# Unified Versioning through Feature Logic

ANDREAS ZELLER and GREGOR SNELTING
Technische Universität Braunschweig

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 an 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 (CPP) representation, users can view and edit multiple versions simultaneously, while 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

This article is a revised and extended version of a paper presented at the Fifth European Software Engineering Conference (ESEC 95), in Sitges, Spain, September, 1995. Early descriptions of the revision and workspace concepts (Sections 4.1 and 4.3) were presented by Zeller [1995], and the featured file system (Sections 5.2 and 5.3) was first discussed by Zeller [1996]. This work was supported by the Deutsche Forschungsgemeinschaft, Grants Sn11/1-2 and Sn11/2-2.

Authors' address: Abteilung Softwaretechnologie, Technische Universität Braunschweig, Bültenweg 88, D-38092 Braunschweig, Germany; email: {zeller; snelting}@ips.cs.tu-bs.de; http://www.cs.tu-bs.de/softech/.

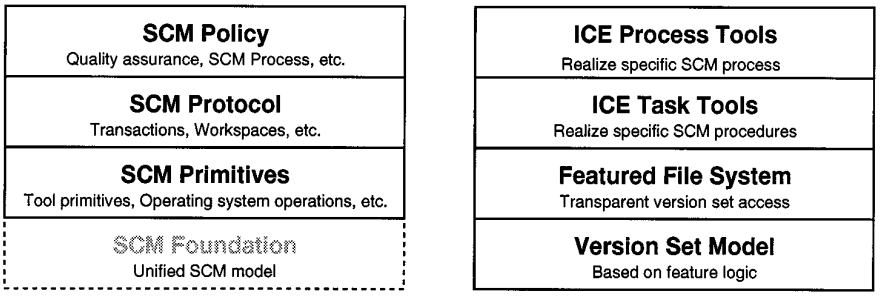


Fig. 1. Federated SCM architecture, as proposed by Brown et al. [1991] (left) and as realized in ICE (right).

configuration management procedures [IEEE 1988; 1990] like *identification* of components and structures, *control* of changes and releases, *status accounting*, or *audit and review*, as well as software-specific tasks [Dart 1991] 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.

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 [Rochkind 1975] or RCS [Tichy 1985] (revision and change control); CPP, the C preprocessor [ISO/IEC 1990] (variant control); or MAKE [Feldman 1979] (manufacture). These days, a new generation has emerged, represented by SCM systems like ADELE [Estublier and Casallas 1994], EPOS [Gulla et al. 1991], or CLEARCASE [Leblang 1994]. These systems provide and integrate support for all SCM aspects through *federated* SCM system architectures [Brown et al. 1991], 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 [Conradi and Westfechtel 1996]. Despite these advances, SCM systems still suffer from three deficiencies:

(1) *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 [Mahler 1994], change propagation across several versions at once [Munch et al. 1993], and consistency checking in abstract (ambiguous) configurations [Schmerl and Marlin 1995].

(2) *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 [Estublier 1995]. At least four diverging SCM models have been identified, each imposing a different SCM process [Feiler 1991]. This is pretty far away from the ideal that an SCM system should adapt to an organization's process.

(3) *Lack of system integration.* Already at the SCM primitive layer, there is considerable disagreement about versioning models [Conradi and Tryggeseth 1995]. 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 [van der Hoek et al. 1996].

In this article, we propose to resolve these deficiencies, using a unified SCM versioning model as a 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 SCM convention to characterize objects by their attributes. Version sets generalize well-known SCM concepts such as components, repositories, workspaces, variant sets, and revision histories. Using *feature logic*, the 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 an 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 C preprocessor (CPP) representation, users can view and edit multiple versions simultaneously, while 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 article 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, that are required for the SCM protocol layer. In Section 5 we turn to practical aspects and

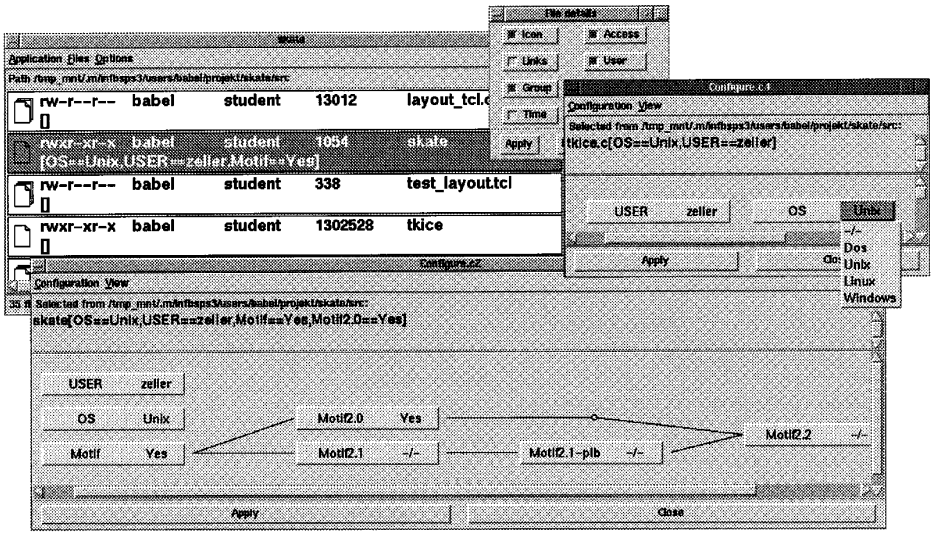


Fig. 2. Exploring the configuration space with the ICE file/configuration browser.

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 toward a more fundamental treatment, still keeping the higher SCM layers in mind.

### 2.1 An 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 stated in the introduction, we identified three key elements of such a foundation:

(1) *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.
(2) *Second foundation: Attributes.* Attribution is one of the few techniques common to the whole SCM area: all known SCM models rely on either versions or changes being 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.
(3) *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 is described 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* [Brachman and Levesque 1984; Nebel 1990; Nebel and Smolka 1989], are used to represent sets of objects by attribute/value combinations, and the semantic analysis of natural language [Kaplan and Bresnan 1982; Kay 1984; Shieber et al. 1983].

In programming languages, attribute/value combinations are used in record structures. Aït-Kaci [1986] was the first to study such structures mathematically, calling them $\psi$-terms. The resulting *$\psi$-term calculus* is the formal foundation of the programming languages LOGIN [Aït-Kaci and Nasr 1986] and LIFE [Aït-Kaci and Podelski 1991], which are similar to PROLOG, but use *feature unification* [Smolka and Aït-Kaci 1990] instead of syntactic unification. In contrast to several description logics, attributes in $\psi$-terms are *functional*: they can have only one value. This is convenient, since objects can be identified by some unique attribute value.

$\psi$-terms have been successfully applied in the context of SCM, notably in the CAPITL system [Adams and Solomon 1995]. 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 $\psi$-terms seem ideal candidates for a unified SCM versioning model, particularly since they have been successfully used

in SCM systems. Unfortunately, in $\psi$-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 an SCM foundation that does not suffer from these restrictions. Boolean operators from *first-order logic* are used in several SCM selection schemes [Estublier and Casallas 1994; Gulla et al. 1991; Mahler 1994; Nicklin 1991; Wiebe 1993]; first-order terms may also be used for identification purposes, using deduction techniques such as *Boolean unification* [Martin and Nipkow 1990] 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, and 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: (1) the Boolean operators and quantifications of first-order logic, in order to express identification and selection schemes; (2) the attribute-oriented formalisms from description logics, denoting how attributes propagate in the SCM process; and (3) the functional attributes of $\psi$-terms, as they uniquely identify objects by their attributes. Such a logic does exist: *feature logic*, as defined by Smolka [1992], 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:

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

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 I, where we denote *variables* by $x$, $y$, $z$; *features* by $f$, $g$, $h$; *constants* by $a$, $b$, $c$; and *feature*

Table I. Syntax of Feature Terms

| 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 iff $T$ holds |
| $\exists x(S)$ | Quantification | There is an $x$ such that $S$ holds |

*terms* by $S$ and $T$.[1] Feature terms are constructed using the well-known Boolean set operations *intersection*, *union*, and *complement*. Each of these set operations may also be interpreted as a 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 let $T = [g: b]$, 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 and de Morgan's law 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 $f: \top$, respectively, $\sim f: \top$ (abbreviated $f \uparrow$). The possibility of specifying complements greatly increases the expressive power of the logic. For example, the term $\sim$[*compiler*: *gcc*] denotes all objects whose feature *compiler* is either undefined or has another value than *gcc*. The term [*compiler*: $\sim$*gcc*] denotes all objects whose feature *compiler* is defined, but with a value other than *gcc*.

A feature term can be interpreted as a representation of the infinite set of all ground (variable-free) terms $T'$ that are *subsumed* by the original term $T$ (i.e., $T \sqsupseteq T'$). 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,

[1]Smolka [1992] 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 Smolka [1992]; they are simple syntactical extensions whose equivalence to simpler operators is shown in Proposition 2.3.1.

$\top \sqsupseteq [fruit: x] \sqsupseteq [fruit: apple] \sqsupseteq [fruit: apple, color: green] \sqsupseteq [fruit: apple, color: green, wormy: no]$, 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 $[os: dos, os: unix]$ is equivalent to $\bot$, the empty set; formally, $[os: dos, os: unix] = [os: [dos, unix]] = [os: \bot] = \bot$ holds. Terms that are equivalent to $\bot$ are called *inconsistent*. Through *feature unification* [Smolka 1992], a constraint-solving technique, one can determine consistency of arbitrary feature terms. For terms without unions and complements, feature unification works similarly 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 2.3.1. *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 [Smolka 1992]:*

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

A feature term is *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 form $\sim a$ 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 they have no common features or variables.

[2]The interpretation of feature terms is formally defined by Smolka [1992].

PROPOSITION 2.3.2. *Every quantifier-free feature term can be rewritten in linear time to an equivalent basic feature term by using the following equivalences [Smolka 1992]:*

$$\sim f\colon S = f \uparrow \sqcup f\colon \sim S, \qquad \sim\bot = \top,$$
$$\sim f \uparrow = f\colon \top, \qquad \sim\top = \bot,$$
$$\sim f \uparrow g = f \uparrow \sqcup g \uparrow \sqcup f \downarrow g, \qquad \sim(S \sqcap T) = \sim S \sqcup \sim T,$$
$$\sim f \downarrow g = f \uparrow \sqcup g \uparrow \sqcup f \uparrow g, \qquad \sim(S \sqcup T) = \sim S \sqcap \sim T,$$
$$\sim\sim S = S, \qquad S \rightarrow T = \sim S \sqcup T,$$
$$S \leftrightarrow T = (\sim S \sqcup T) \sqcap (\sim T \sqcup S).$$

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 2.3.3. *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 [Zeller 1997b]. □

## 2.4 Consistency

We now discuss the notion of *consistency*, stating if feature terms denote empty sets, and we 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 nonempty. A feature term is called incoherent or *inconsistent* if it is not consistent.

PROPOSITION 2.4.1. *Consistency, subsumption, and equivalence of feature terms are linear-time reducible to each other [Smolka 1992]:*

$$S \text{ inconsistent} \Leftrightarrow S \sqsubseteq \bot \Leftrightarrow S = \bot,$$

$$S \sqsubseteq T \Leftrightarrow S \sqcap \sim T \text{ inconsistent},$$

$$S = T \Leftrightarrow S \sqsubseteq T \wedge T \sqsubseteq S.$$

PROPOSITION 2.4.2. *Deciding inconsistency, subsumption, and equivalence of quantifier-free feature terms are co-NP-complete problems [Smolka 1992].*

PROOF. The proof follows from the satisfiability problem of propositional logic, as shown by Smolka [1992]. □

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 2.4.3. *Deciding inconsistency of simple feature terms is of quadratic time complexity [Smolka 1992].*

As transformation of nonsimple feature terms to DNF is NP-complete, time complexity of Smolka's algorithm is exponential in the worst case, complying with Proposition 2.4.2. 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 2.4.3 we have already seen that determining consistency of a simple feature term can be done 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 if and only if (iff) $S$ and $T$ are consistent.

PROPOSITION 2.4.4. *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 [Zeller 1997b]. □

Another efficient algorithm is obtained using principles of *partial evaluation* [Jones et al. 1993]. 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 can thus deduce inconsistency. This proposition holds only if $S$ and $T$ are variable-free.

PROPOSITION 2.4.5. *Let $S$ and $T$ be consistent and variable-free feature terms; let $T$ also be simple, and let $S$ be in basic form. $S \sqcap T$ is inconsistent iff $S \sqcap T$ can be rewritten to $\bot$ using the equivalences*

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

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 $T$ [Zeller 1997b]. □

PROPOSITION 2.4.6. *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 2.3.2, the term $S$ can be rewritten to basic form in linear time, such that Proposition 2.4.5 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 [Zeller 1997b]. □

## 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 an $S' \sqsupseteq S$ that 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 to simplify $S$ afterward. This can be done by adding a few more equivalences to the rewrite system from Proposition 2.4.5.

PROPOSITION 2.5.1. *In $S \sqcap T$, the term $S$ may be further reduced in size by expanding* (1) *with*

$$\begin{aligned} S \sqcap S &= \top \sqcap S, & f \uparrow \sqcap a &= \top \sqcap a, \\ \sim b \sqcap a &= \top \sqcap a, & \sim a \sqcap f\colon T &= \top \sqcap f\colon T \end{aligned} \tag{2}$$

*and subsequent simplification*

$$\begin{aligned} &S \sqcap \bot = \bot, \quad S \sqcap \top = S, \quad S \sqcup \top = \top, \quad S \sqcup \bot = S, \quad \sim \top = \bot, \\ &\bot \sqcap S = \bot, \quad \top \sqcap S = S, \quad \top \sqcup S = \top, \quad \bot \sqcup S = S, \quad \sim \bot = \top. \end{aligned} \tag{3}$$

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 the inconsistency decision and simplification with an example. Consider the term $S \sqcap T$, where $S = [f\colon a, g\colon \{b, \sim c\}]$ and $T = g\colon d$. We can decompose $S$ to $S_1 \sqcap (S_2 \sqcup S_3) = f\colon a \sqcap (g\colon b \sqcup g\colon \sim c)$. For each primitive $S_i$, we check the consistency of $S_i \sqcap T$ and simplify $S_i$ with respect to $T$. Beginning with $S_1 = f\colon a$, 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 $S_2 = g\colon b$, we have $S_2 \sqcap T = g\colon b \sqcap g\colon d$, which can be rewritten as $S_2 \sqcap T = g\colon [b, d] = g\colon \bot = \bot$; $S_2 \sqcap T$ is inconsistent.

Considering $S_3 = g\colon \sim c$, we have $S_3 \sqcap T = g\colon \sim c \sqcap g\colon d = g\colon [\sim c, d] = g\colon d$; 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 $S \sqcap T = S_1 \sqcap (S_2 \sqcup S_3) \sqcap T = S_1 \sqcap (\bot \sqcup \top) \sqcap T = S_1 \sqcap \top \sqcap T = S_1 \sqcap T = [f\colon a] \sqcap T = [f\colon a, g\colon d]$. 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 $S' = [f\colon a]$, since $S \sqcap T = S' \sqcap T$ holds.

Our presentation of feature logic is now complete. In the remainder of this article, we will 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 configuration are modeled by means of set operations, as provided by feature logic.

### 3.1 Versions and Components

In accordance with the SCM standards [IEEE 1988; 1990], we consider the object of interest in SCM as a family of *software products*. Each of these software products breaks down into 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 [Estublier and Casallas 1994], the Context Model [Nicklin 1991], EPOS [Lie et al. 1989], JASON [Wiebe 1993], and SHAPE [Mahler 1994]. 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 that must be satisfied by the selected versions, similar to a classical selection in databases. In conditional inclusion, as exemplified by 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 {*state*: *proposed* , *state*: *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 ~[*operating-system* : *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 ~[*operating-system* : *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 [ *object*: $\top$], the set of all component versions.

We now define the notions of versions and components. A *version set* is any set $V \sqsubseteq$ [*object*: $\top$]. A *version* is a singleton version set, that is, a set $V \sqsubseteq$ [*object*: $\top$] such that $|V| = 1$. A *component* is a set $K \sqsubseteq$ [*object*: $k$], 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 $K \sqsubseteq$ [*object*: $k$] 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 $V_1, V_1, \ldots, V_n$, 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 [*object*: $k$]) 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:

$$printer_1 = [object: printer, print\text{-}language: postscript],$$

$$printer_2 = [object: printer, print\text{-}language: ascii].$$

The *printer* component is then denoted as

$$printer = printer_1 \sqcup printer_2$$
$$= [object: printer, print\text{-}language: \{postscript, ascii\}].$$

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 $S = [print\text{-}language: postscript]$ from *printer* returns $printer_1$, since $printer \sqcap S = (printer_1 \sqcup printer_2) \sqcap S = (printer_1 \sqcap S) \sqcup (printer_2 \sqcap S) = printer_1 \sqcup \bot = printer_1$. Here, $printer_2 \sqcap S = \bot$ 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 $S'$, 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 that contains 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; 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 [Ploedereder and Fergany 1989] or language-specific knowledge [Sachweh and Schäfer 1995; Snelting et al. 1991]. 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 a configuration as the *intersection* of the component features, excluding inconsistent combinations. For instance, we cannot build a 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*] $= \bot$. 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

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

As an example of configuration consistency, consider Figure 3. We see three source components of a text editor, where each component comes in

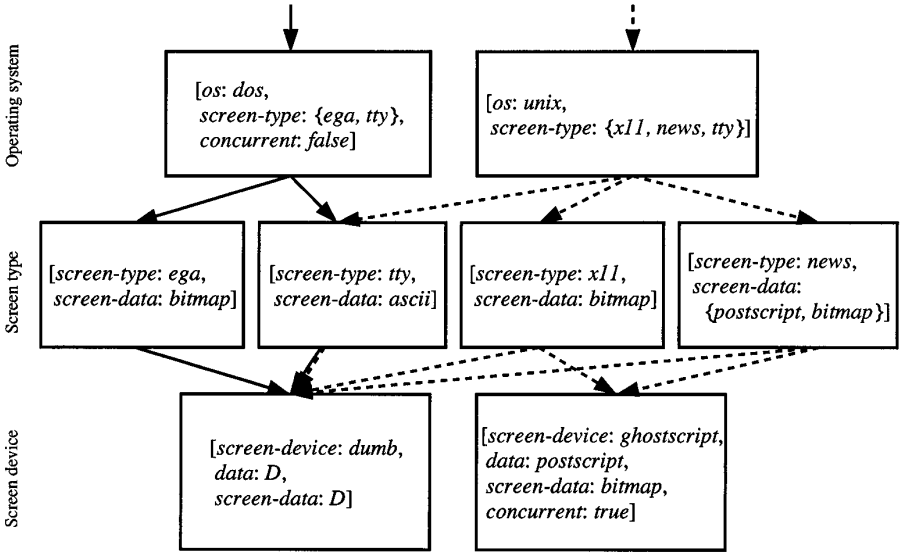


Fig. 3. Consistent configurations in a text/graphic editor.

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,

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

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 solid lines.

As a 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 *ega* and *tty* variants, inheriting the features of its source components. As an alternative, consider the choice [*os*: *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 con-

straints 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 that we have chosen a well-established foundation that addresses all of 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 this is to prefix all independent features $f$ with the component name $k$, resulting in orthogonal features like *tty-author* or *screen-status* [Zeller and Snelting 1995]. A far better alternative is to express this dependency explicitly in feature logic, using implications $[object : k] \rightarrow T$ 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 it 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 $I = \{f_1: \top, f_2: \top, \ldots, f_m: \top\}$ be a feature term denoting independent features, where $f_i \neq object$ 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

$$K_i = [object: k_i] \sqcap K_i' \sqcap K_i''$$

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 (dependent) 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

$$K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n = \mathop{\boxplus_I}_{1 \leq i \leq n} K_i = \mathop{\sqcap}_{1 \leq i \leq n} K_i' \sqcap \left( \left[ object: k_i \right] \rightarrow K_i'' \right).$$

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

PROPOSITION 3.3.1. *Let $S \sqsubseteq [object: s]$, $T \sqsubseteq [object: t]$ denote components, and $I$ denote independent features, as described above. Then,*

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

*holds.*[3]

PROOF. Let $T = [object: t] \sqcap T' \sqcap T''$, as defined above. Then, $U = [object: s] \sqcap (S \boxplus_I T) = [object: s] \sqcap (S' \sqcap T' \sqcap ([object: s] \rightarrow S'') \sqcap ([object: t] \rightarrow T''))$. But, since $[object: s] \sqcap ([object: s] \rightarrow S'') = [object: s] \sqcap (\sim[object: s] \sqcup S'') = [object: s] \sqcap S''$ and $[object: s] \sqcap ([object: t] \rightarrow T'') = [object: s] \sqcap (\sim[object: t] \sqcup T'') = [object: s]$, we have $U = [object: s] \sqcap (S' \sqcap T' \sqcap S'') = ([object: s] \sqcap S' \sqcap S'') \sqcap T' = S \sqcap T' \sqsubseteq S$. □

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 $K_i \sqsubseteq [object: k_i]$, and a term $I$ denoting the independent features, the configuration $C$ is identified by

$$\begin{aligned} C &= \left[object: k_1 \sqcup k_2 \sqcup \cdots \sqcup k_n\right] \sqcap K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n \\ &= \left[object: k_1 \sqcup k_2 \sqcup \cdots \sqcup k_n\right] \sqcap \boxplus_{1 \le i \le n, I} K_i; \end{aligned} \tag{7}$$

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:

$$screen = [object: screen, author: lisa, resolution: \{high, medium\}],$$

$$driver = [object: driver, author: tom, resolution: high].$$

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

$$C = [object: \{screen, driver\}, resolution: high,$$

$$(object: screen \rightarrow author: lisa), (object: driver \rightarrow author: tom)].$$

Besides unifying the dependent features of *screen* and *driver* to *resolution*: *high*, the term $C$ properly selects Lisa's *screen* object and Tom's *driver* object, that is, $C \sqcap [object: screen] \sqsubseteq [author: tom]$ and $C \sqcap [object: driver] \sqsubseteq [author: lisa]$.

We close by defining some *properties* of configurations, following (7). Formally, a *configuration* is a set $C \sqsubseteq [object: c]$, 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 nonzero. A configuration $C$ is called unambiguous or *bound* if it is an aggregation of component versions; formally, $C$ is bound

[3]In Zeller [1995], an alternate definition of the aggregation operator is given, for which (6) does not hold.

if it is a set $C \sqsubseteq [object: c]$ such that $|C| = |c|$. If it is not bound ($|C| > |c|$ holds), a configuration $C$ is called ambiguous, dynamic, or *abstract*.

### 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 $K \sqsubseteq [object: k]$ derived from $n$ source components $K_1, K_2, \ldots, K_n$, and a term $I$ denoting the independent features, $K$ is identified by

$$\begin{aligned} K &= \left[object: k\right] \sqcap K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n \\ &= \left[object: k\right] \sqcap \overline{\boxplus}_{1 \le i \le n,\, I} K_i, \end{aligned} \tag{8}$$

where the explicit setting of the *object* feature removes all implications generated by the aggregation operator—only dependent 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 $[object: os, author: tom]$, $[object: st, author: lisa]$, and $[object: sd, author: john]$, respectively; let the independent features be $I = [author: \top]$. If we derive an *editor* component from a DOS/EGA configuration, it is identified by

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

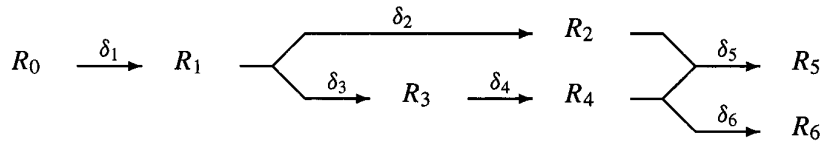


Fig. 4. A revision history.

$$= [\mathit{object}: \mathit{editor}, \mathit{screen\text{-}type}: \mathit{ega}, \mathit{concurrent}: \mathit{false},$$

$$\mathit{screen\text{-}data}: \mathit{bitmap}, \mathit{screen\text{-}device}: \mathit{dumb}, \mathit{data}: \mathit{bitmap}];$$

that is, the *object* features and independent features of the source components are stripped, and all other features are unified. In Zeller [1997a], a MAKE extension is discussed that uses this mechanism to create and reuse 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), and composition (configurations) [Estublier 1995; Reichenberger 1989]. It is a well-known goal of SCM to integrate these dimensions: the concepts of *orthogonal versioning* [Reichenberger 1989] and *three-dimensional versioning* [Estublier 1995], 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.

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 distinguish 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*) [Winkler 1987]. 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 $R_0$, $R_1$, $R_2$, .... Each revision $R_i$ is created by applying a change

(denoted by $\delta_i$) to some originating revisions $R_j, \ldots, R_k$. Consider revision $R_5$, which was created from $R_2$ and $R_4$ by applying the change $\delta_5$.

In version-oriented versioning, each change implies several previous changes. In our example, having the change $\delta_4$ applied requires the previous application of change $\delta_3$; likewise, $\delta_5$ implies all other changes except $\delta_6$. As several configurations are excluded—there simply is no way to include the change $\delta_5$ without also having $\delta_2$ applied—it is quite easy to analyze the impact of a single change. However, version-oriented versioning becomes a problem when changes are largely independent of each other, that is, when one wants a configuration with certain changes applied, but with others excluded. These weaknesses are addressed by *change-oriented versioning* [Harter 1989; Gulla et al. 1991; Munch et al. 1993], 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 $\Delta_i = [\delta_i\colon \top]$ if the change $\delta_i$ has been applied; $R$ is a subset of $\nabla_i = {\sim}\Delta_i = [\delta_i \uparrow]$ if the change $\delta_i$ has *not* been applied. The revision $R_4$ in Figure 4, for instance, would be identified by

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

that is, only the changes $\delta_1$, $\delta_3$, and $\delta_4$ 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 combinations through *revision constraints*.

—*Mutual exclusions.* For 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 $\delta_5$ and $\delta_6$ 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 $\delta_i$ and $\delta_j$ 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. Assume that $R$ contains no version where the change $\delta_9$ has been applied, but not $\delta_7$. We would say that the change $\delta_9$ *implies* the change $\delta_7$. This implication becomes explicit by making $R$ a subset of $\Delta_9 \rightarrow \Delta_7$; in this case, $R \sqcap (\Delta_9 \sqcap \nabla_7) \sqsubseteq (\Delta_9 \rightarrow \Delta_9 \rightarrow \Delta_7) \sqcap (\Delta_9 \sqcap \nabla_7) = (\Delta_9 \rightarrow \Delta_7) \sqcap {\sim}(\Delta_9 \sqcap \Delta_7) = \bot$ holds, effectively excluding the change application.

Generally, to ensure that a change $\delta_i$ implies a change $\delta_j$ in a version set $R$, it suffices to make $R$ a subset of the revision constraint $\Delta_i \rightarrow \Delta_j$.

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 $(\Delta_{211} \rightarrow \Delta_{210}) \sqcap (\Delta_{210} \rightarrow \Delta_{209}) \sqcap \cdots \sqcap (\Delta_2 \rightarrow \Delta_1)$. 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 $(\Delta_2 \rightarrow \Delta_1)$, since $\delta_2$ relies on $\delta_1$, and $R$ must also be a subset of $\nabla_2 \sqcup \nabla_6$, as the changes $\delta_2$ and $\delta_6$ are mutually exclusive. In fact, $R$ can be entirely represented through revision constraints, denoting the complete revision history:

$$R = (\Delta_2 \rightarrow \Delta_1) \sqcap (\Delta_3 \rightarrow \Delta_1) \sqcap (\Delta_4 \rightarrow \Delta_3) \sqcap (\Delta_5 \rightarrow \Delta_2) \qquad (10)$$

$$\sqcap (\Delta_5 \rightarrow \Delta_4) \sqcap (\Delta_6 \rightarrow \Delta_4) \sqcap (\Delta_2 \sqcap \Delta_3 \rightarrow \Delta_5) \sqcap (\nabla_2 \sqcup \nabla_6).$$

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 $\delta_i$, $\delta_j$, $\delta_{\overline{i,j}}$, and $\delta_{\underline{i,j}}$, respectively; the version sets

$$\Delta_i = [\delta_i : \top], \ \Delta_j = [\delta_j : \top], \ \Delta_{\overline{i,j}} = [\delta_{\overline{i,j}} : \top],$$

and

$$\Delta_{\underline{i,j}} = [\delta_{\underline{i,j}} : \top]$$

are defined as usual. Should $R_{\underline{i,j}}$ not exist, then $\Delta_{\underline{i,j}} = \bot$ holds. Now let $C_{i,j}$ be a *formal revision constraint* defined as

$$C_{i,j} = (\Delta_i \sqcup \Delta_j \rightarrow \Delta_{\overline{i,j}}) \sqcap (\Delta_i \sqcap \Delta_j \rightarrow \Delta_{\underline{i,j}}). \qquad (11)$$

If a change $\delta_i$ implies a change $\delta_j$, 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 $\delta_i$ and $\delta_j$ are

mutually exclusive, $C_{i,j} \sqsubseteq (\Delta_i \sqcap \Delta_j \rightarrow \Delta_{\underline{i,j}}) = (\Delta_i \sqcap \Delta_j \rightarrow \bot) = \nabla_i \sqcup \nabla_j$ holds.

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

PROPOSITION 4.2.1. *A revision set $R$ can be represented as the 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:*

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

PROOF. See Zeller [1997b]. □

In our example, the representation in (10) is obtained via (12) and by 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 4.2.1 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 checkin/checkout 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 [*current*: $\top$] that contains the current variants by implying certain revisions. An implication ([*current*: $\top$, *os*: *unix*] $\rightarrow \nabla_5$) ensures that, whenever the current *unix* variant is requested, the change $\delta_5$ 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 us to create, select, and revise arbitrary revision/variant/component combinations as in orthogonal version management [Reichenberger 1989], while still allowing refinement and inheritance as in object-oriented SCM [Wiebe 1993].

### 4.3 Cooperation through Locks and Workspaces

Besides components, variants, and revisions, the 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 of configuration is locked, other developers are excluded from creating new revisions. Using version sets, locks are managed like currency: the set $[locked: \top]$ contains all locked versions, and $\sim[locked: \top] = [locked \uparrow]$ contains the unlocked versions. An SCM system locking a component version $K$ would do so by changing its features such that $K \sqsubseteq [locked: \top]$; any selection of $K$ from $[locked \uparrow]$ 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* [Berliner 1990; Courington 1989]. Rather than preventing concurrent changes, they instead attempt to integrate changes later. The central concept here is the notion of a *workspace*, the individual area of a developer, isolating him or her from changes made by other developers and isolating others from his or her changes.

In our model, a user's workspace is just a variant identified by a feature term $W \sqsubseteq [user: \top]$; that is, $[user: lisa]$ denotes Lisa's workspace, and $[user: tom]$ denotes Tom's workspace. As the *user* feature may have only one value, all workspaces are disjoint; that is, developer Lisa in her workspace $[user: lisa]$ will not see any changes from the $[user: tom]$ workspace. Tom may create new revisions $\Delta_i$ in his workspace or change currency; as his changes are always subsumed by $[user: tom]$, 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 $[user: lisa]$ with the contents of the originating version set $\sim[user: \{lisa, tom\}]$; 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 $[user: tom, os: unix]$, thus confining all changes to his workspace and to the UNIX version. Lisa may wish to work on the current revision only, but to include all variants, thus choosing her workspace as $[user: lisa, current: \top]$. 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 [Lampen and Mahler 1988] or pattern-matching rules as in CLEARCASE [Leblang 1994]. The basic idea is that the first matching rule is applied. An alternate scheme is realized in preference clauses [Lacroix and Lavency 1987], 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*:

$$S_1 \quad \text{and-then} \quad S_2 = \begin{cases} S_1 & \text{if } S_1 \text{ is bound,} \\ S_1 \sqcap S_2 & \text{otherwise,} \end{cases}$$

$$S_1 \quad \text{or-else } S_2 = \begin{cases} S_1 & \text{if } S_1 \neq \bot, \\ S_2 & \text{otherwise,} \end{cases}$$

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, $S = ([\mathit{current}: \top] \text{ or-else } [\mathit{fixed}: \mathit{true}])$ first selects the current version and, if there is none, a "fixed" version; $S = ([\Delta_2, \nabla_3] \text{ and-then } [\mathit{os}: \mathit{unix}])$ selects the second revision and, should this choice be ambiguous, the UNIX variant.

Another practical extension is the use of additional constraints, which can express properties whose mutual consistency cannot be decided in feature logic alone. Useful examples include arithmetic constraints ($\mathit{date} < 1997$) or function interfaces ($\mathit{gcd}: \mathit{int} \times \mathit{int} \rightarrow \mathit{int}$). 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 [Snelting 1991].

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. ICE provides access uses to version sets through a virtual file system called the FFS. The FFS represents version sets in the well-known *#if* $\cdots$ *#endif*

Table II. Translating Feature Terms into CPP Expressions

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

format, which identifies differences between versions. Using the FFS as an 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 CPP representation. Code pieces relevant for certain versions only are enclosed in #*if* $C \cdots$ #*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 even 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 uses CPP terms (i.e., Boolean C expressions) to represent feature terms. In the CPP representation, feature names are expressed as CPP symbols. In Table II we summarize 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

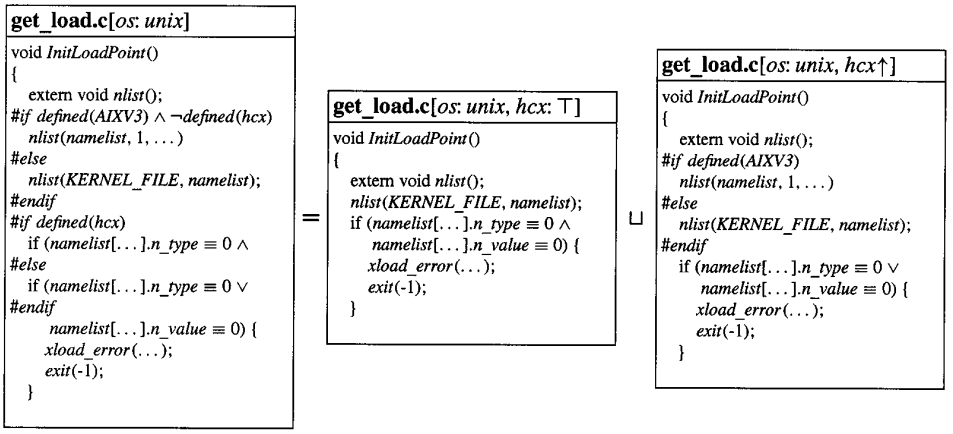


Fig. 5. Version sets represented as CPP files.

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* $C \cdots$ *#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 2.5.1. 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*: $\top$] = *get_load.c*[*os*: *unix*, *hcx*: $\top$] and a non-*hcx* version *get_load.c*[*os*: *unix*][*hcx* $\uparrow$] = *get_load.c*[*os*: *unix*, *hcx* $\uparrow$] (note the simplified CPP expressions). Further selection and refinement are 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 [Miller and Myers 1985] 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

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

where the DIFF algorithm determines a compact representation for the generated version set *get_load.c*[*os*: *unix*]; all governing expressions are simplified with respect to [*os*: *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) subdatabases of the repository [Estublier and Casallas 1994]. 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 [Courington 1989], *n*-DFS [Fowler et al. 1994], and CLEARCASE [Leblang 1994]. 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 a 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 *viewpathing*), sharing common files between several developers.

We have chosen the CPP representation, as introduced above, as the base for a virtual file system in ICE, called the FFS, and by 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:

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

—*Read.* Read access to $F[S]$ is accomplished by selection, as shown; opening the virtual file *tty.c*[*user*: *tom*] gives access to the version set [*user*: *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 multiversion 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 redetermines the differences and CPP directives in the original file $F$. This is very similar to using a multiversion editor [Sarnak et al. 1988], except that the maintenance of multiple versions is done at the file system level.

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

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

—*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* $\neg$(*data* $\equiv$ *postscript*) $\cdots$ *#error* $\cdots$ *#endif*; any attempt to read *printer.c*[$\sim$*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 as 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 shown in Figure 6; in Lisa's workspace, that is, the .[*user*: *lisa*] directory version, *newtty.c* is nonexistent.

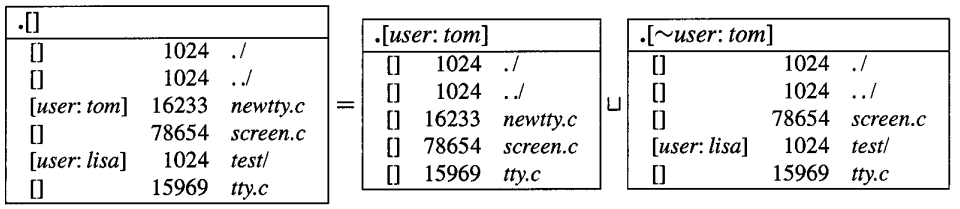


Fig. 6. Versioned directories.

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 affect only Tom's workspace.

By changing the current directory, users can switch between workspaces and versions. Entering *cd* .[*current*: ⊤]/.[*os*: ~*dos*] (or, shorter, *cd* [*current*: ⊤]/[*os*: ~*dos*]) ensures that all subsequent changes apply to the current revision in the non-DOS variants only. As illustrated in Figure 7, such directory changes may 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 .[*tested* ↑] makes the current directory and all contained items available in the [*tested*: ⊤] 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, *testdir*/[*user*: ~*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 [Sandberg et al. 1985], 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 [Zeller 1996]. 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

[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 implementation should add an extra indirection level here.

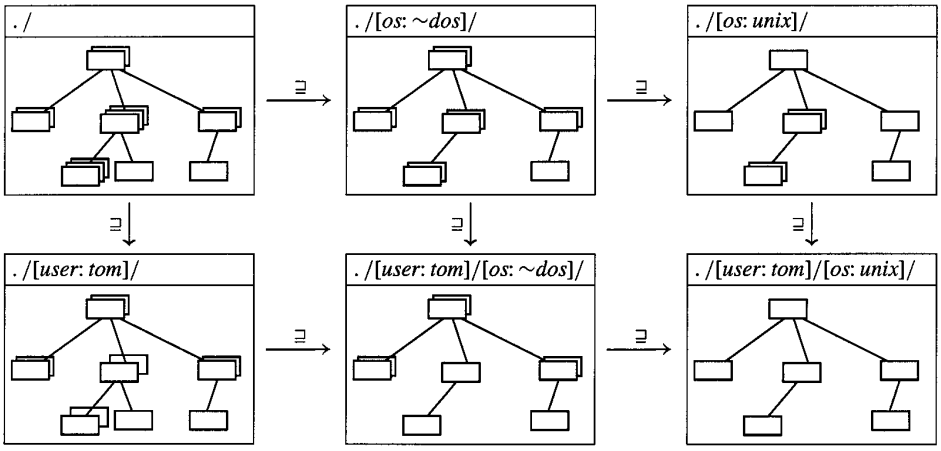


Fig. 7. Narrowing the configuration space in the FFS.

performance as an ordinary NFS server. Should this still be considered too slow, alternate FFS realizations such as dynamic system libraries as in $n$-DFS [Fowler et al. 1994] or virtual device drivers as in CLEARCASE [Leblang 1994] 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.

In contrast to the virtual file systems realized 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 the most unusual circumstances. Rather, each developer will work in some private workspace like .[*user*: *lisa*, *current*: $\top$] and use SCM tools that realize specific SCM policies by changing the contents of [*current*: $\top$]. 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 [1991] survey on configuration management models in commercial environments. We show how to implement these protocols on top of the FFS and give some ideas on how these protocols can be integrated. The number of SCM protocols and 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.

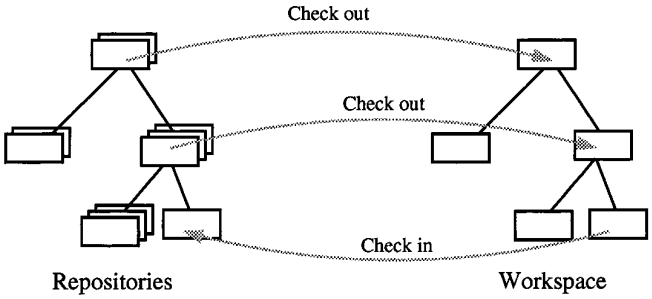


Fig. 8. Checkin/checkout protocol.

### 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 (*checkin*) and retrieve them back again (*checkout*), 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 $[current\colon \top] \rightarrow [r_i\colon \top]$ in $R$.

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

—*Checkin.* 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 $F$ $[current\colon \top \sqcap locked \uparrow]$ does not exist, the *current* revision is locked; abort the operation.
(2) *Store the 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. In this way, including the $\delta_i$ 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 $F[[r_i\colon \top] \rightarrow \Delta_i]$, such that accessing $F[r_i\colon \top]$ returns $F[\Delta_i]$.

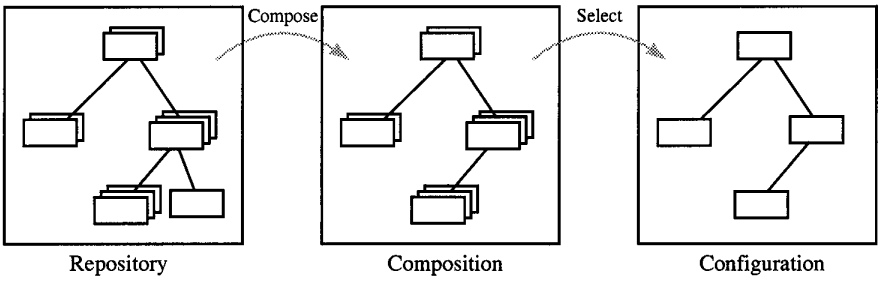


Fig. 9. Composition protocol.

(5) *Maintain currency.* The old currency is invalidated by renaming $F$ to $F[current \uparrow]$. The new currency is established by renaming $F$ to $F[[current: \top] \rightarrow \Delta_i]$.

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

—*Checkout.* To check out the current revision, copy $F[current: \top]$ to some file $F'$. To check out some earlier revision $R_i$, copy $F[r_i: \top]$ to some file $F'$.

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

—*Unlock.* To unlock any revision $R_i$ locked by a user $u$, rename $F[r_i: \top \sqcap locked: u]$ to $F[r_i: \top]$.

The *checkin* operation is quite complex here, so let us illustrate it with 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 $F[R \sqcap (current: \top \rightarrow \Delta_6)]$, 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* $\Delta_7 \cdots$ *#endif* or *#if* $\nabla_7 \cdots$ *#endif*. But, now, selecting an older revision $R_i$ returns a nonsingleton version set, as $[r_i: \top]$ implies neither $\Delta_7$ nor $\nabla_7$. This is handled in step (3): by changing $R$ to $R' = R \sqcap (\Delta_7 \rightarrow \Delta_6) \sqcap [r_7: \top]$, selecting $F[r_5: \top]$ excludes the $\Delta_7$ change, because $R \sqsubseteq ([r_5: \top] \rightarrow \nabla_6)$, and hence, $R' \sqcap [r_5: \top] \sqsubseteq (\Delta_7 \rightarrow \Delta_6) \sqcap \nabla_6 \sqsubseteq \nabla_7$ holds. The remaining steps (4) and (5) ensure that both $F[r_7: \top]$ and $F[current: \top]$ 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 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 [Zeller 1996].

Composition and selection are realized most efficiently if $T$ is a simple feature term, as stated in Proposition 2.4.6; 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 combinatorial 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 a few configurations remain. Whether these properties apply to today's software systems and how they affect their configurability are open issues.

### 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 Envi-*

[6]See Zeller [1997a] for a discussion of how to represent and version relationships.

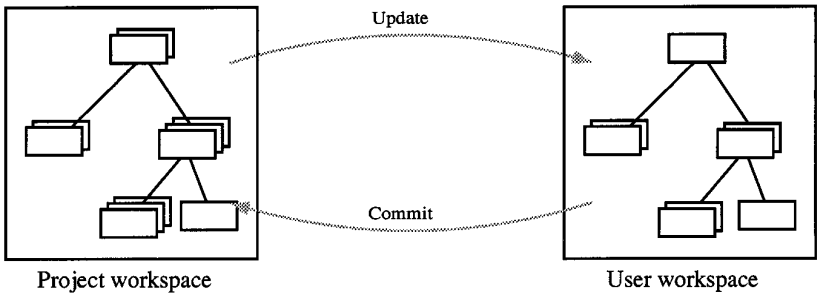


Fig. 10. Long transaction protocol.

*ronment* (*NSE*) [Courington 1989]. 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 .[*user*: $u$]. The common project state is identified by .[*user*: *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 or her workspace on the current revision(s) by entering .[*user*: $u$, *current*: $\top$]. 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 [Feiler 1991]. Our setting assumes an optimistic cooperation strategy and, thus, the existence of *change integration* tools. Several change integration algorithms are known, either text-based [Berliner 1990], syntax-based [Westfechtel 1991], or semantic-based [Binkley et al. 1995]; for our purposes, these algorithms must be extended to handle version sets [Zeller 1996].

The operations of the long transaction protocol are as follows:

—*Originate.* To create a new workspace for a user $u$, rename .[*user*: *project*] to .[*user*: {*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 .[*user*: *project*] to a user's workspace .[*user*: $u$], determine $r_i$ such that $U$ = .[*user*: $u$, $r_i$: $\top$] = .[*user*: *project*, $r_i$: $\top$] 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 .[*user*: *tom*] and .[*user*: *lisa*].

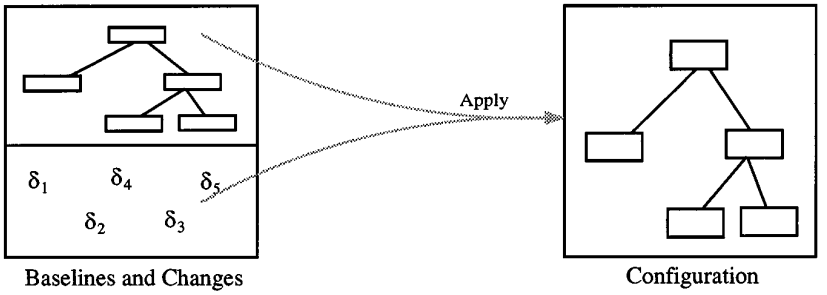


Fig. 11. Change set protocol.

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.

### 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 $\delta_i$ 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 $\delta_i$. If $\delta_i$ implies other changes $\delta_j, \ldots, \delta_k$, rename $F[\Delta_i]$ to $F[\Delta_i \sqcap \Delta_j \sqcap \cdots \sqcap \Delta_k]$.

—*Apply.* To apply a change set $\delta_i, \ldots, \delta_j$ 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 by implication: conflicting changes $\delta_i$ and $\delta_j$ 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 been 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 2.4.2, 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 is this not so in existing SCM systems? Basically, there are three causes, each reducing complexity by imposing constraints on the general problem.

(1) *Simplification.* In existing SCM systems, components are either identified or selected using simple feature terms; the general case of having nonsimple feature terms for both identification and selection never occurs. Hence, the preconditions for Proposition 2.4.6 apply: whether a version is a 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.

(2) *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 $(\Delta_{42} \rightarrow \Delta_{41}) \sqcap (\Delta_{41} \rightarrow \Delta_{40}) \sqcap \cdots$. 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 $\Delta_j$, and repeat the process for $\Delta_j$. Likewise, for each $\nabla_i \sqsupseteq S$, replace all $(\Delta_j \rightarrow \Delta_i) = (\nabla_i \rightarrow \nabla_j)$ by $\nabla_j$, and repeat the process for $\nabla_j$. This scheme allows for efficient selection from "classical" revision histories, as realized in today's SCM systems.

(3) *Orthogonality.* As stated in Proposition 2.4.4, 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 that are orthogonal to all existing ones. Furthermore, orthogonality simplifies the separation of concerns. For

**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 = "";
```

Fig. 12. Multirevision file.

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, and 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

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

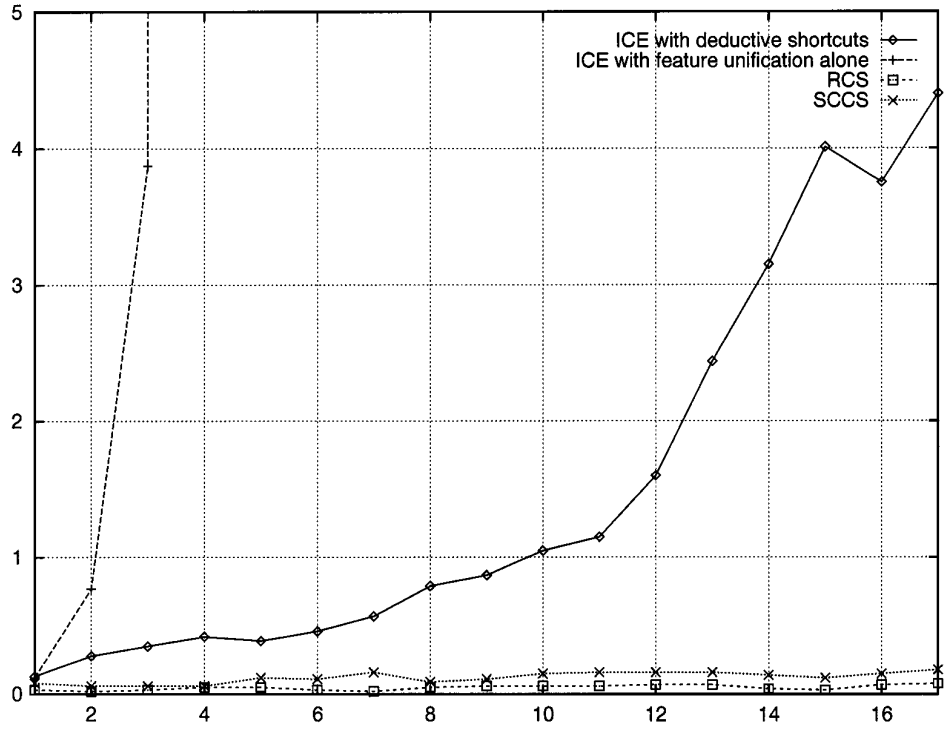


Fig. 13. Revision checkin times (in seconds) for ICE, RCS, and SCCS.

effects of the deductive shortcuts, we also made ICE run without deductive shortcuts and rely on feature unification alone.

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 the 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 taken 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 some unexpected problems. In Figure 13 we list 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 the figure. 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, which compare the new revision only with the previous revision.

The checkin problem could easily be solved by realizing the RCS/SCCS approach and comparing only the latest revisions. The data above show 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 are unquestioned. But, if we have multiple variants in multiple revisions, all sharing some common code, then which are the "latest" revisions that 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, and changes that apply only to certain variants 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 [Krone and Snelting 1994], restructuring software in order to eliminate them is still at its beginning [Snelting 1996]. Future research and experience will show how far nonorthogonal 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 an 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, yet is 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 an SCM primitive layer like

the FFS. These features make ICE an environment that adapts 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:

(1) *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 nonorthogonal SCM concepts, especially variance. With 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.

(2) *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 nonfunctional features called *roles* [Smolka 1992]. This extension will introduce and unify versioning concepts in graph-structured applications such as computer-aided design [Katz 1990] and graph-based software development environments [Engels et al. 1992; Schürr et al. 1995]; first results are given by Zeller [1997a].

(3) *Support of the SCM process.* Focusing 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 behavior may be verified against the SCM process. We imagine organizing the SCM process entirely by manipulating component features, that is, by changing their state from *proposed* via *tested* to *released*; SCM procedures might be modeled by pre- and postconditions specified as feature terms. Unfortunately, there is no true methodology yet on how components and versions should be attributed with feature terms; experiences from other attribute-oriented SCM systems or faceted classification [Prieto-Díaz 1987] 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 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/.

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

ACKNOWLEDGMENTS

Many thanks go 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

ADAMS, P. AND SOLOMON, M. 1995. An overview of the CAPITL software development environment. In *Software Configuration Management: Selected Papers/ICSE SCM-4 and SCM-5 Workshops*, J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 1–34.

AÏT-KACI, H. 1986. An algebraic semantics approach to the effective resolution of type equations. *Theor. Comput. Sci. 45,* 293–351.

AÏT-KACI, H. AND NASR, R. 1986. Login: A logic programming language with built-in inheritance. *J. Logic Program. 3,* 186–215.

AÏT-KACI, H. AND PODELSKI, A. 1991. Towards a meaning of LIFE. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming*, J. Maluszyński and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 528. Springer-Verlag, New York, 255–274.

BERLINER, B. 1990. CVS II: Parallelizing software development. In *Proceedings of the 1990 Winter USENIX Conference* (Washington, D.C.). USENIX Assoc., Berkeley, Calif.

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

BRACHMAN, R. J. AND LEVESQUE, H. J. 1984. The tractability of subsumption in frame-based description languages. In *Proceedings of the 4th National Conference of the American Association for Artificial Intelligence* (Austin, Tex., Aug.). AAAI, Menlo Park, Calif., 34–37.

BROWN, A., DART, S., FEILER, P., AND WALLNAU, K. 1991. The state of automated configuration management. Tech. Rep. CMU/SEI-ATR-91, Software Engineering Inst., Carnegie Mellon Univ., Pittsburgh, Pa., Sept.

CONRADI, R. AND TRYGGESETH, E. 1995. Versioning models. In *Software Configuration Management: Selected Papers/ICSE SCM-4 and SCM-5 Workshops*, J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 80.

CONRADI, R. AND WESTFECHTEL, B. 1996. Version models for software configuration management. Tech. Rep. AIB 96-10, RWTH Aachen, Germany, Oct.

COURINGTON, W. 1989. The network software environment. Tech. Rep. FE 197-0, Sun Microsystems, Inc., Mountain View, Calif., Feb.

DART, S. 1991. Concepts in configuration management systems. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (Trondheim, Norway, June). ACM Press, New York, 1–18.

ENGELS, G., LEWERENTZ, C., NAGL, M., SCHÄFER, W., AND SCHÜRR, A. 1992. Building integrated software development environments—Part 1: Tool specification. *ACM Trans. Softw. Eng. Methodol. 1,* 2 (Apr.), 135–167.

ESTUBLIER, J. 1995. Process session. In *Software Configuration Management: Selected Papers/ICSE SCM-4 and SCM-5 Workshops*, J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 136–137.

ESTUBLIER, J. AND CASALLAS, R. 1994. The Adele configuration manager. In *Configuration Management*, W. F. Tichy, Ed. Trends in Software, vol. 2. John Wiley and Sons, Chichester, England, 99–133.

FEILER, P. H. 1991. Configuration management models in commercial environments. Tech. Rep. CMU/SEI-91-TR-7, Software Engineering Inst., Carnegie Mellon Univ., Pittsburgh, Pa., Mar.

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

FOWLER, G., KORN, D., AND RAO, H. 1994. *n*-DFS: The multiple dimensional file system. In *Configuration Management*, W. F. Tichy, Ed. Trends in Software, vol. 2. John Wiley and Sons, Chichester, England, 135–154.

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

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

IEEE. 1988. IEEE guide to software configuration management. ANSI/IEEE Standard 1042-1987, The Institute of Electrical and Electronics Engineers, New York.

IEEE. 1990. IEEE guide to software configuration management plans. ANSI/IEEE Standard 828-1990, The Institute of Electrical and Electronics Engineers, New York.

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

JONES, N. D., GOMARD, C. K., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Engelwood Cliffs, N.J.

KAPLAN, R. M. AND BRESNAN, J. 1982. Lexical-functional grammar: A formal system for grammatical representation. In *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed. MIT Press, Cambridge, Mass., 173–381.

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

KAY, M. 1984. Functional unification grammar: A formalism for machine translation. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence* (Stanford, Calif.). 75–78.

KRONE, M. AND SNELTING, G. 1994. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering* (Sorrento, Italy, May). IEEE Computer Society Press, Los Alamitos, Calif., 49–57.

LACROIX, M. AND LAVENCY, P. 1987. Preferences: Putting more knowledge into queries. In *Proceedings of the 13th International Conference on Very Large Data Bases* (Brighton, England). VLDB Endowment Press, Saratoga, Calif., 217–225.

LAMPEN, A. AND MAHLER, A. 1988. An object base for attributed software objects. In *Proceedings of the Fall 88 EUUG Conference* (Cascais, Oct.). 95–105.

LEBLANG, D. B. 1994. The CM challenge: Configuration management that works. In *Configuration Management*, W. F. Tichy, Ed. Trends in Software, vol. 2. John Wiley and Sons, Chichester, England, 1–37.

LIE, A., CONRADI, R., DIDRIKSEN, T. M., KARLSSON, E.-A., HALLSTEINSEN, S. O., AND HOLAGER, P. 1989. Change oriented versioning in a software engineering database. In *Proceedings of the 2nd International Workshop on Software Configuration Management* (Princeton, N.J., Oct.). ACM Press, New York, 56–65.

MAHLER, A. 1994. Variants: Keeping things together and telling them apart. In *Configuration Management*, W. F. Tichy, Ed. Trends in Software, vol. 2. John Wiley and Sons, Chichester, England, 39–69.

MARTIN, U. AND NIPKOW, T. 1990. Boolean unification—The story so far. In *Unification*, C. Kirchner, Ed. Academic Press, London, 437–455.

MILLER, W. AND MYERS, E. 1985. A file comparison program. *Softw. Pract. Exper. 15,* 11 (Nov.), 1025–1040.

MUNCH, B. P., LARSEN, J.-O., GULLA, B., CONRADI, R., AND KARLSSON, E. A. 1993. Uniform versioning: The change-oriented model. In *Proceedings of the 4th International Workshop on Software Configuration Management* (preprint) (Baltimore, Md., May). 188–196.

NEBEL, B. 1990. *Reasoning and Revision in Hybrid Representation Systems*. Lecture Notes in Artificial Intelligence, vol. 422. Springer-Verlag, New York.

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

NICKLIN, P. 1991. Managing multi-variant software configurations. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (Trondheim, Norway, June). ACM Press, New York, 53–57.

PLOEDEREDER, E. AND FERGANY, A. 1989. The data model of the configuration management assistant. In *Proceedings of the 2nd International Workshop on Software Configuration Management* (Princeton, N.J., Oct.). ACM Press, New York, 5–13.

PRIETO-DÍAZ, R. 1987. Classifying software for reusability. *IEEE Softw. 4,* 1 (Jan.).

REICHENBERGER, C. 1989. Orthogonal version management. In *Proceedings of the 2nd International Workshop on Software Configuration Management* (Princeton, N.J., Oct.). ACM Press, New York, 137–140.

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

SACHWEH, S. AND SCHÄFER, W. 1995. Version management for tightly integrated software engineering environments. In *Proceedings of the 7th International Conference on Software Engineering Environments* (Noordwijkerhout, Netherlands, Apr.). IEEE Computer Society Press, Los Alamitos, Calif.

SANDBERG, R., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. 1985. Design and implementation of the Sun Network filesystem. In *Proceedings of the Summer 1985 USENIX Conference* (Portland, Oreg., June). USENIX Assoc., Berkeley, Calif., 119–130.

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

SCHMERL, B. D. AND MARLIN, C. D. 1995. Designing configuration management facilities for dynamically bound systems. In *Software Configuration Management: Selected Papers/ICSE SCM-4 and SCM-5 Workshops*, J. Estublier, Ed. Lecture Notes in Computer Science, vol. 1005. Springer-Verlag, New York, 88–100.

SCHÜRR, A., WINTER, A. J., AND ZÜNDORF, A. 1995. Graph grammar engineering with PROGRES. In *Proceedings of the 5th European Software Engineering Conference*, W. Schäfer and P. Botella, Eds. Lecture Notes in Computer Science, vol. 989. Springer-Verlag, New York, 219–234.

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

SMOLKA, G. 1992. Feature-constrained logics for unification grammars. *J. Logic Program. 12,* 51–87.

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

SNELTING, G. 1991. The calculus of context relations. *Acta Inf. 28* (May), 411–445.

SNELTING, G. 1996. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. Softw. Eng. Methodol. 5,* 2 (Apr.), 146–189.

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

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

VAN DER HOEK, A., HEIMBIGNER, D., AND WOLF, A. L. 1996. A generic, peer-to-peer repository for distributed configuration management. In *Proceedings of the 18th International Conference on Software Engineering* (Berlin, Germany, Mar.). IEEE Computer Society Press, Los Alamitos, Calif., 308–317.

WESTFECHTEL, B. 1991. Structure-oriented merging of revisions of software documents. In *Proceedings of the 3rd International Workshop on Software Configuration Management* (Trondheim, Norway, June). ACM Press, New York, 86–79.

WIEBE, D. 1993. Object-oriented software configuration management. In *Proceedings of the 4th International Workshop on Software Configuration Management* (preprint) (Baltimore, Md., May). 241–252.

WINKLER, J. F. H. 1987. Version control in families of large programs. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, Calif., Mar.). IEEE Computer Society Press, Los Alamitos, Calif., 91–105.

ZELLER, A. 1995. A unified version model for configuration management. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, G. Kaiser, Ed. *Softw. Eng. Notes 20,* 4, 151–160.

ZELLER, A. 1996. Smooth operations with square operators—The version set model in ICE. In *Proceedings of the 6th International Workshop on Software Configuration Management*, I. Sommerville, Ed. Lecture Notes in Computer Science, vol. 1167. Springer-Verlag, New York, 8–30.

ZELLER, A. 1997a. Versioning software systems through concept descriptions. Comput. Sci. Rep. 97-01, Abteilung Softwaretechnologie, Technical Univ. of Braunschweig, Braunschweig, Germany, Jan.

ZELLER, A. 1997b. Configuration management with version sets. Ph.D. thesis, Abteilung Softwaretechnologie, Technical Univ. of Braunschweig, Braunschweig, Germany, Nov.

ZELLER, A. AND SNELTING, G. 1995. Handling version sets through feature logic. In *Proceedings of the 5th European Software Engineering Conference*, W. Schäfer and P. Botella, Eds. Lecture Notes in Computer Science, vol. 989. Springer-Verlag, New York, 191–204.

Received March 1996; revised November 1996; accepted March 1997