Unified Versioning through Feature Logic

Andreas Zeller and Gregor Snelting

Informatik-Bericht No. 96-01
Überarbeitete Fassung, 15. Februar 1997

To appear in ACM Transactions on Software Engineering and Methodology 6(3), July 1997

Unified Versioning through Feature Logic

Andreas Zeller and Gregor Snelting*
Technische Universität Braunschweig, Germany

Abstract

Software Configuration Management (SCM) suffers from tight coupling between SCM versioning models and the imposed SCM processes. In order to adapt SCM tools to SCM processes, rather than vice versa, we propose a unified versioning model, the *version set model.* Version sets denote versions, components, and configurations by *feature terms,* that is, boolean terms over (*feature*: *value*)-attributions. Through *feature logic,* we deduce consistency of abstract configurations as well as features of derived components and describe how features propagate in the SCM process; using *feature implications,* we integrate change-oriented and version-oriented SCM models. We have implemented the version set model in a SCM system called ICE for *Incremental Configuration Environment.* ICE is based on a *featured file system* (FFS), where version sets are accessed as virtual files and directories. Using the well-known C Preprocessor representation, users can view and edit multiple versions simultaneously, while still only the differences between versions are stored. It turns out that all major SCM models can be realized and integrated efficiently on top of the FFS, demonstrating the flexible and unifying nature of the version set model.

Categories and Subject Descriptors: D.2.6 [**Software Engineering**] Programming environments; D.2.7 [**Software Engineering**] Distribution and Maintenance—*version control;* D.2.9 [**Software Engineering**] Management—*software configuration management; programming teams;* D.4.3 [**Operating Systems**] File Systems Management; I.2.3 [**Artificial Intelligence**] Deduction and theorem proving; I.2.4 [**Artificial Intelligence**] Knowledge representation formalisms and methods

General Terms: Management, Theory, Standardization

Additional Key Words and Phrases: Feature logic, Version sets

1 Introduction

Software Configuration Management, or SCM for short, is the discipline for controlling the evolution of software systems. SCM encompasses general configuration management procedures [21, 22] like *identification* of components and structures, *control* of changes and releases, *status accounting,* or *audit and review,* as well as software-specific tasks [12] like *manufacture, process management,* and *team work.* SCM is one of the basic prerequisites for process improvement, stipulated by the ISO 9000 standard or the SEI capability maturity model, and thus attracts more and more attention from professional software development.

*This article is a revised and extended version of a paper [64] presented at the Fifth European Software Engineering Conference, Sitges, September 1995. Early descriptions of the revision and workspace concepts (sections 4.1 and 4.3) were presented in [60]. The featured file system (sections 5.2 and 5.3) was first discussed in [61].
This work was supported by the Deutsche Forschungsgemeinschaft, grants Sn11/1-2 and Sn11/2-2.
Authors' address: Technische Universität Braunschweig, Abteilung Softwaretechnologie, Bültenweg 88, D-38092 Braunschweig, Germany; email: {zeller, snelting}@ips.cs.tu-bs.de; http://www.cs.tu-bs.de/softech/.

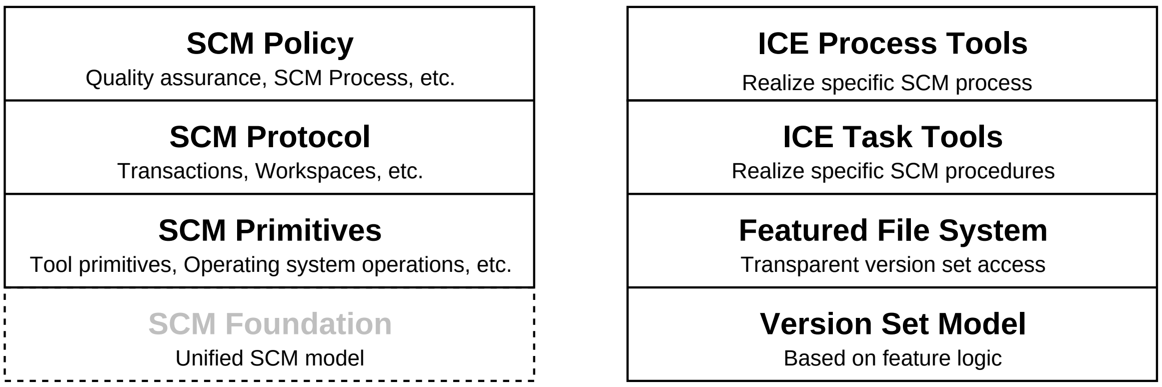

SCM Policy
Quality assurance, SCM Process, etc.
SCM Protocol
Transactions, Workspaces, etc.
SCM Primitives
Tool primitives, Operating system operations, etc.
SCM Foundation
Unified SCM model
ICE Process Tools
Realize specific SCM process
ICE Task Tools
Realize specific SCM procedures
Featured File System
Transparent version set access
Version Set Model
Based on feature logic

Figure 1: A federated SCM architecture, as proposed in [8] (left) and as realized in ICE (right)

As all configuration items are accessible on-line, SCM is typically supported and enforced by automated SCM tools and systems. The early days of SCM were characterized by dedicated SCM tools like SCCS [43] or RCS [55] (revision and change control); CPP, the C preprocessor [23] (variant control); or MAKE [17] (manufacture). These days, a new generation has emerged, represented by SCM systems like ADELE [15], EPOS [19], or CLEARCASE [31]. These systems provide and integrate support for all SCM aspects through *federated* SCM system architectures [8], as illustrated in figure 1: a *primitive layer* provides basic versioning and access capabilities, a *protocol layer* realizes SCM tasks and procedures, and a *policy layer* implements organization-specific standards.

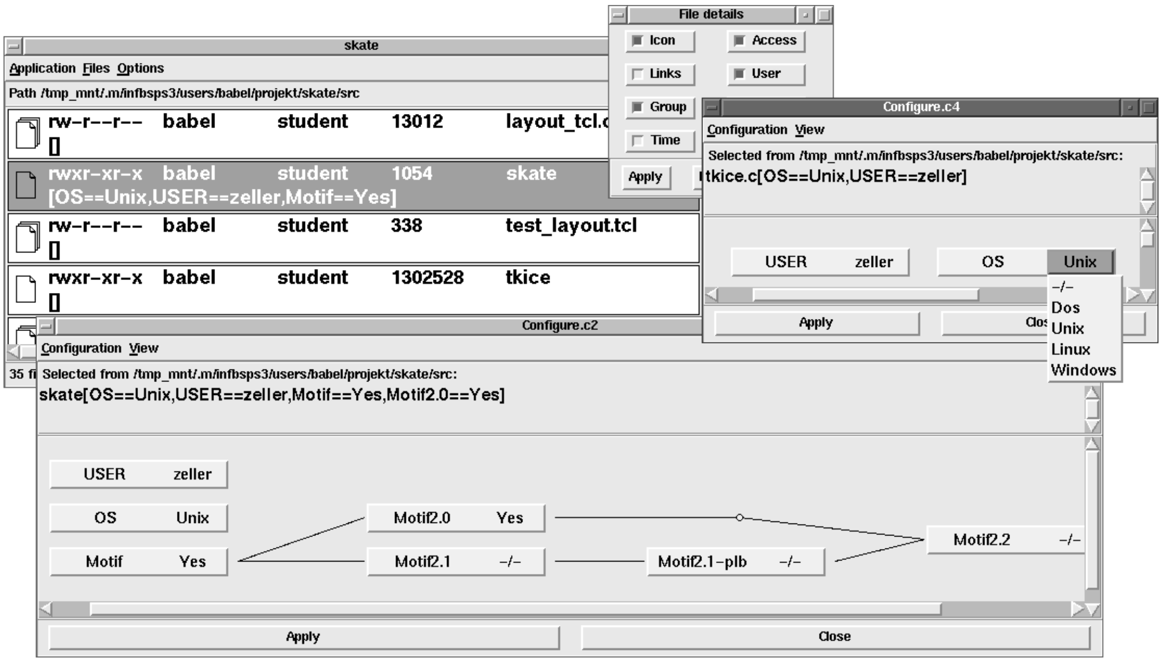
Today, several SCM vendors compete with each other by means of an ever-growing number of product features. This has the benefit that users can choose between a large number of SCM systems, each with an individual set of features [10]. Despite these advances, SCM systems still suffer from three deficiencies:

Lack of ambiguity tolerance. SCM systems generally provide poor support for treating several items at once. This includes lack of support for manipulating and identifying permanent variants [33], change propagation across several versions at once [36], or consistency checking in abstract (ambiguous) configurations [47].

Lack of process flexibility. SCM systems are frequently used to enforce a specific software process. Unfortunately, nearly every SCM system relies on its own predefined and inflexible product life cycle [14]; at least four diverging SCM models have been identified, each imposing a different SCM process [16]. This is pretty far away from the ideal that a SCM system should adapt to an organization's process.

Lack of system integration. Already at the SCM primitive layer, there is considerable disagreement about versioning models [9]. Consequently, the SCM layers are not interchangeable, resulting in SCM systems that neither interoperate nor integrate. Furthermore, the basic layers constrain higher layers: flexibility decreases the higher the layer considered [56].

In this paper, we propose to resolve these deficiencies through a unified SCM versioning model as common SCM foundation. Our *version set model* integrates the common SCM models, increases flexibility at the protocol and policy layers, and tolerates ambiguity at all levels. 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



```
Path /tmp_mnt/.m/infbsps3/users/babel/projekt/skate/src
rw-r--r--   babel   student   13012     layout_tcl.c
[]
rwxr-xr-x   babel   student   1054      skate
[OS==Unix,USER==zeller,Motif==Yes]
rw-r--r--   babel   student   338       test_layout.tcl
[]
rwxr-xr-x   babel   student   1302528   tkice
[]
35 fi
```

Selected from /tmp_mnt/.m/infbsps3/users/babel/projekt/skate/src:
tkice.c[OS==Unix,USER==zeller]

USER zeller — OS Unix

Selected from /tmp_mnt/.m/infbsps3/users/babel/projekt/skate/src:
skate[OS==Unix,USER==zeller,Motif==Yes,Motif2.0==Yes]

USER zeller — OS Unix — Motif Yes — Motif2.0 Yes — Motif2.1 -/- — Motif2.1-plb -/- — Motif2.2 -/-

Figure 2: Exploring the configuration space with the ICE file/configuration browser

SCM convention to characterize objects by their attributes. Version sets generalize well-known SCM concepts such as components, repositories, workspaces, variant sets, or revision histories. Using *feature logic,* intersection, union, and complement operations on version sets are realized in order to express and generalize the semantics of SCM models. 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.

We have implemented the version set model in a SCM system, called ICE for *Incremental Configuration Environment.* ICE integrates within software development environments through its *featured file system* (FFS), where version sets are represented as files and directories. Arbitrary programs can access version sets and realize version operations through file manipulations. Through specialized configuration browsers, as shown in figure 2, users can incrementally explore the configuration space and have ICE deduce consistency even for incomplete configurations. Using the well-known CPP representation, users can view and edit multiple versions simultaneously, while still only the differences between version sets are stored. All four major SCM models can be realized and integrated on top of the FFS, demonstrating the unifying nature of the version set model.

This paper is organized like the federated SCM architecture shown in figure 1. We begin at the lowest SCM layer by motivating and presenting feature logic as a formal SCM foundation. Section 3 introduces the version set model and shows how primitive SCM concepts are modeled through version sets. In section 4, we discuss the modeling of advanced SCM concepts such as change implications and workspaces, required for the SCM protocol layer. In section 5, we turn to practical aspects and demonstrate how the FFS realizes the SCM primitive layer through transparent version set access. In section 6, we treat the SCM protocol layer and demonstrate the realization of SCM protocols on top of the FFS. Section 7 discusses performance and complexity issues, treating the integration of SCM protocols. We close with a summary and suggestions for future work in section 8.

2 Feature Logic

Most of the existing SCM literature is product-oriented, describing and evaluating a set of SCM concepts as realized in some specific implementation. We think that this view hinders a deeper understanding of SCM concepts, as the concept in question cannot be separated from its implementation. In order to support a large variety of SCM versioning concepts, we must thus abstract from specific SCM products and turn towards a more fundamental treatment—still keeping the higher SCM layers in mind.

2.1 A SCM Foundation

The formal foundation we have chosen for capturing SCM versioning concepts is called *feature logic.* Feature logic denotes sets of objects by their properties and provides elemental set operations to manipulate these sets. In our SCM domain, we use feature logic to denote sets of components by their features and to describe the semantics of SCM operations.

So, why did we use feature logic as a formal foundation? Relying on the three SCM deficiencies as stated in the introduction, we identified three key elements of such a foundation.

First foundation: Sets. Ambiguity tolerance imposes the necessity to treat sets of versions and configurations as first-class objects. SCM procedures thus should be set-oriented rather than item-oriented, as manipulating sets generalizes manipulating items. For instance, editing a set of versions or checking a set of configurations for consistency subsumes editing a single version or examining a single configuration.

Second foundation: Attributes. Attribution is one of the few techniques common to the whole SCM area: all known SCM models rely on that either versions or changes be tagged with attributes. Identification and selection schemes should be attribute-based; attribution support includes a description of how attributes propagate in the SCM process, such that composed and derived objects can be identified.

Third foundation: Unification. The usual selection process in SCM systems consists of determining the objects whose attributes are consistent with those of a specific environment. Typically, objects are described by a conjunction of attribute values and the environment by an attribute expression; but the inverse scheme is also found, as in CPP. In order to encompass both schemes, selection and identification should both rely on attribute expressions, *unifying* attribute expressions instead of matching attribute expressions against a conjunction of attribute values.

There are several formalisms that denote sets of objects by their attributes, subsumed under the term *description logics* or *terminological logics.* Their most important domains are the areas of knowledge representation, where *concept descriptions,* also called *frames* [7, 37, 38], are used to represent sets of objects by attribute/value combinations, and the semantic analysis of natural language [25, 27, 49].

In programming languages, attribute/value combinations are used in record structures. Aït-Kaci was the first to study such structures mathematically, calling them *ψ-terms* [2]. The resulting *ψ-term calculus* is the formal foundation of the programming languages LOGIN [3] and LIFE [4], which are similar to PROLOG, but use *feature unification* [51] instead of syntactic unification. In contrast to several description logics, attributes in ψ-terms are *functional*: they can have only one value. This is convenient, since objects can be identified by some unique attribute value.

ψ-terms have been successfully applied in the context of SCM, notably in the CAPITL system [1]. CAPITL uses a variant of LOGIN, called CONGRESS, to denote the attributes of components and tools and to describe how these attributes propagate from source components to derived components. As CAPITL is also among the most advanced and well-founded SCM systems in terms of building and attributing derived components, descriptions like ψ-terms seem ideal candidates for a unified SCM versioning model—the more as they have been successfully used in SCM systems. Unfortunately, in ψ-terms, only *conjunctions* of attribute/value combinations are allowed; negations or disjunctions are not supported. This restriction would severely constrain SCM identification and selection schemes.

There is an alternative candidate for a SCM foundation that does not suffer from these restrictions. Boolean operators from *first-order logic* are used in several SCM selection schemes [39, 15, 19, 58, 33]; first-order terms may also be used for identification purposes, using deduction techniques such as *boolean unification* [34] to match identification and selection terms. The problem with first-order logic is that it is far too general; it lacks the central property of being attribute-oriented. This implies that all SCM functionality like selection through attributes, attribute propagation, or inheritance of abstract configurations requires explicit formalization using first-order axioms and rules.

For a formal SCM foundation, we need the best of three worlds: the boolean operators and quantifications of first-order logic, in order to express identification and selection schemes, the attribute-oriented formalisms from description logics, denoting how attributes propagate in the SCM process, and the functional attributes of ψ-terms, as they uniquely identify objects by their attributes. Such a logic does exist: *Feature logic,* as defined by Smolka [50], is a well-founded description logic that includes quantification, disjunction, and negation over functional attribution terms, forming a full boolean algebra.

2.2 Feature Logic in a Nutshell

We begin with an informal overview of feature logic. A *feature term* denotes a set of objects characterized by certain features. A *feature* is a functional property or attribute of abstract objects. 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 include literals, variables, and (nested) feature terms.

As an example, consider the following feature term T, which expresses the linguistic properties of a natural language fragment:

$$T = \begin{bmatrix} \mathit{tense}\colon \mathit{present}, \\ \mathit{predicate}\colon [\mathit{verb}\colon \mathit{sing}, \mathit{agent}\colon x, \mathit{what}\colon y]\,, \\ \mathit{subject}\colon [x, \mathit{num}\colon \mathit{singular}, \mathit{person}\colon \mathit{third}]\,, \\ \mathit{object}\colon y \end{bmatrix}$$

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, and so on: T denotes the sentence template "x sings y".

The syntax of feature terms is summarized in table 1, where we denote *variables* by x, y, z; *features* by f, g, h; *constants* by a, b, c; and feature terms denoted by S and T.[1] Feature terms are constructed using the well-known boolean set operations *intersection, union,* and *complement.*

[1]Smolka [50] writes $\sim S$ as $\neg S$, $S = T$ as $S \sim T$, and $S \sqsubseteq T$ as $S \preccurlyeq T$. Implications and equivalences do not occur in [50]; they are simple syntactical extensions whose equivalence to simpler operators is shown in proposition 1.

Notation	Name	Interpretation
$\top$ (also $[]$)	Top	Universe
$\bot$ (also $\{\}$)	Bottom	Empty set; Inconsistency
a	Atom	Singleton set containing a
x	Variable	
$f{:}S$	Selection	The value of f is in 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
$\sim S$	Complement	S does not hold
$S \sqcap T$ (also $[S, T]$)	Intersection	Both S and T hold
$S \sqcup T$ (also $\{S, T\}$)	Union	S or T holds
$S \rightarrow T$	Implication	If S holds, then T holds
$S \leftrightarrow T$	Equivalence	S holds if and only if T holds
$\exists x(S)$	Quantification	There is an x such that S holds

Table 1: The syntax of feature terms

Each of these set operations may also be interpreted as logical constraint on the object features, representing the set of objects satisfying this constraint. For instance, let $S = [f{:}a]$, the set of all objects whose feature f has the value a, and $T = [g{:}b]$, the set of all objects whose feature g has the value b. Then, $S \sqcap T = [f{:}a, g{:}b]$ is the intersection of $[f{:}a]$ and $[g{:}b]$, namely the set of objects whose feature f is a *and* whose feature g is b. Similarly, $S \sqcup T = \{f{:}a, g{:}b\}$ is the union of $[f{:}a]$ and $[g{:}b]$—that is, the set of objects whose feature f is a *or* whose feature g is b. As feature terms form a boolean algebra, all boolean transformations like distribution, de Morgan's law etc. hold for feature terms as well.

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 $\sim[\mathit{compiler}{:}\mathit{gcc}]$ denotes all objects whose feature *compiler* is either undefined or has another value than *gcc*. The term $[\mathit{compiler}{:}\sim\mathit{gcc}]$ 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 (variable-free) 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. Feature terms thus always allow for further specialization, like classes in object-oriented models. For instance, $\top \sqsupseteq [\mathit{fruit}{:}x] \sqsupseteq [\mathit{fruit}{:}\mathit{apple}] \sqsupseteq [\mathit{fruit}{:}\mathit{apple}, \mathit{color}{:}\mathit{green}] \sqsupseteq [\mathit{fruit}{:}\mathit{apple}, \mathit{color}{:}\mathit{green}, \mathit{wormy}{:}\mathit{no}]$, and so on.

Atoms like *apple*, *green*, or *gcc* denote singleton sets containing some unique object without any features; the equivalences $a \sqcap b = \bot$ and $a \sqcap f{:}\top = \bot$ hold for all atoms a, b and for any feature f. This leads to a simple *consistency notion*: As feature logic assumes that each feature can have only one value, the term $[\mathit{os}{:}\mathit{dos}, \mathit{os}{:}\mathit{unix}]$ is equivalent to $\bot$, the empty set; formally, $[\mathit{os}{:}\mathit{dos}, \mathit{os}{:}\mathit{unix}] = [\mathit{os}{:}[\mathit{dos}, \mathit{unix}]] = [\mathit{os}{:}\bot] = \bot$ holds. Terms which are equivalent to $\bot$ are called *inconsistent.* Through *feature unification* [50], a constraint-solving technique, one can determine consistence 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).

2.3 Properties of Feature Terms

We now give some properties of feature terms. Two feature terms S and T are called *equivalent* (written $S =^{\mathcal{F}} T$ or $S = T$ where unambiguous) if they denote the same set of objects for every interpretation.[2] Using equivalence, most of the introduced feature term forms are redundant and may be reduced to six primitive forms.

Proposition 1 *Every feature term can be rewritten in linear time to an equivalent feature term containing only the forms a, x, $f{:}\,S$, $S \sqcap T$, $\sim S$, and $\exists x(S)$ by using the following equivalences* [50]*:*

$$\begin{array}{rlrl} f\uparrow &= \sim(f{:}\top) & \bot &= x \sqcap \sim x \\ f\downarrow g &= \exists x(f{:}x \sqcap g{:}x) & \top &= \sim\bot \\ f\uparrow g &= \exists x(f{:}x \sqcap g{:}\sim x) & S \sqcup T &= \sim(\sim S \sqcap \sim T) \\ S \rightarrow T &= \sim(S \sqcap \sim T) & S \leftrightarrow T &= \sim(S \sqcap \sim T) \sqcap \sim(T \sqcap \sim S) \end{array}$$

A feature term is called *closed* if it has no free variables. A feature term is *ground* if it has no variables, agreements, or disagreements. A feature term is *quantifier-free* if it contains no quantifications $\exists x(S)$. A feature term is *basic* if it is quantifier-free, contains no implications, and contains only complements of the from $\sim a$ or $\sim x$. A feature term is *simple* if it is basic and contains no unions. A feature term is in *disjunctive normal form* (DNF) if it has the form $S_1 \sqcup \cdots \sqcup S_n$, where all $S_1, \ldots, S_n$ are simple feature terms. Two feature terms are called *orthogonal* if have no common features or variables.

Proposition 2 *Every quantifier-free feature term can be rewritten in linear time to an equivalent basic feature term by using the following equivalences* [50]*:*

$$\begin{array}{rlrl} \sim f{:}\,S &= f\uparrow \sqcup f{:}\sim S & \sim\bot &= \top \\ \sim f\uparrow &= f{:}\top & \sim\top &= \bot \\ \sim f\uparrow g &= f\uparrow \sqcup g\uparrow \sqcup f\downarrow g & \sim(S \sqcap T) &= \sim S \sqcup \sim T \\ \sim f\downarrow g &= f\uparrow \sqcup g\uparrow \sqcup f\uparrow g & \sim(S \sqcup T) &= \sim S \sqcap \sim T \\ \sim\sim S &= S & S \rightarrow T &= \sim S \sqcup T \\ & & S \leftrightarrow T &= (\sim S \sqcup T) \sqcap (\sim T \sqcup S) \end{array}$$

A feature term S is said to be included or *subsumed* by a feature term T (written $S \sqsubseteq T$ or $T \sqsupseteq S$) if the set denoted by S is a subset of the set denoted by T under every possible interpretation.

Proposition 3 *Let $\mathcal{F}$ be the set of feature terms, as defined above. Then $(\mathcal{F}, \sqcup, \sqcap, \sim, \bot, \top)/{=^{\mathcal{F}}}$ is a* boolean algebra. *$\mathcal{F}$ and subsumption constitute a* subsumption lattice $(\mathcal{F}, \sqsubseteq)/{=^{\mathcal{F}}}$ *with a supremum of $S \sqcup T$ and an infimum of $S \sqcap T$ for all $S, T \in \mathcal{F}$.*

PROOF. As follows from definitions, all properties required for boolean algebras (commutativity, associativity, idempotency, absorption, distribution, etc.) apply under the equivalence $=^{\mathcal{F}}$. $(\mathcal{F}, \sqsubseteq)/{=^{\mathcal{F}}}$ being a subsumption lattice follows from $(\mathcal{F}, \sqcup, \sqcap, \sim, \bot, \top)/{=^{\mathcal{F}}}$ being a boolean algebra [62]. □

[2]The interpretation of feature terms is formally defined in [50].

2.4 Consistency

We now discuss the notion of *consistency,* stating whether feature terms denote empty sets, and devise algorithms that decide consistency. A feature term S is called coherent or *consistent* if there is an interpretation such that the denoted set is non-empty. A feature term is called incoherent or *inconsistent* if it is not consistent.

Proposition 4 *Consistency, subsumption, and equivalence of feature terms are linear-time reducible to each other* [50]*:*

$$\begin{aligned} S \text{ inconsistent} &\Leftrightarrow S \sqsubseteq \bot \Leftrightarrow S = \bot \\ S \sqsubseteq T &\Leftrightarrow S \sqcap {\sim}T \text{ inconsistent} \\ S = T &\Leftrightarrow S \sqsubseteq T \wedge T \sqsubseteq S \end{aligned}$$

Proposition 5 *Deciding inconsistency, subsumption, and equivalence of quantifier-free feature terms are co-NP-complete problems* [50].
PROOF. Follows from the satisfiability problem of propositional logic, as shown in [50]. □

For quantifier-free feature terms, Smolka has devised an algorithm that decides the inconsistency of arbitrary quantifier-free feature terms. The basic idea behind this so-called *feature unification* is that the feature term S is transformed into DNF $S = S_1 \sqcup S_2 \sqcup \cdots \sqcup S_n$; consistency of each conjunct S_i can then be determined using a quadratic-time algorithm.

Proposition 6 *Deciding inconsistency of simple feature terms is of quadratic time complexity* [50].

As transformation of non-simple feature terms to DNF is NP-complete, time complexity of Smolka's algorithm is exponential in the worst case, complying with proposition 5. It is thus unsuitable for practical problems as soon as the feature terms exceed a certain size.

By imposing certain conditions upon feature terms, time complexity of feature unification can be dramatically reduced. In proposition 6, we have already seen that deciding consistency of a simple feature term can be decided in quadratic time. The unification problem can be broken down even more for terms of the form $S \sqcap T$. First, if S and T are orthogonal, $S \sqcap T$ is consistent iff S and T are consistent.

Proposition 7 *Let S and T be orthogonal. Then, $S \sqcap T = \bot \Leftrightarrow S = \bot \vee T = \bot$ holds.*
PROOF. Via algebraic induction over S and T; there can be no intersection of primitives that would lead to inconsistency [62]. □

Another efficient algorithm is obtained using principles of *partial evaluation* [24]. We observe that the unification problem $S \sqcap T = \bot$ is much simplified if T is a simple feature term of the form $T = T_1 \sqcap T_2 \sqcap \cdots \sqcap T_n$: for each primitive T_i, we can check whether $S \sqcap T_i = \bot$ in linear time by (syntactically) comparing T_i with the primitives from S and thus deduce inconsistency. This proposition holds only if S and T are variable-free.

Proposition 8 *Let S and T be consistent and variable-free feature terms; let T also be simple and S be in basic form. $S \sqcap T$ is inconsistent iff $S \sqcap T$ can be rewritten to $\bot$ using the equivalences*

$$\begin{array}{rclrclrcl}
S \sqcap (T_1 \sqcap T_2) &=& (S \sqcap T_1) \sqcap T_2 & S \sqcap \bot &=& \bot & f{\uparrow} \sqcap f{:}T &=& \bot \\
(S_1 \sqcap S_2) \sqcap T &=& (S_1 \sqcap T) \sqcap (S_2 \sqcap T) & \bot \sqcap T &=& \bot & f{:}S \sqcap f{\uparrow} &=& \bot \\
(S_1 \sqcup S_2) \sqcap T &=& (S_1 \sqcup T) \sqcap (S_2 \sqcup T) & f{:}S \sqcap a &=& \bot & a \sqcap b &=& \bot \\
f{:}S \sqcap f{:}T &=& f{:}(S \sqcap T) & a \sqcap f{:}T &=& \bot & {\sim}a \sqcap a &=& \bot \\
 & & & f{:}\bot &=& \bot & a \sqcap {\sim}a &=& \bot
\end{array} \tag{1}$$

Proof. The first four equivalences in (1) handle union, intersection, and selection operators; the remaining equivalences identify all combinations of primitives that might lead to inconsistency. Correctness follows from algebraic induction over S and T [62]. □

Proposition 9 *Let S and T be consistent and variable-free feature terms; let T also be simple. Then, inconsistency of $S \sqcap T$ can be decided in time complexity $\mathcal{O}(s \cdot \log t)$, where s is the number of primitives in S and t is the number of primitives in T.*
Proof. According to proposition 2, the term S can be rewritten to basic form in linear time, such that proposition 8 applies. Complexity follows from the fact that in the worst case, every primitive of S must be (syntactically) searched in T, which can be done in logarithmic time [62]. □

2.5 Simplification

Often, we are not only interested in deciding consistency of $S \sqcap T$, but also in *simplifying* S with respect to a given T; that is, to find a $S' \sqsupseteq S$ which is (syntactically) smaller than S, but for which $S' \sqcap T = S \sqcap T$ holds. The basic idea is to replace all literal occurrences of T in S by $\top$ and simplify S afterwards. This can be done by adding a few more equivalences to the rewrite system from proposition 8.

Proposition 10 *In $S \sqcap T$, the term S may be further reduced in size by expanding* (1) *with*

$$\begin{array}{rclrcl}
S \sqcap S &=& \top \sqcap S & f{\uparrow} \sqcap a &=& \top \sqcap a \\
{\sim}b \sqcap a &=& \top \sqcap a & {\sim}a \sqcap f{:}T &=& \top \sqcap f{:}T
\end{array} \tag{2}$$

and subsequent simplification

$$\begin{array}{lllll}
S \sqcap \bot = \bot & S \sqcap \top = S & S \sqcup \top = \top & S \sqcup \bot = S & {\sim}\top = \bot \\
\bot \sqcap S = \bot & \top \sqcap S = S & \top \sqcup S = \top & \bot \sqcup S = S & {\sim}\bot = \top
\end{array} \tag{3}$$

The first equivalence in (2), $S \sqcap S = \top \sqcap S$, is the essence of simplification: every literal occurrence in S of a primitive in T can be replaced by $\top$. The remaining equivalences in (2) eliminate superfluous negations. The equivalences in (3) propagate the new $\top$ values in S; complexity is unaffected.

Let us illustrate inconsistency decision and simplification by an example. Consider the term $S \sqcap T$, where $S = \big[f{:}a, g{:}\{b, {\sim}c\}\big]$ and $T = g{:}d$. We can decompose S to $S_1 \sqcap (S_2 \sqcup S_3) = f{:}a \sqcap \big(g{:}b \sqcup g{:}{\sim}c\big)$. For each primitive S_i, we check the consistency of $S_i \sqcap T$ and simplify S_i with respect to T. Beginning with $S_1 = f{:}a$, we find that S_1 and T have no common features; thus, $S_1 \sqcap T$ is consistent and S_1 cannot be simplified. Regarding $S_2 = g{:}b$, we have $S_2 \sqcap T = g{:}b \sqcap g{:}d$, which can be rewritten to $S_2 \sqcap T = g{:}[b, d] = g{:}\bot = \bot$; $S_2 \sqcap T$ is inconsistent. Considering $S_3 = g{:}{\sim}c$, we have $S_3 \sqcap T = g{:}{\sim}c \sqcap g{:}d = g{:}[{\sim}c, d] = g{:}d$; the term S_3 can thus be replaced by $\top$, as

$S_3 \sqcap T = T = \top \sqcap T$. The original term $S \sqcap T$ becomes $S \sqcap T = S_1 \sqcap (S_2 \sqcup S_3) \sqcap T = S_1 \sqcap (\bot \sqcup \top) \sqcap T = S_1 \sqcap \top \sqcap T = S_1 \sqcap T = [f{:}a] \sqcap T = [f{:}a, g{:}d]$. Hence, $S \sqcap T \neq \bot$—that is, the term $S \sqcap T$ is consistent. As a side effect, we find that S can be simplified to $S' = [f{:}a]$, since $S \sqcap T = S' \sqcap T$ holds.

Our presentation of feature logic is now complete. In the remainder of this article, we always interpret feature terms as sets of objects, unless otherwise specified. “Traditional” set notation will not be required, with one single exception: We write $|S|$ to express the *cardinality* (the number of elements) of a set denoted by the feature term S under a given interpretation. All other required notation is already provided by feature logic, as introduced above.

3 The Version Set Model

Having introduced feature logic, we can now return to the SCM domain. We begin with the SCM primitive layer, that is, basic versioning and access capabilities. We show how to capture SCM states by means of *version sets,* that is, sets of software components identified by their attributes. The basic SCM operations of selecting a version and composing a consistent configurations are modeled by means of set operations, as provided by feature logic.

3.1 Versions and Components

According with the SCM standards [21, 22], we consider that the object of interest in SCM is a family of *software products.* Each of these software products breaks down in several *components,* each of which may exist in several *component versions.* A component version is an unbreakable, unambiguous configuration item.

In the SCM domain, the common method for identifying component versions is *attribution,* as found in ADELE [15], the Context Model [39], EPOS [32], JASON [58], or SHAPE [33]. Using attribution, every component version is identified by a conjunction of attribute/value pairs describing its features; version selection is done through a (boolean) attribute expression which must be satisfied by the selected versions—similar to a classical selection in databases. In conditional inclusion, as exemplified by the C preprocessor (CPP), this setting is reversed: versions are identified by boolean attribute expressions and selected through a conjunction of attribute/value pairs describing the features of the environment.

Our model uses *feature terms* for both version identification and version selection. Every component version is assigned a feature term describing its features and uniquely identifying both version and component; versions are also selected by feature terms. Besides encompassing and integrating both the database and the CPP scheme, this setting also has a number of advantages for SCM users:

Alternative properties. Using feature terms, we are not restricted to a pure enumeration of features to identify versions. For instance, we can use *unions* like $\{\mathit{state}{:}\mathit{proposed}, \mathit{state}{:}\mathit{tested}\}$ to identify alternatives. In the database setting, such alternatives can only be used when *selecting* versions, but not to identify them. This ability to express alternative component properties is essential for treating version sets as unique items.

Configuration constraints. Feature terms may also express component properties that must *not* apply. For instance, we may use the term $\sim[\mathit{operating\text{-}system}{:}\mathit{unix}]$ to identify a version that must *not* be used under the UNIX operating system. Such a feature expresses a *constraint* on the environment, notably on other components in the configuration. In CPP, such constraints are realized through the *#error* directive. But in contrast to CPP, we can still use arbitrary selection terms—a selection term $\sim[\mathit{operating\text{-}system}\colon \mathit{unix}]$ would exclude all UNIX versions, but still include the non-WINDOWS version.

At the primitive layer, we do not impose specific requirements on the existence and the meaning of features; but to associate the versions of a component with each other, we must have at least one common feature across all component versions. We thus assume that each component can be identified uniquely via an *object* feature assigning each component a simple (unambiguous) component identifier. Our *configuration universe* then becomes the set denoted by $[\mathit{object}\colon \top]$—the set of all component versions.

We now define the notions of versions and components. A *version set* is any set $V \sqsubseteq [\mathit{object}\colon \top]$. A *version* is a singleton version set; that is, a set $V \sqsubseteq [\mathit{object}\colon \top]$ such that $|V| = 1$. A *component* is a set $K \sqsubseteq [\mathit{object}\colon k]$, where k is a simple feature term uniquely identifying the component. A *component version* is both a component and a version; that is, a set $K \sqsubseteq [\mathit{object}\colon k]$ with $|K| = 1$.

The features of a component are modeled as *alternatives* over the features of each component version. So, if we have a component K in n component versions $V_1, V_2, \ldots, V_n$, the component K is determined as

$$K = V_1 \sqcup V_2 \sqcup \cdots \sqcup V_n = \bigsqcup_{1 \le i \le n} V_i \ . \tag{4}$$

Features F of the component itself (as $[\mathit{object}\colon k]$) are the same across all component versions, and hence can be factored out through $(F \sqcap V_1) \sqcup (F \sqcap V_2) = F \sqcap (V_1 \sqcup V_2)$.

As a simple example, consider a *printer* component occurring in two component versions:

$$\begin{aligned} \mathit{printer}_1 &= [\mathit{object}\colon \mathit{printer}, \mathit{print\text{-}language}\colon \mathit{postscript}] \\ \mathit{printer}_2 &= [\mathit{object}\colon \mathit{printer}, \mathit{print\text{-}language}\colon \mathit{ascii}] \ . \end{aligned}$$

The *printer* component is then denoted as

$$\begin{aligned} \mathit{printer} &= \mathit{printer}_1 \sqcup \mathit{printer}_2 \\ &= \big[\mathit{object}\colon \mathit{printer}, \mathit{print\text{-}language}\colon \{\mathit{postscript}, \mathit{ascii}\}\big] \ . \end{aligned}$$

To retrieve a specific version, we specify a *selection term* S giving the features of the desired version. For any selection term S and a version set T, we can identify the versions satisfying S by calculating $T' = T \sqcap S$—that is, the version set that is a subset of S as well as a subset of T. If $T' = \bot$, selection fails—T' does not denote any existing version. In our example, selecting $S = [\mathit{print\text{-}language}\colon \mathit{postscript}]$ from *printer* returns $\mathit{printer}_1$, since $\mathit{printer} \sqcap S = (\mathit{printer}_1 \sqcup \mathit{printer}_2) \sqcap S = (\mathit{printer}_1 \sqcap S) \sqcup (\mathit{printer}_2 \sqcap S) = \mathit{printer}_1 \sqcup \bot = \mathit{printer}_1$. Here, $\mathit{printer}_2 \sqcap S = \bot$ holds since the *print-language* feature may have only one value. As T' is just another version set, we may give a second selection term S' and select $T'' = T' \sqcap S'$, give a third selection term S'', and so on, narrowing the choice set incrementally until a singleton set is selected, containing the desired version.

3.2 Composing Consistent Configurations

A *configuration,* in our setting, is a set of components. In our model, as in several attribution-oriented SCM versioning models, features of the components are propagated to the configurations;

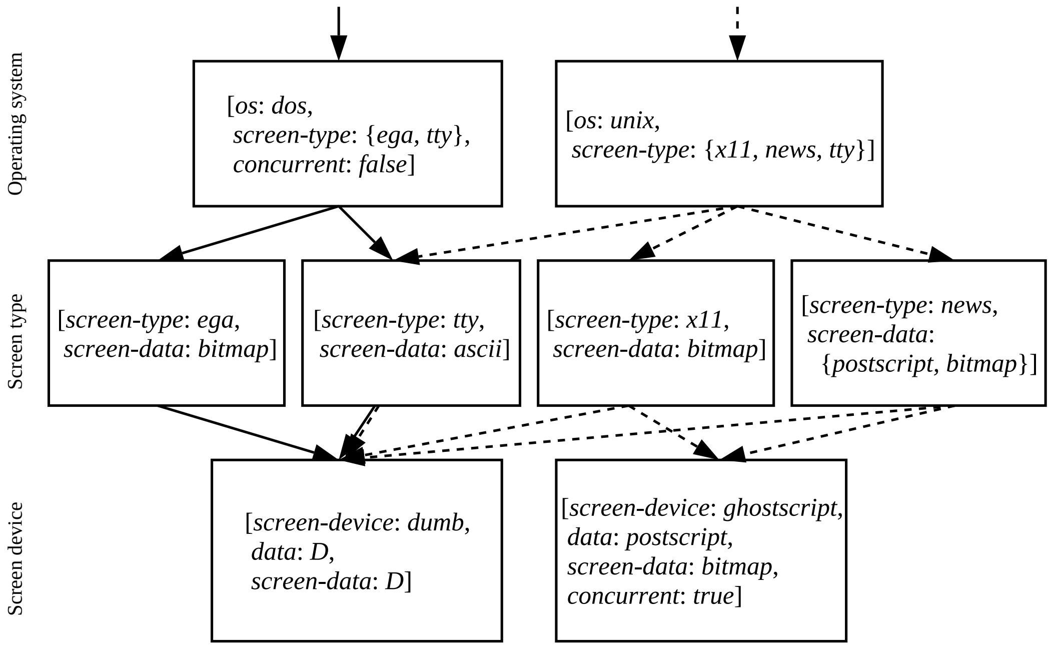

Operating system
[os: dos, screen-type: {ega, tty}, concurrent: false]
[os: unix, screen-type: {x11, news, tty}]
Screen type
[screen-type: ega, screen-data: bitmap]
[screen-type: tty, screen-data: ascii]
[screen-type: x11, screen-data: bitmap]
[screen-type: news, screen-data: {postscript, bitmap}]
Screen device
[screen-device: dumb, data: D, screen-data: D]
[screen-device: ghostscript, data: postscript, screen-data: bitmap, concurrent: true]

Figure 3: Consistent configurations in a text/graphic editor

one says that configurations *inherit* the features from their components. The crucial point when composing configurations from components is to ensure that the configuration is well-formed or *consistent.*

To determine the internal consistency of a configuration, most SCM tools rely on either separate tools [40] or language-specific knowledge [54, 44]. Consistency with respect to an external specification is usually combined with configuration selection; each consistency constraint becomes part of the selection term.

In the version set model, configuration constraints can be specified in the selection term, but also occur in the features of a component. For this purpose, both inheritance and consistency are realized by modeling the features of an configuration as *intersection* of the component features, excluding inconsistent combinations. For instance, we cannot build an configuration from two components having the features [*operating-system*: *windows-nt*] and [*operating-system*: *unix*], since the *operating-system* feature can have only one value: formally, [*operating-system*: *windows-nt*] $\sqcap$ [*operating-system*: *unix*] $= \perp$. If we have a configuration C composed of n components with the features $K_1, K_2, \ldots, K_n$, the configuration C has the features

$$C = K_1 \sqcap K_2 \sqcap \cdots \sqcap K_n = \prod_{1 \le i \le n} K_i \quad . \tag{5}$$

As an example of configuration consistency, consider figure 3. We see three source components of a text editor, where each component comes in several variants. We can choose between two operating systems (*dos* and *unix*), four screen types (*ega*, *tty*, *x11* and *news*), and two screen device drivers (*dumb* and *ghostscript*). The *dumb* driver assumes that the screen type can handle the data directly (expressed through the variable D); the *ghostscript* driver is a separate process that can convert postscript data into a bitmap. The component features imply that at most one version of each component can be included in a bound configuration.

Let us now compose a consistent configuration from these three source components. We begin by selecting the operating system, and choose the *dos* version. This implies that we cannot choose the *x11* or *news* screen types, since (in our example), *dos* does not support them: Formally,

$$\big[\mathit{os}\colon \mathit{dos}, \mathit{screen\text{-}type}\colon \{\mathit{ega}, \mathit{tty}\}\big] \sqcap \big[\mathit{screen\text{-}type}\colon \{\mathit{x11}, \mathit{news}\}\big] = \bot$$

due to the differing *screen-type* features—we cannot use *x11* or *news* screen types. We can, however, choose *ega* or *tty* screen types, as indicated by plain lines.

As final component, we must choose a screen device driver. *ghostscript* cannot be chosen, since it requires *concurrent* to be true, which is not the case under *dos*. The *dumb* driver remains; D is instantiated to *bitmap* or *ascii*, depending on the screen type, making our choice complete: *editor* can be built in a *ega* and a *tty* variants, inheriting the features of its source components. As an alternative, consider the choice $[\mathit{os}\colon \mathit{unix}]$, as indicated by dashed lines. Again, each path stands for a consistent configuration.

The ability of treating component features as configuration constraints allows for arbitrary *localization* of configuration constraints: components can be tagged with constraints regarding their usage, but global constraints regarding (sub-)systems are permitted as well. In short, every constraint usually expressed in version selection is also permissible as a component feature, and applies to the configuration as soon as the component is included.

The benefit of localization is that one single language can be used to specify constraints, to specify the component features, and to select component versions. But this benefit is also a drawback: the chosen language must be expressive enough to encompass existing SCM selection schemes, yet simple enough to keep mutual consistency of configuration constraints decidable. Checking constraint consistency can be a hard task; at least the existing SCM selection and identification schemes should be handled efficiently. With feature logic, we hope having chosen a well-established foundation which addresses all these issues.

3.3 Features of Configurations

Pure intersection is not appropriate for all features. For features like *author* or *status*, it makes perfect sense to differ across components; *object* features differ by definition. These *independent features* must depend on the specific component. A possible approach to do so is to prefix all independent features f with the component name k, resulting in orthogonal features like *tty-author* or *screen-status* [64]. A far better alternative is to express this dependency explicitly in feature logic, using implications $[\mathit{object}\colon k] \to T$ that enforce the version T whenever the component k is required.

To construct such implications, we define a special *aggregation operator.* The operator "$\boxplus_I$" is similar to "$\sqcap$", but has a special handling of independent features: instead of unifying them, it makes them dependent on the specific component; *object* features are stripped altogether.

Let $I = \{f_1\colon \top, f_2\colon \top, \ldots, f_m\colon \top\}$ be a feature term denoting independent features, where $f_i \neq \mathit{object}$ holds for all $1 \leq i \leq m$, and let $K_1, \ldots, K_n$ denote components. For each component K_i, let k_i, $K_i' \sqsubseteq I$, and $K_i'' \not\sqsubseteq I$ be chosen such that

$$K_i = [\mathit{object}\colon k_i] \sqcap K_i' \sqcap K_i''$$

holds—that is, k_i is the unique component identifier, K_i'' denotes the independent features of K_i, and K_i' denotes the ordinary (non-independent) features of K_i. The *aggregation* of all K_i, written $K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n$, is then defined as

$$K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n = \mathop{\boxplus}_{1\leq i\leq n}{}_I\, K_i = \bigsqcap_{1\leq i\leq n} K_i' \sqcap \big([\mathit{object}\colon k_i] \to K_i''\big) \ .$$

Given an aggregation $C = S \boxplus_I T$, we can properly select S and T by intersecting C with $[\mathit{object}\colon s]$ and $[\mathit{object}\colon t]$, respectively:

Proposition 11 *Let $S \sqsubseteq [\mathit{object}\colon s]$ and $T \sqsubseteq [\mathit{object}\colon t]$ denote components, and I denote independent features, as described above. Then,*

$$[\mathit{object}\colon s] \sqcap (S \boxplus_I T) \sqsubseteq S \tag{6}$$

holds.[3]

Proof. Let $T = [\mathit{object}\colon t] \sqcap T' \sqcap T''$, as defined above. Then, $U = [\mathit{object}\colon s] \sqcap (S \boxplus_I T) = [\mathit{object}\colon s] \sqcap \big(S' \sqcap T' \sqcap \big([\mathit{object}\colon s] \to S''\big) \sqcap \big([\mathit{object}\colon t] \to T''\big)\big)$. But since $[\mathit{object}\colon s] \sqcap \big([\mathit{object}\colon s] \to S''\big) = [\mathit{object}\colon s] \sqcap \big({\sim}[\mathit{object}\colon s] \sqcup S''\big) = [\mathit{object}\colon s] \sqcap S''$ and $[\mathit{object}\colon s] \sqcap \big([\mathit{object}\colon t] \to T''\big) = [\mathit{object}\colon s] \sqcap \big({\sim}[\mathit{object}\colon t] \sqcup T''\big) = [\mathit{object}\colon s]$, we have $U = [\mathit{object}\colon s] \sqcap (S' \sqcap T' \sqcap S'') = \big([\mathit{object}\colon s] \sqcap S' \sqcap S''\big) \sqcap T' = S \sqcap T' \sqsubseteq S$. □

Using the aggregation operator, we can extend (5) with *object* features and independent features and formally define how features propagate from components to configurations. If we have a configuration C composed of n components $K_1, K_2, \ldots, K_n$ with $K_i \sqsubseteq [\mathit{object}\colon k_i]$, and a term I denoting the independent features, the configuration C is identified by

$$\begin{aligned} C &= [\mathit{object}\colon k_1 \sqcup k_2 \sqcup \cdots \sqcup k_n] \sqcap K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n \\ &= [\mathit{object}\colon k_1 \sqcup k_2 \sqcup \cdots \sqcup k_n] \sqcap \mathop{\boxplus_I}_{1 \le i \le n} K_i \quad , \end{aligned} \tag{7}$$

that is, *object* features are united, independent features are made dependent on the respective component, and all other features are unified.

As an example, consider two components

$$\begin{aligned} \mathit{screen} &= \big[\mathit{object}\colon \mathit{screen}, \mathit{author}\colon \mathit{lisa}, \mathit{resolution}\colon \{\mathit{high}, \mathit{medium}\}\big] \\ \mathit{driver} &= [\mathit{object}\colon \mathit{driver}, \mathit{author}\colon \mathit{tom}, \mathit{resolution}\colon \mathit{high}] \quad . \end{aligned}$$

Let $I = [\mathit{author}\colon \top]$ be the set of independent features. According to (7), the configuration C containing *screen* and *driver* is

$$\begin{aligned} C = \big[&\mathit{object}\colon \{\mathit{screen}, \mathit{driver}\}, \mathit{resolution}\colon \mathit{high}, \\ &(\mathit{object}\colon \mathit{screen} \to \mathit{author}\colon \mathit{lisa}), (\mathit{object}\colon \mathit{driver} \to \mathit{author}\colon \mathit{tom})\big] \quad . \end{aligned}$$

Besides unifying the non-independent features of *screen* and *driver* to *resolution*: *high*, the term C properly selects Lisa's *screen* object and Tom's *driver* object—that is, $C \sqcap [\mathit{object}\colon \mathit{screen}] \sqsubseteq [\mathit{author}\colon \mathit{tom}]$ and $C \sqcap [\mathit{object}\colon \mathit{driver}] \sqsubseteq [\mathit{author}\colon \mathit{lisa}]$.

We close by defining some *properties* of configurations, following (7). Formally, a *configuration* is a set $C \sqsubseteq [\mathit{object}\colon c]$, where c is a feature term identifying the set of configuration components. A configuration C is called *consistent* with respect to its features if $C \neq \bot$—that is, if the number of possible configurations is non-zero. A configuration C is called unambiguous or *bound* if it is an aggregation of component versions; formally, C is bound if it is a set $C \sqsubseteq [\mathit{object}\colon c]$ such that $|C| = |c|$. If it is not bound ($|C| > |c|$ holds), a configuration C is called ambiguous, dynamic, or *abstract.*

[3] In [60], we gave an alternate definition of the aggregation operator, for which (6) did not hold.

3.4 Features of Derived Components

In the SCM context, we must not only describe how features propagate from components to configurations. An important SCM topic is the identification of *derived* components, constructed automatically from a configuration of source components—using the well-known MAKE program or one of its successors.

To determine the features of derived components, we use a variation of (7). Again, derived components must be consistent, which implies that the source configuration be consistent as well. To ensure consistency across multiple derivation stages, each derived component must inherit the features of its source components, just as a configuration inherits the features of its components.

Formally, if we have a component $K \sqsubseteq [\mathit{object}\colon k]$ derived from n source components $K_1, K_2, \ldots, K_n$, and a term I denoting the independent features, K is identified by

$$\begin{aligned} K &= [\mathit{object}\colon k] \sqcap K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n \\ &= [\mathit{object}\colon k] \sqcap \mathop{\boxplus_I}_{1 \le i \le n} K_i \quad , \end{aligned} \tag{8}$$

where the explicit setting of the *object* feature removes all implications generated by the aggregation operator—only non-independent features remain to be unified.

As an example of derivation, consider the editor example from figure 3. Let us denote the three components operating system, screen type, and screen device by $[\mathit{object}\colon \mathit{os}, \mathit{author}\colon \mathit{tom}]$, $[\mathit{object}\colon \mathit{st}, \mathit{author}\colon \mathit{lisa}]$, and $[\mathit{object}\colon \mathit{sd}, \mathit{author}\colon \mathit{john}]$, respectively; let the independent features be $I = [\mathit{author}\colon \top]$. If we derive an *editor* component from a DOS/EGA configuration, it is identified by

$$\begin{aligned} K = {} & [\mathit{object}\colon \mathit{editor}] \\ & \sqcap \big([\mathit{object}\colon \mathit{os}, \mathit{author}\colon \mathit{tom}, \mathit{screen\text{-}type}\colon \{\mathit{ega}, \mathit{tty}\}, \mathit{concurrent}\colon \mathit{false}] \\ & \quad \boxplus_I [\mathit{object}\colon \mathit{st}, \mathit{author}\colon \mathit{lisa}, \mathit{screen\text{-}type}\colon \mathit{ega}, \mathit{screen\text{-}data}\colon \mathit{bitmap}] \\ & \quad \boxplus_I [\mathit{object}\colon \mathit{sd}, \mathit{author}\colon \mathit{john}, \mathit{screen\text{-}device}\colon \mathit{dumb}, \mathit{data}\colon D, \mathit{screen\text{-}data}\colon D]\big) \\ = {} & [\mathit{object}\colon \mathit{editor}, \mathit{screen\text{-}type}\colon \mathit{ega}, \mathit{concurrent}\colon \mathit{false}, \\ & \quad \mathit{screen\text{-}data}\colon \mathit{bitmap}, \mathit{screen\text{-}device}\colon \mathit{dumb}, \mathit{data}\colon \mathit{bitmap}] \quad , \end{aligned}$$

that is, the *object* features and independent features of the source components are stripped, and all other features are unified. In [63], we discuss a MAKE extension using this mechanism to create and re-use derived components from consistent source configurations.

4 Versioning Dimensions

We now turn to the SCM protocol layer, introducing specific *versioning dimensions.* SCM literature distinguishes four versioning dimensions: historic (revisions), logic (variance), cooperative (workspaces), composition (configurations) [42, 14]. It is a well-known goal of SCM to integrate these dimensions: the concepts of *orthogonal versioning* [42], and *three-dimensional versioning* [14], for instance, each integrate three of these four dimensions. The problem is that these models use different sets of queries and services due to the differing motivations, which results in a lack of orthogonality.

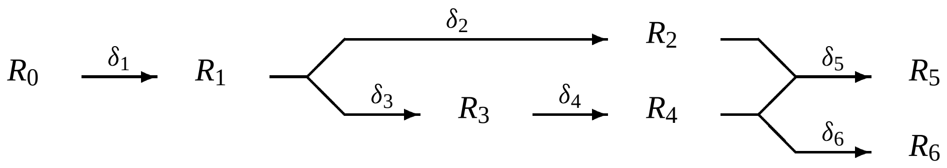


Figure 4: A revision history

In this section, we show that each of these versioning dimensions can be realized in the version set model. The underlying foundation, feature logic, is *uniform:* all versions are identified with their features, regardless of their versioning dimension; the SCM primitive layer makes no such distinction as well. At the protocol level, however, we can introduce *diversity:* by giving special meanings to features, we distinguishing versioning dimensions. We have already seen how to handle variance and composition dimensions; in this section, we turn to the more specific historic and cooperative dimensions.

4.1 Revisions and Changes

As initial concepts, we show how to realize *changes* and *revisions.* A revision is a version intended to supersede another version (in contrast to a *variant*) [59]. Typically, a revision is the product of a *change* applied to an existing revision. In traditional SCM, these changes are controlled by *version-oriented versioning.* Version-oriented versioning controls the impact of changes by *serializing* them—one change is applied after the other, forming a *revision history.* As an example, consider the revision history in figure 4, where individual revisions of a version set are denoted by $R_0, R_1, R_2, \dots$ and so on. Each revision R_i is created by applying a change (denoted by δ_i) to some originating revisions $R_j, \dots, R_k$. As an example, consider revision R_5, which was created from R_2 and R_4 by applying the change δ_5.

In version-oriented versioning, each change implies several previous changes. In our example, having the change δ_4 applied requires the previous application of change δ_3; likewise, δ_5 implies all other changes except δ_6. As several configurations are excluded—there simply is no way to include the change δ_5 without also having δ_2 applied—, it is quite easy to analyse the impact of a single change. However, version-oriented versioning becomes a problem when changes are largely independent of each other—i.e., when one wants a configuration with certain changes applied, but other excluded. These weaknesses are addressed by *change-oriented versioning* [20, 19, 36], where versions are merely the product of applying a change or *delta* to a *baseline,* an already existing version set.

In the version set model, we have adopted change-oriented versioning. Each revision is identified by a conjunction of *delta features* standing for the change application. A revision R is a subset of $\Delta_i = [\delta_i : \top]$, if the change δ_i has been applied; R is a subset of $\nabla_i = \sim\Delta_i = [\delta_i \uparrow]$ if the change δ_i has *not* been applied. The revision R_4 in figure 4, for instance, would be identified by

$$R_4 = \Delta_1 \sqcap \nabla_2 \sqcap \Delta_3 \sqcap \Delta_4 \sqcap \nabla_5 \sqcap \nabla_6 \quad , \tag{9}$$

that is, only the changes δ_1, δ_3, and δ_4 have been applied. Again, revisions are identified and selected just like any other versions, using features.

While a selection scheme enumerating the applied changes is convenient for changes that can be applied independently from each other, it becomes a pain when, say, revision 211 must be selected by enumerating 211 changes to be applied. A unified versioning model thus must find a way to accommodate both the convenience of version-oriented versioning as well as the freedom of change-oriented versioning. The idea is to exclude certain change combinations through *revision constraints.*

Mutual exclusions. As an example, consider a version set R where selecting an arbitrary change combination S should result in a consistent product $R \sqcap S$—except for $R \sqcap (\Delta_5 \sqcap \Delta_6)$, which should be inconsistent ("The changes δ_5 and δ_6 do not integrate"). This can be achieved by making R a subset of $\sim(\Delta_5 \sqcap \Delta_6) = \nabla_5 \sqcup \nabla_6$; it is easy to see that $R \sqcap S \sqsubseteq (\nabla_5 \sqcup \nabla_6) \sqcap S$ becomes inconsistent when $S \sqsubseteq (\Delta_5 \sqcap \Delta_6)$ holds. Generally, to exclude the combination of two changes δ_i and δ_j in a version set R, it suffices to make R a subset of the revision constraint $\nabla_i \sqcup \nabla_j$.

Change implications. Another problem is how to make changes rely on each other. Let us assume that R contains no version where the change δ_9 has been applied, but not δ_7—we would say, the change δ_9 *implies* the change δ_7. This implication becomes explicit by making R a subset of $\Delta_9 \to \Delta_7$; in this case, $R \sqcap (\Delta_9 \sqcap \nabla_7) \sqsubseteq (\Delta_9 \to \Delta_7) \sqcap (\Delta_9 \sqcap \nabla_7) = (\Delta_9 \to \Delta_7) \sqcap \sim(\Delta_9 \sqcap \Delta_7) = \bot$ holds, effectively excluding the change application. Generally, to ensure that a change δ_i implies a change δ_j in a version set R, it suffices to make R a subset of the revision constraint $\Delta_i \to \Delta_j$.

A simple example of revision constraints is a linear revision history, where each change implies all previous changes. As an example, let a revision set R be a subset of $(\Delta_{211} \to \Delta_{210}) \sqcap (\Delta_{210} \to \Delta_{209}) \sqcap \cdots \sqcap (\Delta_2 \to \Delta_1)$. We can easily select revision 211 just by selecting $R \sqcap \Delta_{211}$: all other changes are automatically implied by the revision constraints. We see how revision constraints effectively control the application of changes and inhibit inconsistent change combinations—simply by assigning appropriate features to version sets.

4.2 Constraints and Histories

By specifying appropriate revision constraints, it is even possible to capture arbitrary revision histories, realizing full version-oriented versioning. As an example, consider the version set R containing $R_0, \ldots, R_6$ created through the changes $\delta_1, \ldots, \delta_6$, as shown in figure 4. Following (4), R could be represented as $R = R_0 \sqcup \cdots \sqcup R_6$, where each R_i is a conjunction of included and excluded changes, as R_4 in (9). A far more elegant representation is obtained through revision constraints. For instance, R must be a subset of $(\Delta_2 \to \Delta_1)$, since δ_2 relies on δ_1, and R must also be a subset of $\nabla_2 \sqcup \nabla_6$, as the changes δ_2 and δ_6 are mutually exclusive. In fact, R can be entirely represented through revision constraints, denoting the complete revision history:

$$\begin{aligned} R = {} & (\Delta_2 \to \Delta_1) \sqcap (\Delta_3 \to \Delta_1) \sqcap (\Delta_4 \to \Delta_3) \sqcap (\Delta_5 \to \Delta_2) \sqcap (\Delta_5 \to \Delta_4) \sqcap (\Delta_6 \to \Delta_4) \\ & \sqcap (\Delta_2 \sqcap \Delta_3 \to \Delta_5) \sqcap (\nabla_2 \sqcup \nabla_6) \ . \end{aligned} \tag{10}$$

How are these constraints obtained? Formally, for any two revisions R_i and R_j, let $R_{\overline{i,j}}$ be their lowest common ancestor in the revision history, and let $R_{\underline{i,j}}$ be their highest common descendant. Let us denote the changes leading up to R_i, R_j, $R_{\overline{i,j}}$, and $R_{\underline{i,j}}$ by δ_i, δ_j, $\delta_{\overline{i,j}}$, and $\delta_{\underline{i,j}}$, respectively; the version sets $\Delta_i = [\delta_i \colon \top]$, $\Delta_j = [\delta_j \colon \top]$, $\Delta_{\overline{i,j}} = [\delta_{\overline{i,j}} \colon \top]$, and $\Delta_{\underline{i,j}} = [\delta_{\underline{i,j}} \colon \top]$ are defined as usual. Should $R_{\underline{i,j}}$ not exist, then $\Delta_{\underline{i,j}} = \bot$ holds. Let now $C_{i,j}$ be a *formal revision constraint* defined as

$$C_{i,j} = (\Delta_i \sqcup \Delta_j \to \Delta_{\overline{i,j}}) \sqcap (\Delta_i \sqcap \Delta_j \to \Delta_{\underline{i,j}}) \tag{11}$$

If a change δ_i implies a change δ_j, the revision constraint $C_{i,j}$ becomes $C_{i,j} = (\Delta_i \sqcup \Delta_j \rightarrow \Delta_i) \sqcap (\Delta_i \sqcap \Delta_j \rightarrow \Delta_j) = \Delta_j \rightarrow \Delta_i$; if δ_i and δ_j are mutually exclusive, $C_{i,j} \sqsubseteq (\Delta_i \sqcap \Delta_j \rightarrow \underline{\Delta_{i,j}}) = (\Delta_i \sqcap \Delta_j \rightarrow \bot) = \nabla_i \sqcup \nabla_j$ holds.

It now turns out that the intersection of all $C_{i,j}$ is equivalent to R:

Proposition 12 *A revision set R can be represented as union of all revisions R_i, each identified by an intersection of included and excluded changes, or as an intersection of revision constraints $C_{i,j}$, as defined in* (11)*. Both representations are equivalent.*

$$R = \bigsqcap_{\substack{1 \leq i \leq n \\ 1 < j < i}} C_{i,j} = \bigsqcup_{0 \leq i \leq n} R_i \quad . \tag{12}$$

Proof. See [62]. □

In our example, the representation in (10) is obtained via (12) and removing superfluous constraints, following the general scheme $(\Delta_i \rightarrow \Delta_j) \sqcap (\Delta_j \rightarrow \Delta_k) \sqsubseteq (\Delta_i \rightarrow \Delta_k)$. We see how proposition 12 realizes version-oriented versioning on top of change-oriented versioning, using appropriate constraints.

The maintenance of these implications is the duty of the SCM protocol layer, hiding them from the end user; in section 6.1, we discuss a simple check-in/check-out protocol realized through revision constraints. Our SCM primitive layer has no notion of revisions—all it knows about are components identified by features, and it does not distinguish between specific feature types. Hence, revision constraints may also be used to express implications between delta features and other features.

In CLEARCASE, for example, users can assign names to edges in the revision history and select revisions through a disjunction of name patterns; such naming of changes is easily expressed through an implication between the name and the appropriate delta features. Another example is *currency:* we cannot simply devise some revision as “current”, because currency may differ across variants. Hence, currency constitutes a part of the SCM protocol, expressed through means of the SCM primitive layer. A simple scheme to denote currency is to use a set $[\mathit{current}\colon \top]$ that contains the current variants by implying certain revisions. An implication $\big([\mathit{current}\colon \top, \mathit{os}\colon \mathit{unix}] \rightarrow \nabla_5\big)$ ensures that whenever the current *unix* variant is requested, the change δ_5 is excluded, possibly excluding subsequent changes through further revision constraints. The maintenance of currency is also illustrated in section 6.1.

By dropping any distinction between delta features and variant- or process-specific features, and by unifying the concepts of attribution and revision histories, our SCM primitive layer allows to create, select, and revise arbitrary revision/variant/component combinations as in orthogonal version management [42], still while allowing refinement and inheritance as in object-oriented SCM [58].

4.3 Cooperation through Locks and Workspaces

Besides components, variants, and revisions, SCM literature distinguishes a fourth versioning dimension. *Team functionality* enables a team of developers to develop and maintain the software product. The most basic team functionality is a cooperation strategy that ensures that the changes of an individual developer are not accidently superseded by another developer.

Using a *conservative cooperation strategy,* developers must *lock* each component version or configuration they wish to change. Locks are exclusive: While a version or configuration is locked, other developers are excluded from creating new revisions. Using version sets, locks are managed like currency: The set $[\mathit{locked}\colon\top]$ contains all locked versions, $\sim[\mathit{locked}\colon\top] = [\mathit{locked}\uparrow]$ the unlocked versions. An SCM system locking a component version K would do so by changing its features such that $K \sqsubseteq [\mathit{locked}\colon\top]$; any selection of K from $[\mathit{locked}\uparrow]$ would fail. As locking is orthogonal to all other features, arbitrary version sets can be locked.

The second generation of SCM systems introduced *optimistic cooperation strategies* [5, 11]. Rather than preventing concurrent changes, they rather attempt to integrate changes later. The central concept here is the notion of a *workspace,* the individual area of a developer, isolating him from changes made by other developers, and isolating others from his changes.

In our model, a user's workspace is just a variant identified by a feature term $W \sqsubseteq [\mathit{user}\colon\top]$—that is, $[\mathit{user}\colon \mathit{lisa}]$ denotes Lisa's workspace, and $[\mathit{user}\colon \mathit{tom}]$ is Tom's workspace. As the *user* feature may have only one value, all workspaces are disjoint; that is, developer Lisa in her workspace $[\mathit{user}\colon \mathit{lisa}]$ will not see any changes from the $[\mathit{user}\colon \mathit{tom}]$ workspace. Tom may create new revisions Δ_i in his workspace, or change currency; as his changes are always subsumed by $[\mathit{user}\colon \mathit{tom}]$, Lisa's workspace will remain unaffected. To apply Tom's changes in her workspace, Lisa must integrate Tom's changes and her own changes. Lisa's changes can be identified by comparing the contents of her workspace $[\mathit{user}\colon \mathit{lisa}]$ with the contents of the originating version set $\sim\big[\mathit{user}\colon \{\mathit{lisa}, \mathit{tom}\}\big]$; Tom's changes can be identified likewise.

In our setting, locks and workspaces are part of the SCM protocol, as are currency and revisions. As they are realized through dedicated features, they can be freely integrated with other features in selections and constraints. Tom may declare his workspace as $[\mathit{user}\colon \mathit{tom}, \mathit{os}\colon \mathit{unix}]$, thus confining all changes to his workspace and the UNIX version. Lisa may wish to work on the current revision only, but including all variants, thus choosing her workspace as $[\mathit{user}\colon \mathit{lisa}, \mathit{current}\colon\top]$. Further dedicated features may be used for modeling teams or geographically distributed sites, ensuring orthogonality and uniformity at the interface between the SCM primitive and SCM protocol layers.

4.4 Practical Extensions

Although our versioning model subsumes all common identification and selection schemes as found in SCM systems, it may prove useful to support additional selection schemes in practice. Some SCM systems select component versions through a set of configuration rules, using PROLOG-like syntax as in SHAPE [30] or pattern matching rules as in CLEARCASE [31]. The basic idea is that the first matching rule is applied. An alternate scheme is realized in preference clauses [29], where each configuration rule refines the results of the previous one, until an unambiguous version is selected. Such schemes cannot be expressed in feature logic directly, since a version S being unambiguous means that $|S| = 1$ holds, and checking the cardinality depends on a specific interpretation. However, the semantics of such selection schemes can be described on top of feature logic, using *preference operators:*

$$S_1 \text{ and-then } S_2 = \begin{cases} S_1 & \text{if } S_1 \text{ is bound,} \\ S_1 \sqcap S_2 & \text{otherwise} \end{cases} \qquad S_1 \text{ or-else } S_2 = \begin{cases} S_1 & \text{if } S_1 \neq \bot, \\ S_2 & \text{otherwise} \end{cases}$$

with the equivalences $T \sqcap (S_1 \text{ and-then } S_2) = (T \sqcap S_1 \text{ and-then } T \sqcap S_2)$ and $T \sqcap (S_1 \text{ or-else } S_2) = (T \sqcap S_1 \text{ or-else } T \sqcap S_2)$. Using "and-then" and "or-else", we can express *preferences* in our selection terms. For instance, $S = \big([\mathit{current}\colon\top] \text{ or-else } [\mathit{fixed}\colon \mathit{true}]\big)$ first selects the current version, and, if there is none, a "fixed" version; $S = \big([\Delta_2, \nabla_3] \text{ and-then } [\mathit{os}\colon \mathit{unix}]\big)$ selects revision 2 and, should this choice be ambiguous, the UNIX variant.

Another practical extension are additional constraints, expressing properties whose mutual consistency cannot be decided in feature logic alone. Useful examples include arithmetic constraints (*date* $<$ 1997) or function interfaces (*gcd*: $int \times int \rightarrow int$). Such constraints can be handled as additional constraints in Smolka's feature unification algorithm when deciding about the inconsistency of simple feature terms; they can be evaluated as soon as their variables (features) are instantiated [52].

When using such extended constraints, users should be aware that the inconsistency of a conjunction of extended constraints cannot always be determined. In practice, one would use well-known constraint solving systems like the Simplex Method or language-specific consistency checkers to determine most inconsistencies.

5 The Featured File System

To find out how the version set model works in practice, we have realized the version set model in an experimental SCM system, called ICE for *Incremental Configuration Environment.* ICE provides access uses to version sets through a virtual file system called FFS. The FFS represents version sets in the well-known *#if* ... *#endif* format, which identifies differences between versions. Using the FFS as example, we explore the feasibility of a repository based on version sets; by defining the effects of basic file operations, we provide a means to describe operations at the SCM protocol layer.

5.1 Representing Version Sets

Upon designing ICE, the first problem that arose was the representation and efficient storage of version sets at the SCM primitive layer. As it was our aim to make ambiguity transparent to developers, we wanted to represent version sets in a format suitable for human readers.

The by far most common representation of multiple versions in a single source is the C preprocessor (CPP) representation. Code pieces relevant for certain versions only are enclosed in *#if C* ... *#endif*, where *C* expresses the condition under which the code piece is to be included. Upon compilation, CPP selects a single version out of this set, feeding it to the compiler. (CPP's additional functionality, such as macro expansion and file inclusion, is of no interest here.)

Using conditional compilation, the programmer may perform changes simultaneously on the whole set of versions. Unfortunately, CPP technology does not scale up: as the number of versions grows, the representation can become so strewn with CPP directives that it is hard to understand, yet harder to change. Except for a small amount of variance, CPP usage is thus deprecated in the SCM community. But as this rejection applies to the tool, not the technique, we could represent version sets in CPP format, giving the user a familiar, well-understood representation.

ICE uses the CPP format to represent version sets and it uses CPP terms (i.e. boolean C expressions) to represent feature terms. In the CPP representation, feature names are expressed as

Feature Term	CPP Expr
$\top$	1
$\bot$	0
a	$-/-$
x	$-/-$
$f{:}a$	$f \equiv a$
$f{:}\sim a$	$f \not\equiv a$
$f{:}\sim 0$	f
$f{:}S$	$-/-$
$f{:}\top$	*defined*(f)
$f\uparrow$	$\neg$*defined*(f)
$f \downarrow g$	$f \equiv g$
$f \uparrow g$	$f \not\equiv g$
$\sim S$	$\neg S$
$S \sqcap T$	$S \wedge T$
$S \sqcup T$	$S \vee T$
$S \rightarrow T$	$\neg S \vee T$
$\exists x(S)$	$-/-$

Table 2: Translating feature terms into CPP expressions

get_load.c[*os*: *unix*]

```
void InitLoadPoint()
{
   extern void nlist();
#if defined(AIXV3) ∧ ¬defined(hcx)
   nlist(namelist, 1, . . . )
#else
   nlist(KERNEL_FILE, namelist);
#endif
#if defined(hcx)
   if (namelist[. . . ].n_type ≡ 0 ∧
#else
   if (namelist[. . . ].n_type ≡ 0 ∨
#endif
         namelist[. . . ].n_value ≡ 0) {
      xload_error(. . . );
      exit(-1);
   }
```

=

get_load.c[*os*: *unix*, *hcx*: ⊤]

```
void InitLoadPoint()
{
   extern void nlist();
   nlist(KERNEL_FILE, namelist);
   if (namelist[. . . ].n_type ≡ 0 ∧
         namelist[. . . ].n_value ≡ 0) {
      xload_error(. . . );
      exit(-1);
   }
```

⊔

get_load.c[*os*: *unix*, *hcx*↑]

```
void InitLoadPoint()
{
   extern void nlist();
#if defined(AIXV3)
   nlist(namelist, 1, . . . )
#else
   nlist(KERNEL_FILE, namelist);
#endif
   if (namelist[. . . ].n_type ≡ 0 ∨
         namelist[. . . ].n_value ≡ 0) {
      xload_error(. . . );
      exit(-1);
   }
```

Figure 5: Version sets represented as CPP files

CPP symbols. In table 2, we have summarized the mapping from feature terms to CPP expressions; for better readability, the C tokens `==`, `!=`, `&&`, `||`, and `!` are represented as $\equiv$, $\not\equiv$, $\wedge$, $\vee$, and $\neg$, respectively.

For nearly every feature term, there is an equivalent CPP expression. Exceptions (denoted by "-/-") include atoms (unless occurring as feature values), variables, and composed feature values. All of these can be used in CPP expressions by enclosing them in square brackets. Vice versa, every CPP expression has an equivalent feature term representation, with the exception of arithmetic CPP expressions, which are treated as atoms in feature terms. The CPP program itself is never used by ICE; only the syntax and semantics of CPP files and expressions are used.

We will now show how to realize selection and union on version sets represented as CPP files. Let F be a CPP file representing all source code versions; that is, a version set in CPP representation. To select a subset of F using a selection term S, that is, the set $F \sqcap S$, we proceed as follows. For each code piece enclosed in *#if C . . . #endif*, the governing feature term C is intersected with the selection term S. If $C \sqcap S = \bot$, the code piece is removed from F. If $C \sqcap S = S$, the *#if* directive is removed, because $S \sqsubseteq C$. Otherwise, C is simplified with respect to S, according to proposition 10. The new (smaller) CPP file can be characterized by S and is written $F[S] = F \sqcap S$ (obviously, $F = F[\top]$).

Figure 5 shows the constrained CPP file *get_load.c* taken from *xload,* a tool displaying the system load for several architectures. It shows two subsets of *get_load.c*[*os*: *unix*]: a *hcx* version *get_load.c*[*os*: *unix*][*hcx*: ⊤] = *get_load.c*[*os*: *unix*, *hcx*: ⊤] and a non-*hcx* version *get_load.c*[*os*: *unix*][*hcx*↑] = *get_load.c*[*os*: *unix*, *hcx*↑] (note the simplified CPP expressions). Further selection and refinement is possible until a singleton version set is obtained—that is, a source file without *#if* directives.

The union of two CPP files $F[S]$ and $F[T]$ can be computed through $F[S] \sqcup F[T] = F[S \sqcup T]$. A compact CPP representation of $F[S \sqcup T]$ can also be constructed even if F does not exist. The idea is to compare the two files textually, using a DIFF algorithm [35] initially ignoring all CPP directives. In the resulting file $F[S \sqcup T]$, text parts occurring only in $F[S]$ or $F[T]$ are governed by $S \sqcap {\sim}T$ or ${\sim}S \sqcap T$, respectively; common parts are governed by $S \sqcup T$. Read from right to left, figure 5 demonstrates that

$$\begin{aligned}
&\mathit{get\_load.c}[\mathit{os}\colon \mathit{unix}, \mathit{hcx}\colon \top] \sqcup \mathit{get\_load.c}[\mathit{os}\colon \mathit{unix}, \mathit{hcx}\uparrow] \\
&\quad = \mathit{get\_load.c}\big[[\mathit{os}\colon \mathit{unix}, \mathit{hcx}\uparrow] \sqcup [\mathit{os}\colon \mathit{unix}, \mathit{hcx}\colon \top]\big] \\
&\quad = \mathit{get\_load.c}[\mathit{os}\colon \mathit{unix}] \ \ ,
\end{aligned}$$

where the DIFF algorithm determines a compact representation for the generated version set $\mathit{get\_load.c}[\mathit{os}\colon \mathit{unix}]$; all governing expressions are simplified with respect to $[\mathit{os}\colon \mathit{unix}]$. We see that feature terms, introduced as a syntactic device for the denotation of version sets, now have a precise semantics in terms of CPP files.

5.2 Transparent Version Set Access

For integration with software development environments, the SCM primitive layer must make its configuration items accessible in some way. The least common denominator for today's environments is a *file system;* and we know of no SCM tool that would not provide a file system interface.

Most of today's SCM tools realize item access by explicit copying of source components from repositories (databases) to individual file systems and vice versa. This approach has the advantage that database technology like transaction safety or advanced query services are available for the repository; workspaces may be realized as (possibly ambiguous) sub-databases of the repository [15]. The drawback is that configuration items are no more under SCM control, once copied to the individual file system.

Recent approaches thus allow configurations and workspaces to be selected and manipulated as virtual file systems, representing individual views of the repository. Typical examples include NSE [11], *n*-DFS [18], and CLEARCASE [31]. In these systems, user workspaces are made part of some classical repository; the actual repository is either hard-wired (as in NSE and CLEARCASE) or generic (as in *n*-DFS). The entire repository is then made accessible as virtual file system. While being convenient for users, this technique also gives the SCM system direct control over user's workspaces. It allows for space savings through *copy-on-write* techniques (also known as *view-pathing*), sharing common files between several developers.

We have chosen the CPP representation, as introduced above, as base for a virtual file system in ICE, called FFS for *featured file system,* and realizing an example SCM primitive layer. In the FFS, all files occurring in multiple versions can be accessed by appending a version specification to the file name—just as in our notation above.[4] The following basic operations are supported by the FFS:

Read. Read access to $F[S]$ is accomplished by selection, as shown; opening the virtual file $\mathit{tty.c}[\mathit{user}\colon \mathit{tom}]$ gives access to the version set $[\mathit{user}\colon \mathit{tom}]$ from the file *tty.c*.

Write. Since $F = F[\sim S \sqcup S] = F[\sim S] \sqcup F[S]$, write access to $F[S]$—that is, changing $F[S]$ to $F'[S]$—is implemented by generating $F' = F[\sim S] \sqcup F'[S]$.

In practice, this means that any version subset $F[S]$ of some multi-version document can be edited and changed by invoking an ordinary text editor. CPP directives indicate the common and differing parts between versions. Upon each write of $F[S]$, the FFS re-determines the differences and CPP directives in the original file F. This is very similar to using a multi-version editor [46], except that the maintenance of multiple versions is done at the file system level.

[4]The current FFS implementation uses the CPP representation in version specifications.

$.[]$		
[]	1024	./
[]	1024	../
[*user*: *tom*]	16233	*newtty.c*
[]	78654	*screen.c*
[*user*: *lisa*]	1024	*test/*
[]	15969	*tty.c*

$=$

.[*user*: *tom*]		
[]	1024	./
[]	1024	../
[]	16233	*newtty.c*
[]	78654	*screen.c*
[]	15969	*tty.c*

$\sqcup$

.[∼*user*: *tom*]		
[]	1024	./
[]	1024	../
[]	78654	*screen.c*
[*user*: *lisa*]	1024	*test/*
[]	15969	*tty.c*

Figure 6: Versioned directories

To express that a file be existent in some configuration only, we use the CPP *#error* directive. The *#error* directive stands for a non-existent file: each *#error* directive in F governed by a feature term S indicates that $F[S]$ is non-existent. We thus add the following FFS operations:

Create. Creating a file $F[S]$, where F was non-existent before, creates F containing an *#error* directive governed by $\sim S$—that is, $F[\sim S]$ is still considered non-existent.

Remove. Removing a file $F[S]$ augments F with an *#error* directive governed by S, such that only $F[\sim S]$ is accessible.

As an example, consider the creation of a file *printer.c*[*data*: *postscript*]. After creation, *printer.c* will contain the lines *#if* ¬(*data* ≡ *postscript*) ... *#error* ... *#endif*—any attempt to read *printer.c*[∼*data*: *postscript*] will fail.

An alternate interpretation of "a file F exists in some specific configuration S only" is "the features of F are $\sim S$". Hence, creation and removal can be used to set and manipulate the features of a file F: To set the features of a file F to S, remove $F[\sim S]$. This operation is called *renaming*:

Rename. Renaming a file F to $F[S]$ is equivalent to removing $F[\sim S]$.

This *tagging* technique is further illustrated when discussing the composition protocol in section 6.2.

5.3 A Versioned File System

Besides versioned files, the FFS provides *versioned directories,* covering state and changes of the entire file system—that is, the whole configuration universe. Basically, a versioned directory has the same format like an ordinary directory, except that each directory entry is associated with a governing feature term.

A directory entry governed by the feature term C is visible only if C is a subset of the selection term S, or $C \sqsubseteq S$. If Tom creates a new file *newtty.c* in his workspace [*user*: *tom*], the *newtty.c* entry in the current directory " . " is governed by the term [*user*: *tom*], as illustrated in figure 6; in Lisa's workspace, that is, the .[*user*: *lisa*] directory version, *newtty.c* is non-existent.

If a versioned directory $D[T]$ is part of the current path, the directory version T affects all contents of the directory, including subdirectories and all files contained therein; any file version $F[S]$ in $D[T]$ will be implicitly read as $F[S \sqcap T]$. Hence, opening a directory .[*os*: *unix*] selects the UNIX variants of all files and subdirectories; all changes applied in a .[*user*: *tom*] directory or below affects Tom's workspace only.

By changing the current directory, users can switch between workspaces and versions. Entering *cd* .[*current*: ⊤]/.[*os*: ∼*dos*] (or, shorter, *cd* [*current*: ⊤]/[*os*: ∼*dos*]) makes sure all subsequent changes apply to the current revision in the non-DOS variants only. As illustrated in figure 7, such

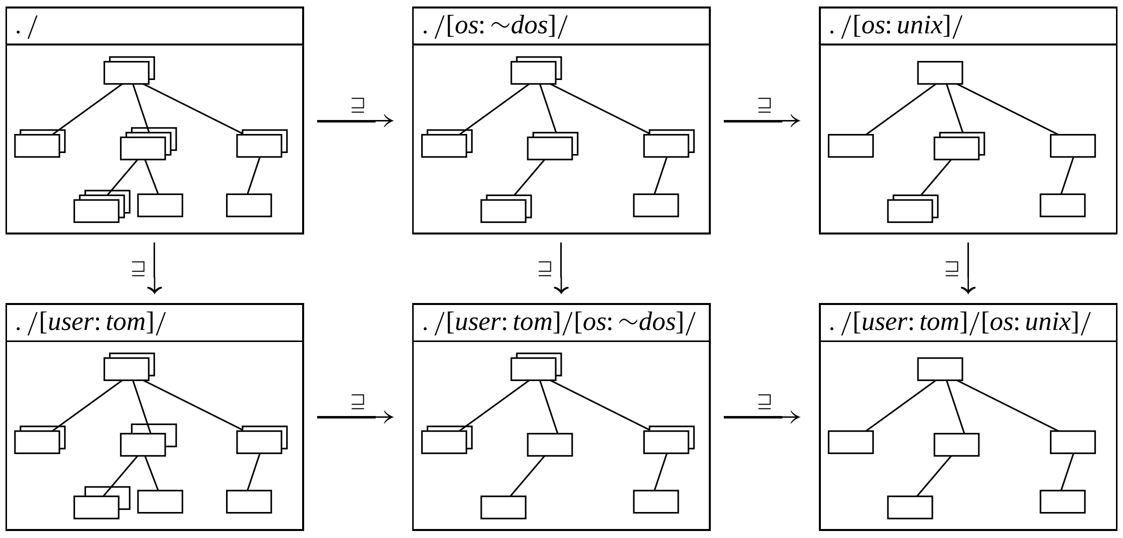


Figure 7: Narrowing the configuration space in the FFS

directory changes may be also be performed incrementally, subsequently narrowing the configuration space as more and more features are specified. Each workspace, variant, or revision is an individual view on the configuration space.

The features of a directory are set like the features of individual files, by removing the complement. Removing the directory version $.[\mathit{tested}\uparrow]$ makes the current directory and all contained items available in the $[\mathit{tested}\colon\top]$ version only. This is convenient for setting the features of all files in one directory or file system subset.

Besides accepting version specifications as parts of the file path, all other features of file systems still apply. The "$..$" directory refers to the second last component from the current path; that is, $\mathit{testdir}/[\mathit{user}\colon\sim\mathit{tom}]/..$ is equivalent to *testdir*. File modes, times, and access restrictions are versioned as well; a file may occur several times in a (versioned) directory, each time with different attributes and a different governing feature term.[5]

Technically, the FFS is realized through a modified NFS server [45], making the FFS available in the network. Version sets are stored as ordinary CPP files, allowing for simple recovery using CPP; a special format is available for binary files [61]. The FFS server keeps version sets in a cache once they are read; changed version sets are also kept in the cache until a superset is requested. Second and later version set accesses are served in constant time. In practice, this means that once a directory version is entered, the FFS server has the same performance as an ordinary NFS server. Should this still be considered too slow, alternate FFS realizations like dynamic system libraries as in *n*-DFS [18] or virtual device drivers as in CLEARCASE [31] could bypass the NFS bottleneck on local file systems and show virtually no difference from direct file access. But still, all files common to several version sets are cached only once, showing the space-saving effects of copy-on-write techniques.

[5]In the current implementation, a file is uniquely identified by its name. While versioning contents and modes of a file exploits a maximum of commonality through the CPP representation, *renaming* a file inhibits a common CPP representation; future FFS implementations should add an extra indirection level here.

In contrast to the virtual file systems realizes in NSE or CLEARCASE, the FFS does not enforce a specific SCM policy. Instead, it provides the basic mechanisms for arbitrary version set access. The specific SCM policy must be realized on top of the FFS by SCM tools that manipulate the version sets. This is in contrast to n-DFS, where the SCM tools are located at the lowest level, realizing repository access as well as basic SCM policies. In practice, we do not expect developers to interact directly with the FFS except for most unusual circumstances. Rather, each developer will work in some private workspace like $.[\mathit{user}\colon \mathit{lisa}, \mathit{current}\colon \top]$ and use SCM tools that realize specific SCM policies by changing the contents of $[\mathit{current}\colon \top]$. This issue is explored further when discussing SCM protocols in section 6.

6 Unified Versioning

In this section, we use the FFS to describe the semantics of the four major *SCM protocols,* taken from Feiler's survey on configuration management models in commercial environments [16]. We show how to implement these protocols on top of the FFS, and we give some ideas on how these protocols can be integrated. The number of SCM protocols an SCM system supports is still an indicator of its flexibility both below and above the protocol layer; it turns out that all four protocols can be realized on top of the FFS, demonstrating the unifying nature of the version set model.

6.1 The Checkin/Checkout Protocol

We begin with the *checkin/checkout protocol,* as realized in the well-known RCS and SCCS tools. As sketched in section 5.2, these SCM tools provide operations to copy revisions from a file system to a repository *(check in)* and retrieve them back again *(check out),* as illustrated in figure 8. Individual developers can *lock* branches of the revision history against further changes.

We now show how to realize the checkin/checkout protocol on top of the FFS. Let each repository be realized through a file $F[R]$, where R is a conjunction of revision constraints as discussed in section 4.2. In order to select an individual revision R_i, we introduce a special feature r_i such that $[r_i\colon \top]$ includes all changes leading up to R_i and excludes all later changes. The term R then contains additional constraints in the form $[r_i\colon \top] \rightarrow \Delta_i \sqcap \nabla_j \sqcap \cdots \sqcap \nabla_k$, where $R_j, \ldots, R_k$ are the revisions immediately derived from R_i; obviously, $R \sqcap [r_i\colon \top] = R \sqcap \Delta_i \sqcap \nabla_j \sqcap \cdots \sqcap \nabla_k = R_i$ holds. The current revision is maintained by a currency constraint $[\mathit{current}\colon \top] \rightarrow [r_i\colon \top]$ in R.

The operations of the checkin/checkout protocol are described below.

Check in. To add a new current revision file F' to the repository R, let i be some unique identifier such that $F[\Delta_i] = F[\nabla_i]$ holds; in other words, i is a yet unused revision number.

1. Check locks. If $F[\mathit{current}\colon \top \sqcap \mathit{locked}\uparrow]$ does not exist, the *current* revision is locked; abort the operation.
2. Store new revision. Overwrite $F[\Delta_i]$ with F'. The new revision is now selected by $F[\Delta_i]$; the old revision set can be accessed as $F[\nabla_i]$.
3. Maintain revision constraints. We require a constraint $C = (\Delta_i \rightarrow \Delta_j \sqcap \cdots \sqcap \Delta_k)$, where $\Delta_j, \ldots, \Delta_k$ are the ancestor revisions. This way, including the δ_i change will automatically include all earlier changes. This is done by renaming F to $F[C]$, such that C becomes a feature of F.
4. Maintain revision selector. Rename F to $F\big[[r_i\colon \top] \rightarrow \Delta_i\big]$, such that accessing $F[r_i\colon \top]$ returns $F[\Delta_i]$.

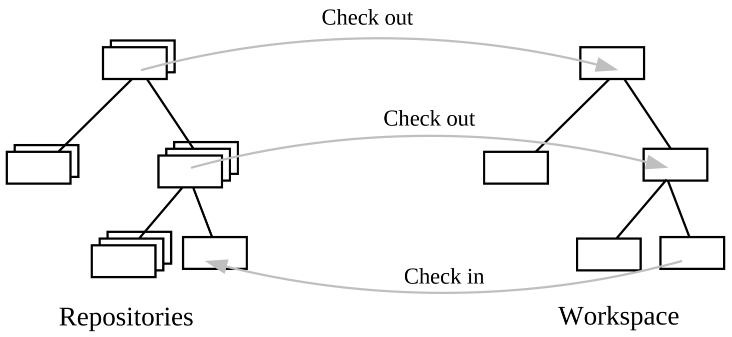


Figure 8: The checkin/checkout protocol

5. Maintain currency. The old currency is invalidated by renaming F to $F[\mathit{current}\uparrow]$. The new currency is established by renaming F to $F\big[[\mathit{current}\colon\top] \rightarrow \Delta_i\big]$.

To add a revision with multiple ancestors, or to add a non-current revision (a *branch*), the constraints are maintained according to (11).

Check out. To check out the current revision, copy $F[\mathit{current}\colon\top]$ to some file F'. To check out some earlier revision R_i, copy $F[r_i\colon\top]$ to some file F'.

Lock. To lock any revision R_i by a user u, first check whether the revision is locked by someone else; If $F[r_i\colon\top \sqcap \mathit{locked}\colon{\sim}u]$ exists, abort the operation. Otherwise, rename $F[r_i\colon\top]$ to $F[r_i\colon\top \sqcap \mathit{locked}\colon u]$, such that $F[r_i\colon\top]$ exists only in a $[\mathit{locked}\colon u]$ version.

Unlock. To unlock any revision R_i locked by a user u, rename $F[r_i\colon\top \sqcap \mathit{locked}\colon u]$ to $F[r_i\colon\top]$.

The *check in* operation is quite complex here, so let us illustrate it by an example. Let F be a repository of revisions $R_0, \ldots, R_6$, as shown in figure 4; let R_5 be the current revision. Hence, the file F exists as $F[R \sqcap (\mathit{current}\colon\top \rightarrow \Delta_6)]$, where R is defined according to (10). Let us now check in a new revision R_7. After step 2, the new version is accessed by $F[\Delta_7]$; the “old” repository is accessible as $F[\nabla_7]$; the differences are enclosed in *#if* $\Delta_7 \ldots$ *#endif* or *#if* $\nabla_7 \ldots$ *#endif*. But now, selecting an older revision R_i returns a non-singleton version set, as $[r_i\colon\top]$ implies neither Δ_7 nor ∇_7. This is handled in step 3: By changing R to $R' = R \sqcap (\Delta_7 \rightarrow \Delta_6) \sqcap [r_7\colon\top]$, selecting $F[r_5\colon\top]$ excludes the Δ_7 change, because $R \sqsubseteq ([r_5\colon\top] \rightarrow \nabla_6)$ and hence $R' \sqcap [r_5\colon\top] \sqsubseteq (\Delta_7 \rightarrow \Delta_6) \sqcap \nabla_6 \sqsubseteq \nabla_7$ holds. The remaining steps 4 and 5 ensure that both $F[r_7\colon\top]$ and $F[\mathit{current}\colon\top]$ return $F[\Delta_7]$.

6.2 The Composition Protocol

The *composition protocol* extends the checkin/checkout protocol with the notions of configurations and consistency. First, a set of components is composed; then, for each component, a version is selected, resulting in a bound consistent configuration, as shown in figure 9. After composition and selection have taken place, the selected components are maintained as in the checkin/checkout protocol; each component has its individual repository.

The composition is usually no more than a simple enumeration of components, obtained by refining dependency relationships[6]; the selection and identification schemes are mostly subsumed

[6] See [63] for a discussion of how to represent and version relationships.

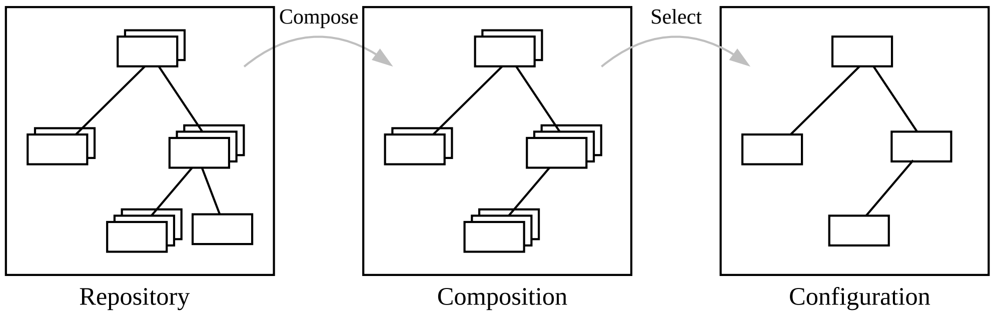


Figure 9: The composition protocol

by feature logic.

To realize the composition protocol, the configurations are maintained in the current directory. The current directory “ . ” records which versions of which components are part of the configuration.

Here are the operations of the composition protocol:

Tag. To assign an attribute T to a file F, rename F to $F[T]$ (or remove $F[\sim T]$). To remove the attribute, make sure that $F[\sim T]$ does not exist, and then rename $F[T]$ to F.

Compose. To compose a set of components, let T be a feature term identifying the composition. If the composition already exists, just enter the directory version $.[T]$. Otherwise, select an originating version $.[S]$ with $S \sqsupseteq T$. In the subset $.[S]/[T]$, set up the configuration by adding or removing files as required.

Select. To make the configuration in $.[T]$ bound, refine T until each component occurs in one version only (unless T was already chosen such that the configuration is bound). This refinement process is best done by an interactive tool that also ensures configuration consistency [61].

Composition and selection are realized most efficient if T is a simple feature term, as stated in proposition 9; a disjunction of configuration rules, as in existing SCM systems, is also handled efficiently.

The single difficult point is to check consistency for ambiguous configurations, as discussed in section 3.2. In theory, we can easily construct examples where each possible configuration must be separately checked for consistency, resulting in a combinatorical explosion and exponential complexity. In practice, we do not expect this to be a problem, due to the principles of *low coupling* and *high cohesion.* Low coupling confines changes to some function or module, leaving the interface intact. This means that the ambiguity has no effect on other components and can thus be factorized out in consistency checking.

On the other hand, high cohesion between functions or modules means that each change implies several other changes: choosing one component version determines the versions of all other components, narrowing the configuration space such that only few configurations remain. Whether these properties apply to today’s software systems and how they affect their configurability is an open issue.

6.3 The Long Transaction Protocol

The *long transaction protocol* is centered around the notion of a *workspace,* as discussed in section 4.3 and realized in Sun’s *Network Software Environment* (NSE) [11].

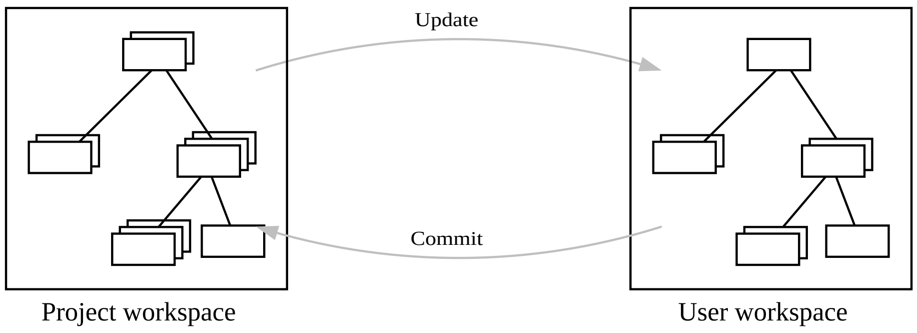


Figure 10: The long transaction protocol

To realize the long transaction protocol on top of the FFS, we use the following setting. Each user u is assigned an individual variant of the project top-level directory, identified by $.[\mathit{user}\colon u]$. The common project state is identified by $.[\mathit{user}\colon \mathit{project}]$, such that it is disjoint from any user's workspace; we call it the *project workspace.* As shown in figure 10, users synchronize their work by propagating changes through the project workspace.

Each workspace has its own revision history. This is realized as in the checkin/checkout protocol, with the current revision being accessed directly through the FFS. Hence, each user u usually works in his workspace on the current revision(s) by entering $.[\mathit{user}\colon u, \mathit{current}\colon \top]$. Entire workspaces can also be versioned.

Some realizations of the long transaction protocol use a conservative strategy and thus rely on component or workspace locking [16]. Our setting assumes an optimistic cooperation strategy and thus the existence of *change integration* tools. Several change integration algorithms are known, either text-based [5], syntax-based [57], or semantic-based [6]; for our purposes, these algorithms must be extended to handle version sets [61].

The operations of the long transaction protocol are as follows:

Originate. To create a new workspace for a user u, rename $.[\mathit{user}\colon \mathit{project}]$ to $.[\mathit{user}\colon \{\mathit{project}, u\}]$, thus (virtually) copying the project workspace to the user's workspace and making it accessible to u.

Update. To propagate changes from the project workspace $.[\mathit{user}\colon \mathit{project}]$ to a user's workspace $.[\mathit{user}\colon u]$, determine the r_i such that $U = .[\mathit{user}\colon u, r_i\colon \top] = .[\mathit{user}\colon \mathit{project}, r_i\colon \top]$ is the common origin of both workspaces. Integrate the changes between the two workspaces, using U as base, and store the result in the workspace of user u.

Commit. To commit all changes from a user workspace to the project workspace, first update the user workspace, as described above. Then create a new current revision of the project workspace containing a (virtual) copy of the user's workspace.

Here is an example of using the long transaction protocol. Let Tom and Lisa each work in their individual workspaces $.[\mathit{user}\colon \mathit{tom}]$ and $.[\mathit{user}\colon \mathit{lisa}]$. Both have made changes to the current revision r_7 of file *tty.c*. Lisa is the first to commit her changes. As no other changes to *tty.c* were made since her last update, a new revision r_8 of the project workspace is created, containing Lisa's changes to *tty.c*. When Tom updates his workspace before his next commit, he must integrate Lisa's changes with his changes, using revision r_7 as a base. The integration is then committed, creating a new revision r_9 of the project workspace incorporating both Lisa's and Tom's changes.

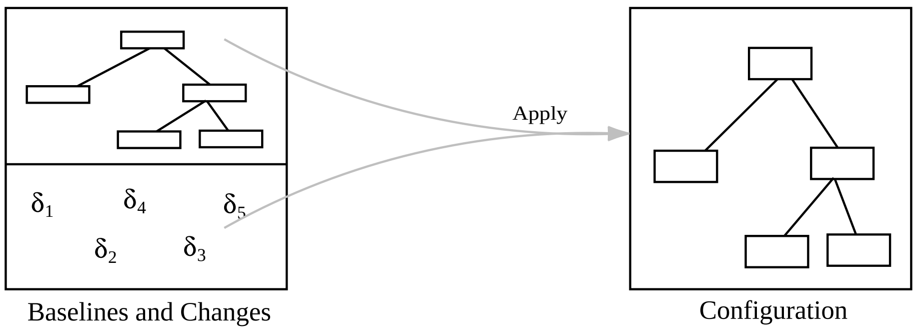


Figure 11: The change set protocol

6.4 The Change Set Protocol

In section 4.1, we have already discussed the difference between version-oriented and change-oriented versioning. In the *change set protocol,* logical *changes* are the primary objects of interest; versions are merely the product of applying *change sets* to a baseline, as shown in figure 11. Change-oriented versioning provides a natural link to *change requests,* as they originate from the SCM process; each configuration can be identified by the incorporated changes.

Our revision concept, as discussed in section 4.1, already assumes that revisions are created by applying changes to an ancestor revision; through appropriate revision constraints, users can denote revisions by specifying change sets as well as by giving revision numbers, as discussed in section 6.1.

Here are the operations of the change set protocol:

Change. To create a change δ_i of a file F, create a new version $F[\Delta_i]$ and change it as desired. The file F may also be a file system subset, such that changes to several files become part of δ_i. If δ_i implies other changes $\delta_j, \dots, \delta_k$, rename $F[\Delta_i]$ to $F[\Delta_i \sqcap \Delta_j \sqcap \cdots \sqcap \Delta_k]$.

Apply. To apply a change set $\delta_i, \dots, \delta_j$ to an arbitrary baseline $F[\nabla_k]$, access $F[\Delta_i \sqcap \cdots \sqcap \Delta_j \sqcap \nabla_k]$. If this version does not exist (because some of the changes are mutually exclusive), create it by integrating the changes, as discussed in section 6.3.

In contrast to the version-oriented protocols, the change-oriented protocol makes extensive use of change integration. Version repositories are thus structured by mutual exclusion rather than implication: conflicting changes δ_i and δ_j are indicated by a constraint $(\nabla_i \sqcup \nabla_j)$. Just like in version-oriented protocols, arbitrary sets of changes, variants, and components can be specified and examined.

7 Performance and Complexity

Having shown how individual protocols are realized on top of the FFS, we can now discuss their complexity issues. At first, this may sound surprising: Obviously, each individual protocol has already be realized efficiently in some existing SCM system, so why bother? First, we must show that this efficiency is not endangered by our formal base—in fact, the efficiency is due to a number of constraints on the organization of features, which we must identify. Second, having understood how these constraints make SCM protocols efficient, we can turn to the problem of *integrating* SCM protocols.

7.1 What is it that Makes Today's SCM Protocols so Efficient?

In proposition 5, we have stated that deciding the inconsistency of a feature term (i.e., deciding whether $S = \bot$ holds) is an NP-complete problem. Several of our SCM principles rely on deciding inconsistency, which should result in exponential complexity. So, why isn't this so in existing SCM systems? Basically, there are three causes, each reducing complexity by imposing constraints on the general problem.

Simplification. In existing SCM systems, components are either identified or selected using simple feature terms; the general case of having non-simple feature terms for both identification and selection never occurs. Hence, the preconditions for proposition 9 apply—whether a version is member of the selection or not can simply be decided by evaluating the selection term with the values furnished in the identification term, or vice versa. This makes the selection operations in section 6 very efficient.

Implication chains. A second issue is specific to revision handling. Applying the revision constraint scheme from section 4.1, revisions are identified by long chains of implications like $(\Delta_{42} \rightarrow \Delta_{41}) \sqcap (\Delta_{41} \rightarrow \Delta_{40}) \sqcap \ldots$. A simple method to decide consistency of such an implication chain R with a selection term S works as follows: for each $\Delta_i \sqsupseteq S$, replace all $(\Delta_i \rightarrow \Delta_j)$ by Δ_j and repeat the process for Δ_j. Likewise, for each $\nabla_i \sqsupseteq S$, replace all $(\Delta_j \rightarrow \Delta_i) = (\nabla_i \rightarrow \nabla_j)$ by ∇_j and repeat the process for ∇_j. This scheme allows for efficient selection from "classical" revision histories, as realized in today's SCM systems.

Orthogonality. As stated in proposition 7, if two feature terms S and T are consistent and have no common features or variables, their intersection is consistent as well—which can be checked in linear time. This property makes the creation of new versions efficient, since they are identified by new features which are orthogonal to all existing ones. Furthermore, orthogonality simplifies the separation of concerns. For instance, maintenance of revisions and variants is dramatically simplified as soon as revision features and variant features do not interact with each other—for example, by placing a CPP file under RCS control.

To conclude: as long as all versions are identified by simple feature terms, as long as we stick to revision histories, as long as we keep revisions, workspaces, and variants separated from each other, we can realize efficient SCM protocols. This is the status quo. But does our common foundation also realize them efficiently?

7.2 A Case Study

To see how ICE handles the major SCM protocols, we have implemented the three methods stated above as deductive shortcuts besides full-fledged feature unification. As a case study, we have chosen the GNU MAKE program, which is publicly available in 17 revisions named 3.55 to 3.74.[7] From the GNU MAKE distribution, we have considered a single file named *commands.c*; this file happened to be modified in each revision. We wanted to know how ICE performs in creating a repository from the 17 revisions of *commands.c*, compared to well-known tools like RCS and SCCS; to see the effects of the deductive shortcuts, we also made ICE run without deductive shortcuts and rely on feature unification alone.

[7]The recent GNU MAKE distribution as well as differences to earlier revisions are available from the GNU FTP server `ftp://prep.ai.mit.edu/pub/gnu/`.

commands.c[]

```
for (d = enter_file(".SUFFIXES")→deps; d ≢ 0; d = d→next)
  {
#if d370
      unsigned int slen = strlen(dep_name(d));
#else
      unsigned int len = strlen(file→name);
#endif
#if d374
      if (len > slen ∧ ¬strncmp(dep_name(d), name + (len − slen), slen))
#elif d370
      if (len > slen ∧ ¬strncmp(dep_name(d), name + len − slen, slen))
#else
      if (len > slen ∧ streq(dep_name(d), file→name + len − slen))
#endif
        {
#if d370
          file→stem = savestring(name, len − slen);
#else
          file→stem = savestring(file→name, len − slen);
#endif
          break;
        }
  }
if (d ≡ 0)
  file→stem = "";
```

Figure 12: A multi-revision file

In figure 12, we see an excerpt of the version set *commands.c*, incorporating all 17 revisions. We see that the change *d370* replaced *file→name* by *dep_name*(*d*) and that change *d374* introduced a parenthesized subexpression. In this excerpt, there is a maximum number of two features that govern code pieces, making the excerpt quite readable. But *commands.c* also contains code pieces governed by four features, which is a little harder to understand—but still an alternative to a set of mutual DIFF runs. From the version set *commands.c*, ICE can extract individual revisions in linear time—due to the efficiency of simplification, selecting a specific revision does not take more time than running the appropriate RCS, SCCS, or CPP command. All results would apply just as well, had we chosen features for identifying workspaces or variants instead of changes.

While reading individual versions easily competes with existing SCM systems, the creation of the repository showed up some unexpected problems. In figure 13, we have listed the execution times for each checkin process in ICE, as well as the checkin times for RCS and SCCS. Initially, we had no deductive shortcuts in ICE, relying on NP-complete feature unification alone, and execution time grew beyond all limits, as shown in figure 13. But even with deductive shortcuts enabled, ICE checkin time still grows with the number of revisions, while the RCS and SCCS checkin times remain fairly constant. The difference with ICE is that ICE compares entire version sets when determining a new compact representation, as discussed in section 5.1; in our example, each new revision is compared with the entire repository, and the ICE inference engine must determine more and more governing feature terms as the number of revisions grows. This is in contrast to RCS and SCCS,

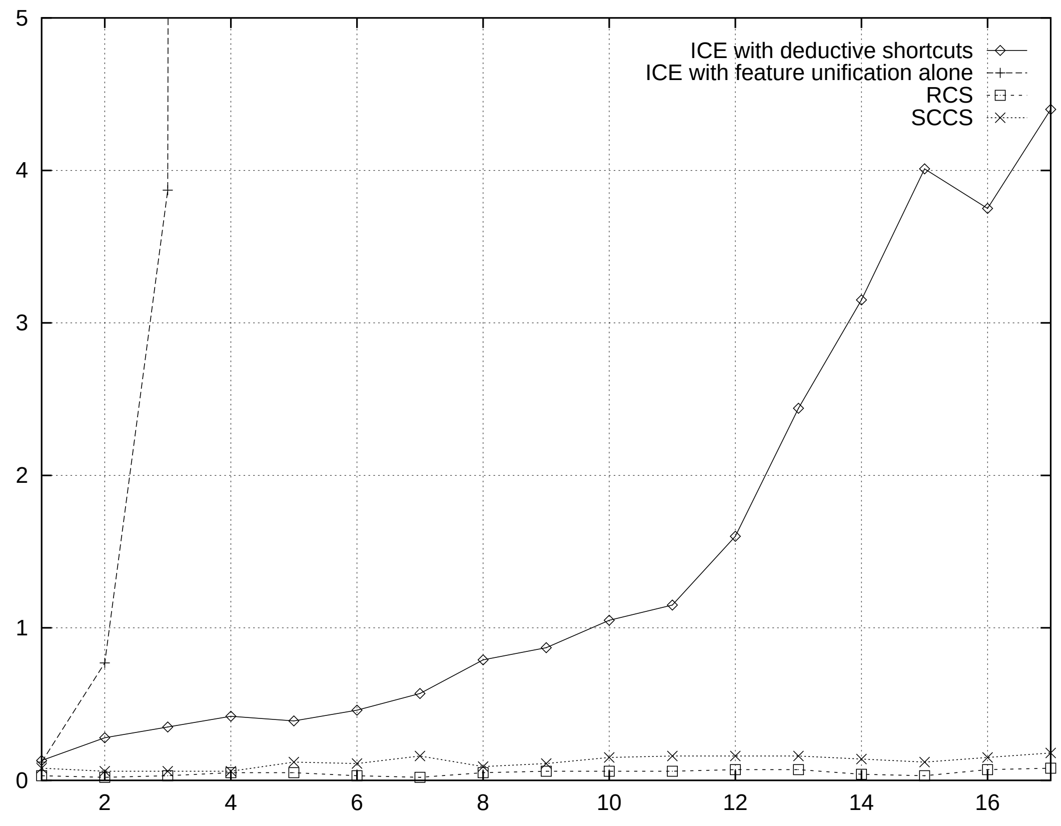


Figure 13: Revision checkin times (in seconds) for ICE, RCS, and SCCS

which compare the new revision with the previous revision only.

The checkin problem could easily be solved by realizing the RCS/SCCS approach and comparing only the latest revisions. The data above shows that ICE is quite efficient when comparing small revision sets; hence, the use of feature logic as a common SCM foundation and the feasibility of a common SCM primitive layer is unquestioned. But if we have multiple variants in multiple revisions, all sharing some common code, which are the "latest" revisions ICE should compare? And to which extent should variants be compared?

The central problem here is the integration of *variance* with other SCM concepts. Workspaces that imply certain variants, variants that imply certain revisions, changes that apply to certain variants only, introduce disjunctions into revision constraints and thus make the deduction process overly complex. Such interferences are indicators of poor structure of the configuration space, showing low coherence and strong coupling between configuration threads. Although these interferences can be uncovered by mathematical concept analysis of configuration structures [28], restructuring software in order to eliminate them is still at its beginning [53]. Future research and experience will show how far non-orthogonal variance can be allowed to interfere with other SCM concepts and how much of the resulting complexity is tolerable in practice. We see that while the realization of an existing SCM protocol imposes no special problems, the integration of SCM concepts remains an open issue.

8 Conclusion

The future of automated SCM lies in a clear separation of primitives, protocol, and policy, based on a clear semantic foundation. We have proposed feature logic and version sets as such a SCM foundation. Version sets integrate and unify current SCM versioning models and provide a well-defined semantics for defining higher SCM layers. Feature logic is powerful enough not to endanger flexibility at higher SCM layers, and yet sufficiently specialized to describe how features propagate in the SCM process.

Our implementation of the version set model in ICE has shown that this foundation has numerous user-visible benefits. Through the feature deduction mechanisms, ambiguity is tolerated at all SCM layers; sets rather than objects are the primary items of interest. The SCM process is not constrained by process-specific decisions in lower SCM layers. All major SCM protocols can be realized efficiently on top of a SCM primitive layer like the FFS. These features make ICE an environment adapting to its users and their process, instead of vice versa.

Besides refining, extending, and evaluating the ICE implementation, especially at the protocol and policy levels, our future work will focus on three subjects.

Efficient integration of SCM concepts. We have seen that each of the four major SCM models can be realized efficiently on top of the version set model. We also have identified complexity problems with non-orthogonal SCM concepts, especially variance. Based on further experience with the FFS and the underlying deduction engine, we want to investigate how far integration of SCM concepts can go without endangering efficiency. Furthermore, we want to see which integrated SCM protocols are feasible, how they can be realized on top of the FFS, and how far the SCM process is determined by these protocols.

Versioned component relations. While our model supports versioned components, it has no notions on relationships between these components. What is required is a means to model versioned component relations—or relations between component versions. Generally, we plan to extend the version set model such that features represent relationships between version sets. $1{:}m$ and $1{:}n$ relationships are modeled through non-functional features called *roles* [50]. This extension will introduce and unify versioning concepts in graph-structured applications such as computer-aided design (CAD) [26], or graph-based software development environments [13, 48]; first results are given in [63].

Support of the SCM process. On the conceptual level, we must find out if and how SCM processes might be formalized using the version set model and whether SCM tool behaviour may be verified against the SCM process. We imagine organizing the SCM process entirely by manipulating component features—changing their state from *proposed* via *tested* to *released*; SCM procedures might be modeled by pre- and post-conditions specified as feature terms. Unfortunately, there is no true methodology yet how components and versions should be attributed with feature terms; experiences from other attribute-oriented SCM systems or faceted classification [41] might help here. Eventually, we hope to model the entire SCM process through operations on version sets denoted by feature logic, providing a uniform semantic foundation for all SCM layers.

ICE and the FFS were developed as part of the NORA project[8] which aims at utilizing inference technology in software tools. ICE and the FFS as well as related technical reports can be accessed

[8]NORA is a figure in Henrik Ibsen's play "A Dollhouse". Hence, NORA is no real acronym.

through the ICE WWW page, `http://www.cs.tu-bs.de/softech/ice/`, and via anonymous FTP from `ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/ice/`.

Acknowledgments

Many thanks to all who have made ICE possible through contributing to the ICE implementation or by making program sources and tools available. Lars Düning implemented the CPP representation using GNU DIFF. Olaf Pfohl built the FFS server on top of a public domain NFS server. Marc Ziehmann implemented Smolka's feature unification algorithm. Dirk Babel, Michael Brandes, and Andreas Mende realized the higher layers of ICE. Finally, we thank the anonymous reviewers for their useful and constructive comments.

References

[1] Adams, P., and Solomon, M. An overview of the CAPITL software development environment. In *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 workshops* (Seattle, Washington, Oct. 1995), J. Estublier, Ed., vol. 1005 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–34.

[2] Aït-Kaci, H. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science 45* (1986), 293–351.

[3] Aït-Kaci, H., and Nasr, R. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming 1986*, 3 (1986), 186–215.

[4] Aït-Kaci, H., and Podelski, A. Towards a meaning of LIFE. In *Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming* (Passau, Germany, Aug. 1991), J. Maluszyński and M. Wirsing, Eds., vol. 528 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 255–274.

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

[6] Binkley, D., Horwitz, S., and Reps, T. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology 4*, 1 (Jan. 1995), 3–35.

[7] 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.

[8] Brown, A., Dart, S., Feiler, P., and Wallnau, K. The state of automated configuration management. Tech. Rep. CMU/SEI-ATR-91, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Sept. 1991.

[9] Conradi, R., and Tryggeseth, E. Versioning models. In *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 workshops* (Seattle, Washington, Oct. 1995), J. Estublier, Ed., vol. 1005 of *Lecture Notes in Computer Science*, Springer-Verlag, p. 80.

[10] Conradi, R., and Westfechtel, B. Version models for software configuration management. Tech. Rep. AIB 96-10, RWTH Aachen, Germany, Oct. 1996.

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

[12] Dart, S. Concepts in configuration management systems. In *Proc. 3rd International Workshop on Software Configuration Management* (Trondheim, Norway, June 1991), P. H. Feiler, Ed., ACM Press, pp. 1–18.

[13] Engels, G., Lewerentz, C., Nagl, M., Schäfer, W., and Schürr, A. Building integrated software development environments—Part 1: Tool specification. *ACM Transactions on Software Engineering and Methodology 1*, 2 (1992), 135–167.

[14] Estublier, J. Process session. In *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 workshops* (Seattle, Washington, Oct. 1995), J. Estublier, Ed., vol. 1005 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 136–137.

[15] Estublier, J., and Casallas, R. The Adele configuration manager. In *Configuration Management*, W. F. Tichy, Ed., vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994, ch. 4, pp. 99–133.

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

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

[18] Fowler, G., Korn, D., and Rao, H. *n*-DFS: The multiple dimensional file system. In *Configuration Management*, W. F. Tichy, Ed., vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994, ch. 5, pp. 135–154.

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

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

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

[22] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Guide to Software Configuration Management Plans.* New York, 1990. ANSI/IEEE Standard 828-1990.

[23] The International Organization for Standardization and The International Electrotechnical Commission. *Programming Languages—C*, Dec. 1990. ISO/IEC International Standard 9899:1990 (E).

[24] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[25] Kaplan, R. M., and Bresnan, J. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed. MIT Press, Cambridge, Mass., 1982, pp. 173–381.

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

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

[28] Krone, M., and Snelting, G. On the inference of configuration structures from source code. In *Proc. 16th International Conference on Software Engineering* (Sorrento, Italy, May 1994), IEEE Computer Society Press, pp. 49–57.

[29] Lacroix, M., and Lavency, P. Preferences: Putting more knowledge into queries. In *Proc. of the 13th International Conference on Very Large Data Bases* (Brighton, 1987), P. M. Stocker and W. Kent, Eds., pp. 217–225.

[30] Lampen, A., and Mahler, A. An object base for attributed software objects. In *Proc. of the Fall '88 EUUG Conference* (Cascais, Oct. 1988), pp. 95–105.

[31] Leblang, D. B. The CM challenge: Configuration management that works. In *Configuration Management*, W. F. Tichy, Ed., vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994, ch. 1, pp. 1–37.

[32] Lie, A., Conradi, R., Didriksen, T. M., Karlsson, E.-A., Hallsteinsen, S. O., and Holager, P. Change oriented versioning in a software engineering database. In *Proc. 2nd International Workshop on Software Configuration Management* (Princeton, New Jersey, Oct. 1989), W. F. Tichy, Ed., ACM Press, pp. 56–65.

[33] Mahler, A. Variants: Keeping things together and telling them apart. In *Configuration Management*, W. F. Tichy, Ed., vol. 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994, ch. 3, pp. 39–69.

[34] Martin, U., and Nipkow, T. Boolean unification—The story so far. In *Unification*, C. Kirchner, Ed. Academic Press, London, 1990, pp. 437–455.

[35] Miller, W., and Myers, E. A file comparison program. *Software—Practice and Experience 15*, 11 (1985), 1025.

[36] Munch, B. P., Larsen, J.-O., Gulla, B., Conradi, R., and Karlsson, E. A. Uniform versioning: The change-oriented model. In *Proc. 4th International Workshop on Software Configuration Management (Preprint)* (Baltimore, Maryland, May 1993), S. Feldman, Ed., pp. 188–196.

[37] Nebel, B. *Reasoning and Revision in Hybrid Representation Systems*, vol. 422 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1990.

[38] Nebel, B., and Smolka, G. Representation and reasoning with attributive descriptions. In *Sorts and Types in Artificial Intelligence* (Eringerfeld, 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.

[39] NICKLIN, P. Managing multi-variant software configurations. In *Proc. 3rd International Workshop on Software Configuration Management* (Trondheim, Norway, June 1991), P. H. Feiler, Ed., ACM Press, pp. 53–57.

[40] PLOEDEREDER, E., AND FERGANY, A. The data model of the configuration management assistant. In *Proc. 2nd International Workshop on Software Configuration Management* (Princeton, New Jersey, Oct. 1989), W. F. Tichy, Ed., ACM Press, pp. 5–13.

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

[42] REICHENBERGER, C. Orthogonal version management. In *Proc. 2nd International Workshop on Software Configuration Management* (Princeton, New Jersey, Oct. 1989), W. F. Tichy, Ed., ACM Press, pp. 137–140.

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

[44] SACHWEH, S., AND SCHÄFER, W. Version management for tightly integrated software engineering environments. In *Proc. of the 7th international Conference on Software Engineering Environments* (Noordwijkerhout, Netherlands, Apr. 1995), IEEE Computer Society Press.

[45] SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun Network filesystem. In *Proc. of the Summer 1985 USENIX conference* (Portland, Oregon, June 1985), pp. 119–130.

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

[47] SCHMERL, B. D., AND MARLIN, C. D. Designing configuration management facilities for dynamically bound systems. In *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 workshops* (Seattle, Washington, Oct. 1995), J. Estublier, Ed., vol. 1005 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 88–100.

[48] SCHÜRR, A., WINTER, A. J., AND ZÜNDORF, A. Graph grammar engineering with PROGRES. In *Proc. 5th European Software Engineering Conference* (Sitges, Spain, Sept. 1995), W. Schäfer and P. Botella, Eds., vol. 989 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 219–234.

[49] 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.

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

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

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

[53] SNELTING, G. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology 5*, 2 (Apr. 1996), 146–189.

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

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

[56] VAN DER HOEK, A., HEIMBIGNER, D., AND WOLF, A. L. A generic, peer-to-peer repository for distributed configuration management. In *Proc. 18th International Conference on Software Engineering* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, pp. 308–317.

[57] WESTFECHTEL, B. Structure-oriented merging of revisions of software documents. In *Proc. 3rd. SCM* (Trondheim, Norway, June 1991), P. H. Feiler, Ed., ACM Press, pp. 86–79.

[58] WIEBE, D. Object-oriented software configuration management. In *Proc. 4th International Workshop on Software Configuration Management (Preprint)* (Baltimore, Maryland, May 1993), S. Feldman, Ed., pp. 241–252.

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

[60] ZELLER, A. A unified version model for configuration management. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Washington, DC, Oct. 1995), G. Kaiser, Ed., vol. 20 (4) of *ACM Software Engineering Notes*, ACM Press, pp. 151–160.

[61] Zeller, A. Smooth operations with square operators—The version set model in ICE. In *Proc. 6th International Workshop on Software Configuration Management* (Berlin, Germany, Mar. 1996), I. Sommerville, Ed., vol. 1167 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 8–30.

[62] Zeller, A. *Configuration Management with Version Sets.* PhD thesis, Technical University of Braunschweig, Germany, Apr. 1997.

[63] Zeller, A. Versioning software systems through concept descriptions. Computer Science Report 97-01, Technical University of Braunschweig, Germany, Jan. 1997. Submitted for publication.

[64] Zeller, A., and Snelting, G. Handling version sets through feature logic. In *Proc. 5th European Software Engineering Conference* (Sitges, Spain, Sept. 1995), W. Schäfer and P. Botella, Eds., vol. 989 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 191–204.