, consider developer Johnson who wishes to check out and lock the current revision in a repository T. Her check-out retrieves $S = [current\colon \top, locked\!\uparrow]$ from T and deletes S from T (by changing T to $T' = T \sqcap \sim S = T \sqcap \{current\!\uparrow, locked\colon \top\}$). In T', the current version is now available only in a locked state, since $T' \sqcap S = \bot$; therefore, subsequent change attempts will fail. Johnson's subsequent check in of the new current version $S' = [current\colon \top, locked\!\uparrow, \delta_i\colon \top]$, first changes the repository T' to $T'' = (T' \sqcap [\delta_i\!\uparrow, current\!\uparrow])$, identifying it by $\delta_i\!\uparrow$ and making its revisions non-current, and then to $T''' = T'' \sqcup S$, adding the new revision. When developer Sanders later checks out $[current\colon \top]$ from T''', he gets the new current version S'.

By introducing and maintaining appropriate features applying to specific revisions, modeling other RCS or SCCS properties such as naming of individual revisions or baselines is straight-forward. As no special distinction is made between delta features and other features, we may create, select and revise arbitrary revision/variant combinations of components as in orthogonal version management [27]. This flexibility is especially useful in workspace management, as we will show in section 7.

6 Composing Consistent Configurations

The *composition model*, as realized in the SHAPE [18, 22], ADELE [5], and CLEARCASE [19] configuration management systems, is an extension of the checkout/checkin model, introducing the notions of a *system* and *system consistency*. A system is an aggregate of components, where each component has its own revision history (and repository). A configuration is created by *composing* the system from its components and *selecting* individual component versions such that the resulting system is consistent. Selection is done through *selection rules*, ensuring consistency.

We have already shown how systems are composed using version sets, using a term

$$T = [\mathit{object}\colon \{c_1, c_2, \ldots, c_n\}] \sqcap F$$

to select the components c_i in versions specified by their features F. Selection rules are accommodated by transforming them in equivalent feature terms. For instance, an ADELE selection rule

$$F = (\mathit{window\text{-}system} = \mathit{x11} \\ \wedge (\mathit{current} \vee \mathit{status} \neq \mathit{experimental}))\ ,$$

expressing "The window system is X11; only current or non-experimental components are to be included", becomes in feature logic

$$F = [\mathit{window\text{-}system}\colon \mathit{x11}, \\ \{\mathit{current}\colon \top, \mathit{status}\colon \sim\mathit{experimental}\}]\ .$$

Selection of F in a version set T yields a version set $T' = T \sqcap F$ denoting all configurations with the desired features. T' may be empty (that is, $T' = \bot$), indicating that no such configuration exists. T' may also contain multiple configurations, in which case more specific selections are required to make T' unambiguous.[3] Individual revisions may be selected by implications like $\{\sim\mathit{object}\colon \mathit{screen}, [\delta_4\colon \top, \delta_5\uparrow]\}$, returning the Δ_4 revision of the *screen* object.

In our model, there is not only consistency of a system against some selection term F, but also consistency of components against others. Due to the consistency notion of feature logic, we cannot compose a system from two components identified by, say, T and T' with

$$T = [\mathit{object}\colon \mathit{gadget}, \mathit{screen\text{-}type}\colon \mathit{tty}]$$
$$T' = [\mathit{object}\colon \mathit{widget}, \mathit{screen\text{-}type}\colon \mathit{x11}]\ ,$$

since $[\mathit{screen\text{-}type}\colon \mathit{tty}] \sqcap [\mathit{screen\text{-}type}\colon \mathit{x11}] = \bot$ and therefore $T \boxplus_I T' = \bot$. Similarly, we cannot compose two components identified by $[\delta_i\colon \top]$ and $[\delta_i\uparrow]$, which makes sure that a change made to several components is not partially excluded. This *intrinsic consistency* notion is explained in detail in [42]; it ensures local and global consistency without special selection rules and allows for incremental exploration of the configuration space.

[3] If all component features are well-determined (that is, component versions are not identified by negated feature terms), F may also include arbitrary relational or arithmetic constraints like $\mathit{x11\text{-}release} \geq 4$ (in feature logic, a first-order-term $\mathit{greater\text{-}or\text{-}eq}(\mathit{x11\text{-}release}, 4)$), which are not generally decidable otherwise. We are currently investigating on integrating arithmetic constraints into feature unification.

7 Concurrent Development in Workspaces

While the composition model ensures consistency, there is no intrinsic support for parallel and distributed development. This is the primary concern of the *long transaction model*, as realized in the SUN NSE system [4]. Here, developers create a *workspace* by selecting a configuration from an environment, which is a repository or another workspace. All changes are local to the workspace until the changes are *committed* to the environment. In contrast to the composition model, developers first select a specific configuration and then perform changes on components and the system model.

Many implementations of the workspace model allow for concurrent changes. Instead of providing a locking mechanism, currency is confined to workspaces; each workspace has its own notion of a current component revision. Only through a synchronization operation is the local currency updated. When changes are committed to the environment, the CM system compares the originating revision with the current one in the environment. If they differ, another user has performed changes in the meantime; these changes must be integrated before committing the new current revision.

In the version set model, a workspace is modeled as a *subset* of an originating system configuration; the origin (the superset) may be the entire system version set as well as another workspace. Just like creating a revision, creating a workspace W splits an original configuration T into two subsets $T \sqcap W$ (the subset where changes are confined to), and $T \sqcap \sim W$ (the original version set).

By applying further changes to the $T \sqcap W$ subset only, a change δ_i is confined to the workspace W; the resulting revisions are identified by $\Delta_{i\text{-}1} = T \sqcap (\sim W \sqcup [\delta_i\uparrow])$ and $\Delta_i = T \sqcap W \sqcap [\delta_i\colon \top]$. The new revision Δ_i is not visible in the workspace complement $\sim W$, since $\Delta_i \sqcap \sim W = \bot$. To commit a change δ_i in W to the originating version set T—that is, make δ_i visible outside of W—, we observe that $T \sqcap W \sqcap [\delta_i\colon \top] = T \sqcap [\delta_i\colon \top]$ holds since $[\delta_i\colon \top] \sqsubseteq W$. Hence, we can rename the set $T \sqcap W \sqcap [\delta_i\colon \top]$ to $T \sqcap [\delta_i\colon \top]$, making δ_i visible in the other workspaces $\sim W$ as well (formally, $T \sqcap [\delta_i\colon \top] \sqcap \sim W \neq \bot$). An additional currency scheme can be used to make $[\delta_i\colon \top]$ the "current" version, as we will illustrate in section 9.

8 Committing Changes via Change Sets

Creation of a workspace may be realized logically, for instance by creating a special view on the environment. A more sophisticated implementation would allow for geographically distributed development where changes are committed by transferring them between workspaces as individual entities. This is the basic idea of the *change set* model, as realized in the EPOS [20, 24] and AIDE-DE-CAMP [11] systems. Here, changes are individual entities which can be applied to a base revision. Changes can be combined to change sets, allowing for the representation of correlated changes. Special care is required to determine whether change sets are in conflict.

In the previous text, we introduced version sets like $T = [object{:}\ editor, \delta_1 : \top, \delta_2\uparrow]$ to represent the *editor* component with the change δ_1 applied and the change δ_2 excluded. To accommodate change sets, version sets T are considered *aggregates* from base versions B and later changes C—that is, $T = B \sqcap C$; in our example, $B = [object{:}\ editor]$ and $C = [\delta_1 : \top, \delta_2\uparrow]$. This allows for the representation of change sets *without* a referred object, as in $C' = C \sqcap {\sim}B = [{\sim}object{:}\ editor, \delta_1 : \top, \delta_2\uparrow]$. C' effectively contains all the changes applied to B (in this case, the differences between the Δ_0 and Δ_1 revisions of B), but not B itself. As changes usually have a smaller representation than version sets, it suffices to transfer only C' between workspaces where T is already known.

As individual changes are combined to aggregates (i.e., change sets), the subsumption lattice is used to determine the consistency of a change set. For instance, for the evolution shown in figure 2, there can be no change set $[\delta_3\uparrow, \delta_4 : \top]$ (that is, δ_4 applied but δ_3 excluded): Since in this case, $[\delta_3 : \top] \sqsupseteq [\delta_4 : \top]$ holds, we have $[\delta_4 : \top] = [\delta_3 : \top] \sqcap X$ and $[\delta_3\uparrow, \delta_4 : \top] = [{\sim}\delta_3 : \top] \sqcap [\delta_3 : \top] \sqcap X = \bot$. Note that in contrast to the checkout/checking model, a change δ_i may apply to many components at once. This implicitly ensures that aggregation of components $[\delta_i : \top]$ (the δ_i change is applied) and components $[\delta_i\uparrow]$ (the δ_i change is not applied) is forbidden.

In many cases, version sets integrating different changes can be generated automatically. This requires the changes to be *independent* from each other; that is, each change can be applied independent from the other, resulting in well-formed components. For instance, the change set $[\delta_3\uparrow, \delta_4 : \top]$ may be generated automatically if we know that δ_4 is a change independent from δ_3—that is, the change set $[\delta_3\uparrow, \delta_4 : \top]$ can be applied, still resulting in a correct system.

Determining independence requires knowledge on the semantics of changes and the involved components (when is a component "well-formed" and when not?); it is not decidable in general. However, there are reasonable approaches to determine change independence and realizing integrated changes—either textually [2, 11], syntactically [35, 39], or even semantically [41]. While such approaches are usually based on integrating two single versions, they can easily be extended for integrating arbitrary version and change sets.

9 A Comprehensive Example

In the following scenario, we show how the techniques from the presented CM models can be combined. Consider the CD-ROM player example, consisting of a *screen* and a *drive* component. Developer Sanders wishes to improve the $screen_1$ component such that it also works with the medium speed drive $drive_2$ as well. He creates a workspace $W = [user{:}\ sanders]$. This "creation" does not have any impact on T, since $T \sqcap W$ and $T \sqcap {\sim}W$ are yet identical. Sanders performs a change δ_1 on the $screen_1$ component, changing its features and making it current. As δ_1 is confined to W, only the W variant of $screen_1$ is changed (that is, $screen_1 \sqcap W$).

$screen_1$ now contains two versions: the original version (δ_1 not applied) identified by

$$screen_{1.0} = [\delta_1\uparrow, speed{:}\ high] \ ,$$

and Sander's version (δ_1 applied), identified by

$$screen_{1.1} = [\delta_1 : \top, speed{:}\ \{medium, high\}, \\ user{:}\ sanders] \ .$$

Hence, $screen_1$ may be written as

$$screen_1 = [object{:}\ screen, resolution{:}\ high] \\ \sqcap (screen_{1.0} \sqcup screen_{1.1}) \sqcap R \ ,$$

where R realizes currency through implications:

$$R = \{{\sim}[user{:}\ sanders, current{:}\ \top], \delta_1 : \top\} \\ \sqcap \{[user{:}\ sanders, current{:}\ \top], \delta_1\uparrow\}$$

making sure that whenever Sanders requests the current version, δ_1 is returned.

Other developers will not see any of Sanders' changes; seen from developer Cherry's workspace, $screen_1$ is still a singleton set, since $screen_1 \sqcap [user{:}\ cherry]$ still returns the original version $screen_{1.0}$. (Some supervisor, however, may examine the entire version set $screen_1$ to trace changes without being confined to a user workspace).

In the meantime, developer Johnson commits an own change δ_2 to $screen_1$ from her workspace. The change δ_2 splits the version set $screen_1$ into two parts $[\delta_2\uparrow]$ and $[\delta_2 : \top]$. $[\delta_2 : \top]$ is the new "current" version; however, this is invisible to Sanders since workspaces maintain their own currency. $screen_1$ becomes a union of three versions

$$screen'_1 = [object{:}\ screen, resolution{:}\ high] \\ \sqcap (screen'_{1.0} \sqcup screen'_{1.1} \sqcup screen'_{1.2}) \sqcap R$$

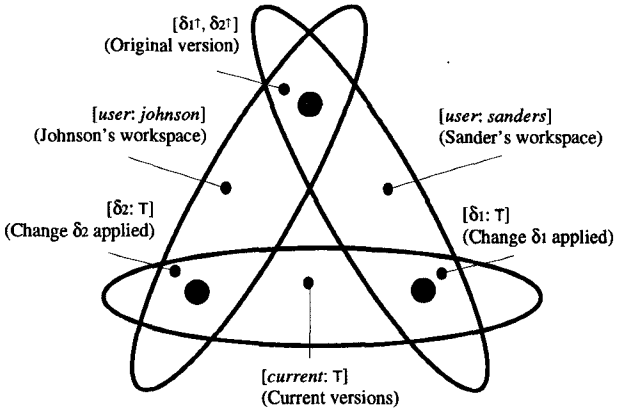

Figure 3: Workspaces as version sets

identified by

$$\begin{aligned} screen'_{1.0} &= screen_{1.0} \sqcap [\delta_2\uparrow] \\ &= [\delta_1\uparrow, \delta_2\uparrow, speed{:}\ high] \ , \end{aligned}$$

the original version,

$$\begin{aligned} screen'_{1.1} &= screen_{1.1} \sqcap [\delta_2\uparrow] \\ &= [\delta_1 : \top, \delta_2\uparrow, speed{:}\ \{medium, high\}, \\ &\quad user{:}\ sanders] \ , \end{aligned}$$

Sander's version, and

$$\begin{aligned} screen'_{1.2} &= screen_{1.0} \sqcap [\delta_2 : \top] \\ &= [\delta_1\uparrow, \delta_2 : \top, speed{:}\ high] \ , \end{aligned}$$

the version changed by Johnson. This somewhat complex set is shown in figure 3. Again, R denotes the currency:

$$\begin{aligned} R = \{&\sim[user{:}\ sanders, current{:}\ \top], \delta_1 : \top\} \\ &\sqcap \{[user{:}\ sanders, current{:}\ \top], \delta_2 : \top\} \end{aligned}$$

Sander's original configuration $[\delta_1\uparrow]$ is now no more the current one. This is detected as Sanders wishes to commit his change δ_1. Since the new current revision should not discard Johnson's changes, Sanders must create an *integrated* revision $[\delta_1 : \top, \delta_2 : \top]$, incorporating both change sets.

Unfortunately, besides changing the $screen_1$ component, Johnson has replaced the $drive_2$ component by a cheaper one, changing its features from [*speed: medium*] to [*speed: low*]. For Johnson, this perfectly makes sense, since in her workspace, the possible configurations stay the same. When Sanders wishes to commit his changes to the environment, this will result in an inconsistency: his new $screen_1$ component can not be combined with Johnson's new $drive_2$ component. As the resulting current version set would be inconsistent ($screen_1 \boxplus_I drive_2 = \bot$ due to the differing *speed* features), the CM system must request Sanders to resolve the conflict with Johnson before committing his changes.

In real life, such a context-sensitive inconsistency may well have gone by unnoticed, resulting in late and expensive conflict resolution. In our example, let us assume Johnson takes back her change to $drive_2$. Sanders may now commit the integrated $screen_1$ changes, resulting in the final version set:

$$\begin{aligned} screen''_1 = [&object{:}\ screen, resolution{:}\ high, \\ &speed{:}\ \{medium, high\}] \\ &\sqcap \{current\uparrow, [\delta_1 : \top, \delta_2 : \top]\} \end{aligned}$$

With this final change, there are no more differences between Sanders' workspace and the environment; any combinations of Sanders' and Johnson's changes are possible, and $[\delta_1 : \top, \delta_2 : \top]$ is the new current version, allowing arbitrary combinations between the *screen* and *drive* versions.

Although the scenario may appear simplified, the combination of revision control, variant control, concurrency control, and consistency checking is a major challenge for existing CM systems. Traditionally, every issue is addressed by a separate mechanism, following a hierarchical layer model—for instance, first a workspace, then a variant, and then a revision is selected. Only through version sets can we provide a uniform denotation with the ability to switch and combine concepts at need.

10 Complexity

Since its core can be reduced to the satisfiability problem, feature unification (that is, deciding if $T \sqcap S = \bot$) is NP-complete. This results in general exponential time complexity for all the CM tasks described herein. However, we have identified and implemented some *deductive shortcuts*. The most important shortcut is the observation that $T \sqcap S = T' \sqcap S$, if $T = T' \sqcap T''$ such that $T'' \sqsupseteq S$ holds (similarly, $T \sqcup S = T' \sqcup S$, if $T = T' \sqcup T''$ such that $S \sqsupseteq T''$); hence, T can be simplified to T' for unification purposes. These simplifications are similar in spirit to *partial evaluation* and are particularly effective in three cases:

- Selecting a single version (or a small version set) if the selection term S is a simple intersection of feature terms;
- Operating on version sets with a given subsumption lattice (especially revision repositories), since terms like $T \sqcup T'$ can be simplified to T (likewise, $T \sqcap T'$ to T') if we know that $T \sqsupseteq T'$;

- Checking the consistency of a version set with respect to a selection rule S if all component versions are identified by intersections of simple feature terms.

Under these conditions, the feature unification problem $T \sqcap S = \bot$ effectively becomes partial evaluation of T with respect to a selection term S, resulting in linear time behavior. Are these conditions too restrictive? We do not think so, since they are also imposed by the discussed CM concepts, and their interaction and combination is not endangered.

One crucial point for complexity remains. For checking the global consistency, feature unification is indispensable. Hence, the time grows exponentially with the number of possible combinations. This problem can be addressed by *incremental configuration*. As configuration consistency is a global property and thus holds for arbitrary version sets, a system can be configured using a bottom-up approach, following the system structure. First, all components for each subsystem are configured, ensuring consistency within the subsystem. If the assembly of the configured subsystems results in an inconsistency, initial choices must be revoked.[4]

11 Representing Version Sets

While complexity can be kept at a low level, the question of an appropriate and efficient version set representation arises. We have realized the version set model in a CM tool, named *Incremental Configuration Engine* (ICE) [42]. ICE uses individual files containing C preprocessor (CPP) directives to represent version sets; each text piece belonging to a specific version set T is governed by a CPP expression equivalent to T.

Using CPP files as representation, selection and union operations on version sets result in a time/space behavior comparable to the "embedded delta" method used by SCCS [29]. While in "reverse delta" tools like RCS [37], selection takes longer the older the selected revision is, ICE shows a constant selection time for arbitrary version sets (usually realized through partial evaluation). Adding version sets to a repository also shows SCCS-like time behavior as the entire version sets are passed through a *diff* algorithm [23] to determine common text pieces across versions. Hence, it is possible to represent version sets without performance penalties larger than those imposed by other SCM systems. Note, however, that the version set model is independent from a specific representation, and that other technologies may be more appropriate and efficient for large-scale purposes.

Right now, ICE is no more than a simulation shell providing creation, deletion and read/write access on version sets. For end users, we are currently extending ICE to an entire virtual file system. This system, more a configuration "environment" than an "engine", will provide arbitrary version set access and realize interactive and incremental configuration by simply integrating version set specifications into file and path names.

12 Conclusion

In [6], Feiler closes with requirements on a unified configuration management model:

> The appropriateness of the concepts as a basis for a unified model has to be validated. The interaction between the concepts has to be understood, and the ability to combine and adapt concepts to process needs has to be possible.

We have shown how the concepts of four central CM models—the checkin/checkout model, the change set model, the composition model, and the long transaction model—are encompassed by the version set model, making it a unified basis for modeling, realizing and integrating CM tasks. We also have identified some conditions under which efficient realization is possible, using the sample ICE implementation.

Still, much work needs to be done. Due to its flexibility, the version set model leaves much room for designing and exploring new CM models, which must be evaluated. There is no true methodology yet how components and versions should be attributed with feature terms; experiences from other attribute-oriented CM systems [5, 10, 24] or faceted classification [26] might help here. Also, one must set up schemes to determine component features from existent software.

Only a part of the entire CM spectrum is yet sufficiently modeled and formalized. *Process management* and *configuration testing* issues are left unaddressed; we think that these should form separate layers on top of the version set model. *Manufacturing* (i.e. software builds) can be described through operations on version sets, building derived objects whose features are inherited from their sources and construction tools, using properties from SHAPE [22] and CAPITL [1, 28], for example. *Build dependencies* must be covered by an additional system model describing the system by defining relations between version sets, in a manner similar to Katz [13], but using feature logic as a formal base. Eventually, such process, testing, and system models could describe the entire CM process through operations on version sets and their features and result in a fully unified CM model.

[4] Such incremental approaches have been discussed extensively for *context relations*, a calculus similar to feature logic; as Snelting shows in [33], reevaluation considering the topological order of dependencies results in dramatically improved response time. A much simpler backtracking scheme is realized in ADELE [5], where one single "best-fitting" variant is automatically selected during program construction.

ICE is part of the inference-based software development environment NORA[5]. NORA aims at utilizing inference technology in software tools; concepts and preliminary results can be found in [9, 17, 21, 34, 35].

ICE and related technical reports are available from ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/. Users with access to the Web can look at the ICE WWW page, http://www.cs.tu-bs.de/softech/software/ice_e.html, for simple access to ICE and all related information.

Acknowledgements. Many thanks to Bernd Fischer, Petra Funk, Franz-Josef Grosch, Christian Lindig, Axel Mahler, and Gregor Snelting as well as to the anonymous referees for their valuable comments and suggestions on earlier versions of this paper.

The CD-ROM scenario was inspired by Michael Crichton's novel *Disclosure*.

References

[1] ADAMS, P., AND SOLOMON, M. An overview of the CAPITL software development environment. In Feldman [8], pp. 3–28.

[2] BERLINER, B. CVS II: Parallelizing software development. In *Proc. of the 1990 Winter USENIX Conference* (Washington, D.C., 1990).

[3] BRACHMAN, R. J., AND LEVESQUE, H. J. The tractability of subsumption in frame-based description languages. In *Proc. of the 4th National Conference of the American Association for Artificial Intelligence* (Austin, Texas, Aug. 1984), pp. 34–37.

[4] COURINGTON, W. The Network Software Environment. Tech. Rep. FE 197-0, Sun Microsystems, Inc., Feb. 1989.

[5] ESTUBLIER, J., AND CASALLAS, R. The Adele configuration manager. In Tichy [38], pp. 99–133.

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

[7] FEILER, P. H., Ed. *Proc. 3rd International Workshop on Software Configuration Management* (June 1991), ACM Press.

[8] FELDMAN, S., Ed. *Proc. 4th International Workshop on Software Configuration Management (Preprint)* (May 1993).

[5]NORA is a figure in Henrik Ibsen's play "A Dollhouse". Hence, NORA is NO Real Acronym.

[9] FISCHER, B., KIEVERNAGEL, M., AND SNELTING, G. Deduction-based software component retrieval. In Köhler et al. [16]. To appear.

[10] GULLA, B., KARLSSON, E.-A., AND YEH, D. Change-oriented version descriptions in EPOS. *Software Engineering Journal 6*, 6 (Nov. 1991), 378–386.

[11] HARTER, R. Version management and change control; systematic approaches to keeping track of source code and support files. *Unix World 6*, 6 (June 1989).

[12] KASPER, R. T., AND ROUNDS, W. C. A logical semantics for feature structures. In *Proc. of the 24th Annual Meeting of the ACL* (Columbia University, New York, 1986), pp. 257–265.

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

[14] KAY, M. Functional unification grammar: A formalism for machine translation. In *Proc. 10th International Joint Conference on Artificial Intelligence* (1984), pp. 75–78.

[15] KOBIALKA, H.-U., AND MEYKE, C. Configurations are versions, too. In Feldman [8], pp. 156–164.

[16] KÖHLER, J., GIUNCHIGLIA, F., GREEN, C., AND WALTHER, C., Eds. *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs* (Aug. 1995). To appear.

[17] KRONE, M., AND SNELTING, G. On the inference of configuration structures from source code. In *Proc. 16th International Conference on Software Engineering* (May 1994), IEEE Computer Society Press, pp. 49–57.

[18] LAMPEN, A., AND MAHLER, A. An object base for attributed software objects. In *Proc. of the Fall '88 EUUG Conference* (Oct. 1988), pp. 95–105.

[19] LEBLANG, D. B. The CM challenge: Configuration management that works. In Tichy [38], pp. 1–37.

[20] LIE, A., DIDRIKSEN, T. M., CONRADI, R., KARLSSON, E.-A., HALLSTEINSEN, S. O., AND HOLAGER, P. Change oriented versioning. In *Proc. 2nd European Software Engineering Conference* (Sept. 1989), C. Ghezzi and J. A. McDermid, Eds., vol. 387 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–202.

[21] LINDIG, C. Concept-based component retrieval. In Köhler et al. [16], pp. 21–25. To appear.

[22] MAHLER, A. Variants: Keeping things together and telling them apart. In Tichy [38], pp. 39–69.

[23] MILLER, W., AND MYERS, E. A file comparison program. *Software—Practice and Experience 15*, 11 (1985), 1025.

[24] MUNCH, B. P., LARSEN, J.-O., GULLA, B., CONRADI, R., AND KARLSSON, E. A. Uniform versioning: The change-oriented model. In Feldman [8], pp. 188–196.

[25] NEBEL, B., AND SMOLKA, G. Representation and reasoning with attributive descriptions. In *Sorts and Types in Artificial Intelligence* (Apr. 1989), K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, Eds., vol. 256 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 112–139.

[26] PRIETO-DIAZ, R. Classifying software for reusability. *IEEE Software 4*, 1 (Jan. 1987).

[27] REICHENBERGER, C. Orthogonal version management. In *Proc. 2nd International Workshop on Software Configuration Management* (Oct. 1989), ACM Press, pp. 137–140.

[28] RICH, A., AND SOLOMON, M. A logic-based approach to system modelling. In Feiler [7], pp. 84–93.

[29] ROCHKIND, M. J. The source code control system. *IEEE Transactions on Software Engineering SE-1*, 4 (Dec. 1975), 364–370.

[30] SHIEBER, S., USZKORZEIT, H., PEREIRA, F., ROBINSON, J., AND TYSON, M. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, J. Bresnan, Ed. SRI International, 1983.

[31] SMOLKA, G. Feature-constrained logics for unification grammars. *Journal of Logic Programming 12* (1992), 51–87.

[32] SMOLKA, G., AND AÏT-KACI, H. Inheritance hierarchies: Semantics and unification. In *Unification*, C. Kirchner, Ed. Academic Press, London, 1990, pp. 489–516.

[33] SNELTING, G. The calculus of context relations. *Acta Informatica 28* (May 1991), 411–445.

[34] SNELTING, G., FISCHER, B., GROSCH, F.-J., KIEVERNAGEL, M., AND ZELLER, A. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik—Forschung und Entwicklung 9*, 3 (Aug. 1994), 116–131. In German.

[35] SNELTING, G., GROSCH, F.-J., AND SCHROEDER, U. Inference-based support for programming in the large. In *Proc. 3rd European Software Engineering Conference* (Oct. 1991), A. van Lamsweerde and A. Fugetta, Eds., vol. 550 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 396–408.

[36] TICHY, W. F. A data model for programming support environments and its application. In *Automated Tools for Information Systems Design*, H.-J. Schneider and A. I. Wasserman, Eds. North-Holland Publishing Company, 1982, pp. 31–48.

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

[38] TICHY, W. F., Ed. *Configuration Management*, vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, England, 1994.

[39] WESTFECHTEL, B. Structure-oriented merging of revisions of software documents. In Feiler [7], pp. 86–79.

[40] WINKLER, J. F. H. Version control in families of large programs. In *Proc. 9th International Conference on Software Engineering* (Mar. 1987), E. Riddle, Ed., IEEE Computer Society Press, pp. 91–105.

[41] YANG, W., HORWITZ, S., AND REPS, T. A program integration algorithm that accommodates semantics-preserving transformations. *ACM Transactions on Software Engineering and Methodology 1*, 3 (July 1992), 310–354.

[42] ZELLER, A., AND SNELTING, G. Handling version sets through feature logic. In *Proc. 5th European Software Engineering Conference* (Sept. 1995), W. Schäfer, Ed., Lecture Notes in Computer Science, Springer-Verlag. To appear.