Vom Fachbereich für Mathematik und Informatik

der Technischen Universität Braunschweig

genehmigte Dissertation

zur Erlangung des Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

Andreas Zeller

Configuration Management with Version Sets

A Unified Software Versioning Model and its Applications

1. April 1997

1. Referent:	Prof. Dr. Gregor Snelting
2. Referent:	Prof. Dr. Walter F. Tichy
Eingereicht am:	1. November 1996

Zeller, Andreas:
Configuration Management with Version Sets.
A Unified Software Versioning Model and its Applications.

Includes bibliographical references and index.

Revision `1.103` of `thesis.tex`
Created: `1997-05-22 21:42:28`
Formatted: `1997-05-22 23:44:00`

Please note: This electronic version differs slightly from the original paper copy. The paper copy uses a MathTime font for mathematical symbols; this font is copyrighted by Y&Y, Inc. and must not be distributed electronically. This electronic version uses a Computer Modern Roman font for mathematical symbols instead. The text itself is unchanged (except for this note); locations of section headings, figures, etc. have not changed as well.

This electronic version is available via the WWW at

`http://www.cs.tu-bs.de/softech/papers/zeller-phd/`

Please use this URL when referring to this work.

As an exception of the copyright rules below, you are hereby granted to reproduce this electronic version for the purposes of viewing its contents on a screen or creating a paper copy for personal use only, provided that the copyright note below is preserved.

Typeset by Andreas Zeller, Braunschweig using Times 10 pt
Printed at the Technische Universität Braunschweig
Bookbinding by Dissertations Druck Darmstadt (DDD), Darmstadt

To my grandfather

Preface

GENTLE READER: This is a book about software configuration management, the discipline to organize and control evolving software systems. In software configuration management, or SCM for short, one deals with the problem of several people developing, building, shipping, and maintaining several copies of software products, each with an individual set of changes applied to make it fit into a particular environment. The aim of an SCM engineer is to identify and control these changes, such that all resulting software products are well-identified and well-defined.

Software configuration management is a hard task, because few things are so easy to change and so easy to propagate as software. Fortunately, a number of automated SCM tools and systems exist that can help enforcing and maintaining SCM procedures. Unfortunately, there are many such tools and each comes with its own SCM policy, which is often centered on a specific environment and thus seldom interoperates, yet alone integrates with other SCM tools. From the SCM engineer's point of view, this is an unfortunate situation as the entire development process must be adapted to a specific SCM policy.

In this work, I have attempted to provide a common formal and adaptive base for the technical aspects of software configuration management. The base I have chosen for this integration is *feature logic,* a logic denoting objects by specifying their possible attributes (or non-attributes). Characterizing objects by their features is a common technique in SCM, and it seemed natural to me to choose a logic based on this technique.

Using feature logic, I have been able to model and integrate common SCM functionality such as attributed components, repositories, work spaces, variant sets, revision histories, or consistency checking in a single concept, called *version sets.* Version sets group versions, components, and configurations by their features. SCM functionality is realized through set operations. Versions are selected and refined through set intersection. Set union realizes the grouping of

versions to repositories. Subsumption and disjointness express inclusion and exclusion of changes, structuring the version space.

Version sets do not introduce new concepts into SCM; instead, they expose new ways of combining and integrating existing concepts and thus provide much more flexibility in adapting SCM systems to their users. In short, I have designed the version set model as an attempt to integrate the current spectrum of SCM functionality into a single, hopefully simple and elegant formalism, allowing for adaptive combinations of SCM concepts with predictable effects.

In science, claims are justified by proofs; in engineering, claims are justified by simulation. Applied computer science is both a scientific and an engineering discipline. I thus have supplied both proofs and an implementation; the resulting SCM system ICE (for *Incremental Configuration Environment*), is presented and evaluated in its own part at the end of this work. The version set model could not have been conceived without its usability and efficiency steadily being verified in ICE.

While conceiving and developing ICE, I have resisted the temptation to enrich the wide spectrum of software engineering with yet another eclectic environment, another eclectic special-purpose formalism, and another eclectic configuration language. Instead, I have designed ICE to work with well-established SCM techniques and representations wherever possible, in order to keep the learning curve flat and the integration smooth. It is my hope that ICE will not only help to demonstrate the effectiveness of the underlying version set model, but also be a useful aid in addressing today's practical SCM problems.

To make this book self-contained, the first part summarizes the state of the art in today's SCM practice and research, followed by an introduction to feature logic. The version set model and ICE come in individual parts, closing with answers to frequently asked questions. In short, this book presents today's SCM concepts, their common foundation, and some new applications. Enjoy!

Braunschweig A. Z.
November 1996

Abstract

Software configuration management (SCM) is the discipline for organizing and controlling evolving complex software systems. Several SCM tools and systems exist that automate and integrate SCM tasks like version identification, system modeling, product construction, or team work coordination. However, the choice of an SCM system is still a long-term commitment: Each SCM system comes with its own SCM policy, which is often centered on a specific environment and thus seldom interoperates, yet alone integrates with other SCM tools. This is unfortunate, as the entire software development process must be adapted to fit the system's SCM policy.

We want SCM systems that adapt to their users, rather than vice versa. As a foundation, 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,* a well-established formalism for knowledge representation and logic programming, we define the semantics of SCM tasks and concepts. Our results are as follows:

Unified versioning. Version sets provide one single formalism to express all versioning dimensions as well as constraints on them, integrating SCM concepts like revisions, variants, workspaces, and configurations in one single model. The SCM policy is not constrained by decisions made in lower SCM layers.

Integration of changes and revisions. Configuration constraints, expressed in feature logic, allow us to capture the entire range of temporal versioning—from the rigidness of versions-oriented models to the flexibility of change-oriented models.

Consistency checking under ambiguity. Through feature logic, we deduce the features and the consistency of configurations as well as derived compo-

nents and thus describe how features propagate in the SCM process. Inconsistencies are detected even when the configuration description is incomplete or ambiguous. Ambiguity is not only tolerated in consistency checking; at all SCM layers, sets rather than single items are the primary objects of SCM tasks and procedures.

We have implemented the version set model in an experimental SCM system named ICE for *Incremental Configuration Environment.* In ICE, the version set model shows up numerous user-visible benefits. Through the FFS, a virtual file system, users can access version sets consisting of arbitrary combinations of revisions, changes, variants, and workspaces. Individual versions are accessed as files; version sets as a whole can be handled via version directories or through the well-known C preprocessor representation. On top of the FFS, specific SCM protocols are realized efficiently through simple file operations on version sets. These features make ICE a universal platform for individual well-structured SCM policies.

Zusammenfassung

Software-Konfigurationsmanagement (SCM, auch *Software-Verwaltung*) befaßt sich mit der Organisation und Kontrolle des Entwicklungsprozesses komplexer Softwaresysteme. Heute gibt es zahlreiche SCM-Werkzeuge und SCM-Systeme, die Aufgaben wie Versionsbezeichnung, System-Modellierung, Programmkonstruktion oder Koordination der Gruppenarbeit automatisieren und integrieren. Allerdings bedeutet die Auswahl eines SCM-Systems immer noch eine langfristige Verpflichtung: Jedes SCM-System bringt sein eigenes Vorgehensmodell mit, das oft auf eine bestimmte Umgebung zugeschnitten ist und deshalb nicht mit anderen SCM-Systemen zusammenarbeitet, von einer Integration ganz zu schweigen. Das ist um so bedauerlicher, da die gesamte Software-Entwicklung an die jeweilige Verfahrensweise angepaßt werden muß.

Wir möchten SCM-Systeme, die sich ihren Anwendern anpassen, statt umgekehrt. Als Grundlage schlagen wir ein vereinheitlichtes Versionierungs-Modell vor, das Modell der *Versionsmengen.* Versionsmengen kennzeichnen Versionen, Komponenten und Konfigurationen durch *Feature-Terme* – Boolesche Terme über Ausdrücke der Art $(\mathit{Eigenschaft}{:}\mathit{Wert})$. Mit *Feature-Logik,* einem etablierten Formalismus für Wissensrepräsentation und logische Programmierung, definieren wir Aufgaben und Konzepte des SCM. Im einzelnen erhalten wir folgende Ergebnisse:

Vereinheitlichte Versionierung. Versionsmengen sind ein einheitlicher Formalismus, mit dem alle Dimensionen der Versionierung als auch Querbeziehungen ausgedrückt werden. Dadurch werden SCM-Begriffe wie Revisionen, Varianten, Arbeitsumgebungen, und Konfigurationen in ein einziges Modell integriert. Das SCM-Vorgehensmodell wird nicht durch Festlegungen in unteren SCM-Schichten eingeschränkt.

Integration von Änderungen und Revisionen. Konfigurationsbedingungen in Feature-Logik decken das gesamte Spektrum zeitlicher Versionierung ab –

von der Strenge der versionsorientierten SCM-Modelle bis zur Kombinationsfreudigkeit der änderungsorientierten SCM-Modelle.

Konsistenzprüfung unter Mehrdeutigkeit. Mit Feature-Logik bestimmen wir die Eigenschaften und Konsistenz von Konfigurationen als auch abgeleiteter Komponenten und beschreiben so, wie sich Eigenschaften im SCM-Prozeß fortpflanzen. Unstimmigkeiten werden auch dann entdeckt, wenn die Konfigurationsbeschreibung unvollständig oder mehrdeutig ist. Mehrdeutigkeit ist nicht nur bei der Konsistenzprüfung zulässig; auf allen SCM-Ebenen arbeiten die SCM-Verfahren mit Versionsmengen statt Versionen.

Wir haben das Modell der Versionsmengen in einem experimentellen SCM-System namens ICE implementiert (ICE = *incremental configuration environment,* inkrementelle Konfigurations-Umgebung). In ICE zeigt das Modell der Versionsmengen zahlreiche Vorteile für den Benutzer. Über das FFS, ein virtuelles Dateisystem, können Anwender Versionsmengen bearbeiten, die aus beliebigen Kombinationen von Revisionen, Varianten und Arbeitsbereichen bestehen. Einzelne Versionen werden als Dateien angesprochen; Versionsmengen als ganzes können über Versions-Verzeichnisse oder über die wohlbekannte C-Präprozessor-Darstellung bearbeitet werden. Mit FFS als Grundlage lassen sich SCM-Verfahren durch einfache Dateioperationen auf Versionsmengen effizient realisieren. Diese Eigenschaften machen ICE zu einer universellen Plattform für individuelle, wohlstrukturierte SCM-Vorgehensmodelle.

Contents

List of Figures

List of Tables

Part One

The State of the Art in SCM

Chapter 1

Configuration Management

We begin with a short presentation of software configuration management. We show why software configuration management (SCM) is important in creating complex software, we show the procedures required by SCM, and we give a brief survey of the SCM models and SCM functionality areas as supported by today's automated SCM systems.

1.1 The Name of the Game

In software development, nothing is as persistent as change. Typically, we find several individuals producing, changing, and exchanging common and individual software parts, all oriented towards a common goal. Often, this common goal is not a single static product, but a dynamic collection of components destined to work with each other, where not all assemblies may result in a complete and consistent product. There may be hundreds or thousands of such components, with several hundred persons at different sites maintaining and changing them; the entire development process becomes a continuous history of changes and improvements. To keep all these multi-version, multi-people activities under control, the need for *configuration management* arises.

Configuration management (CM) is the discipline for organizing and controlling evolving systems. Configuration management is an old discipline, born out of systems manufacturing. CM mandates procedures for identification of components and their assemblies, for controlling releases and changes, for recording the product status, and for validating the completeness and consistency of a product [IEE88, IEE90]. Recent CM definitions [Dar91] also include areas like construction management, process management, and team work control.

1.2 From CM to SCM

Software configuration management (SCM) goes beyond these CM procedures in several ways. First, few things are as malleable as software. This adds special complexity to configuration management because changes are easy to make, and, in fact, occur more often than in traditional CM areas. Second, software is easily duplicated. There may be multiple copies of a software component, some private, some public, each having its individual set of changes which may diverge in time. Third, software is complex. Applying a change in a single component may induce hard-to-trace failures in other components. It is these properties that make software development difficult, and which make CM significantly harder when applied to software development.

SCM also differs from traditional CM since all components are under computer control. Hence, software configuration management can be widely automated, compensating for the added complexity. Automation applies to most of the identification and control tasks, to construction management as well as to completeness and consistency maintenance. Also, SCM tools can be integrated into software development tools, which run on computers as well. Today, there are several SCM tools available that automate SCM procedures. Some SCM systems encompass the entire SCM process by combining several tools and techniques. In this chapter, we give a brief survey of SCM functionality, as addressed by these systems.

1.3 SCM Procedures

A standard definition of configuration management [IEE88, IEE90] mandates the following *CM procedures* (cited from [Dar91]):

Identification. Reflects the structure of the product, identifies components and their type, making them unique and accessible in some form.[1]

Control. Controls the release of a product and changes to it throughout its life cycle by having controls in place that ensure consistent software via the creation of a baseline product.

Status Accounting. Records and reports the status of components and change requests, and gathers vital statistics about components in the product.

[1] The IEEE SCM standards [IEE88, IEE90] denote components by *configuration items*; the synonyms *configuration object* or simply *object* are also found.

Audit and Review. Validates the completeness of a product and maintains consistency among the components, ensuring that the product is a well-defined collection of components.

Recent SCM surveys [Dar91] broaden this definition to include procedures like construction management, process management, and team work control:

Manufacture. Manages the construction and building of the product in an optimal manner.

Process Management. Ensures the carrying out of the organization's procedures, policies and life cycle model.

Team work. Controls the work and interactions between multiple developers.

When applied to software development, these CM procedures can be easily carried out with computer support, since all software components are under computer control. Several software configuration management (SCM) tools and systems are available today, automating some or all of these CM procedures and providing a wide range of functionality.

1.4 SCM Models

In [Fei91a], Peter H. Feiler made a first approach to classify SCM functionality. He examines the software process as it is enforced by existing SCM systems and distinguishes four *configuration management models,* each introducing specific functionality:

Checkin/Checkout Model. The basic SCM model introduces the concept of a *repository* holding multiple *versions* of a product component. Developers can copy versions from (check out) and to (check in) the repository.

Change-Oriented Model. As its name says, the Change-Oriented Model focuses on *changes* rather than on versions. In this model, versions are the product of *change set* applied to a baseline. This model is useful for propagating and combining changes across users and sites.

Composition Model. The Composition Model extends SCM from the component level to the system level, introducing *system models* describing the system structure and *configurations* denoting versions of several components. *Consistency* issues are also found here.

Long Transaction Model. The Long Transaction Model introduces the notion of a *workspace,* where developers are isolated from each other's changes.

Since Feiler's survey, many new SCM systems have emerged, and many have extended their initial functionality to incorporate functionality that was previously found in other SCM models. Although all of todays SCM systems are essentially based on one of these SCM model, and although no significantly new SCM models have emerged, a more fine-grained approach is required to capture the entire spectrum of functionality in SCM systems.

1.5 SCM Functionality Areas

In [Dar91], Susan Dart uses a typical SCM scenario to define a set of SCM *functionality areas* users expect from today's SCM systems, reproduced in figure 1.1 on the facing page. Although some SCM aspects are missing (notably *variants* and *distribution),* it still constitutes a valid schema to capture SCM functionality.

Dart distinguishes between team-centered and process-centered functionality areas. The *team-centered* functionality areas deal with the *technical aspects* of software configuration management:

Components. Identify, classify, store and access the components that form the product.

Structure. Represent the architecture of the product.

Construction. Support the construction of the product and its artifacts.

Team. Enable a project team to develop and maintain a family of products.

In contrast to the team-centered areas, the *process-centered* functionality areas (shown in grey) cover management issues:

Auditing. Keep an audit trail of the product and its process.

Accounting. Gather statistics about the product and its process.

Controlling. Control how and when changes are made.

Process. Support the management of how the product evolves.

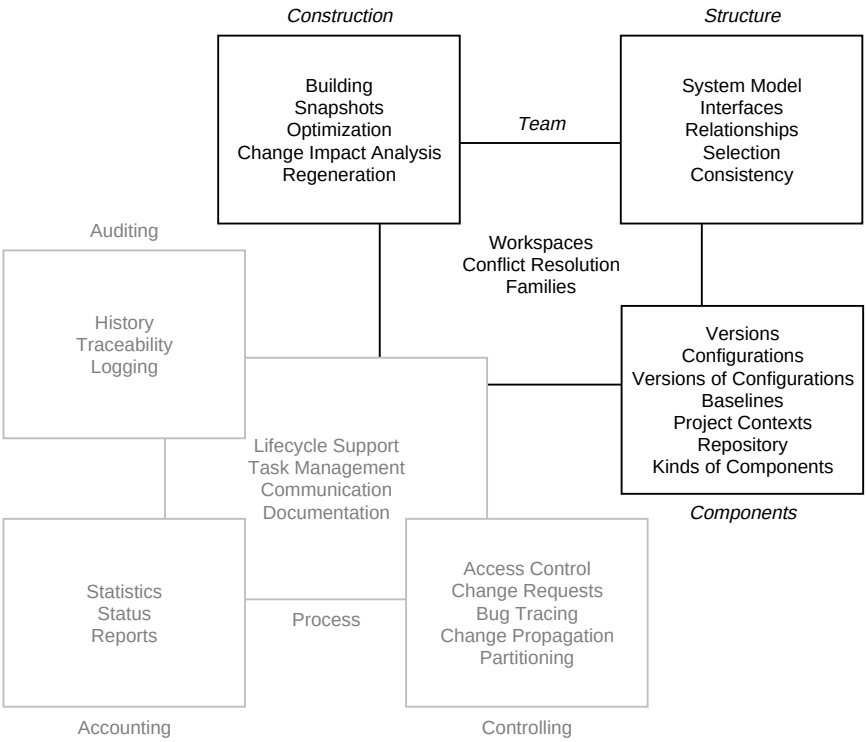

Figure 1.1: CM functionality requirements (after [Dar91])

In this part, we give an overview on the spectrum of functionality in today's SCM systems, following the classification of Dart's survey, and treating SCM models with their typical concepts. As our work is primarily concerned with the technical aspects of SCM rather than with the process areas, we focus on the team-centered functionality areas and only sketch the process-centered functionality areas. As a conclusion, we identify requirements for future SCM systems.

La maintenance des logiciels de grande taille est très coûteuse.
Cependant, ce thème est souvent ignoré des chercheurs.

— JEAN-MARIE FAVRE, Vers une représentation multi-langages et multi-versions des programmes

Chapter 2

Components Functionality

We present the component functionality area, as realized in the Checkin/Checkout Model. *A central repository, shared among developers, holds all component versions as they are created. Versions are accessed by copying component versions from the repository to a private space (check out) and copy them back again into the repository (check in). The Checkin/Checkout Model is the simplest and oldest SCM model; its typical realizations are Rochkind's* Source Code Control System *(SCCS)* [Roc75] *and Tichy's* Revision Control System *(RCS)* [Tic85]. *As an alternative, we also take a look at the* Change-Oriented Model, *which focuses on managing* changes *instead of versions.*

2.1 Versioning Dimensions

Software products are commonly broken down into several *components,* which are created and maintained by different people. As these people apply changes to software components, they create new component *versions.* Each version is one of several instances of a single component. This implies that two versions of a component should be more similar to each other than any two components are. Depending on the context, the unqualified word component denotes either all component versions or one single version.

Depending on the intentions of the creator, SCM literature divides versions into three *versioning dimensions* [EC95]; ideally, all these dimensions should be fully orthogonal to each other.

Historical versioning. Versions that are created to *supersede* a specific version, e.g. for maintenance purposes, are called *revisions* [Win87]. When a new

revision is created, evolution of the original version is phased out in favor of the new revision. In practice, a revision of a component is usually created by modifying a copy of the most recent revision. The old revisions are permanently stored for maintenance and documenting purposes; they form the version history or *revision history* of the component.

Logical versioning. In contrast to revisions, a variant is created as an *alternative* to a specific version. They are created in *branches,* that is, parallel development threads that may eventually be *merged* with the main development thread. *Permanent variants* are created when the product is adapted to different environments. Variance can again arise in several dimensions, including varying user requirements and varying system platforms, but also variants for testing and debugging. These *variance dimensions* need no more be orthogonal and be subject to several constraints.

Cooperative versioning. A *temporary variant* is a variant that will later be integrated (or merged) with another variant. Temporary variants are required, for example, to change an old revision while the new revision is already under development. We will discuss temporary variants in the context of cooperation strategies in section 5.5.

2.2 Versioning Models

Figure 2.1 on the next page illustrates the difference between the various version kinds. The boxes denote various versions as they are created; an arrow from version *A* to version *B* indicates that *B* was created based on *A*.[1] The entire graph is the *version graph* of the component, showing how each version was created.

In a version graph, the SCM distinction between revisions and variants is pragmatic; deciding whether a version is a revision or a temporary variant or a permanent variant can only be decided *a posteriori* when taking the later version graph into account. Upon creation of a new version, the developer must choose a version kind depending on the expected history. Since the motives of the developer may change, it should be possible to change the version kind later.

In most SCM tools and systems, the versioning dimensions are addressed by separate concepts; changing the version kind thus is a non-trivial task. Also, early version control tools like SCCS [Roc75] or RCS [Tic85] were primarily conceived for revision and change control; variance was managed by dedicated variant control tools like the C preprocessor (CPP) [KR89], discussed in section 2.6.1.

[1] As stated in section 3.2.2, this is called a *is-derived-from* relation.

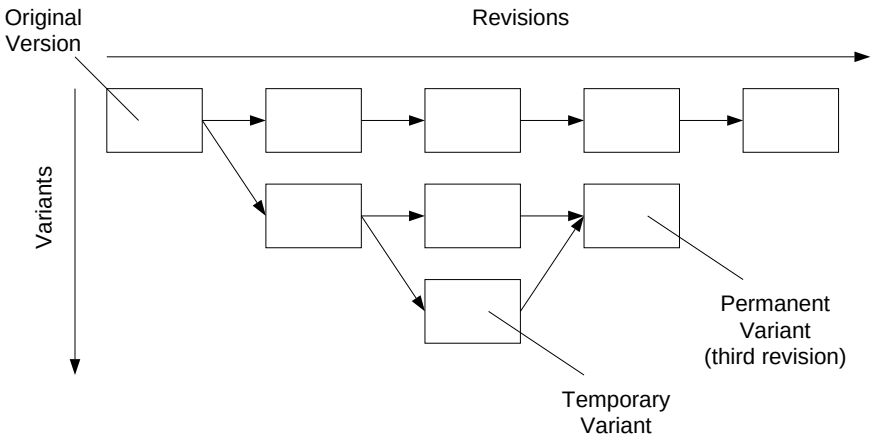

Figure 2.1: Version kinds in a version graph

Recently, new versioning models have emerged that overcome the limitations of version graphs. In [Rei89], Reichenberger coined the term *orthogonal version management,* as implemented in the VOODOO SCM tool [Rei95]. In orthogonal version management, the universe of all components, variants, and revisions constitutes a three-dimensional space, the *object pool,* from which *projections* can be chosen to select groups of variants, revisions, or components, as illustrated in figure 2.2 on the following page.

In [EC95], Estublier and Casallas also propose a three-dimensional versioning model, using the historical, logical, and cooperative dimensions, as discussed above. In contrast to Reichenberger's orthogonal version management, however, each dimension is accessed using different kinds of queries or services, according to the specific needs.

2.3 Identifying Component Versions

Along with the creation of new versions comes a consistent version identification scheme. It is common practice in SCM to use different identification schemes for revisions as well as permanent and temporary variants.

2.3.1 Identifying Revisions

Revisions are typically identified by *revision numbers* which reflect their creation date: the most recent revision is the one with the highest revision number.

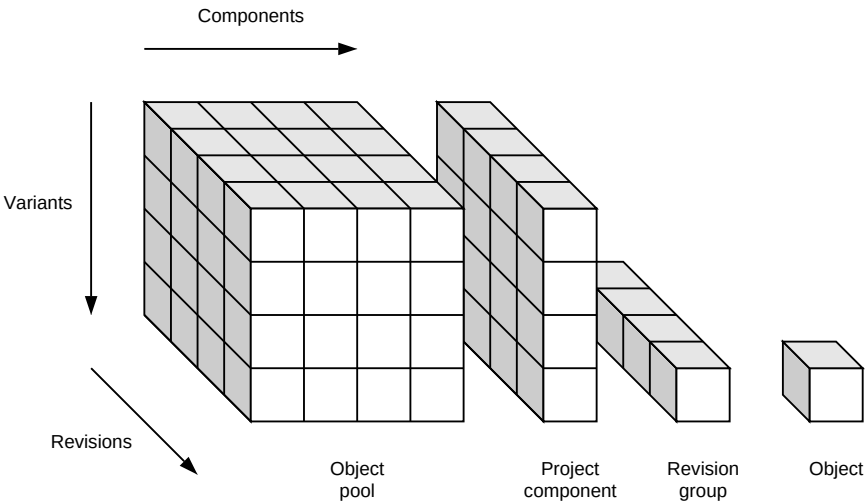

Figure 2.2: The object pool and some of its projections (after [Rei89])

Numbering schemes include single integers—the first revision is named 1, the second 2, and so on, as in CLEARCASE [Leb94]—and pairs of integers as in SCCS [Roc75] or RCS [Tic85], sometimes called the release number and the level number. An increment in the release number (for instance, from 2.2 to 3.1) indicates a major change, an increment in the level number (from 3.1 to 3.2) indicates a minor change. All revision control tools allow for identifying revisions by the *revision date* (e.g. the time the revision was created).

In the Change-Oriented Model, revisions are identified by a list of changes applied to the baseline, as discussed in section 2.7. The individual changes are named; a version identified by bugfix-3, extension-5 thus has the changes bugfix-3 and extension-5 applied.

2.3.2 Identifying Variants

Permanent variants are usually named instead of numbered, since they are not implicitly ordered. One method, realized in the CLEARCASE system is to assign names to edges in the version graph; zbuf.c@@/main/new_GUI/color denotes a path in the version graph of the component zbuf.c. First, the path to the main variant is chosen, then the new graphical user interface (GUI) from this main variant, then the color variant of the GUI. As shown in the example, this identification

scheme imposes a hierarchical order on the variants and is restricted to paths in the version graph: the specification zbuf.c@@/color does not make sense, because the major variants are not specified.

Other SCM systems use an approach independent from the version graph. They assign a set of *attribute*/*value* pairs, where each attribute reflects a variance dimension. For instance, a component occurring in several variants for multiple languages and multiple operating systems can be identified by two variance dimensions *language* and *operating-system*. *language* may take values like *english*, *german*, *french* and so on, while *operating-system* is either *unix*, *windows*, or *mac*. Such schemes are also called *attribution schemes.* They are used in high-level SCM systems like ADELE [Est85, Est88, EC94] or in the *attributed file system* (AtFS) of SHAPE [LM88, Mah94] as well as in low-level variant control tools like the C preprocessor, which we discuss in section 2.6.1.

For temporary variants, various identification schemes exist. The RCS and SCCS way is to introduce additional numbering levels. That is, a variant of the original revision 1.2 is named 1.2.1.1, with subsequent revisions 1.2.1.2, 1.2.1.3, and so on. In CLEARCASE, there is no special distinction in identifying version kinds.

2.4 Determining Version Differences

In order to determine changes made to software, users must be able to determine the differences between versions. A basic procedure for this task is *text file comparison* [Tic84, MM85], as realized in the UNIX DIFF program. DIFF takes two text files *A* and *B* as input and generates a minimal set of changes (i.e. line deletions and inclusions) that are necessary to convert *A* into *B*. In figure 2.3 on the next page, we show the output of DIFF applied to two text files **tichy-cm** and **dart-cm**; in the DIFF output, lines occurring in **tichy-cm** only are prefixed with "<"; lines occurring in **dart-cm** are prefixed with ">". DIFF and DIFF-like tools are the base of many SCM tools and systems, since they are convenient for reducing the size of repositories (see section 2.5 for details).

Using DIFF is accurate for text data, since we can easily distinguish common lines from differing lines and manual changes are usually confined to small regions. Differences that affect the entire file are not well handled by the DIFF algorithm. This is especially true for non-text files, such as pictures, machine code files, or compressed files, where a minimal change can affect the contents of the entire file. In the last years, several improvements on the original DIFF algorithm have thus been developed; it has been empirically shown that these improvements show better performance than DIFF, notably on binary data [HVT96].

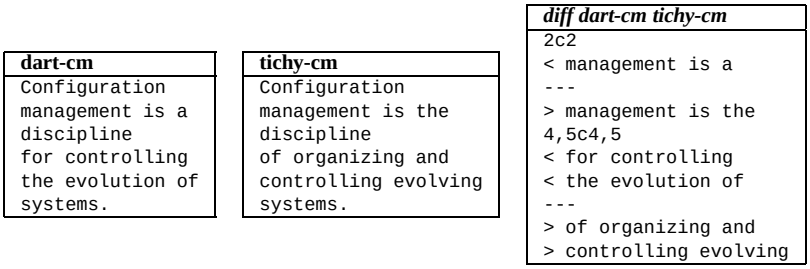

Figure 2.3: Finding textual differences with DIFF

To determine code differences in well-structured data, such as programs, a structured representation is more effective than the textual representation. In section 5.6, we discuss methods to determine code differences in abstract syntax trees.

Historically, the primary objective of determining differences between versions is to save space when storing multiple versions at once, as discussed in section 2.5. However, the difference must also be interpretable by humans to find out what exactly changed. For these purposes, editors that keep track of changes [MAM93, WG95] have been created; they track and express differences in terms of user interactions rather than in terms of changed blocks of data.

2.5 Storing Component Versions in Repositories

As developers create new versions of components, old and new versions must be stored persistently such that one can identify the product evolution and such that earlier versions can be reconstructed.

2.5.1 SCM Repositories

Early SCM tools like SCCS and RCS introduced the concept of a *repository,* where the component versions are stored together with the related SCM information. A repository does not store each version on its own, since that would require too much space. Instead, it exploits the commonality between versions by storing only the *difference* (also called *deltas*) between versions. The mechanisms used vary from SCM system to SCM system. RCS stores the most recent version as full text together with the differences (so-called *reverse deltas*) to earlier versions. In the SCCS system, each text block is tagged with the version(s) the block belongs to. Most today's SCM systems are based on either an RCS or SCCS approach.

2.5.2 Database Repositories

Emerging from the requirements of computer-aided design (CAD), substantial efforts have been made to store composite and versioned objects in databases. The common approach is to extend entity-relationship models by explicit *version relationships* like *derived-from* relationships, *is-part-of* relationships, and so on; we discuss such relationships in section 3.2.2.

Recently, such database technology has also been introduced in software engineering environments [Dit89]. For instance, the IPSEN software engineering environment is centered around a *graph database* which supports version relationships [SS95]. But the IPSEN authors also state that still there is no database fulfilling all needs of software engineering environments [ESW93].

2.6 Managing Variance

2.6.1 CPP "Repositories"

A completely different "repository" concept used for variant management is realized in the C programming language. All variants are stored in a single component visible to the programmer; variant-specific parts are enclosed in C preprocessor (CPP) `#if` ... `#endif` directives. As part of the compilation, CPP selects a single variant from the source code determined by a conjunction of attribute/value pairs. CPP evaluates each `#if`-expression, with any attribute being replaced by its respective value. The code piece enclosed by the `#if` ... `#endif` is included only if the `#if`-expression evaluates to a non-zero value.

As shown in figure 2.4 on the following page, invocation of CPP with the attributes `TICHY` set to `true` and `DATE` set to `1995` selects exactly the version tagged with the formula `TICHY && DATE >= 1994`.

Using CPP, specific environments are described by *configuration files* that define attribute values reflecting the properties of a specific environment. Such definition files can also be generated automatically. Tools like AUTOCONF [Mac94] run a series of tests to determine the features of the environment and create an appropriate configuration file.

It is common to see *conditional compilation,* as exemplified by CPP, as a programming language feature. In our context, conditional compilation should rather be regarded as a compiler- and language-independent version-control technique. In fact, preprocessor use for other purposes than version control is highly discouraged. In [Str94], Bjarne Stroustrup, the designer of the C++ programming language, states that one of the aims of C++ was to make CPP redundant and

cm-defs

```
Configuration
#if TICHY && DATE >= 1994
management is the
#else
management is a
#endif
discipline
#if TICHY && DATE >= 1994
of organizing and
controlling evolving
#else
for controlling
the evolution of
#endif
systems.
```

cpp -D TICHY=true -D DATE=1995 cm-defs

```
Configuration
management is the
discipline
of organizing and
controlling evolving
systems.
```

cpp -D TICHY=false cm-defs

```
Configuration
management is a
discipline
for controlling
the evolution of
systems.
```

Figure 2.4: Selecting versions with CPP

to "banish CPP into the program development environment with the other extra-linguistic tools where it belongs".

In the extra-linguistic context of SCM, conditional compilation is recognized as a "flexible and general scheme" [GJM91] and called "normal industry practice" [GMSW89]. The main advantage of conditional compilation is that variance is explicitly placed under the control of the programmer, who can view and edit several variants at once. Conditional compilation is thus frequently used to enrich revision-oriented SCM systems with orthogonal variance support. Unfortunately, as variance grows, the CPP file can become so strewn with CPP directives that it is hard to understand, yet harder to change. Hence, the need for dedicated variant-handling tools arises.

2.6.2 Multi-Variant Editors

On the component level, the problem of handling multiple variants was addressed by *variant-specific editors,* exemplified by the P-EDIT and MVPE editors developed by IBM [SBK88]. These editors follow the CPP paradigm, but allow for editing arbitrary version subsets. Only a single version is presented and edited, but the color of each text part indicates whether the text part (and the subsequent change) applies to the single version only or to several versions at once. For transparency, the user can change the presented version while editing. A similar functionality was implemented by Abrahamsen in the CPP-parse-edit-mode for

the GNU EMACS editor [Abr95], allowing users to examine and edit a restricted view of a CPP file. The CPP-parse-edit-mode also allows users to color text and mark text as read-only based on the CPP variable settings. A third approach is presented by Narayanaswamy [Nar89], where a variant-specific editor encloses differing code pieces in CPP-like directives.

When program code is stored not as text, but as an *abstract syntax tree,* structure editors can make variance explicit by supporting versioned subtrees and allowing the user to switch between variants. Such interactive variant selection is found in the PSG [BS86, SGS91, Sch95] and IPSEN [ELN+92, SS95] program development environments. As the common code is stored in the common supertree, the user can apply changes to all configurations by changing the common supertree only. As shown in [Sch95], such approaches can be combined with syntactical and semantical analysis, resulting in automatic consistency checking. The problem is that changes occurring near the top of the syntax tree result in distinct version subtrees, which may have identical, but unshared subtrees.

Multi-variant editors have not gained much acceptance. This may be due to the fact that traditional techniques (such as conventional text editors and CPP usage) suffice in practice, or that users prefer open, tool-based environments to specialized program development environments. Another reason may be that recent SCM research introduced other concepts for applying changes to several versions at once, as discussed in section 2.7.

2.7 Managing Changes

In the concepts discussed so far, individual *versions* of components were identified and managed. As an alternative, one can see a version as the result of *changes* applied to some original version or *baseline.* This is the basic idea of the *Change-Oriented Model,* as realized in the SCM systems EPOS [LCD+89, MLG+93] and AIDE-DE-CAMP [Har89], where changes, rather than versions, are identified, composed and applied on baselines.

In the Change-Oriented Model, changes are individual entities. For instance, DIFF output, as discussed in 2.4 on page 13, may be regarded as a change representation. Related changes, which may involve several components, can be grouped into *change sets* (also called *patches*) to ensure that they be applied as a single entity.

Using a specialized stream editor, like the UNIX PATCH program, one can apply change sets on a baseline and create the changed version from the original version or vice-versa. As an example, consider figure 2.5 on the following page, where the patch **tichy-patch** (the output of the DIFF run in figure 2.3 on page 14) is applied to the baseline **dart-cm**. In **dart-cm,** PATCH removes all lines prefixed

dart-cm

```
Configuration
management is a
discipline
for controlling
the evolution of
systems.
```

tichy-patch

```
2c2
< management is a
---
> management is the
4,5c4,5
< for controlling
< the evolution of
---
> of organizing and
> controlling evolving
```

patch dart-cm* < *tichy-patch

```
Configuration
management is the
discipline
of organizing and
controlling evolving
systems.
```

Figure 2.5: Applying changes with PATCH

with "<" and inserts the lines prefixed with ">", resulting in the "patched" text on the right (which is actually the **tichy-cm** text from figure 2.5).

The main differences between change-oriented and version-oriented models are summarized in table 2.1 on the next page. The principal advantages of the Change-Oriented Model over version-oriented models are:

A natural link to SCM processes. Most SCM processes are *change-driven*: A customer or developer issues a *change request* (CR), which is considered by a *configuration control board* (CCB), and finally incorporated into the product after CCB approval. The Change-Oriented Model allows changes to be identified as separate entities and thus linking them with change requests as these are processed.

Support for accounting and controlling is improved. Knowing the set of applied changes is important for determining the features of the final product. For instance, one can always determine whether certain faults have been corrected or whether special extensions have been made. Also, change sets may reveal dependencies between components that do not show up in the system model.

Changes may be applied to several variants at once. Representing changes as individual entities allows users to perform a change on a single version and to *propagate* that change to a whole set of versions (called *ambition*), just as a patch can be applied to files other than those it was generated from.

Many change combinations are possible. In the version-oriented models, each version incorporates all changes leading up to that version. In the Change-Oriented Model, one can choose for each change set whether it should be

	Version-oriented models	Change-oriented models
Version space	version graphs (revisions and variants); version attributes	product-level changes; attributes controlling change application
Configuration	$\sum$ component versions	base version + $\sum$ changes
Product structure	white box approach (query references the structure)	black box approach (structure transparent to the query)
Version rules	expressions over version attributes	expressions over change attributes
Constraints	conditions on version attributes (e.g. consistent variant selection)	conditions on change combinations (e.g. c_1 implies c_2)
Versioning	explicit (members of the version graph)	implicit (any change combination)
Combinability	v^m (m modules in v versions)	2^v (v changes)

Table 2.1: Version-oriented vs. change-oriented models (from [CW96a])

applied or not. For instance, one may create a version that excludes all changes but the latest one, which is not possible in version-oriented models.

A problem with change propagation is that the user may not survey how his change to a single version is propagated to the remaining versions. Another problem occurs with the ability of applying and combining arbitrary changes: one must make sure that illegal combinations are excluded. Each application of a change set C must ensure that all changes C' that C relies upon are applied as well.

Until recently, change-oriented SCM systems did not allow users to specify such mutually exclusive changes. Only combinations resulting in a *conflict* were automatically excluded—that is, the change cannot be applied because the original lines are not found in the base line.[2] In [Mun96], Munch describes the HICOV system, a constraint-based system that allows users to structure the configuration space. It remains open, however, whether these constraints could actually be used to model "traditional" version graphs and thus result in a unified SCM model.

Another recent approach that attempts to unify change-oriented and version-oriented models is the ASGARD system [MC96], which is realized on top of CLEARCASE. In ASGARD, each user groups his changes according to a specific *activity.* An activity is a group of related changes (e.g. fixing bug #327, extending the editor, changing the font resolution, and so on) and can thus be defined as a

[2] See section 5.6 for a description of conflicts.

process resulting in a *change set.* This simple and intuitive scheme is useful for organizing SCM tasks and will help to introduce change-oriented versioning in practice.

2.8 Discussion

We have identified several concepts used for maintaining evolving components. Various versioning models are used to denote variants, revisions, and components. As Conradi states in [Est95, p. 80], there is not yet a common agreement on basic versioning models. At least, the versioning models can be identified and classified; see [CW96b] for a detailed discussion.

Tools like DIFF can determine the difference (or change, or delta) between versions automatically; this is useful for maintaining repositories in which a multitude of versions can be stored in a compact fashion. Using tools like CPP or multi-variant editors, users can apply changes to several versions at once. More advanced tools, especially suitable for structured texts (e.g. programs) will be discussed in section 5.6.

In contrast to the Checkin/Checkout Model, where developers copy individual *versions* from and to a central repository, the Change-Oriented Model focuses on *changes* being applied to a baseline. Managing changes instead of versions allows for smooth integration into common SCM processes and provides much flexibility in combining change sets. Until recently, the Change-Oriented Model lacked a notion of inconsistency across change sets. This is now addressed by constraint-based systems like HICOV, although it still seems difficult to integrate both version-oriented and change-oriented versioning in a unified model.

Both the Checkin/Checkout Model and the Change-Oriented Model are primarily concerned with single components; support for component *relationships* is poor. Such structure functionality is found in the *Composition Model,* which is discussed in the following chapter.

My second remark is that our intellectual powers
are rather geared to master static relations
and that our powers to visualize processes evolving in time
are relatively poorly developed.

— EDSGER W. DIJKSTRA, Go To Statement Considered Harmful

To look back to antiquity is one thing, to go back to it is another.

— CHARLES CALEB COLTON

Chapter 3

Structure Functionality

We extend SCM from the component level to the system level, using the concepts of the Composition Model. *The central concepts in the Composition Model are a* system model *describing the system structure and* configuration rules *describing which component versions are to be selected. Developers operate on configurations by composing a system from its components and by selecting the desired version for each component.*

3.1 Describing the System Structure

To build a software product, components are assembled to form a *software system.* To keep the terminology simple, we denote the set of all software components that form a product as *software system,* any subset thereof as *software subsystem,* and any unbreakable item as *component.* A software system together with any non-software items (such as documentation) forms the *software product.*

An unstructured set of components is not enough to describe a software system. A *system model* is required that describes the architecture of a software system, that is, its structure, its components, and how to build it [Dar91]. Since the system model evolves with the software system, it must be subject of CM procedures; it is a basic CM principle that the system model must be explicit, unambiguous and managed as an item in its own right [Whi91].

3.2 System Models for SCM

System models are commonly defined by describing the *relationships* between the software items—that is, software components, subsystems, and systems. The simplest system model describes a system as the aggregation of its components.

Its basic relationship is *is-a-part-of*: An item *A* is said to be *part* of an item *B* if *B* contains *A*. Using *is-a-part-of*, one can decompose a system into subsystems and atomic components and thus describe item hierarchies.

Recent time has seen considerable advances in system modeling, especially with the introduction of modular and object-oriented approaches. For SCM purposes, specialized system models have been developed. Besides *is-a-part-of* relationships, these also reflect the relationships between versions.

3.2.1 AND/OR Relationships

Among the first concepts that included version concepts in a system model were *AND/OR graphs* [MNR83, Tic81]. In an AND/OR graph, aggregates (systems and subsystems) are modeled by AND nodes; an edge leading from an AND node *A* to a component *C* indicates that *C* is a part of *A* (*is-a-part-of* relation). To model version alternatives, special OR nodes are introduced. Each edge leading from an OR node *O* to a component *C* indicates a possible alternative; *C* is a possible *version* of *O* (*is-a-version-of* relation).

As an example, consider the AND/OR graph shown in figure 3.1 on the facing page. The system *S* is present in two versions 1.0 and 2.0. Version 1.0 consists of the subsystem *R* and the component *C*. *R* itself comes in two versions 1.0 and 2.0; version 1.0 of *R* is built from two arbitrary versions of the components *A* and *B*; version 2.0 of *R* requires specific versions of *A* and *B*.

3.2.2 Database Relationships

One of the drawbacks of the AND/OR graph model is that it does not distinguish between different version kinds: there is no way to determine an ordering between versions. Such distinctions were introduced in later models. In his survey on version modeling in engineering databases [Kat90], Katz replaces the *is-a-version-of* relation by two new relations: The *is-derived-from* relation models revision histories; the *is-a-kind-of* relation models *generic components*—the set of all versions of a component. His system model distinguishes four types of relationships:

is-a-part-of: A component *A* is said to be *part* of a component *B* if *B* contains or uses *A*. *B* is thus either a client of *A*, using *A*'s functionality, or an aggregate containing *A*. *is-a-part-of* relationships model component hierarchies.

is-derived-from: A component *A* is *derived* from a component *B* if *A* is a version based on *B*. Typically, *A* is a revision of *B*; Katz does not distinguish

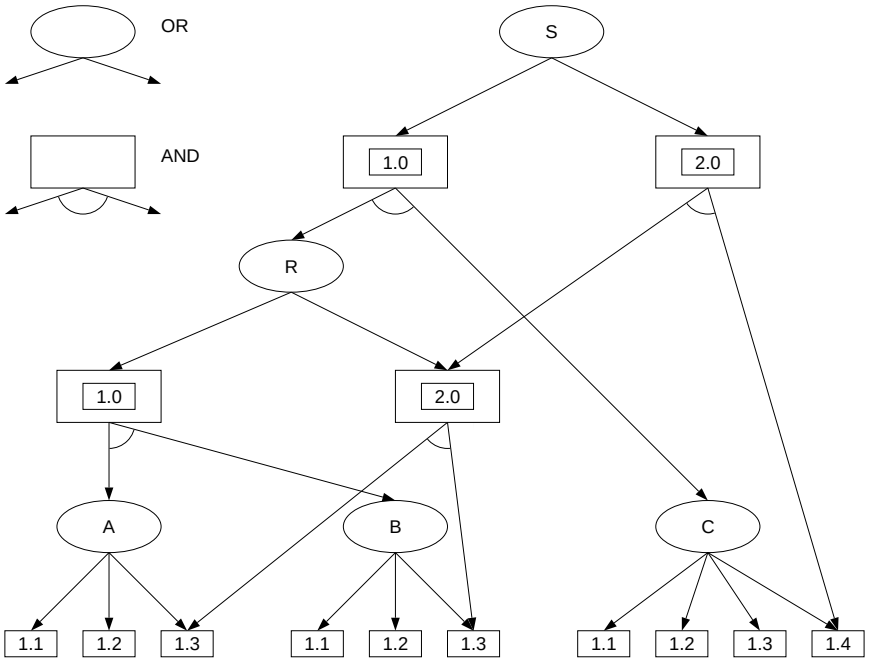

Figure 3.1: An AND/OR graph (from [Tic88])

between revisions and variants. Using *is-derived-from* relationships, one can determine the version graph.[1]

is-a-kind-of: A component *A* is a *kind* of *B* if *A* is an instance of the generic component *B*. *is-a-kind-of* relations unite specific versions of a single component.

is-equivalent-to: Some applications, especially CAD, provide a variety of component *representations.* These can be tied together using *is-equivalent-to* relationships.

An example of *is-derived-from* and *is-a-kind-of* hierarchies is shown in figure 3.2 on the next page. The component *ALU.Layout* comes in the five versions

[1] Note that the term *derivation* is more frequently used for denoting the relationship between source components and derived components.

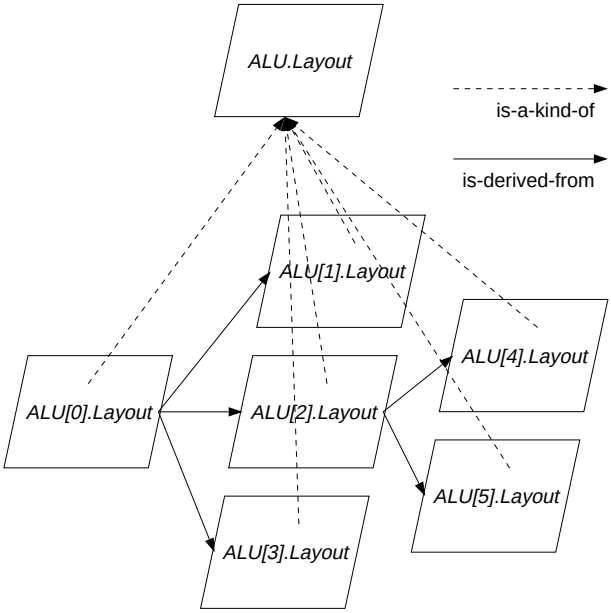

Figure 3.2: A database relationship graph (from [Kat90])

ALU[0].*Layout* to *ALU*[5].*Layout*. *ALU*[0].*Layout* is the original version; both *ALU*[4].*Layout* and *ALU*[5].*Layout* are derived from the version *ALU*[2].*Layout*.

Since Katz's system model originates from maintaining design data, it provides no relationships between target components derived from source components, as discussed in chapter 4; Consistency issues (see section 3.6) are left unaddressed as well. Such issues, specific to software construction, were introduced in specific SCM models, such as the one realized in the *Configuration Management Assistant* (CMA), discussed in section 3.6.2. It remains unclear, though, how SCM *operations*—transitions between relationship graphs—are to be modeled and how constraint relationships such as consistency or compatibility are to be verified.

3.3 Selecting System Configurations

From a system model, the SCM system (and the developers) can determine what components are part of the system. To work on a particular set of components, they determine a *configuration.* A configuration is a collection of components

tailored for a specific purpose [Whi91]. Typically, a configuration meets the needs of a particular environment or user, which is identified by *configuration rules* denoting the components and their respective versions.

The configurations described by configuration rules can be grouped into three *configuration types.*

Bound configuration. A *bound configuration* [LCS88] describes an unambiguous configuration independent from a specific context, as the current time or the state of other components. Bound configurations are typically used to identify product releases as shipped to customers.

Generic configuration. In contrast to bound configurations, a *generic configuration* [Tic88] (also called *partially bound* [LCS88]) describes an unambiguous configuration dependent on the context; for instance, a rule specifying the most recent version of a component. Generic configurations are typically used in software development and production.

Abstract configuration. Both bound and generic configurations denote an unambiguous set of components and versions. In case the rules are ambiguous, the configuration specification is incomplete. We call such a configuration *abstract* because of the similarity to abstract superclasses in object-oriented design (see section 3.3.6 for details); the synonyms *dynamic configuration* [Kat90], *configuration template* [Fei91a, Sch95], *configuration family* [PF89], and *ambition* [LDC+89, MLG+93] are also found.[2] Abstract configurations allow for describing sets of configurations and have recently found increased interest in the domain of dynamically composed systems (DCS) [SM95a, SM95b].

The configuration rules as realized by SCM systems are discussed below.

3.3.1 Tagging Configurations

Simple SCM tools like SCCS and RCS provide bound configurations: specific versions are tagged with a label (a *configuration tag*) identifying the configuration. This allows for the definition of a *configuration baseline.* RCS and SCCS do not

[2]The term *dynamic configuration* is prone to confusion, since it is widely used in the context of adaptive systems as the ability to modify the structure of an application while the application continues to operate [WS95]. The term *configuration template* suggests an instantiation instead of a refinement. The term *configuration family* implies a finite, well-defined set of possible configurations, which need not be, and the term *ambition* is too closely related with change propagation, as discussed in section 2.7.

allow for specifying the set of components actually included in the configuration. This is handled by the *Concurrent Versions System* (CVS) [Ber90], which extends the tagging mechanism to software systems and thus identifies the set of components in the configuration. In all these simple SCM systems, generic configuration is supported only through selection of the most recent version.

3.3.2 Boolean Attribute Queries

The configuration rules of more advanced SCM systems reflect the respective identification schemes, as discussed in section 2.3. The basic idea is to use boolean expressions which must be satisfied by the identification term of selected version.

The *option space* as described by Lie *et al.* [LCD+89] is closely related with the Change-Oriented Model, where each change can be applied or not. Consequently, configurations are described by a formula in propositional logic, where each proposition (called *option*) may be true, standing for a change to be applied, or false, meaning that the change not be applied.

In ADELE [Est85, Est88, EC94], variants are identified by *attributes,* where each attribute can have an arbitrary value; thus, one is not restricted to boolean values as in the option space. The user can designate a configuration by specifying a boolean term based upon the desired attributes. The ADELE configuration rule

$$\textit{window-system} = \textit{x11} \wedge (\textit{current} \vee \textit{status} \neq \textit{experimental})$$

includes all components in a configuration whose window system is X11; only current or non-experimental components are to be included. Revisions are selected in a similar fashion by imposing constraints on the *date* attribute (e.g. $\textit{date} < 18\_02\_89$). Through this flexible and general scheme, ADELE supports both bound and generic configurations.

In Nicklin's *context model* [Nic91], a similar scheme is used. As an extension, attributes can be undefined: referencing an undefined attribute results in an undefined value of the selection term. The richest model of boolean queries, however, is found in the JASON system [Wie93], where full first-order logic may be used, including existential and universal quantifiers. These queries can also be used as general *configuration constraints,* as discussed in section 3.6.1.

3.3.3 Preferences and Defaults

As most SCM systems cannot handle ambiguity, they provide means to make selections unambiguous. The idea is to provide special configuration rules for these tasks:

Preferences. A *preference* rule applies if the selection is ambiguous. It selects one "most preferred" version out of the selection.

Defaults. A *default* rule applies if the selection is empty. It makes the selection contain one "default" version.

As an example for preferences and defaults, consider the SHAPE system. In SHAPE, configuration rules are specified in a PROLOG-like syntax. Each rule denotes alternatives of boolean conjunctions; the rules are specified according to their preference: the most preferred versions come first, the least (the default) comes last.

Figure 3.3 shows a SHAPE configuration rule that implements a change of a component status from "saved" to "proposed" (components are either saved, proposed, or published).

```
i_test_rule :– gt(status, saved), max(version);
               eq(status, proposed), eq(test_switch, on);
               ge(status, published), max(version);
               cut(Cannot bind $+ — something's wrong here!).
```

Figure 3.3: A SHAPE configuration rule

The first preference clause selects the most recently published version with status saved or better. If the first clause fails, such a version is is not available. Hence, the second clause chooses a proposed version dedicated for testing (with a test switch set to *on*). If this clause again fails, the next *default* clause applies, stating that all remaining objects are to be chosen from the home baseline—that is, the most recently published version. If this clause also fails, the final clause issues a diagnostic and aborts the selection.

3.3.4 Preferences in Queries

Another approach for specifying preferences and defaults is found in *database queries.* When databases are used as component repositories, database queries are used to retrieve specific component versions. In [LL87], Lacroix and Lavency point out that traditional database query languages are not sufficient for selection of configurations. Since configuration queries are *intensional,* they denote objects by their properties rather than by their name (or exact version specifications). But intensional queries may be ambiguous and result in more than one selected version; the SCM user must select the best suitable version manually. The

```
select the instances of CONF
  having
    the version of MAIN
      having
        same TARGET as the version
          of PROCESS-DATA and
        same TARGET as the version
          of GET-DATA
from which
  prefer those
    having
      the version of MAIN
        having STATUS = tested
  prefer those
    having
      the version of PROCESS-DATA
        having STATUS = tested
```

Figure 3.4: A database selection rule with preferences (from [LL87])

authors thus suggest to extend database query languages by preferences and defaults to make the selection process explicit. A self-documenting example of such a database query, selecting component versions with a certain status, is shown in figure 3.4.

3.3.5 Search Paths in the Version Graph

All query mechanisms discussed so far rely on versions tagged with a set of attribute/value pairs; each query mechanism can be expressed by specifying a first-order boolean formula which the selected versions must satisfy (for database selection rules, second-order formulas may be required). Systems relying on other identification schemes provide alternate configuration rules.

As discussed in section 2.3.2, CLEARCASE identifies versions by labeling edges in the version graph. The CLEARCASE configuration rules are thus *search paths* in the version graph. Search options can include the work areas, variants, and revisions in either all components or selected subsets.

Figure 3.5 on the facing page illustrates the usage of configuration rules in the CLEARCASE system. Each rule, beginning with the keyword element, contains a *wildcard* denoting the components it applies to ("*" applies to all components) and a *version graph query.*

```
— Rules for maintenance to an old release:
— if the file is checked out, use this version.
element   *   CHECKEDOUT
— otherwise, use latest version on maintenance branch.
element   *   .../vs_fixes/LATEST
— otherwise, use the official V2 released version.
element   *   V2 -mkbranch v2_fixes
```

Figure 3.5: CLEARCASE configuration rules (from [Leb94])

If a query finds one or more versions, the latest version is taken; otherwise, the next rule is tried. Each developer is assigned a set of rules describing his particular environment.

3.3.6 Refinement of Configurations

Rather than disambiguating selections as soon as possible, a few SCM systems also handle abstract configurations, as discussed in section 3.3, and allow for operating with several configurations at once.

The JASON system [Wie93] uses partial attribute descriptions to denote abstract configurations. Abstract configurations are used as abstract superclasses of further instantiated configurations; subclassed configurations inherit the attributes of their superclasses. JASON thus realizes an *object-oriented* SCM model.

Figure 3.6 on the next page illustrates JASON configuration descriptions. The configuration *EMailSpec* is defined as a subclass of *DesignSpec*: an abstract configuration denoting all electronic mail systems, inheriting all *DesignSpec* attributes like *contents*, *version*, or *revision*.

Even more concrete (less abstract) configurations may be obtained through further subclassing: Starting with an abstract configuration like *EMailSpec*, the set of configurations is constrained through additional attribute specifications until a fully instantiated (bound or generic) configuration is obtained.

3.4 Integrated Configuration Languages

Recently, specialized *configuration languages* have been developed that attempt to integrate all SCM aspects of system modeling into one single formalism. PCL, the configuration language of the PROTEUS system [TGC95], allows to express variability in the composition of a system, including relationships between components and versions, as well as the selection of a bound consistent configuration (called *binding* in PROTEUS).

In figure 3.7 on the facing page, we see a PCL example modeling a family of calculator programs named CalcProg. The **attributes** section declares the attributes by which the individual versions differ—in this case, one version has a graphical user interface (xgui = true), and the other does not.

The **parts** section declares the components of the CalcProg family; calc is a member of the Calculator family, while math is a member of the mathlib family. The user interface part, is only present in the graphical user interface version, as a member of the XGUI component family; the non-graphical version (xgui = false) does not require such a component.

In PROTEUS, primitive entities like Calculator are mapped to physical files. Again, this mapping can be subject to variability, as shown in figure 3.8 on page 32—if the expression attribute is set to infix, the files expr.C and expr.h are chosen, and if expression is set to reverse_polish, the files rpn_expr.C and rpn_expr.h are chosen.

Version selection is done by a simple instantiation of attributes; for instance, by assigning the value true to the xgui attribute and the value reverse_polish to the expression attribute. PROTEUS also allows *partial instantiations* to refine the selection incrementally.

The benefit of a full-fledged configuration language like PROTEUS is that it integrates several SCM aspects—in this case, system modelling, configuration selection, and manufacturing—into one single formalism. The question is how far such a formalism is more than the sum of its parts. If each SCM aspect is represented by yet another language feature, the language gets easily overloaded by individual, non-orthogonal features.

```
DesignSpec: class
{
        system:   String,
        contents: Document,
        version:  Integer,
        revision: Integer
}
EMailSpec: family of DesignSpec
{
        system = "Electronic Mail System"
}
```

Figure 3.6: JASON configuration descriptions (from [Wie93])

```
family CalcProg
    attributes
        ...
        xgui: boolean default false;
    end
    parts
        ui    ⇒  if xgui = true then XGUI endif;
        calc  ⇒  Calculator;
        math  ⇒  mathlib;
    end
end
```

Figure 3.7: Structural variability in PCL (from [TGC95])

3.5 Visualizing the Configuration Space

To keep track of the growing number of possible configurations, users must be able to conceptualize and visualize the configuration space. In this section, we present some visualization techniques.

Version graphs. The first approach to visualizing the version space, and still by far the most popular, is to display component-based version graphs and let the user choose versions interactively. Version graphs are useful for single components only and thus useful for SCM tools realizing the Checkin/ Checkout model. In figure 3.9 on page 33, we see a revision graph as displayed in RCE [Xcc95, Tic95], an RCS successor providing a graphical user interface.

Version threads. To illustrate version selection for systems built from several components, *version threads* have been suggested as notation, as shown in figure 3.10 on page 34. Each system revision (shown on the left) consists of one revision of each system component, as indicated by the specific version thread. This notation does not support variants, even temporary ones, and does not visualize consistency constraints.

Constraint formalisms. Both version graphs and version threads only show a set of existing configurations, rather than visualizing the set of possible configurations. In [Gul93], Bjørn Gulla presents a visualization of configuration constraints using graphs. Nodes indicate configuration options, arrows implications between options, diamonds stand for disjunctions and thick dot-

```
family Calculator
    attributes
        ...
        expression: expr_type default infix;
    end
    physical
        calc  ⇒  ("Calculator.C", "Calculator.h");
        expr  ⇒  if expression = infix then
                     ("expr.C", "expr.h")
                 elsif expression = reverse_polish then
                     ("rpn_expr.C", "rpn_expr.h")
                 endif;
    end
end
```

Figure 3.8: Mapping variability in PCL (from [TGC95])

ted lines represent mutually exclusive sets. Different abstraction levels are obtained by defining new options as subexpressions (or subgraphs).

In figure 3.11 on page 35, users can choose between one of the mutually exclusive options PM, X11, or SunView. After choosing X11, users have the choice between Hp9000, Dec, and Sun3, while PM implies the IBM machine just as SunView or Sparc imply the Sun3 machine.

As no technique is fully satisfying, it is obvious that the work on visualization of configurations is still in its infancy. As Gulla himself states, "this is a first proposal that will probably need refinements and validation in an industrial environment."

3.6 Interfaces and Consistency

Selecting an arbitrary configuration from a collection of components does not suffice; as stated in section 1.3, the configuration must be consistent. In SCM systems, we find maintenance of external consistency (respective to some specification) and of internal consistency (the syntactic and static correctness of a program).

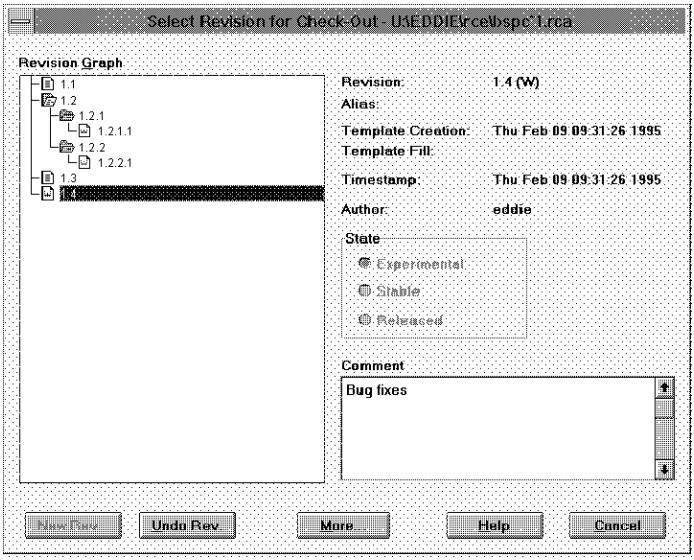

Figure 3.9: Version selection from a RCE revision graph (from [Xcc95])

3.6.1 External Consistency

External consistency is consistency respective to a specification separated from the software components. Typically, such a specification is coupled with the identification scheme; it can be expressed through *consistency constraints* in the configuration selection rule as discussed in section 3.3.

As consistency constraints usually apply to each possible configuration, they are often separated from the actual selection rules. Each consistent configuration, selected in a separate process, must satisfy these constraints. The JASON system, for instance, allows to specify *configuration constraints* as first-order boolean formulas on version attributes including universal and existential quantifiers. The scheme is general enough to specify module interconnection constraints like "No resource is provided by more than one component", as illustrated in figure 3.12 on page 36.

Another generic approach is found in the *Configuration Management Assistant* (CMA). In [PF89], Ploedereder and Fergany introduce the following relationships to model source/target and consistency dependencies:

is-instance-of: *Instance relationships* are used to model dependencies between

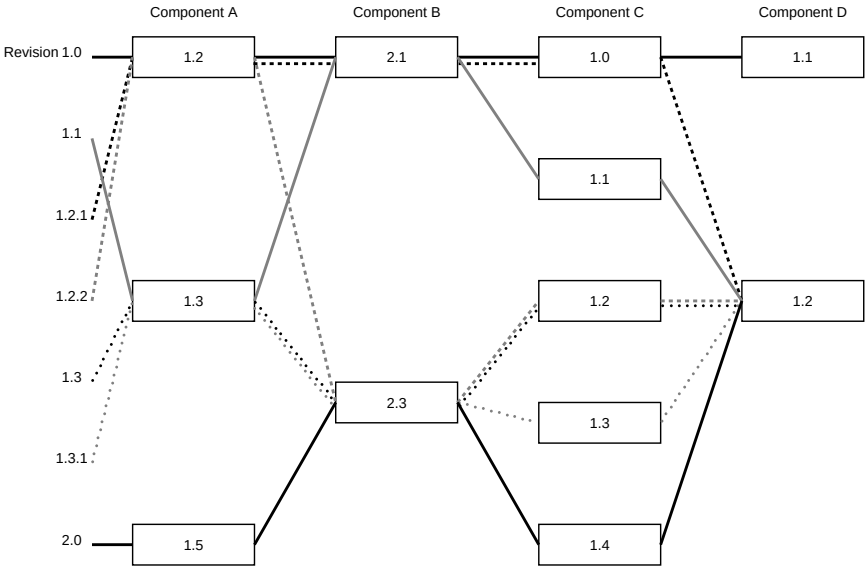

Figure 3.10: Version threads (after [Gul93])

source components (e.g. source code) and derived components (e.g. object code).

is-consistent-to: Two components are said to be consistent with each other if "they correctly operate together".

is-compatible-to: Two versions of a component are called *compatible* if replacing one version with another still results in a consistent system.

Based on the semantics of the version attributes and these relationships, the CMA can determine the consistency of a configuration. However, as in other SCM systems, consistency largely relies on user specifications.

3.6.2 Internal Consistency

In some cases, consistency violations can be determined automatically when the actual contents of the software components are taken into account. For instance, violations of the *static correctness* of a software system can be verified. The simplest way to determine violations is to rely on the build tools and check for failing

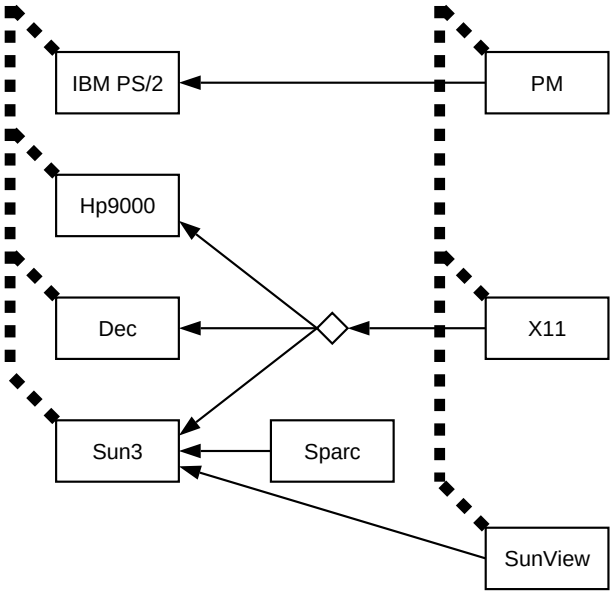

Figure 3.11: A constraint diagram (after [Gul93])

build attempts; in chapter 4, we discuss how SCM systems covering software builds maintain the static correctness by determining the impact of a component change and rebuilding all dependent components.

Besides this basic functionality, some SCM systems infer and use *interface information* for maintaining the static correctness for a configuration. Such an approach is found in the IPSEN software development environment [ELN+92, SS95]. Based on the module interfaces as specified in the components and the inferred dependency graph, IPSEN can ensure the syntactic and static correctness of a configuration. In the proposed versioning model for the PSG system [SGS91, Sch95], such consistency violations can even be deduced for fine-grained changes within components.

3.7 Discussion

The Composition Model extends SCM from the component level to the system level. The system structure is expressed in a system model. Developers operate on configurations by first composing a system from its components and

```
Rule-2: constraint on (config: Configuration)
    for-all comp-1, comp-2 in config.components:
        comp-1 ≠ comp-2 implies
            for-all resource in comp-1.provides:
                not comp-2.provides(resource)
```

Figure 3.12: A JASON constraint specification (from [Wie93])

then by selecting the desired version for each required component. Several selection schemes exist, from pattern-matching search paths in the version graph via first-order boolean formulas to full-fledged database queries. Consistency is ensured through appropriate selection schemes or through additional constraints; SCM systems tailored for specific programming languages may also check for internal consistency.

The Composition Model does not support changes as individual entities, as does the Change-Oriented Model. As such, the Composition Model does not provide special construction or team facilities. These facilities shall be discussed in the following chapters.

Mahler: Is a configuration a description
or is it the result of applying the description?
Audience: Yes! (Laughter)

— SUMMARY OF SVCC'88 PLENARY DISCUSSION [Win88]

Chapter 4

Construction Functionality

Building a software system requires a system model enhanced with build information. The simplest of these system models is a build command file *containing a procedural description of the processing steps to build all derived components of a configuration from the source components. Through more advanced system models, a SCM system can support automated incremental software construction and perform management of derived components.*

4.1 Component Dependencies

For large systems, building a system from scratch can be very expensive, especially, if the system must be completely rebuilt after each change. The solution to that problem is to determine the components affected by a change in a source component. In general, a component *A* is said to *depend* upon a component *B* if a change in *B* might require changes in *A* such that *A* remains correct. Whitgift [Whi91] distinguishes four types of dependency:

1. An *implementation* of a component depends upon its specification.
2. A *derived component* depends upon its source components.
3. A software component depends upon the components whose *functionality* it uses.
4. *Documentation* and *program* code depend upon each other.

Most of these dependencies must be resolved manually after a change, but dependencies of type 2 can be processed automatically through *incremental construction.*

4.2 Incremental Construction

One of the first approaches for incremental software construction and probably one of the most successful software tools ever written, was Feldman's MAKE tool [Fel79]. In MAKE, the system model is represented through a *Makefile.* The Makefile declares the dependencies between source and derived components and the processing steps to build derived components. At each MAKE run, MAKE checks the last modification date of all source and derived components. Each derived component that does not exist or that is dependent on a younger source component is rebuilt.

As an example, consider the simple Makefile shown in figure 4.1. Each dependency is shown by a declaration of the form D: $S_1\, S_2 \cdots S_n$, meaning that the derived component D depends on the n source components $S_1, \ldots, S_n$. The actual commands building D follow the dependency declaration. For instance, the tty.o component depends on the source components tty.c and common.h; to build it, the command cc -c tty.c is issued. For convenience, OBJECTS defines a list of objects referenced as $OBJECTS.

```
OBJECTS = tty.o display.o
editor: $(OBJECTS)
    cc -o editor $(OBJECTS)
tty.o: tty.c common.h
    cc -c tty.c
display.o: display.c common.h
    cc -c display.c
```

Figure 4.1: A simple Makefile

Should the tty.c component be changed after a build, the display.o component will not be rebuild, because it does not depend on tty.c. Only the tty.o and editor components will be rebuilt. Should the common.h component change, all objects must be rebuilt, since all depend on common.h.

The problem with MAKE when used in an SCM context is that MAKE does not determine dependencies and that it does not know about component versions; some MAKE extensions like GNU MAKE at least include conditional evaluation and automatic check-out from RCS repositories. Also, relying only on the modification date to determine changes may result in unnecessary rebuilds. These problems were addressed by later build tools that allowed for automatic dependency determination, versioned source access using the configuration selection rules and for automatic identification of derived components with their prove-

nance and build environment.

4.3 Determining Dependencies Automatically

With language-specific knowledge, build tools can automatically deduce dependencies and the impact of changes. The ODIN system [Cle88, Cle93], for example, can automatically deduce dependencies by scanning source components for appropriate statements. This scanning is language-dependent; for instance, components written in the C or C++ programming language are scanned for #include directives. ODIN saves its derivation history across builds; this allows for deleting intermediate components such as object files when the final system does not need to be rebuilt.

Another language-specific approach is found in the RATIONAL software development environment [FDD88, Mor88]. RATIONAL can determine the impact of changes to ADA programs—for instance, a change applying to comments only does not cause any rebuilds.

An elegant and language-independent method for determining dependencies is undertaken in CLEARCASE. Through its virtual file system, discussed in section 5.4.2, the CLEARCASE MAKE utility (called CLEARMAKE) monitors all file accesses performed by the build commands and thus determines all dependencies while the system is being built. For each derived component *C*, each file accessed is considered a source component that *C* is dependent upon.

4.4 Versioned Software Construction

In all SCM systems supporting software construction, building a system is done by specifying the desired configuration, as discussed in section 3.3. The main problem is the identification of derived components, which must take the entire build environment into account—that is, the versions of the source components as well as the versions, parameters, and environment variables of the build tools.

In CLEARCASE, each derived component is tagged with a *bill of material* (also called *bound configuration thread* or BCT) describing the build environment. The bill of material is determined automatically file access monitoring. The unfortunate side effect is that minor changes in the environment—for instance, the change of an environment variable unrelated with software builds—may result in an unnecessary rebuild. CLEARCASE thus allows to distinguish between *critical* environment aspects (those that cause a change in the derived components) and *non-critical* aspects (whose change does not imply a rebuild).

In the SHAPE system, the user has a similar control about the settings that influence rebuilds. For each variant, the user can specify by which MAKE variables

it is dependent upon. Hence, the change of a compilation flag may result in a rebuild, while the change of the installation directory may not. Similar approaches have been undertaken by Kielmann [Kie92], who uses PROLOG for software construction.

4.5 Attribute Propagation

The CAPITL system [RS91, AS95] uses a description logic called *Persistent objects with logic* (POL) to identify components and to infer build plans. POL terms are conjunctions of *name* $\Rightarrow$ *value* pairs, called *attributes.* Each component is tagged with a POL term denoting its attributes.

For the purpose of planning and building, six attributes are used:

code: a list of possible build expressions;

contents: the contents (e.g. source or object code) of the component;

provenance: the record of how the component was created;

form: its type when used as argument to a tool;

functionality: a description of what the component does; and

references: other components this component depends upon.

Through the provenance attribute, each derived component is tagged with its *derivation history* and thus uniquely identified. Just as in SHAPE, users can control which attributes cause differing variants and how attributes are propagated from tools and source components to derived components.

As an example for attribute propagation, consider the tool specification rule in figure 4.2 on the next page. The specification Cc_debug describes an executable C compiler whose functionality is to generate an object_code from a c_source. The functionality F, which matches an entire POL term, is propagated from the source component to the object component. However, the dbg_sym and opt attributes of the generated object codes differ. The Cc_debug tool generates debugging symbols and thus sets the dbg_sym attribute to yes; as it does not optimize, the opt attribute is set to no. Using the Cc_opt tool, these attribute values are just inverted.

By making attribute propagation explicit and through its underlying well-defined attribute logic, CAPITL provides the most versatile identification scheme for derived components found in today's SCM systems. As POL terms can also be denoted as graphs (an alternate name is *cyclic terms*), they also provide a means

```
Cc_debug: obj(
      form ⇒ executable,
      functionality ⇒
           func(in  ⇒ obj(form ⇒ c_source, functionality ⇒ F),
                out ⇒ obj(form ⇒ object_code(dbg_sym ⇒ yes, opt ⇒ no,
                               functionality ⇒ F),
      contents ⇒ "⟨actual Cc executable code⟩"
),
Cc_opt: obj(
      form ⇒ executable,
      functionality ⇒
           func(in  ⇒ obj(form ⇒ c_source, functionality ⇒ F),
                out ⇒ obj(form ⇒ object_code(dbg_sym ⇒ no, opt ⇒ yes,
                               functionality ⇒ F),
      contents ⇒ "⟨actual Cc executable code⟩"
)
```

Figure 4.2: Tool specifications in CAPITL (after [AS95])

to unify attributes and relationships: each relation $X \to Y$ is represented by an attribute in X with a value of Y and a name standing for the relation kind.

4.6 Optimized Software Construction

Most SCM repositories only store source components, since determining the difference between derived components (often binary files) does not lead to efficient compression of the repository. Many SCM systems provide a *cache* for derived components (also called *object pool* or *binary pool*), where frequently used derived components are stored.

When components are unchanged across versions, building a derived component can be avoided when the derived component is still cached as the result of a previous build. Such techniques are found in SHAPE and CLEARCASE; of course, the source components must not have changed in between. Besides caching derived components, CLEARCASE gains additional speed through distributed and parallel construction. The correctness criteria for such build optimizations have been formalized by Gunter [Gun96].

4.7 Conclusion

Most SCM construction tools are descendants of MAKE. Typical extensions include automatic generation of dependencies, versioned software construction that propagate version identification from source components and tools to derived components, and optimizations to reuse derived components from a central cache.

The more innocuous the modification appears to be,
the further its influence will extend
and the more the design will have to be redrawn.

— FYFE'S SECOND LAW OF REVISION

Chapter 5

Team Functionality

To allow for parallel work, SCM systems provide the notion of a workspace*, isolating developers from each other's changes. SCM systems differ in the way workspaces are realized and in the specific* cooperation strategy*—that is, how changes are propagated across workspaces.*

5.1 Cooperation through Workspaces

One of the central functionality areas in SCM is *team functionality.* Team functionality enables a team of developers to develop and maintain the software product. The benefit of team functionality is that developers can work in parallel, isolating individual developer's changes from each other and coordinating the propagation of changes.

The central concept in team functionality is the *workspace* (also called *long transaction,* due to a similarity with database transactions [EGLT76, Gra81]). A workspace is the individual area of a developer, isolating him from changes made by others, and isolating others from his changes. Any propagation of changes across a workspace boundary is an explicit SCM operation.

A workspace is usually accessed as a file system. This is necessary because the vast majority of software development tools cannot access its sources directly from the repository, but requires sources in a file system instead. Hence, workspaces perform the *integration* of a SCM system into a software development environment.

Other aspects of team functionality are *cooperation strategies* and *conflict resolution.* When developers work in parallel, the SCM system must ensure that their changes do not conflict with each other. This is realized through a cooperation

strategy that either relies on locking components against changes or on merging parallel changes. Finally, the SCM system must provide support for projects that span multiple sites.

5.2 Workspaces as Private Directories

The simplest workspace concept is that of a private file system (e.g. a user's directory), copying versions from and to the central repository. This is the base of the Checkin/Checkout Model, as discussed in chapter 2. Developers must copy (or *check out*) components from the repository into their workspace (a private directory), work with them and copy them back (*check in*) into the repository after changes have been made. Besides the components the developer wants to change, the workspace must also contain all components required for compilation, testing, or searching; these must be checked out as well.

This component-based approach can be extended to systems; in fact, most repository-based SCM systems following the Composition Model use this scheme. The CVS system, for instance, allows for checking out all components of a system at once, creating a private copy of the entire system source for each developer. CVS provides an automatic scheme that exports all changes from the private workspace to the central repository and vice-versa, synchronizing the workspace with the repository.

This "to-and-fro copying" scheme has one advantage, its simplicity. It also has several disadvantages.

Copying is waste. Giving each developer a private copy of the entire system may require huge amounts of storage resources. Copying can be affordable for medium-sized projects; in fact, the CVS developers state that the purchase of additional mass storage for a new developer can be neglected when compared to other work costs. But maintaining a copy for each developer is unlikely for large systems with thousands and thousands of developers—especially because every developer must build his own system copy.

Sharing is non-transparent. Some SCM systems suited for large systems provide sharing mechanisms that allow developers to share environments. Unfortunately, sharing is non-transparent to the developers, who must take additional care when accessing shared versions.

Components are copied away from version control. This is the central problem with copying schemes: a checked out component is no more under SCM control. Neither can the SCM system save space by determining the

version differences, nor can one use SCM tools to determine the state of a checked-out component, nor can build tools exploit equality of derived components across workspaces. Developers can propagate changes and component versions directly between workspaces, bypassing the SCM system.

These problems have led to the development of methods that allow developer tools to access the repository directly, without the need of copying to and from a repository. Using these methods, workspaces are actually parts of the repository and fully under SCM control.

5.3 Workspaces through Application Interfaces

The first approach to overcome to-and-fro copying was the development of "standard" repositories that could be accessed through an application programmer interface (API). That is, all development tools must be extended such that they access source components through the repository interface instead of the file system.

This approach has several advantages; in particular, it allows to overcome the shortcomings of a file system, such as transaction insecurity, inappropriate object identification, and so on. A developer's workspace would consist of a configuration rule, identifying the components and the respective versions. Developers can share source components and derived components (which are stored in the repository). For a survey of repository-based software engineering environments, and the required repository techniques, see [BESS96].

The single, but fatal disadvantage of such encapsulated environments is that still, a file system is the smallest common denominator between nearly all development tools; the consequence is that even when using a standard repository, users must still copy versions from and to the repository.

5.4 Workspaces through Virtual File Systems

The most successful approach to realize direct repository access is to provide a *virtual file system* mapping the repository into a file system. This ensures that derived components are created within the workspace, placing them under SCM control.

5.4.1 Explicit Version Access

On the component level, the SHAPE toolkit provides a dynamically linked library that interprets file names containing *version specifications*. This allows arbitrary programs to access the SHAPE repository directly, providing transparent version

access. For instance, opening a virtual file like prog.c:3.1 returns version 3.1 of the file prog.c. A similar approach is found the RATIONAL system.

A *generic* approach is pursued in the *multiple dimensional file system* (*n*-DFS), as discussed by Fowler *et al.* in [FKR94]. In the *n*-DFS, arbitrary *services* can be attached to a file system. For instance, a versioning service may provide direct repository access through means of virtual file names.

Instead of extending file names with versioning information, RCE provides a library that hooks into the *user interface.* RCE extends the standard file selection dialog with a version selection dialog, as shown in figure 3.9 on page 33. Whenever a user selects a file for processing, he may also select a version to work upon.

One problem is common to all these approaches: Versioning is explicit. There is no way to switch between versions implicitly, without embedding the version in the path name—or specifying the version in an interactive dialog. It may be desirable, though, to access several components from a specific configuration, without having to specify the version of each single component. This is realized through implicit version access, as described below.

5.4.2 Explicit/Implicit Version Access

Instead of appending a version specification to a path name, the CAPITL extensible file system (EFS) *prepends* the version specification. Through changing the current directory, a *current version* can be selected that applies by default: Through changing the current directory to 3.2.1:, all subsequent file accesses refer to the respective 3.2.1 version. This method allows for both implicit version access (using relative paths from a versioned directory) as well as explicit version access (using absolute paths containing the version specification).

In the CLEARCASE system, explicit and implicit access are handled by different methods. *Explicit* version access is achieved by appending the version specifier to the component name, as shown above. A CLEARCASE version specifier has the form "@@/", followed by the path in the version graph. The color variant of component zbuf.c can thus be accessed under the name zbuf.c@@/color, for instance.

Additionally, CLEARCASE allows versioned access to entire file systems via *configuration rules,* discussed in section 3.3.5. If a component is accessed without a version specifier, the version according to the configuration rules is selected. Using this two-fold scheme, CLEARCASE allows explicit version access (by appending a version specifier) as well as implicit access (by specifying the configuration rule). A CLEARCASE workspace is thus defined by a configuration rule,

providing a specific *view* on the repository.

Another approach realizing both implicit and explicit version access is realized in the SUN Network Software Environment (NSE) [Cou89], which realizes the so-called *Long Transaction Model.* In NSE, workspaces are also views on a central repository. The workspace is mounted as a virtual file system in the user's directory; upon mounting, a specific configuration must be selected. NSE *per se* thus allows only implicit version access; by mounting different configurations at different places, explicit version access can be realized.

5.4.3 Realizing Virtual File Systems

To realize virtual file systems, three major approaches can be found.

Replace the system libraries. In the SHAPE AtFS, the *n*-DFS, and RCE, virtual file access is realized through extended variants of the system libraries. That is, file accesses containing version specifications are diverted to access the repository instead. Programs must be linked with the specialized library in place of the system library; in case the operating system supports shared libraries, replacing the shared system library will suffice for dynamically linked programs.

The advantage of this approach is its good performance; the disadvantage is that, depending on the operating system, some or even all programs must be relinked to include virtual file system access. Another problem is that process size is increased with repository access code.

Provide a specialized NFS server. The NSE and the CAPITL EFS are realized on top of a modified *network file system* (NFS) [SGK+85] server. NFS was originally indented to allow network-wide file system access, but it can also be used to create virtual file systems by modifying the NFS server.

The advantage of the NFS-based approach is that any programs can access the virtual file system without modification; the NFS server is easily installed and incorporated in existing heterogeneous environments. The disadvantage is that NFS lowers performance significantly, especially in contrast to direct local file system access.

Extending the system kernel. The CLEARCASE system bypasses the NFS bottleneck by extending the operating system kernel with specialized device drivers, providing an abstract file system interface or directly replacing disk device drivers.

As all programs access their file systems through the kernel, the kernel extension approach allows for a wide range of system-specific optimizations. The NFS bottleneck for local file systems is also avoided. The drawback is that realization and installation are non-trivial tasks.

5.5 Cooperation Strategies

When several people work in parallel, it is important that their changes be coordinated such that one change does not, by accident, undo the effects of another change. As this is a key element in SCM, each SCM system realizes a specific *cooperation strategy.*

5.5.1 Conservative Cooperation Strategies

Conservative cooperation strategies prevent conflicting changes using a simple *locking scheme.* Developers working on a specific component version or configuration can lock it against further changes. While a version or configuration is locked, other developers are excluded from creating new revisions. They are allowed, however, to create temporary variants, that is, a branch in the revision history.

Explicit locking is the scheme followed by RCS and SCCS; it is also used in systems using the Composition Model such as CLEARCASE. In CLEARCASE, a workspace initially is read-only: to change a component, a developer must explicitly create a temporary variant and ensure that his configuration rule gives him access to this variant. Besides explicit locking, this scheme has the benefit that read-only components are shared across workspaces; hence, creating a workspace in CLEARCASE does not require additional resources.

Locking a version or configuration is inappropriate when a developer makes a major change over a long time, since this prevents other developers from making quick fixes. Hence, developers are allowed to create temporary variants instead, starting an individual development path. All changes made in this individual path must eventually be integrated with the changes made in the original development path, which may or may not be difficult.

5.5.2 Optimistic Cooperation Strategies

In contrast to conservative strategies, an *optimistic* strategy by default allows parallel changes; changes are integrated in a later stage. In an optimistic strategy, each developer is assigned individual temporary variants to work upon. The CVS

and NSE systems, for example, realize optimistic cooperation strategies through workspaces.

When a CVS or NSE workspace is created, temporary variants are created for *all* configuration components, resulting in a multitude of branches. This scheme allows developers to perform changes to any component without further explicit branching. Despite abundant branching and creation of temporary variants, optimistic strategies need not be inefficient: NSE implements a "copy-on-write" policy where unchanged components are shared between the originating version in the repository and the derived workspace; a similar technique is found as *view-pathing* in the *n*-DFS.

Optimistic strategies are appropriate when the number of expected conflicts is low—for instance because parallel development is made on disjunct subsystems, making conflicting changes unprobable.

5.6 Merging and Conflict Resolution

In both conservative and optimistic cooperation strategies, parallel changes must eventually be integrated or *merged.* To see how this can be done, we take a look at the conflict resolution strategies as found in SCM systems. Each of the following strategies creates a so-called *merged version* that integrates the changes from two or more temporary variants.

5.6.1 Textual Merging

The most frequently found mechanism for change merging is *textual merging,* as realized in the UNIX tool DIFF3. The DIFF3 program performs a three-way comparison between two temporary variants V_1 and V_2 and their common ancestor V_0, the so-called *base version.* V_1, V_2, and V_0 are scanned in parallel. Each text fragment that occurs in V_1 and V_2 is included in the merged version M. If a text fragment differs between V_1 and V_2, then only the text fragment different from V_0 (that is, the changed one) is included in M. A text fragment different in all three versions V_1, V_2, and V_0 indicates a *conflict*: the text fragment has been changed both in V_1 and V_2. Such a conflict must then be resolved manually.

The principal limitation of textual merging is that the content of the text is not considered. Whether two changes conflict or not is simply determined by size of text fragments compared: the smaller the textual distance between two changes, the higher are the chances that they be flagged as in conflict with each other. Even if no conflicts are detected, the results of textual merging must be carefully inspected.

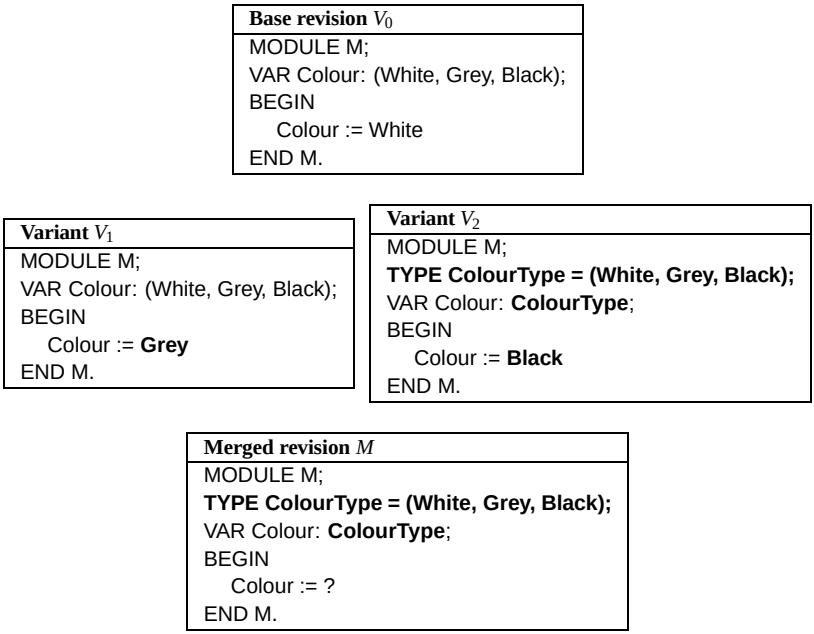

Figure 5.1: Syntax-based merging (from [Wes91])

5.6.2 Syntax-Based Merging

Automatic merging becomes more effective if internal consistency is ensured, as discussed in 3.6 on page 32. This requires knowledge about syntactical invariants that must hold after merging operations.

In [Wes91], Westfechtel describes a generic merging algorithm working on abstract syntax trees, realized in the IPSEN system. Each node class (identifier, structure, or list) is treated by a different *merge rule.* As an example, consider figure 5.1. In variant V_1, the assignment to Colour was changed from White to Grey. In variant V_2, a new type ColourType was introduced, the type of the Colour variable was adapted, and the Colour assignment was changed to Black.

The merge rule for lists states that insertions in one variant be applied in the merged version M as well. Hence, M contains the new type ColourType introduced in V_2. Name changes applied in one variant only are also reflected in M; hence the type change for the Colour variable in V_2 is propagated to M. Conflicts may still occur if a substructure is changed in both variants. Hence, the third

change in V_2, the Colour assignment value conflicts with the change in V_1 and must be resolved manually. Using textual merging, all three changes would have been in conflict because they are too close together.

Westfechtel's syntax-based merging also ensures a certain amount of internal consistency by preserving the context-free correctness and detecting context-free conflicts. Besides the context-free syntax, it also takes the binding of identifiers to their declarations into account, detecting anomalies and conflicts with respect to binding changes. However, it relies on determining the differences between abstract syntax trees, which is expensive, or on logs of tree manipulations generated by the editor.

Westfechtel's work has been extended by Schroeder in [Sch95], ensuring the correctness of the statical semantics even for incomplete subtrees, using PSG context relations [Sne91, SGS91]. Recent work in syntax-based merging includes collaborative work in structure editors, as in the MJØLNER project [MAM93, MA96], as well as the integration of incremental analysis with version management [WG95]. Syntax-based merging programs that do not rely on an external abstract syntax tree have also been presented [Buf95].

5.6.3 Semantics-Based Merging

While syntax-based merging guarantees the syntactic correctness of the merge result M, one still has no guarantee about how the execution *behavior* of M relates to the execution behavior of the merged variants V_1 and V_2. A first attempt, based on denotational semantics, is found in [Ber94], but the first approach that performed true *semantics-based* merging was presented by Horwitz, Prins, and Reps in [HPR89]. Their algorithm relies on the assumption that *behavior differences,* rather than textual or structural differences, are significant and must be preserved in M.

The algorithm works on a *program dependency graph* (PDG) representation for the programs to be merged. Each node stands for a program statement; edges indicate control and data dependence. A *program slice* is the subgraph of a PDG that can reach a given component. To determine interference of changes, the algorithm determines the *program slices* in V_1 and V_2 that are changed from the base V_0 and the slices that are unchanged from V_0. The changed and unchanged slices are then merged, and if there is no interference, a merged program M is produced from the merged slices. The algorithm ensures that M captures the changed behavior of both V_1 and V_2 as well as the behavior that was unchanged from the base V_0.

While the original algorithm [HPR89] had severe restrictions on the class of

programs it could be applied upon, it was later refined by Binkley, Horwitz, and Reps in [BHR95] and now constitutes a mature algorithm for multi-procedure merging.

5.7 Multi-Site Development

SCM is not only a problem of several people working on multiple versions. Often, these people also work at *multiple sites.* This imposes another technical challenge on SCM systems, as local version access must not be slowed down by low connectivity between the sites.

Distributed SCM is a relatively new feature in SCM systems. We can distinguish four ways to realize distribution:

Use a central repository server. Both RCS and CVS have been extended for distribution. The resulting DRCS [OG90] and DCVS [HK92] tools rely on a client/server relationship between local RCS or CVS clients and a central repository server. For instance, if a local user checks out a RCS version, the local RCS client fetches the version from the remote central RCS repository server. The drawbacks of DRCS and DCVS are that all operations depend on the reachability of one single server and that traffic is huge since entire versions (or configurations, as in DCVS) are transferred.

Propagate changes across sites. Communication overhead between sites can be reduced if sites share a common baseline and transmit changes instead of versions, as in the Change-Oriented Model. This approach has been undertaken in the MISTRAL tool [Gad95], realizing distributed SCM in the ADELE system. However, all difficulties of change propagation apply, as discussed in section 2.7.

Assign each site an individual workspace. Another possibility to manage distributed SCM is to assign each site an individual workspace or temporary variant. This is the base of the MULTISITE tool [AFK+95], which enhances the CLEARCASE system with distributed CM. To maintain consistency, each site has branches in its repository representing the other sites; these branches are updated periodically. Each site can only modify its local branch, but merge in changes made at other sites. This simple and realistic solution fits practical users needs, as the authors claim, but relies on frequent merging.

Use a distributed repository. The most recent approach to distributed CM is the usage of a *distributed repository* that allows to access versions transpar-

ently from arbitrary sites. On top of the *Network for unified configuration management* (NUCM) prototype [vdHHW96], a variety of CM models can be realized through a combination of three generic models (storage, access, and distribution). The initial implementation of NUCM realizes a distributed, decentral repository using peer-to-peer relationships between local CM repositories.

5.8 Process Functionality Areas

So far, we have discussed the *team-centered* aspects of SCM. In contrast to these more *technical* issues, the *process-centered* functionality areas cover *management issues.* As this is beyond the scope of this work, we only give a brief introduction on each of these functionality areas, following Dart's survey [Dar91].

5.8.1 Auditing Functionality

An important feature in SCM systems is an *audit trail* or *change history* where the SCM system logs all changes made to the developed product. Such an audit trail usually includes a *change comment,* details on the reason and effects of the change. Every SCM system that supports revisions maintains such audit trails and provides simple tools to print, filter or analyze the trail.

5.8.2 Accounting Functionality

The *accounting* functionality area, as found in SCM systems, includes mechanisms to record statistics about the product and the process. The questions that accounting must answer include the current status of a component, whether a change request (CR) has been approved by the configuration control board, which component version implements a specific CR or how many faults per month are detected and corrected.

5.8.3 Controlling Functionality

Controlling functionality assigns work to individual developers. *Access control* means granting or revoking version access. *Change control* provides procedures by which changes are requested, authorized, scheduled, and tracked. Change control includes on-line support for *change requests,* a developer's request to change a component, and *problem reports,* stressing the circumstances and consequences of a fault, as well as procedures to propagate changes across different versions of a

product (e.g. from an experimental version to the released version). Finally, controlling functionality also must track faults and report how, when, and by whom they are dealt with.

5.8.4 Process Functionality

The functionality areas discussed so far can be subsumed as *process functionality.* Process functionality is the significant area of all non-technical SCM functionality. In short, SCM systems should support the life cycle model and policies of the user's organization; identify tasks to be done, how and when they are completed; as well as basic facilities to direct information about relevant events to the appropriate people and facilities for documenting the product knowledge.

5.9 Conclusion

The central SCM concept to realize cooperative work is the notion of a *workspace,* preventing developers from interfering with one another's work. A workspace usually comes as a file system and thus integrates the SCM system into the software development environment. Various concepts for the realization of workspaces exist, the most advanced being a virtual file system with both explicit and implicit version access.

To coordinate changes, SCM systems either provide conservative cooperation strategies that rely on version locking, or optimistic cooperation strategies that rely on a later conflict resolution between parallel changes. Conflict resolution is realized through merging of changes, where textual merging is the most versatile and semantics-based merging the most secure approach.

Recent SCM systems also support development at geographically distributed sites with low connectivity. The pragmatic approach is to assign each site a separate workspace; future repositories may be realized in a distributed manner.

Besides the technical, team-centered functionality, process functionality areas cover the management part of SCM, which is not discussed in this work.

While process management and control are necessary
for a repeatable, optimized development process,
a solid configuration management foundation for that process is essential.

— DAVID W. EATON, Configuration Management Frequently Asked Questions

In any case, it must be borne in mind that,
tools can be encapsulated whilst users can not.

— JACKY ESTUBLIER and RUBBY CASALLAS,
The ADELE Configuration Manager

Chapter 6

Future SCM Requirements

There can be no doubt that today's SCM systems largely satisfy Dart's requirements on CM functionality [vdHHW95]. *For each functionality, we have identified a large number of SCM concepts as realized in one or more SCM systems. Some commercial SCM systems, such as CLEARCASE, provide satisfactory solutions for each required CM functionality.*

Since Dart's survey, new requirements and problems have emerged. We identify five major problems in current SCM systems, which also constitute requirement areas for future SCM systems.

6.1 Improved Support for Variant Sets

SCM still has poor support for manipulating sets of configurations, or abstract configurations. As a simple example, consider the editing of multiple versions. The number one technique for variation in the small, the C preprocessor (CPP) fails when variance becomes too large. As Gentleman *et al.* state in [GMSW89],

> Code containing conditional compilation directives becomes quite unreadable when variants associated with different factors interact.

In fact, large variance leads to a lose-lose situation. Either commonality between variants is exploited, then the CPP directives become too complex, or commonality is not exploited, then code duplication follows:

> Interleaved directives are incomprehensible, and the code expansion of conditional compilation directives can be intolerable.

The alternate technique, change propagation from a single variant X to the remaining variants Y as discussed in section 2.7, is still considered inferior than "classical" approaches such as preprocessing. In [Whi91, p. 44], Whitgift states:

> This approach is better than revising both X and Y manually, but it only works well when X and Y are very similar. Even then the techniques described in the next two subsections [CPP and multi-variant editors] are a more reliable way of managing similar permanent variants.

The only consequence can be to keep the number of permanent variants as small as possible. Not only can they seldom be handled by SCM systems. More even, common software engineering principles like abstraction, parameterization, generalization, and localization are far better ways to keep software variable than to introduce variants for every new environment. But these techniques can only apply to permanent, *planned* variance, not to temporary variance as it may result anytime during parallel development. Hence, the need to manipulate several variants at once is still present, and insufficiently covered by todays SCM systems [Mah94].

6.2 Consistency of Abstract Configurations

Another problem of SCM systems regarding abstract configurations is the lack of determining their *consistency*. As Schmerl and Marlin point out in [SM95a], this is especially important in the domain of dynamically composed systems (DCS):

> DCS are composed incrementally, and therefore some of the components may not yet be bound (meaning that it is a partial configuration). It is still desirable to analyse this partially bound configuration so that we can answer questions about what comprises the system, and whether or not the partially bound configuration is inconsistent.

Unfortunately, today's SCM systems rely on completely bound configurations to determine consistency. Even where ambiguity is allowed, as in ADELE configuration rules, heuristics to find the single "best-fitting" variant are applied to make the configuration bound.

6.3 Beyond Version Graphs

Lack of support for abstract configurations may be founded in inadequate versioning models that do not tolerate ambiguity. Among the few SCM concepts that

in principle tolerate ambiguity is the Change-Oriented Model, as it allows to apply changes to several versions at once. The advantages of the change-oriented model are the disadvantage of the version-oriented models and vice versa:

Change-oriented models: the drawback of flexibility. The strength of change-oriented models is that arbitrary change combinations are possible—that is, all change combinations that do not result in a conflict. This strength is also its major weakness, as users cannot ensure that the change application results in a consistent configuration.

Version-oriented models: few change combinations. Version-oriented models focus on the creation of versions, instead of changes. Hence, the number of actually existing versions is much smaller. Each change resulting in the creation of a new revision implies all previous changes leading up to that revision, thus ensuring change consistency. But this rigidity also has its drawbacks: creation of versions including arbitrary changes is always explicit, as is the application of changes to multiple versions at once.

Unfortunately, both models cannot be used to simulate each other. In the change-oriented models, recent approaches like HICOV [Mun96] have begun to introduce consistency constraints. But it is still unclear how a "classical" version graph would be realized through these constraints. On the other side, simulating the Change-Oriented Model through version-oriented models reveals the weakness of the version graph paradigm, since the arbitrary combination of changes results in a much larger number of potential versions than could possibly be maintained through revision graphs. Moreover, it is still an open question how revisions and changes are to be integrated with logical and cooperative versioning [EC95].

6.4 Unified Versioning Models

The divergence of change-oriented and version-oriented models is the largest difference between SCM versioning models, but by far not the only one. SCM in general suffers from a multitude of incompatible versioning models, as Conradi and Tryggeseth complain in [Est95, p. 80]:

> Is the versioning model linked to the data model, the product model (schema), the transaction model (uni-version subdatabases), or is it independent? At what granularity are "deltas" expressed, computed and merged—on the base of whole files, text lines, or syntactical entities? And how is versioning combined with e.g. inheritance and

> parameterization? Does basic versioning only apply to atomic and textual objects, and not to composites or to the entire database?
>
> How to version relationships, and thus configurations? How to express intentional version selection, and how to express constraints, defaults and preferences for such selections? Is the selection based on symbolic attribute values, that together constitute a version space? Can the constraints and attribute domains evolve over time? Given a system model with objects and relationships: is the product selection (AND-closure) done before the version selection within each group (OR-choices), or vice versa, or intertwined?

It is also symptomatic that hardly no visualization techniques beyond version graphs exist. To summarize, citing Gulla from [Gul93]:

> The lack of proper conceptual models and visualization techniques is a serious draw-back that limits the use and usefulness of current tools.

6.5 Flexible Process Support

The multitude of versioning models may be the effect of the multitude of SCM processes and models as they are realized in SCM systems. In his survey on configuration management models in commercial environments [Fei91a], Peter H. Feiler closes with:

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

Things have not much changed since Feiler's study, except that the problem is generally accepted. In the fifth international workshop on software configuration management [Est95, p. 136], Jacky Estublier states:

> There is a large consensus, including SCM designers and vendors, that SCM must include, in one way or another, some process support. This is a major change in relationship to previous workshops, where most industrials considered this topic as academic.

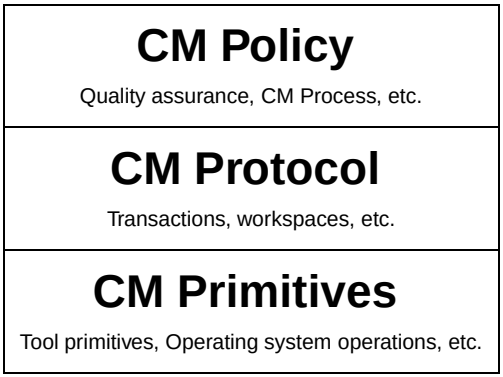

Figure 6.1: Three levels of CM services (from [BDFW91])

Estublier also points out that almost all today's SCM systems ignore other process tools, and that only a few, including EPOS and ADELE, provide a layer on top of which process support tools can be built. He concludes with:

> Most think the major challenge for future SCM tools will be the process dimension. In the future, it is expected an SCM tool will be selected based on its ability to support processes. The current state of practice is pretty far away from ideals.

6.6 Improved SCM System Architectures

Good process support means a flexible process support. This flexibility must be obtained through the architecture of SCM systems.

In their report on the state and future of automated configuration management, Brown *et al.* suggest a *federated architecture* for SCM systems, as shown in figure 6.1. Each service domain represents a virtual machine layer of services [BDFW91]:

CM Primitives layer. The CM primitives layer provides a set of primitive operations that would be supported in a particular CM tool, or provided as part of an environment framework. For example, basic versioning capabilities, data object locking, and access control are typical of the services at this level.

CM Protocol layer. The CM protocol layer supports one or more of the CM concepts and models. At this level the operations are independent of underlying implementation techniques. For example, operations of check in/out of data items from workspaces, transaction management, and coordination of change sets would be provided.

CM Policy layer. The CM policy layer makes use of the CM protocol operations to encode some procedures specific to an organization. For example, these could be company standards for handling change requests, quality assurance procedures, and so on.

As Brown *et al.* state,

> The advantage of using three layers of service domains in providing CM support is that many of the issues that are often confused can be drawn out in isolation, and the relationships between different elements more clearly expressed.

In [vdHHW95], van der Hoek, Heimbigner, and Wolf recognize that most of today's SCM systems follow this architecture. But they also state that there is an increasing lack of flexibility, the higher the level considered:

> CM systems allow some restricted flexibility at the low level (e.g., one can choose to use RCS, a file system, or a DBMS), and even less flexibility at the middle level (e.g. the naming and locking mechanisms are usually fixed). At the high level of process, second-generation CM systems either provide no explicit support for expressing policies or they provide particular processes for a specific task, such as change control. (ADELE is a notable exception to this.)

Van der Hoek *et al.* conclude that the lack of flexibility at the lower architectural levels is the cause for bad process support, and that alternative architectural views might lead to novel CM solutions.

6.7 A Unified SCM Model

For Brown *et al.*, the key to flexibility in SCM lies in the combination of a federated architecture and a unified CM model. As they summarize in [BDFW91],

> We believe that progress will have to be made in three areas in order that future CM support as outlined in our federated vision can be realized in practice.

First, the spectrum of concepts and the four conceptual models have to be integrated into a unified CM model whose semantics are well-defined. This will result in a common set of interfaces to CM services.

Second, the service-based approach of the federated environment architecture can provide a migration path from the current state of CM services (being provided in a fragmented manner by CM tools, environment frameworks, and CASE tools) toward the notion of a common repository and shared environment framework services, but still accommodating heterogeneity in software development environments. CM will be a key component of such a federated environment architecture by being a service domain in the form of a set of protocols, which are derived from the unified CM services model.

Third, the set of CM services reflected in the unified model will provide a virtual machine layer on top of which process adaptation can be performed. Process adaptation results in encoding elements of the software process in a software development environment, in this case those aspects of the software process that relate to CM.

These are the issues we have addressed in this work.

Although there is a bunch of appropriate techniques
and powerful tools, none of them is sufficient
for solving all involved problems.

— AXEL MAHLER, Variants

Part Two

Feature Logic

Chapter 7

A SCM Foundation

In chapter 6, we have found that "the major challenge for future SCM tools will be the process dimension" and that a flexible CM policy can only be attained through flexibility at the lowest levels, notably a unified configuration management model. *This unified SCM model, as postulated by Brown* et al. [BDFW91] *must integrate all four conceptual SCM models as discussed by Feiler* [Fei91a] *and have a well-defined semantics.*

In this chapter, we try to determine a formal foundation *for such a unified SCM model. We discuss the properties of such a unified SCM model, using the requirements of chapter 6 and their implications, and identify SCM foundations fitting these properties.*

7.1 First Foundation: Sets

As stated in section 6.1, most of todays SCM systems lack support for manipulating variant sets. But also configuration sets, that is, *abstract configurations,* lack proper SCM support. Generally, version and configuration sets play an important role in three areas:

Inheritance. Abstract configurations can be used as templates for further refinement. See section 3.3.6 for details.

Ambiguity support. Abstract configurations and version sets allow manipulating several versions and configurations at once. See the CPP concepts in section 2.6.1 and the *ambition* concept in section 2.7 for a discussion.

Consistency. In dynamically composed systems, inconsistency in configurations must be detected even if the configuration is incomplete. See section 6.2 for an example.

We conclude that a unified SCM model should be *set-oriented* rather than *object-oriented*[1], as manipulating sets generalizes manipulating single objects. For instance, editing a set of versions or checking a set of configurations for consistency subsumes editing a single version or checking a single configuration. Consequently, the unified SCM model should support version and configuration sets as first-class objects.

7.2 Second Foundation: Attribution

Attributes and relationships play an important role in SCM versioning models.

Identification. All of the selection schemes discussed in section 3.3 rely on that either versions or changes be tagged with attributes. Attribution is one of the few techniques common to the whole SCM area. We recognize attribution as a key element for identification and selections in a unified SCM model.

Propagation. As any SCM identification scheme must include composed and derived objects as well, there should be a well-defined relationship between the attributes of a simple component version and the attributes of a set of objects. This includes the propagation of attributes from versions to components, from components to configurations, from source components to derived components, and from changes to change sets.

Relationships. To handle propagation, the unified SCM model must allow describing the relationships between components, such as *is-instance-of* relationships to model derivation or *is-a-part-of* relationships to model composition.

The most advanced SCM system in this field is the CAPITL system, discussed in section 4.5; its attribution and propagation schemes should be considered in a unified SCM model. It also shows how attribution can be generalized to include relationships, provided the underlying attribution model is rich enough.

We conclude that the unified SCM model should be attribute-oriented: attributes should be used for identification and selections. It should also describe how attributes propagate between components, using the component relationships.

[1] Pun intended.

7.3 Third Foundation: Unification

In SCM, attribute expressions are used for both identification and selection. This duality is illustrated by the CPP and JASON systems:

Strong identification, weak selection. Across all SCM systems, CPP, the C preprocessor, realizes the most general *identification scheme.* Arbitrary logical and arithmetic expressions involving attributes are used for variant identification; see section 2.6.1 for details. Version selection in CPP is done using a conjunction of attributes.

Strong selection, weak identification. The most general *selection scheme* is realized in the JASON system, which uses full first-order logic over attribute expressions, as discussed in section 3.6.1. In JASON, individual versions are identified by a conjunction of attributes.

It is remarkable that strength in identification comes with weakness in selection, and vice versa. Such restrictions are necessary to keep selection decidable.[2] Unless we decide to ignore variant set support such as provided by CPP, the unified SCM model should support the smallest common superset of both approaches and thus rely on *unification* techniques to match selection terms with identification terms.

7.4 Putting it all Together

We have found that the unified SCM model should be

set-oriented: Supports manipulating consistent sets of versions and configurations.

attribute-oriented: Follows SCM conventions for the identification and selection of objects and allows for predictable identification of composed and derived objects.

unification-oriented: Encompasses the largest possible common subset of SCM identification and selection schemes.

We now discuss adequate foundations to express the semantics of our unified SCM model. Basically, there are three candidates for this *SCM foundation,* each with its own pros and cons.

[2] If we combined the strength of both systems, we would be challenged by general arithmetical problems; for instance, whether a version identified by the CPP expression $n > 2$ is matched by the JASON selection term $\exists a,b,c \in \mathbb{N}(a^n + b^n = c^n)$. Such problems are undecidable in general, although some of them may be eventually proved [Wil95].

7.5 First Candidate: First-Order Logic

The first candidate for an SCM foundation is very general and widely known. Boolean *first-order logic* is the base of several SCM selection schemes, including JASON's; even CPP's arithmetic version identification may be replaced by boolean first-order terms without much loss. First-order terms may be used for both identification and selection, using *boolean unification* [Boo47, BJSS90] to match identification and selection terms.

The expressive power of first-order logic is no doubt sufficient for describing the semantics of a unified SCM model. But first-order logic is far too general; it lacks the central property of being attribute-oriented. As we have already seen how important attributes are in the SCM area, this implies that all SCM functionality like selection through attributes, attribute propagation, or inheritance of abstract configurations requires explicit formalization using first-order axioms and rules. We would have to set up another formal layer in terms of first-order logic in order to describe these attribute fundamentals.

7.6 Second Candidate: Description Logics

As an alternative to first-order logic, there are several formalisms that denote sets of objects by their attributes (called *roles*), subsumed under the term *description logics* or *terminological logics.* Their most important domains are:

Knowledge representation. In the domain of knowledge representation, *concept descriptions,* also called *frames* [BL84, Neb90, NS89], are used to represent sets of objects by attribute/value combinations.

Configuration of technical systems. To configure technical systems, *terminological configuration systems* like CLASSIC [BMPS$^+$91a, BMPS$^+$91b], K-REP [MDW91], BACK [Pel91], LOOM [Mac91], or KRIS [BH91, BFH$^+$94] are more and more preferred to domain-specific configuration systems like XCON [McD82, McD84] or customizable systems like PLAKON [CGS91]. These terminological systems rely on description logic as a semantic foundation to identify component properties as well as to express configuration constraints.

All these description logics combine attribute descriptions with full boolean set semantics, including set union (disjunction) and set complement (negation). This makes them ideal choices for SCM selection and identification schemes—and last but not least, they already have been used to describe and solve configuration problems. However, attribute propagation from components to composites must be explicitly stated for each single role.

7.7 Third Candidate: Feature Logics

A special subset of description logics are *feature logics.* Here, attributes are called *features*. In contrast to roles, features are *functional*: each feature of a component can have only one value. The features of composite objects are implicitly determined from the unified features of their components. Typical applications of feature logics are:

Language analysis. In the semantic analysis of natural language [KB82, Kay84, SUP+83], feature logics are used to represent and propagate grammatical information—for instance, how the features of a sentence are determined by the features of its verb.

Programming. In programming languages, attribute/value combinations are frequently used in *record structures.* Aït-Kaci was the first to study such structures mathematically, calling them ψ*-terms* [AK86]. The resulting ψ-term calculus is the formal foundation of the PROLOG-like programming languages LOGIN [AKN86] and LIFE [AKP91], using *feature unification*[SAK90] instead of PROLOG's syntactic unification. A variant of LOGIN, called CONGRESS, is the base of the CAPITL build planner discussed in section 4.5.

The advantage of feature logics is that they provide a natural way of attribute propagation from components to composites—a property that already has been successfully exploited in the SCM domain. The disadvantage of the feature logics listed is that only *conjunctions* of attribute/value combinations are supported; negations or disjunctions are not allowed. This restriction would severely constrain identification and selection schemes, not to speak of CPP arithmetic expressions, or quantifiers in JASON.

7.8 Conclusion

For the SCM domain, we need the best of three worlds:

Boolean operations as in first-order logic. This is a must for modelling SCM identification and selection schemes.

Attribute descriptions and set operations as in description logics. These formalisms are needed for identifying versions according to their properties.

Attribute propagation and unification as in feature logics. This is needed to describe the features of derived and composed objects.

Fortunately, there is a special feature logic that includes quantification, disjunction, and negation over attribution terms, forming a full boolean algebra while preserving the functional nature of features and describing how features propagate from components to composites. This logic, described by Smolka in 1992, and simply called *feature logic,* is presented in chapter 8.

I was to learn later in life
that we tend to meet any new situation
by reorganizing;
and a wonderful method it can be
for creating the illusion of progress
while producing confusion, inefficiency, and demoralization.

— PETRONIUS ARBITER

Chapter 8

Feature Logic

After a short excursion into the evolution of feature logic, we give an informal overview. For a deeper understanding, we present the formal syntax and semantics of feature logic, based on [Smo92].

8.1 The Evolution of Feature Logic

Feature descriptions and feature logic have two sources. The first source is oriented towards boolean formulae, providing for the declaration and specification of linguistic knowledge. The *lexical-functional grammar* [KB82], of Bresnan and Kaplan, as well as Shieber's PATR-II formalism [SUP$^+$83] and Johnson's *attribute-value logic* [Joh88] use boolean combinations of features, constants, and variables.

The second source is oriented towards set-denoting feature expressions, called *feature terms* in this work, used in programming languages and knowledge representation. This includes Kay's *functional unification grammar* [Kay84], Aït-Kaci's ψ*-term calculus* [AK86, SAK90], and the logic of Kasper [KR86] and Rounds [MR87]. These feature terms also have much in common with *concept descriptions* used in knowledge representation [BL84, Neb90, NS89].

In [Smo92], Smolka unified these two approaches and showed that the different feature descriptions can be embedded into first-order predicate logic with equality.

We have chosen Smolka's feature logic as a SCM foundation. Not only does it provide a simple and clear semantics, but it also allows us to describe SCM concepts by attribution without losing the expressiveness of boolean first-order logic.

8.2 Feature Logic in a Nutshell

We begin with an informal overview of feature logic. *Feature terms* denote sets 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:

$$T = \left[\begin{array}{l} \mathit{tense}:\mathit{present}, \\ \mathit{predicate}:[\mathit{verb}:\mathit{sing}, \mathit{agent}:x, \mathit{what}:y], \\ \mathit{subject}:[x, \mathit{num}:\mathit{singular}, \mathit{person}:\mathit{third}], \\ \mathit{object}:y \end{array}\right]$$

This term says that the language fragment is in present tense, third person singular, that the agent of the predicate is equal to the subject, and so on. In other words, T denotes the sentence template "x sings y".

The syntax of feature terms is summarized in table 8.1 on the facing page, where we denote *variables* by x, y, z; *features* by f, g, h; *constants* by a, b, c; and feature terms denoted by S, T, and U. 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 logical constraint on the object features, representing the set of objects satisfying this constraint. For instance, let $S = [f:a]$, the set of all objects whose feature f has the value a, and $T = [g:b]$, the set of all objects whose feature g has the value b. Then, $S \sqcap T = [f:a, g:b]$ may be read as the intersection of S and T as well as 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 S and T as well as the set of objects whose feature f is *a or* whose feature g is b. As feature terms form a boolean algebra, all boolean transformations like distribution, de Morgan's law etc. hold for feature terms as well.

Feature terms have two important properties which make them especially suitable in the context of SCM.

Each feature of an object may have only one value. This property is due to the functional nature of features. For instance, the term $[\mathit{os}:\mathit{dos}, \mathit{os}:\mathit{unix}]$ is equivalent to $\bot$, the empty set. This property is useful for selection and consistency checking.

Notation	Name	Interpretation
$\top$ (also $[]$)	Top	Ignorance
$\bot$ (also $\{\}$)	Bottom	Inconsistency
a	Atom	
x	Variable	
$f{:}S$	Selection	The value of f is S
$f{:}\top$	Existence	f is defined
$f{\uparrow}$	Divergence	f is undefined
$f \downarrow g$	Agreement	f and g have the same value
$f \uparrow g$	Disagreement	f and g have different values
$\sim S$	Complement	S does not hold
$S \sqcap T$ (also $[S,T]$)	Intersection	Both S and T hold
$S \sqcup T$ (also $\{S,T\}$)	Union	S or T holds
$S \rightarrow T$	Implication	If S holds, then T holds
$S \leftrightarrow T$	Equivalence	S holds if and only if T holds
$\exists x(S)$	Quantification	There is an x such that S holds

Table 8.1: Syntax and interpretation of feature terms

Feature terms always allow for further specialization. Every feature term can be refined by specifying further features, like subclasses in object-oriented models. This property allows for attribute propagation and abstract configurations.

In this chapter, we give a formal definition of features and feature terms, closely following Smolka's definitions in [Smo92] and further clarified by Fischer in [Fis93]. For each operator in table 8.1, we give its denotational semantics and show its respective properties.

8.3 Features and Feature Algebras

The definition of features as functional properties implies that we can model features as *partial functions* that, applied to abstract objects, result in a single *feature value.* For instance, the feature *os* of a component X may be $os(X) = unix$. The functional nature of features also implies that each feature of an object may have only one value.

We now define these properties of features formally, introducing *feature algebras* as interpretations of feature descriptions.

Definition 8.1 (Feature algebra, Feature) A *feature algebra* I is a pair $(\mathbf{D}^I, \cdot^I)$ consisting of a nonempty set $\mathbf{D}^I$, called the *domain* of I, and an *interpretation function* $\cdot^I$ assigning to every atom a an element $a^I \in \mathbf{D}^I$ and to every feature f a set of ordered pairs $f^I \subseteq \mathbf{D}^I \times \mathbf{D}^I$ such that the following conditions are satisfied:

1. If (d,e) and (d,e') are in f^I, then $e = e'$ *(features are functional),*
2. If $a \neq b$, then $a^I \neq b^I$ *(unique-name assumption),*
3. If f is a feature and a is an atom, then there exists no $d \in \mathbf{D}^I$ such that $(a^I, d) \in f^I$ *(atoms are primitive).* □

The first condition captures the functional nature of features; the third definition restricts the application of features to non-primitive objects.

For the denotation of variables, we introduce *I-assignments*:

Definition 8.2 (Assignment) Let I be a feature algebra. An *I-assignment* is a mapping from the set of all variables to the domain of I. □

The set of all I-assignments is denoted as $\mathrm{Ass}[I]$.

8.4 Syntax and Semantics of Feature Terms

We now introduce *feature terms,* a denotation for sets in feature algebras. For each construct, we give its syntax, followed by its *denotation* $S^I_\alpha \subseteq \mathbf{D}^I$, where I is a feature algebra, and $\alpha \in \mathrm{Ass}[I]$ an I-assignment.

8.4.1 Top and Bottom

Top denotes the entire universe of objects, bottom the empty set.

Definition 8.3 (Top) The symbol $\top$ denotes the entire domain of the feature algebra I:

$$\top^I_\alpha = \mathbf{D}^I$$

□

Definition 8.4 (Bottom) The symbol $\bot$ denotes the empty set:

$$\bot^I_\alpha = \emptyset$$

□

8.4.2 Atoms and Variables

The primitives of feature logic are atoms and variables.

Definition 8.5 (Atom) An *atom a* is a primitive object for which no features are defined. An atom denotes a singleton set containing itself:

$$a^I_\alpha = \{a^I\}$$

□

Definition 8.6 (Variable) A *variable x* is a placeholder for some feature term. Its denotation is the term it stands for:

$$x^I_\alpha = \{\alpha(x)\}$$

□

A variable is called *free* if it is not bound by any quantifier.

8.4.3 Selection

The basic operation of feature logic is *selection,* denoting the objects where a feature has a specific value.

Definition 8.7 (Selection) The term $f{:}S$ denotes the set of all objects whose feature f has a value S:

$$(f{:}S)^I_\alpha = \{d \in \mathbf{D}^I \mid \exists e \in S^I_\alpha{:}\,(d,e) \in f^I\}$$

□

For instance, the feature term *tested*:*true* denotes all objects whose feature *tested* has a value of *true*.

In feature logic, there is no distinction between objects and feature values. Hence, feature values may be feature terms again, denoting other objects. As an example, consider *existence.* As follows from definitions 8.3 and 8.7, a term $f{:}\top$ (*Existence*) denotes all objects for which the feature f is defined with an arbitrary value:

$$(f{:}\top)^I_\alpha = \{d \in \mathbf{D}^I \mid \exists e \in \mathbf{D}^I{:}\,(d,e) \in f^I\}$$

Note that the suggestive $f{:}\bot$ does *not* stand for all objects whose feature f is undefined, but for the empty set instead. As follows from definitions 8.4 and 8.7, $f{:}\bot = \bot$ holds for all I and α:

$$\begin{aligned}(f{:}\bot)^I_\alpha &= \{d \in \mathbf{D}^I \mid \exists e \in \bot^I_\alpha{:}\,(d,e) \in f^I\}\\ &= \{d \in \mathbf{D}^I \mid \exists e \in \emptyset{:}\,(d,e) \in f^I\}\\ &= \emptyset\\ &= \bot^I_\alpha\end{aligned}$$

Hence, we need an alternate construct to capture undefined features.

8.4.4 Divergence

A feature may be *undefined* on certain objects.

Definition 8.8 (Divergence) The set $f{\uparrow}$ is the set of all objects whose feature f is undefined:

$$(f{\uparrow})^I_\alpha = \{d \in \mathbf{D}^I \mid \forall e \in \mathbf{D}^I{:}\,(d,e) \notin f^I\}$$

□

8.4.5 Agreement and Disagreement

Special notations exist for sets of objects whose features have equal or unequal values.

Definition 8.9 (Agreement) The set $f{\downarrow}g$ is the set of all objects for which the feature f has the same value as the feature g:

$$(f{\downarrow}g)^I_\alpha = \{d \in \mathbf{D}^I \mid \exists e \in \mathbf{D}^I{:}\,(d,e) \in f^I \cap g^I\}$$

□

Definition 8.10 (Disagreement) The set $f{\uparrow}g$ is the set of all objects for which the feature f has another value than the feature g:

$$(f{\uparrow}g)^I_\alpha = \{d \in \mathbf{D}^I \mid \exists e,e' \in \mathbf{D}^I{:}\,(d,e) \in f^I \wedge (d,e') \in g^I \wedge e \neq e'\}$$

□

Assuming that we classify compilers by their host and target architectures, we may thus specify a cross-compiler as *host-arch*$\uparrow$*target-arch*. Note that agreement and disagreement imply that both features f and g be actually defined.

8.4.6 Complement

The set complement respective to $\top$ is denoted by the complement sign $\sim$.[1]

Definition 8.11 (Complement) The set $\sim S$ denotes all objects other than those denoted by S:

$$(\sim S)^I_\alpha = \mathbf{D}^I - S^I_\alpha$$

□

When speaking of features rather than objects, the term $\sim S$ may also be read as *negation.* Hence, the term $T = \textit{operating-system}{:}\sim\textit{windows}$ denotes all objects whose feature *operating-system* is *not windows*. This must not be confounded with the term $T' = \sim\textit{operating-system}{:}\textit{windows}$, which consists of the objects of T as well as of the objects whose feature *operating-system* is undefined.

The definition implies that the well-known equivalences[2] for set complements apply:

$$\sim\top = \bot$$
$$\sim\sim S = S$$

8.4.7 Intersection

Intersections are used to denote objects by several features.

Definition 8.12 (Intersection) We write $S \sqcap T$ for the intersection of S and T:

$$(S \sqcap T)^I_\alpha = S^I_\alpha \cap T^I_\alpha$$

□

When speaking of features instead of objects, terms like $S \sqcap T$ may also be read as *conjunction.* As an example, the term $T = \textit{author}{:}\textit{zeller} \sqcap \textit{status}{:}\textit{experimental}$ denotes all objects whose author is Zeller *and* whose status is experimental.

Feature conjunctions occur very frequently. We thus introduce the more convenient *matrix notation* for feature terms, which traces back to the very roots of

[1] Smolka [Smo92] uses the $\neg$ symbol.

[2] See definition 8.25 on page 85 for a formal definition of equivalence.

feature logic [Kay79]. In matrix notation, conjunctions are surrounded by square brackets, such that the following equivalences hold:

$$\begin{aligned}[] &\equiv \top \\ [S] &\equiv S \\ [S_1, S_2, \ldots, S_n] &\equiv S_1 \sqcap S_2 \sqcap \cdots \sqcap S_n\end{aligned}$$

Hence, the feature term $T = \mathit{age}{:}30 \sqcap \mathit{mood}{:}\mathit{happy}$ may also be written as $T = [\mathit{age}{:}30, \mathit{mood}{:}\mathit{happy}]$.

Definition 8.12 implies associativity, commutativity, and idempotency of intersection:

$$\begin{aligned}(S \sqcap T) \sqcap U &= S \sqcap (T \sqcap U) \\ S \sqcap T &= T \sqcap S \\ S \sqcap S &= S\end{aligned}$$

The neutral element respective to intersection is $\top$; the zero element respective to intersection is $\bot$.

$$\begin{aligned}S \sqcap \top &= S \\ S \sqcap \bot &= \bot\end{aligned}$$

Intersection with a complement is the empty set:

$$S \sqcap {\sim}S = \bot$$

Intersections and complements as feature values can be *lifted* to the top level:

$$\begin{aligned}[f{:}(S \sqcap T)] &= [f{:}S] \sqcap [f{:}T] \\ [f{:}{\sim}S] &= [f{:}\top] \sqcap {\sim}[f{:}S]\end{aligned}$$

8.4.8 Union

Unions are used to denote alternatives.

Definition 8.13 (Union) The term $S \sqcup T$ denotes the union of S and T:

$$(S \sqcup T)^I_\alpha = S^I_\alpha \cup T^I_\alpha$$

□

Again, when speaking of features instead of objects, the union operator has the meaning of a boolean *disjunction* operator. As an example, consider the term $T = \mathit{operating\text{-}system}{:}\,\mathit{dos} \sqcup \mathit{operating\text{-}system}{:}\,\mathit{unix}$, denoting all objects whose operating system is DOS *or* UNIX.

In matrix notation, conjunctions are surrounded by curly braces, such that the following equivalences hold:

$$\{\} \equiv \bot$$
$$\{S\} \equiv S$$
$$\{S_1, S_2, \ldots, S_n\} \equiv S_1 \sqcup S_2 \sqcup \cdots \sqcup S_n$$

Hence, the term $T = \mathit{status}{:}\,\mathit{proposed} \sqcup \mathit{status}{:}\,\mathit{experimental}$ may also be written as $T = \{\mathit{status}{:}\,\mathit{proposed}, \mathit{status}{:}\,\mathit{experimental}\}$ or, according to definition 8.13, as $T = \mathit{status}{:}\,\{\mathit{proposed}, \mathit{experimental}\}$.

Again, we can deduce associativity, commutativity, and idempotency:

$$(S \sqcup T) \sqcup U = S \sqcup (T \sqcup U)$$
$$S \sqcup T = T \sqcup S$$
$$S \sqcup S = S$$

Respective to union, the neutral element is $\bot$; the zero element is $\top$.

$$S \sqcup \bot = S$$
$$S \sqcup \top = \top$$

Union with a complement is the universe.

$$S \sqcup {\sim}S = \top$$

Regarding unions and intersections, distribution and absorption rules apply:

$$(S \sqcup T) \sqcap U = (S \sqcap U) \sqcup (T \sqcap U)$$
$$(S \sqcap T) \sqcup U = (S \sqcup U) \sqcap (T \sqcup U)$$
$$S \sqcup (S \sqcap T) = S$$
$$S \sqcap (S \sqcup T) = S$$

De Morgan's laws apply as well.

$${\sim}(S \sqcap T) = {\sim}S \sqcup {\sim}T$$
$${\sim}(S \sqcup T) = {\sim}S \sqcap {\sim}T$$

Unions as feature values can also be lifted to the top level:

$$[f{:}\,(S \sqcup T)] = [f{:}S] \sqcup [f{:}T]$$

8.4.9 Implication

The implication $S \to T$ is a short-hand notation for $\sim S \sqcup T$:

Definition 8.14 (Implication) The term $S \to T$ denotes all objects which are in T or not in S:

$$(S \to T)^I_\alpha = \left(\mathbf{D}^I - S^I_\alpha\right) \cup T^I_\alpha$$

□

All rules for implications are deduced from the equivalence

$$S \to T = \sim S \sqcup T$$

The following absorption rules have practical relevance:

$$(S \to T) \sqcap S = T$$
$$(S \to T) \sqcap \sim T = \sim S$$

8.4.10 Equivalence

The equivalence $S \leftrightarrow T$ is a short-hand notation for $(S \to T) \sqcap (T \to S)$:

Definition 8.15 (Feature equivalence) The term $S \to T$ denotes the objects that are either in S and T or in neither S nor T:

$$(S \leftrightarrow T)^I_\alpha = \left(\left(\mathbf{D}^I - S^I_\alpha\right) \cup T^I_\alpha\right) \cap \left(\left(\mathbf{D}^I - T^I_\alpha\right) \cup S^I_\alpha\right)$$

□

All rules for equivalences are deduced from

$$S \leftrightarrow T = (S \to T) \sqcap (T \to S)$$

which can also be expressed as

$$S \leftrightarrow T = (S \sqcap T) \sqcup (\sim S \sqcap \sim T) \quad .$$

8.4.11 Quantification

The final element in the syntax of feature terms is existential quantification.

Definition 8.16 (Existential quantification) The term $\exists x(S)$ defines the union of all sets S where x is instantiated by some object:

$$\big(\exists x(S)\big)^I_\alpha = \bigcup_{d \in \mathbf{D}^I} S^I_{\alpha[x \leftarrow d]}$$

□

Here, the term $\alpha[x \leftarrow d]$ stands for the *instantiation* of x with d: If α is an I-assignment and $d \in \mathbf{D}^I$, then $\alpha[x \leftarrow d]$ denotes the I-assignment obtained from α by mapping x to d rather than to $\alpha(x)$.

Again, $\exists x(S)$ may be interpreted as denoting features rather than objects: a term $\exists x(S)$ then denotes all objects where there exists an x such that S is satisfied.

Our presentation of the syntax and semantics of feature terms is now complete. As a summary, consider table 8.2 on the following page: Given a feature algebra I and a I-assignment α, the *denotation* of a feature term S in I under α is a subset of $\mathbf{D}^I$ defined inductively as shown.

8.4.12 An Interpretation Example

As a simple example for the interpretation of feature terms, let

$$\mathbf{D}^I = \{\text{BICYCLE}, \text{CAR}, \text{TRUCK}, \text{ONE}, \text{TWO}, \text{FOUR}, \text{SIX}\}$$

be some domain, let WHEELS and PASSENGERS be features, and let $\cdot^I$ be an interpretation function such that

$$\begin{aligned} \mathit{wheels}^I &= \text{WHEELS} \\ &= \big\{(\text{BICYCLE}, \text{TWO}), (\text{CAR}, \text{FOUR}), (\text{TRUCK}, \text{SIX})\big\} \end{aligned}$$

and

$$\begin{aligned} \mathit{passengers}^I &= \text{PASSENGERS} \\ &= \big\{(\text{BICYCLE}, \text{ONE}), (\text{CAR}, \text{FOUR}), (\text{TRUCK}, \text{TWO})\big\} \quad . \end{aligned}$$

Furthermore, let us interpret the atoms $1, 2, 4, 6$ as $1^I = \text{ONE}$, $2^I = \text{TWO}$, $4^I = \text{FOUR}$, and $6^I = \text{SIX}$.

$$
\begin{aligned}
\top_\alpha^I &= \mathbf{D}^I \\
\bot_\alpha^I &= \emptyset \\
a_\alpha^I &= \{a^I\} \\
x_\alpha^I &= \{\alpha(x)\} \\
(f{:}S)_\alpha^I &= \{d \in \mathbf{D}^I \mid \exists e \in S_\alpha^I{:}(d,e) \in f^I\} \\
(f\uparrow)_\alpha^I &= \{d \in \mathbf{D}^I \mid \forall e \in \mathbf{D}^I{:}(d,e) \notin f^I\} \\
(f\downarrow g)_\alpha^I &= \{d \in \mathbf{D}^I \mid \exists e \in \mathbf{D}^I{:}(d,e) \in f^I \cap g^I\} \\
(f\uparrow g)_\alpha^I &= \{d \in \mathbf{D}^I \mid \exists e,e' \in \mathbf{D}^I{:}(d,e) \in f^I \wedge (d,e') \in g^I \wedge e \neq e'\} \\
(\sim S)_\alpha^I &= \mathbf{D}^I - S_\alpha^I \\
(S \sqcap T)_\alpha^I &= S_\alpha^I \cap T_\alpha^I \\
(S \sqcup T)_\alpha^I &= S_\alpha^I \cup T_\alpha^I \\
(S \to T)_\alpha^I &= (\mathbf{D}^I - S_\alpha^I) \cup T_\alpha^I \\
(S \leftrightarrow T)_\alpha^I &= ((\mathbf{D}^I - S_\alpha^I) \cup T_\alpha^I) \cap ((\mathbf{D}^I - T_\alpha^I) \cup S_\alpha^I) \\
(\exists x(S))_\alpha^I &= \bigcup_{d \in \mathbf{D}^I} S_{\alpha[x \leftarrow d]}^I
\end{aligned}
$$

Table 8.2: Formal denotation of feature terms

The denotation of the feature term $S = [\mathit{passengers}{:}\,2]$ under the feature algebra $I = (\mathbf{D}^I, \cdot^I)$ and some I-assignment α is then determined as

$$
\begin{aligned}
S_\alpha^I &= [\mathit{passengers}{:}\,2]_\alpha^I \\
&= \{d \in \mathbf{D}^I \mid \exists e \in 2_\alpha^I{:}(d,e) \in \mathit{passengers}^I\} \\
&= \{d \in \mathbf{D}^I \mid \exists e \in \{\text{TWO}\}{:}(d,e) \in \text{PASSENGERS}\} \\
&= \{d \in \mathbf{D}^I \mid (d,\text{TWO}) \in \text{PASSENGERS}\} \\
&= \{\text{TRUCK}\} \ .
\end{aligned}
$$

The term $T = \exists x[\mathit{passengers}{:}\,x, \mathit{wheels}{:}\,x]$ is interpreted as

$$
\begin{aligned}
T_\alpha^I &= \exists x[\mathit{passengers}{:}\,x, \mathit{wheels}{:}\,x]_\alpha^I \\
&= \bigcup_{d \in \mathbf{D}^I} [\mathit{passengers}{:}\,x, \mathit{wheels}{:}\,x]_{\alpha[x \leftarrow d]}^I
\end{aligned}
$$

$$= \cdots \cup [\mathit{passengers}{:}\,x, \mathit{wheels}{:}\,x]^I_{\alpha[x\leftarrow \text{FOUR}]} \cup \cdots \quad .$$

We focus upon the assignment of x with FOUR, giving

$$T^I_\alpha = \cdots \cup \left([\mathit{passengers}{:}\,x]^I_{\alpha[x\leftarrow \text{FOUR}]} \cap [\mathit{wheels}{:}\,x]^I_{\alpha[x\leftarrow \text{FOUR}]} \right) \cup \cdots$$

which reduces to

$$\begin{aligned} T^I_\alpha &= \cdots \cup \left(\left\{ d \in \mathbf{D}^I \mid \exists e \in x^I_{\alpha[x\leftarrow \text{FOUR}]} : (d,e) \in \mathit{passengers}^I \right\} \cap \cdots \right) \cup \cdots \\ &= \cdots \cup \left(\left\{ d \in \mathbf{D}^I \mid \exists e \in \{\text{FOUR}\} : (d,e) \in \mathit{passengers}^I \right\} \cap \cdots \right) \cup \cdots \\ &= \cdots \cup \left(\left\{ d \in \mathbf{D}^I \mid (d, \text{FOUR}) \in \text{PASSENGERS} \right\} \right. \\ &\qquad \left. \cap \left\{ d \in \mathbf{D}^I \mid (d, \text{FOUR}) \in \text{WHEELS} \right\} \right) \cup \cdots \\ &= \cdots \cup \left(\left\{ d \in \mathbf{D}^I \mid (d, \text{FOUR}) \in \text{PASSENGERS} \wedge (d, \text{FOUR}) \in \text{WHEELS} \right\} \right) \cup \cdots \end{aligned}$$

This leaves only the CAR element as possible interpretation:

$$T^I_\alpha = \cdots \cup \left(\{\text{CAR}\} \cap \{\text{CAR}\} \right) \cup \cdots$$

All other assignments for x result in the empty set, giving

$$\begin{aligned} T^I_\alpha &= \emptyset \cup \left(\{\text{CAR}\} \cap \{\text{CAR}\} \right) \cup \emptyset \\ &= \{\text{CAR}\} \quad . \end{aligned}$$

Existential quantification $\exists x(S)$ in feature terms, as in the example above, imposes some decidability and complexity problems. Existential quantification is thus often implicitly expressed through equivalent agreement and disagreement terms. The term $T = \exists x[\mathit{passengers}{:}\,x, \mathit{wheels}{:}\,x]$ can be expressed through the equivalent $T = \mathit{passengers}\downarrow \mathit{wheels}$, for instance. The algorithms discussed in this work all require that their feature terms be free of existential quantifiers.

8.5 Properties of Feature Terms

8.5.1 Redundant Forms

Smolka observes that most of the introduced feature term forms are redundant and may be reduced to six primitive forms.

Definition 8.17 (Primitive feature term) A feature term is called *primitive* if it contains only the forms a, x, $f{:}S$, $S \sqcap T$, $\sim S$, and $\exists x(S)$. □

Proposition 8.18 Every feature term can be rewritten in linear time to an equivalent primitive feature term by using the following equivalences:

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

PROOF. Follows from definitions. □

8.5.2 Special Feature Terms

We now introduce the notions of *closed, quantifier-free, basic,* and *simple* feature terms.

Definition 8.19 (Closed feature term) A feature term is called *closed* if it has no variables. □

Definition 8.20 (Quantifier-free feature term) A feature term is *quantifier-free* if it contains no quantifications $\exists x(S)$. □

Definition 8.21 (Basic feature term) A feature term is *basic* if it is quantifier-free and contains only complements of the from $\sim a$ or $\sim x$. □

Every quantifier-free feature term can be transformed into a basic feature term, where negations occur only at the atom and variable level.

Proposition 8.22 Every quantifier-free feature term can be rewritten in linear time to an equivalent basic feature term by using the following equivalences:

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

PROOF. Follows from definitions. □

Definition 8.23 (Simple feature term) A feature term is *simple* if it is basic and contains no unions. □

Definition 8.24 (Disjunctive normal form) 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. □

8.5.3 Equivalence

The meaning of an expression $S = T$ is the intuitive one.

Definition 8.25 (Term equivalence) Two feature terms S and T are called *equivalent* (written $S =^I T$ or $S = T$ where unambiguous) if $S^I_\alpha = T^I_\alpha$ for every feature algebra I and an I-assignment α.[3] □

Using this equivalence notion, we find that feature terms constitute a boolean algebra.

Proposition 8.26 Let f be the set of feature terms, as defined in section 8.4. Then $(f, \sqcup, \sqcap, \sim, \bot, \top)$ is a boolean algebra under the equivalence $=^I$.
PROOF. All properties required for boolean algebras (commutativity, associativity, idempotency, absorption, distribution, etc.) apply. □

8.5.4 Subsumption

In our SCM context, *subsumption* is frequently needed for eliminating redundant feature terms and to express implications.

Definition 8.27 (Subsumption) A feature term S is said to be *subsumed* by a feature term T (written $S \sqsubseteq T$ or $T \sqsupseteq S$) if $S^I_\alpha \subseteq T^I_\alpha$ holds for every feature algebra I and every I-assignment α.[4] □

The following propositions hold for all feature terms S, T, U:

$$S \sqsubseteq S \tag{8.1}$$

$$S \sqsubseteq T \sqcap T \sqsubseteq S \Rightarrow S = T \tag{8.2}$$

$$S \sqsubseteq T \sqcap T \sqsubseteq U \Rightarrow S \sqsubseteq U \tag{8.3}$$

[3] Smolka [Smo92] writes $S \sim T$ instead of $S = T$.
[4] Smolka [Smo92] says that S is *included* by T, written $S \preccurlyeq T$.

as well as

$$S \sqcap T \sqsubseteq S \qquad f{:}S \sqsubseteq f{:}T \Leftrightarrow S \sqsubseteq T \qquad S \sqsubseteq \top$$
$$S \sqsubseteq S \sqcup T \qquad \sim S \sqsubseteq \sim T \Leftrightarrow T \sqsubseteq S \qquad \bot \sqsubseteq S \ .$$

As subsumption is reflexive (8.1), antisymmetric (8.2), and transitive (8.3), it imposes a *partial order* on feature terms—for instance, we have $\top \sqsupseteq [\mathit{fruit}{:}\mathit{apple}] \sqsupseteq [\mathit{fruit}{:}\mathit{apple}, \mathit{color}{:}\mathit{green}] \sqsupseteq [\mathit{fruit}{:}\mathit{apple}, \mathit{color}{:}\mathit{green}, \mathit{wormy}{:}\mathit{no}] \sqsupseteq \cdots \sqsupseteq \bot$. This order constitutes a lattice structure in the set of feature terms.

Proposition 8.28 The set of all feature terms f and subsumption constitute a *subsumption lattice* $(f, \sqsubseteq)$ with a supremum of $S \sqcup T$ and an infimum of $S \sqcap T$ for all $S, T \in f$.
PROOF. Follows from proposition 8.26 on the page before. □

8.5.5 Consistency

The notion of *consistent* feature terms is important for defining the consistency of a configuration.

Definition 8.29 (Consistency) A feature term S is called coherent or *consistent* if there exists a feature algebra I and an I-assignment α such that $S^I_\alpha \neq \emptyset$. A feature term is called incoherent or *inconsistent* if it is not consistent. □

Definition 8.30 (Mutual consistency) Two feature term S and T are called *consistent with each other* if their intersection is consistent—that is, if $S \sqcap T$ is consistent. □

Definition 8.31 (Disjointness) Two feature terms S and T are called *disjoint* if their intersection is inconsistent—that is, if $S \sqcap T$ is inconsistent. □

Both deciding subsumption and equivalence can be tracked down to deciding consistency.

Proposition 8.32 Consistency, subsumption, and equivalence of feature terms are linear-time reducible to each other:

$$S \text{ inconsistent} \Leftrightarrow S = \bot \tag{8.4}$$
$$S \sqsubseteq T \Leftrightarrow S \sqcap \sim T \text{ inconsistent} \tag{8.5}$$
$$S = T \Leftrightarrow S \sqsubseteq T \wedge T \sqsubseteq S \tag{8.6}$$

PROOF. Follows from definitions. □

8.6 Conclusion

Feature logic combines boolean formulas with attribute descriptions. Its basic notions are *features,* functional properties or attributes of abstract objects, and *feature terms,* denoting sets of objects by their features. Feature logic has a convenient and natural set notation, describes objects by attributes, and provides a suitable notion of consistency. Feature logic thus fulfills our requirements for a SCM foundation, as discussed in chapter 7.

In this work, we always interpret feature terms as sets of objects, unless otherwise specified. "Traditional" set notation will not be required, with one single exception: We write $|S|$ to express the *cardinality* (the number of elements) of a set denoted by the feature term S under a given interpretation. All other required notation is already provided by feature logic, as introduced above. Having provided the necessary foundation, we now apply feature logic in the context of SCM, developing a layer of CM primitives on top of feature logic.

In the first place, Herodotus,
you must understand what it is that words denote,
in order that by reference to this
we may be in a position to test opinions, inquiries, or problems,
so that our proofs may not run untested ad infinitum,
nor the terms we use be empty of meaning.

— EPICURUS, Diog. Laert, Epicurus, X, 37

Part Three

The Version Set Model

Chapter 9

Versions and Components

Let us now return to the SCM domain. In this part, we show how feature logic can be used to describe SCM tasks and concepts, and how a unified SCM model can integrate the common four SCM models.

We begin with the SCM primitives layer, that is, basic versioning and access capabilities. We introduce the concept of version sets, *sets of component versions denoted by feature terms. We show how the features of components are modeled as alternatives over the features of the individual versions, and demonstrate how specific versions are selected by intersection.*

9.1 Identifying Versions

According with the SCM standards, as stated in section 1.3, we consider that the object of interest in SCM is a family of *software products.* Each of these software products breaks down in several *components,* each of which may exist in several *component versions.* A component version is an unbreakable, unambiguous configuration item.

In our setting, a component is a *set* of component versions and thus identifiable by a feature term. Each of the individual component versions is identified by a singleton subset. To bind these component versions together, we must assume at least one common feature across all component versions. Hence, we assume that each component can be identified uniquely via an *object* feature assigning each component a simple (unambiguous) component identifier.

Definition 9.1 (Component) A *component* is a set $K \sqsubseteq [object{:}k]$, where k is a simple feature term uniquely identifying the component. □

For instance, $[\mathit{object}\colon \mathit{printer}]$ denotes a component, but $[\mathit{fruit}\colon \mathit{apple}]$ does not.

A *component version* is uniquely identified by a singleton component. As a matter of convenience, we use the same name for singleton sets and the object they denote. Hence, a *component version* means both a singleton set and the object contained in that set.

Definition 9.2 (Component version) A *component version* is a component K with $K \sqsubseteq [\mathit{object}\colon k]$ and $|K| = 1$, where k is a simple feature term uniquely identifying the component. □

Definition 9.3 (Abstract component) A component K is called generic or *abstract* if it occurs in more than one version, i.e. $|K| > 1$. □

As an example, let

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

denote versions of a *printer* component, distinguished by their input language (PostScript or ASCII). The term $[\mathit{object}\colon \mathit{printer}]$ then denotes an abstract component, since it occurs in (at least) two versions.

Definition 9.4 (Bound component) A component K is called unambiguous or *bound* if it occurs in exactly one version, i.e. $|K| = 1$. □

Following our example, if both component versions $\mathit{printer}_1$ and $\mathit{printer}_2$ are bound, the abstract component $[\mathit{object}\colon \mathit{printer}]$ comes in exactly two versions.

We now abstract from components and speak of versions alone. A collection of arbitrary components in arbitrary versions is called a *version set.* We still assume that a *object* feature exists.

Definition 9.5 (Version set) A *version set* is any set $V \sqsubseteq [\mathit{object}\colon \top]$. □

For consistency, a *version* is a singleton version set. This implies that a component version is both a component and a version.

Definition 9.6 (Version) A *version* is any version set $V \sqsubseteq [\mathit{object}\colon \top]$ such that $|V| = 1$. □

The features of a component are modeled as *alternatives* over the features of each component version. That is, a component is the *union* of its individual component versions:

Definition 9.7 (Components vs. Component versions) A component K existing in n component versions $V_1, V_2, \ldots, V_n$, is determined as the union of all V_i:

$$K = V_1 \sqcup V_2 \sqcup \cdots \sqcup V_n = \bigsqcup_{1 \le i \le n} V_i \ . \tag{9.1}$$

□

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 an example, reconsider our printer setting. The *printer* component itself is determined as the union of $printer_1$ and $printer_2$:

$$\begin{aligned} printer &= printer_1 \sqcup printer_2 \\ &= \left[object{:}printer, print\text{-}language{:}\{postscript, ascii\}\right] \ . \end{aligned}$$

The term *printer* can be read either as union of the component versions $printer_1$ and $printer_2$, or as the features of the *printer* component, which is "the printer language is PostScript or ASCII".

9.2 Selecting Versions

To retrieve individual versions of a version set, the version set is intersected with a *selection term* containing the desired features. For any version set T and a selection term S, 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.

For instance, consider the printer example from section 9.1. In figure 9.1 on the following page, we have represented some version sets using the well-known *Venn diagrams;* each curve represents a set enclosing the denoted objects. We see that selecting $S = [print\text{-}language{:}postscript]$ from *printer* returns $printer_1$, since $printer_1$ is a subset of S (that is, $printer_1 \sqsubseteq S$), while $printer_2$ is not (that is, $printer_2 \sqsubseteq \sim S$).

Formally, we have $printer \sqcap S = (printer_1 \sqcup printer_2) \sqcap S = (printer_1 \sqcap S) \sqcup (printer_2 \sqcap S) = printer_1 \sqcup \bot = printer_1$. Here, $printer_2 \sqcap S = \bot$ holds since the *print-language* feature may have only one value.

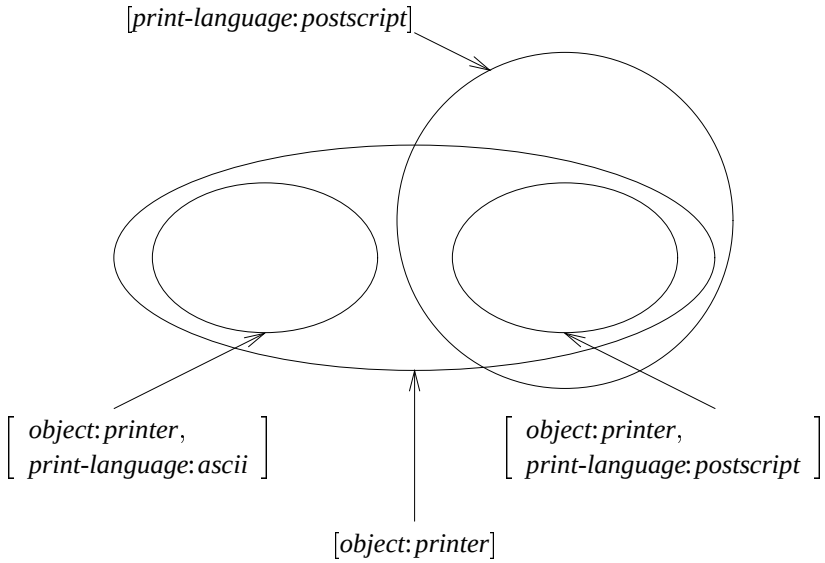

Figure 9.1: Selecting component versions

The selection term may be an arbitrary feature term. For instance, we may select any printer except $printer_1$, by selecting

$$\begin{aligned} S &= \sim printer_1 \\ &= \sim[object{:}printer, print\text{-}language{:}postscript] \\ &= \sim[object{:}printer] \sqcup \sim[print\text{-}language{:}postscript] \end{aligned}$$

Obviously, we have $printer \sqcap S = printer_2$, since $(printer_1 \sqcup printer_2) \sqcap \sim printer_1 = printer_2$ holds.

Due to the semantics of feature logic, there is a potential danger in selections. Since every non-existing feature must be specified as explicitly as every existing feature, a selection with non-specified, orthogonal features may result in counter-intuitive results. For instance, selecting $S = [colors{:}4]$ from $printer = printer_1 \sqcup printer_2$ would result in the entire *printer* set, although the *colors* feature is neither defined nor undefined in each $printer_1$ and $printer_2$. Which is even worse, $printer \sqcap S$ results in a new term *augmented* with the *colors* feature from S:

$$\begin{aligned} &\mathit{printer} \sqcap [\mathit{colors}: 4] \\ &\qquad = [\mathit{object}: \mathit{printer}, \mathit{print\text{-}language}: \{\mathit{postscript}, \mathit{ascii}\}, \mathit{colors}: 4] \end{aligned}$$

Although this behavior makes sense in a set-theoretic context, it is undesirable for SCM selection purposes. Fortunately, this behavior can easily be avoided in an implementation by disallowing non-orthogonal selection terms. In section 19.5, we discuss techniques for safe interactive exploration of the version space.

To conclude, the ability to use boolean expressions for both identification and selection complies with the requirement for *unification* as stated in section 7.3. It allows our model to encompass attribute-oriented identification schemes as well as attribute-oriented selection schemes. Alas, the expressiveness of feature logic comes with the cost of NP-completeness, which implies exponential time complexity for selections in the worst case. Fortunately, all of today's SCM tasks can be realized efficiently, as we discuss in chapter 14.

9.3 Making Selections Unambiguous

As our selection scheme is set-oriented, the result of each selection T' of a selection $T' = T \sqcap S$ is just another version set and may thus be ambiguous. To make our selection unambiguous, we may give a second selection term S' and select $T'' = T' \sqcap S'$, give a third selection term S'', and so on, narrowing the choice set incrementally until a singleton set is selected, containing the desired version. Such techniques can be used to explore the configuration space interactively, narrowing and extending the selection as desired. We discuss such interactive tools in section 19.5.

As discussed in section 3.3, most SCM tools make their selection unambiguous as soon as possible, using *configuration rules* to express preferences and defaults. The semantics of such complex selection schemes can be described on top of feature logic, by defining *preference operators* and *default operators* which handle ambiguity and inconsistency.

Definition 9.8 (Preferences and defaults) The *preference operator* "and-then" and the *default operator* "or-else" are defined as

$$S_1 \text{ and-then } S_2 = \begin{cases} S_1 & \text{if } S_1 \text{ is unambiguous (i.e. } |S_1| = 1), \\ S_1 \sqcap S_2 & \text{otherwise} \end{cases}$$

$$S_1 \text{ or-else } S_2 = \begin{cases} S_1 & \text{if } S_1 \text{ is non-empty (i.e. } S_1 \neq \bot), \\ S_2 & \text{otherwise} \end{cases}$$

where the equivalences $T \sqcap (S_1 \text{ and-then } S_2) = (T \sqcap S_1 \text{ and-then } T \sqcap S_2)$ and $T \sqcap (S_1 \text{ or-else } S_2) = (T \sqcap S_1 \text{ or-else } T \sqcap S_2)$ hold. □

Using "and-then" and "or-else", we can express preferences and defaults in our selection terms. For instance, $S = \big([\mathit{current}: \top] \text{ or-else } [\mathit{fixed}: \mathit{true}]\big)$ first selects the current version, and, if there is none, a "fixed" current version. The selection $S = \big([\mathit{os}: \mathit{unix}] \text{ and-then } [\mathit{unix\text{-}flavour}: \mathit{bsd}]\big)$ selects the UNIX version and, should this choice be ambiguous, the BSD variant.

Another practical extension are additional constraints, for instance quantification, arithmetic constraints or function interfaces. 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 [Sne91]. Well-known constraint solving systems like the Simplex Method or language-specific consistency checkers, as discussed in section 3.6.2, may help to decide about inconsistence. Such constraints are discussed in section 18.5.

Users must be aware, however, that the usage of preferences or additional constraints may lead to unresolved constraints due to undecidability. Such unresolved constraints can be avoided by using extensions either only for version identification or only for version selection, making preferences and additional constraints useful extensions in many environments.

9.4 Dynamic Version Creation

So far, we have thought of components as a union over a finite set of versions. But it is also conceivable that specific versions are *dynamically created and instantiated* just as they are requested. As an example, consider a component *network-interface* that can be customized with a specific network address. As the number of network addresses is (in theory) infinite, the *network-interface* component is the union over an infinite set of possible versions. Hence, the features of *network-interface* become

$$\mathit{network\text{-}interface} = [\mathit{object}: \mathit{network\text{-}interface}, \mathit{address}: \top]$$

which means that for any version (subset) of *network-interface*, the *address* feature must be defined. A SCM system may now be set up such that a selection $\mathit{network\text{-}interface} \sqcap [\mathit{address}: 127.0.0.1]$ would actually *instantiate* the generic *network-interface* component with a version for the address 127.0.0.1, creating versions on-the-fly as needed. In practice, this specific example would probably not be implemented via a SCM system, but through some run-time configuration mechanism (which may again realize the version set model).

As a more SCM-specific example, consider *change sets,* as discussed in section 2.7. As (more or less) arbitrary combinations of change sets are possible, an SCM system should be set up such that these versions are created only when requested. A selection term like $[\mathit{change\text{-}41}\colon\top, \mathit{change\text{-}42}\colon\top]$, for instance, would result in the creation of a version with the changes 41 and 42 applied. We further discuss this idea of representing change sets and dynamic version creation in chapter 11.

9.5 Assigning Features to Versions

We close this chapter by discussing the question which features of components and versions are significant and how these should be modeled in feature logic. The specific attribution methodology is part of higher SCM layers (notably the protocol and policy layers); in order to maintain flexibility at these layers, we do not impose more meaning than necessary on specific features.

There are only few existing attribution methodologies; we have already discussed the CAPITL methodology in section 4.5; another frequently-cited scheme is *faceted classification* [PD87, OHPDB92]. However, we can supply some general guidelines imposed by feature logic.

9.5.1 Variants must be Disjoint

Definition 9.2 requires that each component version be singleton and thus unambiguous. This implies that the intersection of any two different component versions V_i and V_j must be empty, or $V_i \sqcap V_j = \bot$. For instance, consider the following terms:

$$\begin{aligned}\mathit{screen}_1 &= [\mathit{object}\colon \mathit{screen}, \mathit{depth}\colon 1]\\ \mathit{screen}_2 &= [\mathit{object}\colon \mathit{screen}, \mathit{x\text{-}resolution}\colon 1024, \mathit{y\text{-}resolution}\colon 1024]\end{aligned}$$

screen_1 and screen_2 do *not* identify two distinct component versions, as their intersection is non-empty:

$$\begin{aligned}&\mathit{screen}_1 \sqcap \mathit{screen}_2\\ &\qquad = [\mathit{object}\colon \mathit{screen}, \mathit{depth}\colon 1, \mathit{x\text{-}resolution}\colon 1024, \mathit{y\text{-}resolution}\colon 1024]\end{aligned}$$

is the set of all screens with depth 1 and a resolution of 1024×1024 pixels. To have screen_1 and screen_2 denote two unambiguous variants, screen_1 must include resolution features, and screen_2 must include a *depth* feature.

9.5.2 Feature Values keep Versions Disjoint

A simple way to keep versions disjoint is to assign each of them a common feature with differing values. For instance, two variants for the UNIX and WINDOWS operating systems would be easily distinguished via an *operating-system* feature:

$$os_1 = [object{:}\, os, operating\text{-}system{:}\, dos]$$
$$os_2 = [object{:}\, os, operating\text{-}system{:}\, unix]$$

As all features, *operating-system* can have only one value. Hence, selecting the UNIX variant $[operating\text{-}system{:}\, unix]$ automatically excludes the DOS variant $[operating\text{-}system{:}\, dos]$ and vice versa.

The alternative, introducing *dos* and *unix* features, is less convenient, since the alternative operating system must be excluded explicitly; this would only make sense if we expected some future version to support both UNIX and DOS variants.

9.5.3 Features Model Variance Dimensions

Re-consider the *screen* example. Let us assume that in fact, arbitrary combinations of depth and resolution are possible. In this case, both depth and resolution constitute orthogonal variance dimensions and should be modeled by different features. With dynamic version creation, each of these instantiations of depth and resolution could be created on-the-fly, making the *screen* component a union over an infinite number of possible component versions, or

$$screen = [object{:}\, screen, depth{:}\, \top, x\text{-}resolution{:}\, \top, y\text{-}resolution{:}\, \top] \ .$$

9.5.4 Alternatives may Denote Multiple Features

Sometimes, a single version supports several alternatives. Let *smart-printer* be a specific printer component may determine automatically the language of its printer data and thus support several languages at once. This can again be modeled as *alternative,* for instance as:

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

where *smart-printer* is a singleton set; thus, the selections

$$smart\text{-}printer \sqcap [print\text{-}language{:}\, ascii] \ ,$$
$$smart\text{-}printer \sqcap [print\text{-}language{:}\, postscript] \ , \text{and}$$
$$smart\text{-}printer \sqcap [print\text{-}language{:}\, pcl]$$

all return the same component version.

9.5.5 Constraints Exclude Feature Combinations

It is often easier to specify the *non-existence* of certain feature combinations rather than to specify all existent combinations. This is especially true for dynamic version creation. Such *feature constraints* are best modeled as common features of the component. A screen with one plane, for instance, is monochrome: it can only show either black or white pixels. This general constraint can be expressed through an implication

$$C = (\mathit{planes}{:}\,1 \to \mathit{colors}{:}\,2) = {\sim}[\mathit{planes}{:}\,1] \sqcup [\mathit{colors}{:}\,2] \ ,$$

stating that the screen has either more than one plane (strictly spoken, any other number of planes than one) or two colors. This constraint may become part of the common features of the *screen* component:

$$\mathit{screen} = (\mathit{planes}{:}\,1 \to \mathit{colors}{:}\,2) \sqcap (\mathit{screen}_1 \sqcup \cdots \sqcup \mathit{screen}_n)$$

which makes the relationship between *planes* and *colors* explicit and saves users from specifying it in each term denoting the component versions.

9.6 Discussion

Using feature terms for both identification and selection of version sets constitutes an expressive and general scheme. By handling version sets instead of individual versions, we allow ambiguity as well as dynamic creation of versions. Through preference and default operators, we can model disambiguation as found in SCM systems.

Flexibility has its drawbacks. Using complex terms for identification as well as for selection may result in exponential time complexity. Selection with orthogonal terms leads to counter-intuitive results. Finally, there are only few attribution methodologies that would help classifying versions according to their features. All three issues must be and can be addressed at the higher SCM layers.

When you have mastered numbers,
you will in fact no longer be reading numbers,
any more than you read words when reading books.
You will be reading meanings.

— HAROLD GENEEN, Managing

Chapter 10

Composing Configurations

Having discussed how individual components are versioned, we can now turn to collections of components, or configurations. *We discuss how features propagate from components to configurations, and how the features of a configuration are determined by the common features of its component versions. We show how common features are used as a means to determine consistency, and discuss how configurations integrate with other versioning concepts discussed so far.*

10.1 Extrinsic and Intrinsic Features

In chapter 9, we have seen how features propagate from versions to components: Each feature of a component version becomes an alternative feature in the component itself. The next questions are: how do features propagate from components to configurations, and how do these features interact with each other? Basically, there are two alternatives.

Feature unification. The features of the configuration are determined by the common (i.e. unified) features of the component versions; these common features determine the component versions. For instance, adding a $[os{:}\,dos]$ version to a configuration makes $[os{:}\,dos]$ a feature of the entire configuration, excluding all non-DOS versions in other components.

Feature union. The features of the configuration are determined by the united features of the component versions; component features do not interact. For instance, when composing a configuration from two components $vector = [object{:}\,vector]$ and $multiset = [object{:}\,multiset]$, the features of the configuration should be $[object{:}\,\{vector, multiset\}]$—that is, the objects are *vector* and *multiset*.

A solution to this dilemma is to distinguish between *extrinsic* features, which are unified, and *intrinsic* features, which are not.

Definition 10.1 (Extrinsic and intrinsic features) Features of a component version are either *extrinsic* or *intrinsic*. A dependent or *extrinsic* feature of a component is a feature that determines the features of other components in a configuration. An independent or *intrinsic* feature is a feature that is not extrinsic. □

Extrinsic features are typically features that must be common across all components, for instance *operating-system customer*, or *bug-fix-377*. Intrinsic features are often process-driven and used for identifying purposes only, like *author*, *date*, or *change-log*. The *object* feature is also an intrinsic feature.

We first discuss the treatment of extrinsic features, including a larger example, and than turn to the integration of intrinsic features.

10.2 Unifying Extrinsic Features

In chapter 9, we have seen that in our model, feature terms may be used for identification as well as for selection purposes. Until now, we have identified a component version by its intrinsic features. But we may also use the feature term of a component version to specify the features of its environment, notably the features of *other* component versions—that is, extrinsic features.

As an example, consider a simple portable CD-ROM player built from a *screen* and a *drive* component. Each comes in two versions $screen = screen_1 \sqcup screen_2$ and $drive = drive_1 \sqcup drive_2$, where

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

that is, $screen_1$ is a high-resolution screen, which requires a high-speed drive, and $screen_2$ is a medium-resolution screen, which also works with medium- or low-speed drives.

Indeed, the version set model does not make a distinction between *providing* and *requiring* features. In the *screen* component, the *drive-speed* feature is required; in the *drive* component, the *drive-speed* feature is provided. The only

statement we can make is that any configuration of the *screen* and *drive* components should exclude $drive_2$ if $screen_1$ is included. This leads us to the general idea that each configuration should *inherit* the features of its components, and that the common features of the components determine the features of the configuration:

Definition 10.2 (Configuration features) Let C be a configuration of n components with the extrinsic features $K_1, K_2, \ldots, K_n$. Then, C has the features

$$C = K_1 \sqcap K_2 \sqcap \cdots \sqcap K_n = \prod_{1 \le i \le n} K_i \quad . \tag{10.1}$$

□

As a simple configuration example, consider the CD-ROM drive. The configuration of *screen* and *drive* has the features

$$\begin{aligned} C = \big(& [\mathit{resolution}\colon \mathit{high}, \mathit{drive\text{-}speed}\colon \mathit{high}] \\ & \sqcup [\mathit{resolution}\colon \mathit{medium}, \mathit{drive\text{-}speed}\colon \{\mathit{high}, \mathit{medium}, \mathit{low}\}] \big) \\ & \sqcap \big([\mathit{drive\text{-}speed}\colon \mathit{high}] \sqcup [\mathit{drive\text{-}speed}\colon \mathit{medium}] \big) \\ = & [\mathit{resolution}\colon \mathit{high}, \mathit{drive\text{-}speed}\colon \mathit{high}] \\ & \sqcup [\mathit{resolution}\colon \mathit{medium}, \mathit{drive\text{-}speed}\colon \mathit{high}] \\ & \sqcup [\mathit{resolution}\colon \mathit{medium}, \mathit{drive\text{-}speed}\colon \mathit{medium}] \quad , \end{aligned}$$

that is, actually three possible configurations with different resolutions and drive speeds.

Even without handling of intrinsic features, we already see that a configuration will again be represented as a set and may be possibly ambiguous. We also see that composing a configuration is very much like selection: each component in the configuration imposes its constraints on the other components. This scheme can be used for checking consistency with regard to the features, as discussed in the next section.

10.3 A Unification Example

As a larger example for illustrating configuration consistency, consider figure 10.1 on the next page. We see three source components of a text editor, where each component comes in several variants. We can choose between two operating systems (*dos* and *unix*), four screen types (*ega*, *tty*, *x11* and *news*), and two screen device drivers (*dumb* and *ghostscript*). The *dumb* driver assumes that the screen type

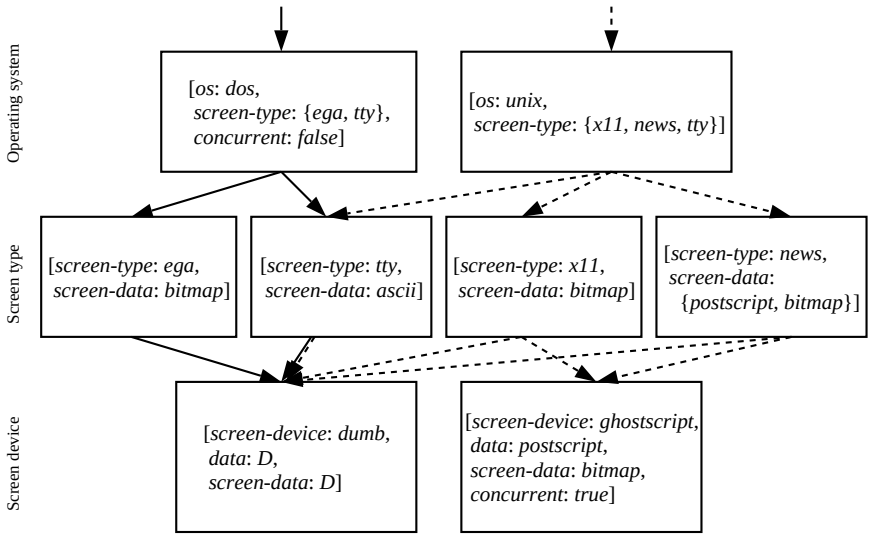

Figure 10.1: Consistent configurations in a text/graphic editor

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 plain lines.

As final component, we must choose a screen device driver. *ghostscript* cannot be chosen, since it requires *concurrent* to be true, which is not the case under *dos.* The *dumb* driver remains; *D* is instantiated to *bitmap* or *ascii*, depending on the screen type, making our choice complete: *editor* can be built in a *ega* and a *tty* variants, inheriting the features of its source components. As an alternative, consider the choice $[os{:}\,unix]$, 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 constraints regarding (sub-)systems are permitted as well. The drawback is that one single language must be used to specify constraints, to specify the component features, and to select component versions. With feature logic, we hope having chosen a well-established foundation with sufficient richness of expression.

10.4 Handling Intrinsic Features

We now show how to propagate intrinsic features in configurations. As stated in the introduction, it makes perfectly sense for intrinsic features like *author* or *status*, to differ across components; *object* features even differ by definition. To keep these intrinsic features from constraining other component versions, we localize them, that is, we make them depend on the specific component.

A possible approach to localize intrinsic features is to prefix all intrinsic features f with the component name k. This would result in orthogonal features like *tty-author* or *screen-status*. A more elegant 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. The idea is to create a *configuration term* with these implications that automatically selects the desired version(s) from each component.

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

Definition 10.3 (Aggregation) Let $I = \{f_1{:}\top, f_2{:}\top, \ldots, f_n{:}\top\}$ be a feature term denoting intrinsic features. Let S and T denote components with

$$\begin{aligned} S &= [object{:}s] \sqcap S' \sqcap S'' \text{ and} \\ T &= [object{:}t] \sqcap T' \sqcap T'' \ , \end{aligned} \tag{10.2}$$

such that $S'', T'' \sqsubseteq I$ denote the intrinsic features, and $S', T' \not\sqsubseteq I$ denote extrinsic features. The *aggregation* of S and T, written $S \boxplus_I T$, is then defined as

$$S \boxplus_I T = S' \sqcap T' \sqcap \big([object{:}s] \rightarrow S''\big) \sqcap \big([object{:}t] \rightarrow T''\big) \ . \tag{10.3}$$

□

Every aggregation $S \boxplus_I T$ selects version subsets from $[object{:}s]$ and $[object{:}t]$:

Proposition 10.4 Let $S \sqsubseteq [object{:}\, s]$ and $T \sqsubseteq [object{:}\, t]$ denote components, and I denote intrinsic features, as described above. Then,

$$[object{:}\, s] \sqcap (S \boxplus_I T) \sqsubseteq S \tag{10.4}$$

holds.

PROOF. Let $T = [object{:}\, t] \sqcap T' \sqcap T''$, as in (10.2), satisfying the requirements of definition 10.3 on the preceding page. Then, we have

$$\begin{aligned} U &= [object{:}\, s] \sqcap (S \boxplus_I T) \\ &= [object{:}\, s] \sqcap \big(S' \sqcap T' \sqcap \big([object{:}\, s] \to S''\big) \sqcap \big([object{:}\, t] \to T''\big)\big) \quad . \end{aligned} \tag{10.5}$$

We reduce the first sub-formula, following the pattern $A \sqcap (A \to B) = A \sqcap B$:

$$\begin{aligned} [object{:}\, s] \sqcap \big([object{:}\, s] \to S''\big) &= [object{:}\, s] \sqcap \big(\sim [object{:}\, s] \sqcup S''\big) \\ &= [object{:}\, s] \sqcap S'' \end{aligned} \tag{10.6}$$

as well as the second, following the pattern $A \sqcap (\sim A \to B) = A$:

$$\begin{aligned} [object{:}\, s] \sqcap \big([object{:}\, t] \to T''\big) &= [object{:}\, s] \sqcap \big(\sim [object{:}\, t] \sqcup T''\big) \\ &= [object{:}\, s] \end{aligned} \tag{10.7}$$

and can reformulate (10.5) using (10.6) and (10.7) to

$$\begin{aligned} U &= [object{:}\, s] \sqcap (S' \sqcap T' \sqcap S'') \\ &= \big([object{:}\, s] \sqcap S' \sqcap S''\big) \sqcap T' \\ &= S \sqcap T' \quad . \end{aligned}$$

Hence, $U = [object{:}\, s] \sqcap (S \boxplus_I T) = S \sqcap T' \sqsubseteq S$, which was to be shown. □

Using the aggregation operator, we can extend definition 10.2 with *object* features and intrinsic features and formally define how all kinds of features propagate from components to configurations.

Definition 10.5 (Configuration vs. components) 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 intrinsic features, the configuration C is identified by

$$\begin{aligned} C &= [object{:}\, k_1 \sqcup k_2 \sqcup \cdots \sqcup k_n] \sqcap K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n \\ &= [object{:}\, k_1 \sqcup k_2 \sqcup \cdots \sqcup k_n] \sqcap \Big[\!\!\Big[\boxplus_I \Big]\!\!\Big]_{1 \le i \le n} K_i \quad , \end{aligned} \tag{10.8}$$

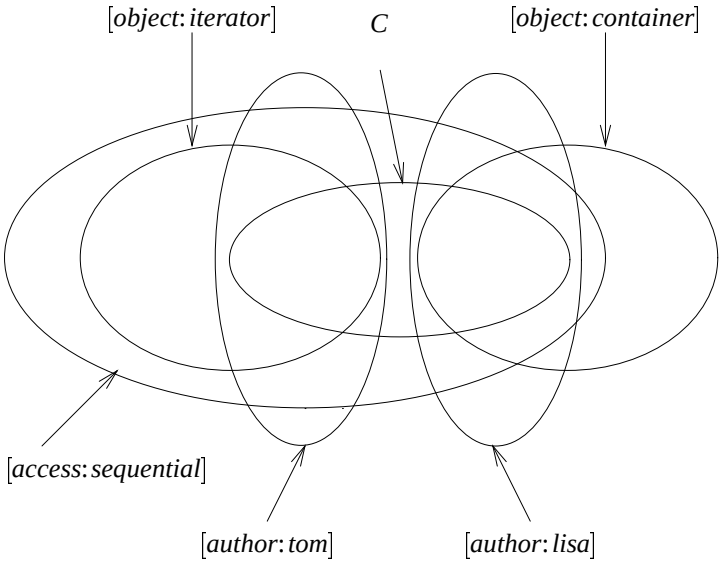

Figure 10.2: Creating a configuration from two components

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

As an example, consider two components

$$\begin{aligned} \mathit{container} &= \big[\mathit{object}\colon \mathit{container}, \mathit{author}\colon \mathit{lisa}, \mathit{access}\colon \{\mathit{sequential}, \mathit{random}\}\big] \\ \mathit{iterator} &= \big[\mathit{object}\colon \mathit{iterator}, \mathit{author}\colon \mathit{tom}, \mathit{access}\colon \mathit{sequential}\big] \ . \end{aligned}$$

Let $I = [\mathit{author}\colon \top]$ be the set of intrinsic features. According to definition 10.5 on the preceding page, the configuration C containing *container* and *iterator* is

$$\begin{aligned} C &= \big[\mathit{object}\colon \{\mathit{container}, \mathit{iterator}\} \sqcap (\mathit{container} \boxplus_I \mathit{iterator})\big] \\ &= \big[\mathit{object}\colon \{\mathit{container}, \mathit{iterator}\}, \mathit{access}\colon \mathit{sequential}, \\ &\qquad (\mathit{object}\colon \mathit{container} \to \mathit{author}\colon \mathit{lisa}), (\mathit{object}\colon \mathit{iterator} \to \mathit{author}\colon \mathit{tom})\big] \ . \end{aligned}$$

Not only does the term C unify the extrinsic features of *container* and *iterator* to $[\mathit{access}\colon \mathit{sequential}]$. As illustrated in figure 10.2, it also ensures that whenever the *container* component is selected, Lisa's version is returned:

$$C \sqcap [\mathit{object}\colon \mathit{container}] \sqsubseteq [\mathit{author}\colon \mathit{lisa}]$$

and that whenever the *iterator* component is required, Tom's version is returned:

$$C \sqcap [\mathit{object}\colon \mathit{iterator}] \sqsubseteq [\mathit{author}\colon \mathit{tom}] \ .$$

Likewise, requesting Tom's version returns the *iterator* component:

$$C \sqcap [\mathit{author}\colon \mathit{tom}] \sqsubseteq [\mathit{object}\colon \mathit{iterator}] \ .$$

10.5 Properties of Configurations

We can now define some properties of configurations formally, according to definition 10.5 on page 106.

Definition 10.6 (Configuration) A *configuration* is a set $C \sqsubseteq [\mathit{object}\colon c]$, where c is a feature term identifying the set of configuration components. □

Definition 10.7 (Consistent configuration) A configuration C is called *consistent* with respect to its features if $C \neq \bot$—that is, if the number of possible configurations is non-zero. □

Definition 10.8 (Bound configuration) A configuration C is called unambiguous or *bound* if it is an aggregation of component versions; formally, C is bound if it is a set $C \sqsubseteq [\mathit{object}\colon c]$ such that $|C| = |c|$ holds. □

Definition 10.9 (Abstract configuration) A configuration C is called ambiguous, dynamic, or *abstract,* if it is not bound; that is, $|C| > |c|$ holds. □

Definition 10.10 (Generic configuration) A configuration C is called partially bound or *generic* if it is abstract and a true subset of the configuration universe; that is, $|c| < |C| < |[\mathit{object}\colon \top]|$ holds. □

We see that the informal definitions given in section 3.3 can now be put more precisely through well-founded formal definitions.

10.6 Configurations and Ambiguity

As configurations are again ordinary version sets (albeit containing several components), all selection properties for component versions apply, as discussed in chapter 9. A configuration can be dynamically created, for example; but it can

also occur in multiple versions. We have already seen how ambiguity in a component propagates to all configurations containing this component. But ambiguity may also affect the actual set of components contained in the configuration.

As a simple example, consider a problem occurring in the 4.1 release of the SunOS operating system. The system library libc comes in two versions: one dynamic version for dynamically linked programs, and one static version for statically linked programs. Both libraries are identical, except for one minor difference: The strerror() function is only contained in the dynamic library. This means that programs using this function must include their own strerror component if compiled statically, and omit this component if compiled dynamically.

For simplicity, let us assume a program with only one component *program*, and without any specific features. Using version sets, the alternative configurations C are modeled as

$$\begin{aligned} C &= [object\colon program, \sim linkage\colon static] \qquad (10.9)\\ &\quad \sqcup [object\colon \{program, strerror\}, linkage\colon static]\\ &= [object\colon program]\\ &\quad \boxplus_I ([object\colon strerror, linkage\colon static] \sqcup \sim [linkage\colon static])\\ &= [object\colon program] \boxplus_I (linkage\colon static \to object\colon strerror) \quad . \qquad (10.10) \end{aligned}$$

The disjunctive form in (10.9) shows what the actual configurations look like. The implication constraint in (10.10), however, explicitly states that whenever static linkage is required, the *strerror* object must be contained as well. These two possibilities of expressing alternatives—enumeration or constraints—will be discussed further when dealing with revisions and changes in chapter 11.

As the components of a configuration may be configurations again, we can describe a full system model by compositions ($\boxplus_I$) and alternatives ($\sqcup$), similar to AND/OR graphs discussed in section 3.2.1. Through transformations of the configuration term according to the rules of feature logic, arbitrary interchanged selection and composition stages are possible. Additionally to compositions and alternatives, complements may be used to express that a specific version set *not* be included in a selection—for instance, $S' = S \sqcap \sim T$ contains all configurations of S that do not contain T. As versions, components, and configurations are all modeled by version sets, all version set operations can be applied equally, making configurations first-class objects.

10.7 Features of Derived Components

Closely related to the composition of configurations is the *derivation* of components from a set of source components, as discussed in chapter 4. To determine the features of derived components, we use a variation of definition (10.8). 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 extrinsic features of its source components, just as a configuration inherits the extrinsic features of its components.

Definition 10.11 (Derivation) Let a component $K \sqsubseteq [object\colon k]$ be derived from n source components $K_1, K_2, \ldots, K_n$, and let a term I denote their intrinsic features. K is then identified by

$$\begin{aligned} K &\sqsubseteq [object\colon k] \sqcap K_1 \boxplus_I K_2 \boxplus_I \cdots \boxplus_I K_n \\ &\sqsubseteq [object\colon k] \sqcap \boxplus_{1 \le i \le n,\, I} K_i \quad , \end{aligned} \tag{10.11}$$

The term $K_1 \boxplus_I \cdots \boxplus_I K_n$ is called *source configuration* of K. □

The explicit setting of the *object* feature removes all implications generated by the aggregation operator—only extrinsic features remain to be unified. As an example, consider the editor example from figure 10.1 on page 104. Let us denote the three components by

$$\begin{aligned} os &= [object\colon os, author\colon tom] \quad , \\ screen\text{-}type &= [object\colon screen\text{-}type, author\colon lisa] \quad , \text{and} \\ screen\text{-}device &= [object\colon screen\text{-}device, author\colon john] \quad , \end{aligned}$$

respectively; let the intrinsic features be $I = [author\colon \top]$. If we derive an *editor* component from a DOS/EGA configuration, it is identified by

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

that is, the *object* features and intrinsic features of the source components are stripped, and all extrinsic features are unified.

10.8 Discussion

By raising the version set model from components to configurations, we have supplied a uniform denotation for components and systems with uniform query mechanisms. In our model, configurations are full first-class objects; in fact, a component is just the special case of a configuration with a single component. Ambiguity is allowed in configurations just as in components; even the set of components can depend on other features. Extrinsic features are propagated to configurations as well as to derived components, while intrinsic features remain dependent on the specific component.

Our configuration setting also has some drawbacks. While feature propagation from versions to components was simple and smooth, feature propagation from components to configurations is much less elegant, due to the variety of feature interactions in configurations. With the distinction between intrinsic and extrinsic features, and the special handling of *object* features, we hope having supplied solutions for modeling the large majority of feature interactions.

Uniting *object* features has the desired effect of excluding all components which are not part of the configuration. But using a union for what should actually be a set value has some unfortunate side-effects, notably when talking about ambiguous configurations. For instance, how shall a configuration $C = [object: \{a, b, c\}]$ be interpreted: as a configuration of three components *a*, *b*, and *c*; or as an ambiguous configuration involving either *a*, *b*, or *c*?

To solve this problem, the best solution for that problem would be a feature logic enhanced with set values. Smolka [Smo92] discusses such an extension of feature logic, generalizing feature terms to *concept descriptions,* using set-valued features called *roles. object* could then be represented as role instead of a feature, allowing multiple *object* values. Unfortunately, Smolka does not give a consistency notion for concept descriptions, let alone a consistency-checking algorithm like feature unification. In [Man94], Manandhar presents an alternative feature logic whose consistency notion encompasses set values. But Manandhar's logic has no complement operator and hence no negation. The integration of set-valued features or roles in a feature logic including a consistency-checking constraint system remains an open problem.

In the absence of roles, there is an *ad hoc* solution for SCM systems interpreting feature terms: always use the *widenest possible set.* In our case, this results in *C* being interpreted as set of three objects, as was our intention. Ambiguity is

still possible as soon as other features are involved. For example, consider

$$C' = \{\,[\mathit{device}{:}\,x11, \mathit{object}{:}\,\{a,b,c\}], [\mathit{device}{:}\,\mathit{win}, \mathit{object}{:}\,\{a,b\}]\,\}\ .$$

Given C', an SCM system would interpret the outer union as alternative, because the united version sets are disjunct; there is no unambiguous widenest possible set. The inner unions, however, can be interpreted as set values, as in C.

As is often the case,
providing information about the system as a whole
implies properties of individual components.

— DEBORAH L. McGUINNESS, LORI ALPERIN RESNICK and CHARLES ISBELL,
Description Logic in Practice

Interchangeable parts won't.

— LAWS OF ASSEMBLY, II

Chapter 11

Changes and Revisions

We shall now turn from planned *versions, that is, versions as they occur in the final product, to* unplanned *versions, that is, versions as they occur during software development and maintenance. In this chapter, we show how to model revisions and changes through feature logic. The basic idea is to identify revisions by the applied changes, as in the change-oriented models. By expressing* revision constraints*, we constrain the versioning space by disallowing specific change combinations—up to revision graphs as in the version-oriented models.*

11.1 Revision Graphs

In section 2.7, we discussed the Change-Oriented Model, where revisions are the result of changes applied to a baseline. In our model, we also assume that new revisions are created by applying changes on existing revisions. In contrast to the Change-Oriented Model, we still focus on versions and do not treat changes as separate entities. However, we *identify* revisions by the changes applied and the changes not applied.

Let us denote the revisions of a version set by $R_0, R_1, R_2, \ldots$, and so on; $\delta_1, \delta_2, \delta_3, \ldots$ denote individual changes. Each revision R_i is created by applying a change δ_i to some originating revisions $R_j, \ldots, R_k$—for instance, the change δ_1 results in revision R_1. The exception to the rule is the *baseline* R_0, which has no associated change.

A simple way to illustrate the relationships between revisions and changes is a *version graph,* as discussed in section 2.2. In this chapter, we shall use a *revision graph* where each derivation between revisions is annotated with the associated

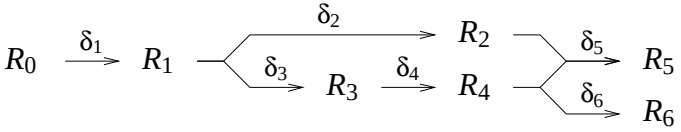

Figure 11.1: A revision graph

change. In short, an edge

$$R_i \xrightarrow{\delta_j} R_j$$

between two revisions R_i and R_j means that R_j was created by applying the change δ_j on R_i. Since this implies that R_i is older than R_j, revision graphs represent the evolution of a version set in time.

As an example of a revision graph, consider figure 11.1. Most revisions have one single origin—for example, revision R_1 was created by applying δ_1 on the baseline R_0. But there is also a case of *multiple origins*: Revision R_5 was created from R_2 and R_4 by applying the change δ_5.

Individual revisions can be uniquely identified by the included and excluded changes. For instance, revision R_1 includes the change δ_1, and excludes all others. Revision R_4 includes δ_1, δ_3, and δ_4, and excludes δ_2, δ_5, and δ_6. Revision R_6 includes all changes except δ_5.

But why should one care about identifying revisions by their changes? The answer is: if there are n changes, there might be up to 2^n revisions—that is, one revision for each combination of included and excluded changes. Assigning revision numbers is convenient for a small set of revisions, but if there is a large number of changes that can be applied independently, any linear numbering scheme for revisions soon runs out of numbers.

With a given revision graph, it suffices to state only a few of the included and excluded changes to identify individual revisions. Let us take a look at figure 11.1. To identify revision R_6, it suffices to state that the change δ_6 should be included. Likewise, to select R_3, we only need to state that δ_3 should be included and that δ_4 should be excluded.

This simplification is possible because revision graphs express *implications* between changes. For instance, applying the δ_4 change implies that the δ_3 change be applied as well—there is no revision including δ_4 and excluding δ_3. Hence, when selecting a revision that includes the change δ_4, we do not need to specify that the implied changes δ_3, δ_2, and δ_1 are to be included as well. Likewise,

excluding the change δ_4 means that the changes implying δ_4 are excluded as well—that is, δ_5 and δ_6, since they imply δ_4 to be applied.

In this chapter, we show how to identify revisions just by stating included and excluded changes, and how to use implications between changes to structure revision graphs.

11.2 Identifying Revisions

We now formally define the notions of changes and revisions. We begin with introducing *delta features*, which we use to identify changes.

Definition 11.1 (Delta feature) A *delta feature* is an identifier δ_i denoting the application of some change δ_i. □

Delta features are convenient for grouping revisions into version sets, called *delta sets.*

Definition 11.2 (Delta set) A *delta set* $\Delta_i = [\delta_i \colon \top]$ is the set of objects where the change δ_i has been applied. □

In figure 11.1 on the facing page, the delta set Δ_4 contains R_4, R_5, and R_6; the delta set Δ_6 contains R_6 alone; and the delta set Δ_1 contains all revisions except R_0.

Since we want to identify revisions by the excluded changes as well, we introduce a short-hand notation for the complement of a delta set:

Definition 11.3 (Nabla set) The complement of a delta set is called *nabla set,* written as $\nabla_i = {\sim}\Delta_i = [\delta_i{\uparrow}]$. It denotes the set of objects where the change δ_i has *not* been applied. □

In figure 11.1, the nabla set ∇_1 identifies R_0 alone, while ∇_5 contains all revisions except R_5.

To ensure that each revision R_i is associated with a delta set Δ_i and a nabla set ∇_i, we define Δ_0 and ∇_0 accordingly.

Definition 11.4 (Δ_0, ∇_0) We define $\Delta_0 = \top$ and $\nabla_0 = {\sim}\Delta_0 = \bot$. □

Intersections of delta and nabla sets are useful for identifying revisions.

Definition 11.5 (Revision features) For a given revision graph, the features of each revision R_k are

$$\begin{aligned} R_k = {} & (\Delta_1 \sqcap \cdots \sqcap \Delta_k) \sqcap (\nabla_{k+1} \sqcap \cdots \sqcap \nabla_m) \\ & \sqcap (\nabla_{m+1} \sqcap \cdots \sqcap \nabla_j) \sqcap (\nabla_{j+1} \sqcap \cdots \sqcap \nabla_n) \end{aligned} \tag{11.1}$$

where each Δ_i is a change leading up to a revision R_i:

- $R_1, \ldots, R_{k-1}$ are ancestors of R_k.
- $R_{k+1}, \ldots, R_m$ are direct descendants of R_k.
- $R_{m+1}, \ldots, R_j$ are indirect descendants of R_k—that is, descendants of the direct descendants $R_{k+1}, \ldots, R_m$.
- $R_{j+1}, \ldots, R_n$ are neither ancestors nor descendants of R_k. □

For the revision graph in figure 11.1, definition 11.5 yields the following revision features:

$$\begin{aligned}
R_0 &= \nabla_1 \sqcap \nabla_2 \sqcap \nabla_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_6 \\
R_1 &= \Delta_1 \sqcap \nabla_2 \sqcap \nabla_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_6 \\
R_2 &= \Delta_1 \sqcap \Delta_2 \sqcap \nabla_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_6 \\
R_3 &= \Delta_1 \sqcap \nabla_2 \sqcap \Delta_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_6 \\
R_4 &= \Delta_1 \sqcap \nabla_2 \sqcap \Delta_3 \sqcap \Delta_4 \sqcap \nabla_5 \sqcap \nabla_6 \\
R_5 &= \Delta_1 \sqcap \Delta_2 \sqcap \Delta_3 \sqcap \Delta_4 \sqcap \Delta_5 \sqcap \nabla_6 \\
R_6 &= \Delta_1 \sqcap \nabla_2 \sqcap \Delta_3 \sqcap \Delta_4 \sqcap \nabla_5 \sqcap \Delta_6
\end{aligned} \tag{11.2}$$

Figure 11.2 on the next page illustrates the relationship between delta sets and revisions for the revision graph given in figure 11.1. We see that each delta set contains exactly those revisions where the change has been applied; likewise, each revision is contained in the delta sets denoting its changes.

If we create a *revision set,* a version set containing revisions (a RCS or SCCS *repository,* for example), we can select individual revisions by stating the excluded or included changes. As an example, let us create a revision set R containing the revisions $R_0, \ldots, R_6$, as determined in (11.2). According to (9.1), R is determined as

$$R = R_0 \sqcup R_1 \sqcup R_2 \sqcup R_3 \sqcup R_4 \sqcup R_5 \sqcup R_6 \quad .$$

Arbitrary version sets can now be selected from R by specifying a conjunction of applied and non-applied changes, denoting paths in the revision graph. For instance, the selection $R \sqcap \Delta_4$ denotes R_4 and its descendants R_5 and R_6, as they all include the δ_4 change (formally, $R_4 \sqcup R_5 \sqcup R_6 \sqsubseteq R \sqcap \Delta_4$); since $R_0, \ldots, R_3$ do not include the δ_4 change ($R_0 \sqcap \cdots \sqcap R_3 \sqsubseteq \nabla_4$), they are excluded ($R_0 \sqcup R_1 \sqcup R_2 \sqcup R_3 \sqsubseteq R \sqcap \nabla_4$). The selection $R \sqcap [\Delta_2, \nabla_5]$ returns the single revision R_2, since R_5, the only other revision including the change δ_2, also includes the change δ_5 and is thus excluded by $\nabla_5 = [\delta_5\uparrow]$.

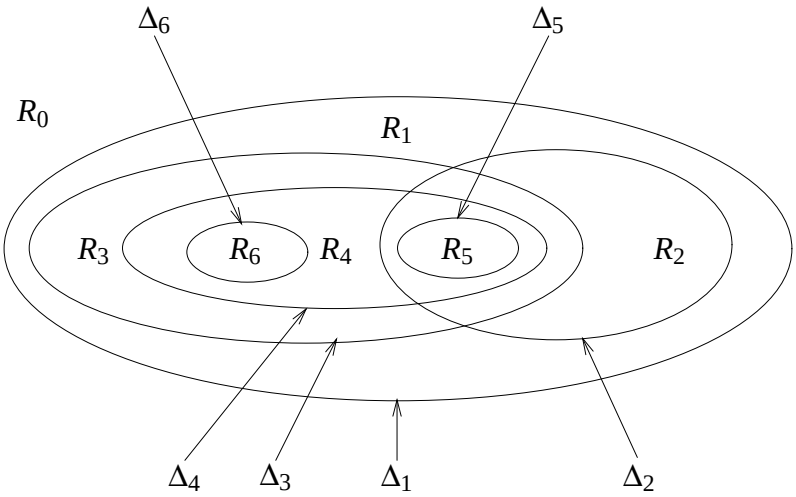

Figure 11.2: Changes and revisions

Generally, to select a single revision, it suffices to include the change leading up to that revision and to exclude the changes leading up to its immediate descendants.

We conclude with a few formal definitions regarding revision sets. In a revision set R, we call a revision R_j an *ancestor* of R_i if R contains no revision including the change δ_i while excluding δ_j—that is, $\Delta_i \sqsubseteq \Delta_j$ holds.

Definition 11.6 (Ancestor, Descendant) In a revision set R, consider a pair of revisions $R_i \sqsubseteq R$ and $R_j \sqsubseteq R$. If $i \neq j$ holds and $R \sqcap (\Delta_i \sqcap \nabla_j)$ is inconsistent, R_j is called *ancestor* of R_i and R_i is called *descendant* of R_j. □

An immediate ancestor is called *origin;* an immediate descendant is called *successor.*

Definition 11.7 (Origin, Successor) In a repository R, let $R_j, \ldots, R_k$ be the ancestors of a revision R_i. Each $R_l \sqsubseteq R_j \sqcup \cdots \sqcup R_k$ is called *immediate ancestor* or *origin* of R_i if there is no change $\delta_m \neq \delta_l$ such that R_m is a descendant of R_l and an ancestor of R_i; revision R_i is called *immediate descendant* or *successor* of R_l.

□

11.3 Revisions and Variants

The introduction of delta features allows us to distinguish revisions from variants formally. Basically, a revision is a version set that cannot be refined any further by specifying more delta features in a selection term. For instance, the feature term $R = [\mathit{object}{:}\mathit{foo}, \Delta_{47}]$ denotes a revision if $R = R \sqcap \Delta_i = R \sqcap \nabla_i$ holds for all $i \neq 47$.

Likewise, a variant is a version set that cannot be refined any further by specifying any more non-delta features in a selection term. For example, the term $V = [\mathit{object}{:}\mathit{bar}, \mathit{tested}{:}\mathit{true}]$ denotes a variant if $V = V \sqcap [f{:}\top] = V \sqcap [f\uparrow]$ holds for all features f such that $f \neq \mathit{object}$ and $f \neq \mathit{tested}$ and f is not a delta feature.

Neither variants nor revisions are necessarily singleton: A variant may still come in multiple singleton revisions, and that a revision may come in multiple singleton variants. If a version set can no more be refined, we have a singleton *version,* following definition 9.6 on page 92.

Here come the formal definitions, beginning with refinement. A term T refines a term S if $T \sqcap S$ is different from S and non-empty. Like cardinality, refinement can only be determined for some given interpretation.

Definition 11.8 (Refinement) A feature term T is said to *refine* a feature term S if $S \sqcap T \neq S$ and $S \sqcap T \neq \bot$ hold. □

If a version set cannot be refined by stating more delta features, we call it a revision.

Definition 11.9 (Revision set, Revision) A *revision set* is a version set R that is a subset of some delta or nabla set. A revision set S is called a *revision* if there is no revision set R such that R refines S. □

If a version set cannot be refined by stating more non-delta features, we call it a variant.

Definition 11.10 (Variant set, Variant) A *variant set* is a version set R that is not a revision set. A variant set S is called a *variant* if there is no variant set V such that V refines S. □

Note that a version V may be a revision as well as a variant:

- If V is distinguished from another version via a delta feature only, V was created by applying a change and is thus a revision.

- If V is distinguished from another version via a non-delta feature only, V is a variant.

- If V is distinguished via delta features as well as via other features, there was a change that affected other features as well; V is a revision as well as a variant.

In section 12.5, we will further investigate the relationships between delta features and other features.

11.4 Revision Constraints

In (11.2), we have seen that the terms R_i denoting the individual revisions may become quite large—each of $R_0, \ldots, R_n$ contains n primitives. If we represent the features of the revision set R in DNF, as stated in (11.2), R contains $(n+1) \times n = n^2 + n$ primitives, resulting in quadratic time behavior for any repository accesses.

In this section, we discuss an alternate representation for R, using an intersection of *revision constraints*, that is, implications between delta sets. Using revision constraints, the revision set R from (11.2) can be expressed as

$$\begin{aligned} R = {} & (\Delta_2 \to \Delta_1) \sqcap (\Delta_3 \to \Delta_1) \sqcap (\Delta_4 \to \Delta_3) \sqcap (\Delta_5 \to \Delta_2) \sqcap (\Delta_5 \to \Delta_4) \\ & \sqcap (\Delta_6 \to \Delta_4) \sqcap (\Delta_2 \sqcap \Delta_3 \to \Delta_5) \sqcap (\Delta_2 \sqcap \Delta_6 \to \bot) \quad , \end{aligned} \tag{11.3}$$

that is, one single implication for each edge in the revision lattice as well as one single implication for each integration. Not only does such a representation save much space, it also immediately reflects the structure of the revision graph. Besides that, the constraint representation is much easier to maintain when new revisions are added, since all we have to do is to intersect R with an additional constraint.

When selecting revisions from R, all revision features are created by applying revision constraints—every revision constraint in R is reduced to some delta or nabla set. As an example, consider the selection $R \sqcap \Delta_5$, which should return R_5, as defined in (11.2). Following the general scheme

$$(\Delta_i \to \Delta_j) \sqcap \Delta_i = (\nabla_i \sqcup \Delta_j) \sqcap \Delta_i = \Delta_i \sqcap \Delta_j \quad , \tag{11.4}$$

we begin with intersecting the constraints involving Δ_5 in (11.3) and obtain

$$\begin{aligned} (\Delta_5 \to \Delta_2) \sqcap \Delta_5 &= \Delta_2 \sqcap \Delta_5 \\ (\Delta_5 \to \Delta_4) \sqcap \Delta_5 &= \Delta_4 \sqcap \Delta_5 \quad , \end{aligned}$$

that is, $R \sqcap \Delta_5 \sqsubseteq \Delta_2$ and $R \sqcap \Delta_5 \sqsubseteq \Delta_4$ hold. Consequently, we can intersect the other constraints with Δ_2 or Δ_4 to eliminate alternatives:

$$(\Delta_2 \to \Delta_1) \sqcap \Delta_2 = \Delta_1 \sqcap \Delta_2$$
$$(\Delta_4 \to \Delta_3) \sqcap \Delta_4 = \Delta_3 \sqcap \Delta_4$$

and find that $R \sqcap \Delta_5 \sqsubseteq \Delta_1$ and $R \sqcap \Delta_5 \sqsubseteq \Delta_3$ hold. $R \sqcap \Delta_5 \sqsubseteq \nabla_6$ also holds:

$$(\Delta_2 \sqcap \Delta_6 \to \bot) \sqcap \Delta_2 = \nabla_6 \sqcap \Delta_2 \ .$$

The remaining constraint is trivially reduced to

$$(\Delta_2 \sqcap \Delta_3 \to \Delta_5) \sqcap \Delta_5 = \top \sqcap \Delta_5 \ .$$

We obtain $R \sqcap \Delta_5$ as

$$\begin{aligned} R \sqcap \Delta_5 &= (\Delta_1 \sqcap \Delta_1 \sqcap \Delta_3 \sqcap \Delta_2 \sqcap \Delta_4 \sqcap \Delta_4 \sqcap \top \sqcap \nabla_6) \sqcap \Delta_5 \\ &= \Delta_1 \sqcap \Delta_2 \sqcap \Delta_3 \sqcap \Delta_4 \sqcap \Delta_5 \sqcap \nabla_6 \\ &= R_5 \ . \end{aligned}$$

As another example, consider the selection $R \sqcap \nabla_1$, which should return R_0, as defined in (11.2). We now rely on a variant of (11.4), namely

$$(\Delta_i \to \Delta_j) \sqcap \nabla_j = (\nabla_i \sqcup \Delta_j) \sqcap \nabla_j = \nabla_i \sqcap \nabla_j \ , \tag{11.5}$$

in order to reduce revision constraints to revision features. Intersecting the first two constraints in (11.3) with ∇_1 yields

$$(\Delta_2 \to \Delta_1) \sqcap \nabla_1 = \nabla_2 \sqcap \nabla_1$$
$$(\Delta_3 \to \Delta_1) \sqcap \nabla_3 = \nabla_3 \sqcap \nabla_1 \ ,$$

that is, $R \sqcap \nabla_1$ is a subset of ∇_2 and ∇_3. Hence, $R \sqcap \nabla_1 = R \sqcap \nabla_1 \sqcap \nabla_3$ holds, and we can intersect the other constraints with ∇_3 to obtain further features:

$$(\Delta_4 \to \Delta_3) \sqcap \nabla_3 = \nabla_4 \sqcap \nabla_3$$
$$(\Delta_5 \to \Delta_4) \sqcap \nabla_4 = \nabla_5 \sqcap \nabla_4$$
$$(\Delta_6 \to \Delta_4) \sqcap \nabla_4 = \nabla_6 \sqcap \nabla_4$$

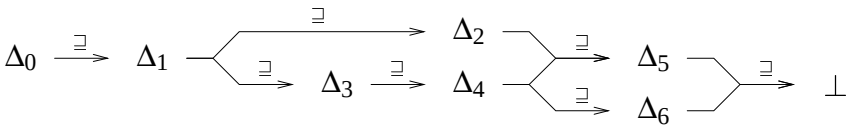

Figure 11.3: A revision graph as subsumption lattice

The last two constraints are easily reduced to $\top$:

$$(\Delta_2 \sqcap \Delta_3 \to \Delta_5) \sqcap \nabla_2 = \top \sqcap \nabla_2$$
$$(\Delta_2 \sqcap \Delta_6 \to \bot) \sqcap \nabla_2 = \top \sqcap \nabla_2$$

We obtain

$$\begin{aligned} R \sqcap \nabla_1 &= (\nabla_2 \sqcap \nabla_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_5 \sqcap \nabla_6 \sqcap \top \sqcap \top) \sqcap \nabla_1 \\ &= \nabla_1 \sqcap \nabla_2 \sqcap \nabla_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_6 \\ &= R_0 \ . \end{aligned}$$

11.5 Constraints and Lattices

How does one obtain these revision constraints? Revision constraints are deduced from the *revision lattice.* The revision lattice is the subsumption lattice obtained from the subsumption relation between delta sets. If $\Delta_i \sqsubseteq \Delta_j$ holds, requesting the change δ_i implies that δ_j be applied as well. Using (8.4) and (8.5), we have

$$\Delta_i \sqsubseteq \Delta_j \ \Leftrightarrow \ \Delta_i \sqcap \nabla_j = \bot \ .$$

Consequently, if $\Delta_i \sqsubseteq \Delta_j$ holds, there is no revision such that δ_i is applied, but δ_j is not.

These subsumption relations between delta sets can be visualized in a graph. The revision lattice for our example is shown in figure 11.3. In the revision lattice, the supremum of any two revision sets Δ_i and Δ_j is the set of ancestor revisions $\Delta_i \sqcup \Delta_j$; their infimum is the (possibly empty) set of integrated revisions $\Delta_i \sqcap \Delta_j$.

We see that the revision lattice is isomorph to the revision graph, as shown in figure 11.3; the only difference is that we have added a $\bot$ element to complete the lattice. The structure similarity does not surprise—the revision graph is structured by change implications $\delta_i \to \delta_j$, which correspond to the subsumption relations $\Delta_i \sqsupseteq \Delta_j$ in the revision lattice.

Using the revision lattice, we can compute *revision constraints*.

Definition 11.11 (Revision constraint) For any two delta sets Δ_i and Δ_j in a revision lattice, let $\Delta_{\overline{i,j}} \sqsupseteq (\Delta_i \sqcup \Delta_j)$ be their lowest common ancestor (supremum), and let $\Delta_{\underline{i,j}} \sqsubseteq (\Delta_i \sqcap \Delta_j)$ be their (possibly empty) highest common descendant (infimum) in the revision lattice. The *revision constraint* $C_{i,j}$ is defined as

$$C_{i,j} = (\Delta_i \sqcup \Delta_j \to \Delta_{\overline{i,j}}) \sqcap (\Delta_i \sqcap \Delta_j \to \Delta_{\underline{i,j}}) \tag{11.6}$$

□

In the common case of changes that imply each other (that is, $\Delta_i \sqsupseteq \Delta_j$), revision constraints take a much simpler form:

Corollary 11.12 If $\Delta_i \sqsupseteq \Delta_j$ holds, the revision constraint $C_{i,j}$ is

$$C_{i,j} = \Delta_j \to \Delta_i \tag{11.7}$$

PROOF. We have $\Delta_{\overline{i,j}} = \Delta_i$ and $\Delta_{\underline{i,j}} = \Delta_j$. Consequently,

$$\begin{aligned} C_{i,j} &= (\Delta_i \sqcup \Delta_j \to \Delta_{\overline{i,j}}) \sqcap (\Delta_i \sqcap \Delta_j \to \Delta_{\underline{i,j}}) \\ &= \big((\nabla_i \sqcap \nabla_j) \sqcup \Delta_i\big) \sqcap \big((\nabla_i \sqcup \nabla_j) \sqcup \Delta_j\big) \\ &= (\nabla_j \sqcup \Delta_i) \sqcap \top \\ &= \Delta_j \to \Delta_i \ , \end{aligned}$$

which was to be shown. □

Constraints involving Δ_0 are trivial.

Corollary 11.13 For all j, $C_{j,0} = C_{0,j} = \top$ holds.
PROOF. Because of (11.7), $C_{j,0} = C_{0,j} = \Delta_j \to \Delta_0 = \Delta_j \to \top = \top$. □

As an example of revision constraints, consider the revision graph in figure 11.3 on the page before, where we have

$$C_{2,4} = (\Delta_2 \sqcup \Delta_4 \to \Delta_1) \sqcap (\Delta_2 \sqcap \Delta_4 \to \Delta_5)$$

as well as

$$C_{2,6} = (\Delta_2 \sqcup \Delta_6 \to \Delta_1) \sqcap (\Delta_2 \sqcap \Delta_6 \to \bot) \ .$$

The conjunction of all revision constraints in a revision graph is called *constraint representation* of the revision graph.

Definition 11.14 (Constraint representation) Given a revision lattice with delta sets $\Delta_1, \ldots, \Delta_n$, the *constraint representation* C of a revision graph is defined as the conjunction

$$C = \prod_{\substack{1 \leq i \leq n \\ 1 < j < i}} C_{i,j} \quad , \tag{11.8}$$

where each revision constraint $C_{i,j}$ is defined according to (11.6). □

It now turns out that this conjunction of constraints, as defined in (11.8), is equivalent to the union of revisions, as defined through (9.1), and thus constitutes a suitable representation for revision graphs, as demonstrated in section 11.4. The proof is given in section 11.6.

11.6 An Equivalence Result

In this section, we show that the conjunction of constraints C, as defined in (11.8), is equivalent to the union of revisions R, as defined through (9.1). The road map to the proof is as follows:

- In lemma 11.15 on the following page, we show that a selection in C stating the included and excluded changes actually contains the desired revisions.
- In lemma 11.16 on page 125, we show that this selection does not return any other revisions.
- Proposition 11.17 on page 127 combines lemmas 11.15 and 11.16 and states that we can select a single revision R_k from C by specifying the included and excluded changes.
- Lemma 11.18 on page 127 applies proposition 11.17 to revision sets and shows that we can select a revision and all its descendants from C.
- Finally, theorem 11.19 on page 128 applies lemma 11.18 to R_0, showing that $R = C$ holds.

We begin with some selection results. First, we show that we can select a revision R_k from C by stating the change leading up to R_k and excluding the changes leading up to its descendants. Lemma 11.18 on page 127 states that R_k is at least a subset of such a selection.

Lemma 11.15 Let C be a constraint representation, as defined in (11.8), and $R = R_0 \sqcup \cdots \sqcup R_n$ be a union of revisions. For all revisions $R_k \sqsubseteq R_n$ with $0 \leq k \leq n$, we have

$$R_k \sqsubseteq C \sqcap \Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l \ , \tag{11.9}$$

where $R_{k+1}, \ldots, R_m$ are the immediate descendants of R_k.

PROOF. According to (8.5), (11.9) holds if and only if U, defined as

$$U = R_k \sqcap {\sim}(C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m)$$

is inconsistent. We apply de Morgan's laws, obtaining

$$U = R_k \sqcap \big({\sim}C \sqcup {\sim}(\Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m)\big) \ .$$

Because of (11.1), $R_k \sqsubseteq \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m$ holds. Hence, we have

$$U = R_k \sqcap {\sim}C$$

and we see that (11.9) holds if and only if $R_k \sqsubseteq C$ holds. We replace C by its definition (11.8) and obtain

$$\begin{aligned} U &= R_k \sqcap {\sim}(C_{1,2} \sqcap \cdots \sqcap C_{n,n-1}) \\ &= R_k \sqcap ({\sim}C_{1,2} \sqcup \cdots \sqcup {\sim}C_{n,n-1}) \\ &= (R_k \sqcap {\sim}C_{1,2}) \sqcup \cdots \sqcup (R_k \sqcap {\sim}C_{n,n-1}) \ , \end{aligned}$$

that is, U is inconsistent if every $R_k \sqcap {\sim}C_i j$ is inconsistent. For each pair i, j, using (11.6), we evaluate $R_k \sqcap {\sim}C_{i,j}$ to

$$\begin{aligned} R_k \sqcap {\sim}C_{i,j} &= R_k \sqcap {\sim}\big((\Delta_i \sqcup \Delta_j \to \Delta_{\overline{i,j}}) \sqcap (\Delta_i \sqcap \Delta_j \to \Delta_{\underline{i,j}})\big) \\ &= R_k \sqcap \big(\big((\Delta_i \sqcup \Delta_j) \sqcap \nabla_{\overline{i,j}}\big) \sqcup \big(\Delta_i \sqcap \Delta_j \sqcap \nabla_{\underline{i,j}}\big)\big) \\ &= \big(R_k \sqcap (\Delta_i \sqcup \Delta_j) \sqcap \nabla_{\overline{i,j}}\big) \sqcup \big(R_k \sqcap \Delta_i \sqcap \Delta_j \sqcap \nabla_{\underline{i,j}}\big) \end{aligned}$$

Let $U' = (\Delta_i \sqcup \Delta_j) \sqcap \nabla_{\overline{i,j}}$ and $U'' = (\Delta_i \sqcap \Delta_j) \sqcap \nabla_{\underline{i,j}}$ such that $R_k \sqcap {\sim}C_{i,j} = U' \sqcup U''$ holds. We distinguish four cases:

1. $R_k \sqsubseteq \Delta_i \sqcap \Delta_j$. Due to (11.1), $R_k \sqsubseteq \Delta_{\overline{i,j}}$ must hold as well, which implies $R_k \sqcap U' \sqsubseteq \Delta_{\overline{i,j}} \sqcap \nabla_{\overline{i,j}} = \bot$. As R_k is the integration of R_i and R_j or a descendant, we have $R_k \sqsubseteq \Delta_{\underline{i,j}}$ and thus $R_k \sqcap U'' \sqsubseteq \Delta_{\underline{i,j}} \sqcap \nabla_{\underline{i,j}} = \bot$.

2. $R_k \sqsubseteq \nabla_i \sqcap \Delta_j$. Revision R_k inherits the features of R_j and all of its ancestors. As in case 1, $R_k \sqsubseteq \Delta_{\overline{i,j}}$ must hold as well; $R_k \sqcap U' = \bot$ holds. Since $R_k \sqsubseteq \nabla_i$, $R_k \sqcap U'' \sqsubseteq \nabla_i \sqcap \Delta_i = \bot$ holds.

3. $R_k \sqsubseteq \Delta_i \sqcap \nabla_j$. Same as case 2, above.

4. $R_k \sqsubseteq \nabla_i \sqcap \nabla_j$. We have $R_k \sqcap U' \sqsubseteq (\nabla_i \sqcap \nabla_j) \sqcap (\Delta_i \sqcup \Delta_j) \sqsubseteq \bot$; as in case 2, $R_k \sqcap U'' = \bot$ holds.

In all four cases, $R_k \sqcap (U' \sqcup U'') = R_k \sqcap {\sim}C_{i,j} = \bot$ holds for each pair i, j, resulting in $U = R_k \sqcap {\sim}C = \bot$. Since U is inconsistent, (11.9) holds, which was to be shown. □

Lemma 11.16 states that R_k is also a superset of the same selection.

Lemma 11.16 Let C and R be defined as in lemma 11.15 on the preceding page. For all revisions $R_k \sqsubseteq R_n$ with $0 \leq k \leq n$, we have

$$R_k \sqsupseteq C \sqcap \Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l \quad , \tag{11.10}$$

where $R_{k+1}, \ldots, R_m$ are the immediate descendants of R_k.
PROOF. As stated in (8.5), (11.10) holds if and only if U, defined as

$$U = {\sim}R_k \sqcap C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m$$

is inconsistent. R_k takes the general form

$$\begin{aligned} R_k = {} & (\Delta_1 \sqcap \cdots \sqcap \Delta_k) \sqcap (\nabla_{k+1} \sqcap \cdots \sqcap \nabla_m) \\ & \sqcap (\nabla_{m+1} \sqcap \cdots \sqcap \nabla_j) \sqcap (\nabla_{j+1} \sqcap \cdots \sqcap \nabla_n) \quad , \end{aligned}$$

where all Δ_i and ∇_i are defined according to definition 11.5 on page 115. The inverted form is

$$\begin{aligned} {\sim}R_k = {} & (\nabla_1 \sqcup \cdots \sqcup \nabla_k) \sqcup (\Delta_{k+1} \sqcup \cdots \sqcup \Delta_m) \\ & \sqcup (\Delta_{m+1} \sqcup \cdots \sqcup \Delta_j) \sqcup (\Delta_{j+1} \sqcup \cdots \sqcup \Delta_n) \quad , \end{aligned}$$

such that U evaluates to

$$\begin{aligned} U = {} & \big((\nabla_1 \sqcup \cdots \sqcup \nabla_k) \sqcup (\Delta_{k+1} \sqcup \cdots \sqcup \Delta_m) \\ & \quad \sqcup (\Delta_{m+1} \sqcup \cdots \sqcup \Delta_j) \sqcup (\Delta_{j+1} \sqcup \cdots \sqcup \Delta_n) \big) \\ & \sqcap C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \quad . \end{aligned}$$

We shall now show that none of the alternatives in $\sim R_k$ can be satisfied. We begin with the alternatives $\nabla_k \sqcup \Delta_{k+1} \sqcup \cdots \sqcup \Delta_m$. These are explicitly excluded by the selection term $\Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m$ and we obtain

$$\begin{aligned} U = \big((\nabla_1 \sqcup \cdots \sqcup \nabla_{k-1}) \sqcup (\Delta_{m+1} \sqcup \cdots \sqcup \Delta_j) \sqcup (\Delta_{j+1} \sqcup \cdots \sqcup \Delta_n) \big) \\ \sqcap C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \ . \end{aligned}$$

We continue with eliminating the ancestor alternatives. Since $R_1, \ldots, R_{k-1}$ are ancestors of R_k, we have $\Delta_{\overline{i,k}} = \Delta_i$ for $1 \leq i \leq k-1$. Consequently, $C \sqsubseteq C_{i,k} = \Delta_k \to \Delta_i$ holds and $C \sqcap \Delta_k \sqsubseteq \Delta_i$ for $1 \leq i \leq k-1$. Hence, the alternatives $\nabla_1 \sqcup \cdots \sqcup \nabla_{k-1}$ cannot be satisfied and may be omitted, resulting in

$$\begin{aligned} U = \big((\Delta_{m+1} \sqcup \cdots \sqcup \Delta_j) \sqcup (\Delta_{j+1} \sqcup \cdots \sqcup \Delta_n) \big) \\ \sqcap C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \ . \end{aligned}$$

We continue with the descendant alternatives. The same applies to the indirect descendants of R_k. Since $R_{k+1}, \ldots, R_m$ are direct descendants of R_k, we have $\Delta_{\overline{i,l}} = \nabla_i$ for $k+1 \leq i \leq m$ and $m+1 \leq l \leq j$. As above, $C \sqsubseteq C_{i,l} = \Delta_l \to \Delta_i = \nabla_i \to \nabla_l$ holds and thus $C \sqcap \nabla_i \sqsubseteq \nabla_l$ for $k+1 \leq i \leq m$ and $m+1 \leq l \leq j$. This removes the alternatives $\Delta_{m+1} \sqcup \cdots \sqcup \Delta_j$, resulting in

$$U = (\Delta_{j+1} \sqcup \cdots \sqcup \Delta_n) \sqcap C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \ .$$

We close with the remaining alternatives. The revisions $R_{j+1}, \ldots, R_n$ are neither ancestors nor descendants of R_k. For each R_i with $j+1 \leq i \leq n$, let us check if R_i and R_k integrate:

- If R_i and R_k integrate in some revision R_l, we have $\Delta_{\underline{i,k}} = \Delta_l$. Then, $C \sqsubseteq C_{i,k} \sqsubseteq (\Delta_i \sqcap \Delta_k \to \Delta_l)$ holds, and we have $C \sqcap \Delta_k = \Delta_i \to \Delta_l = \nabla_l \to \nabla_i$. But R_l is a descendant of Δ_k. As shown above, this implies that $C \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \sqsubseteq \nabla_l$ and we have $C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \sqsubseteq \nabla_i$.
- If R_i and R_k do not integrate, we have $\Delta_{\underline{i,k}} = \bot$. In this case, $C \sqsubseteq C_{i,k} \sqsubseteq (\Delta_i \sqcap \Delta_k \to \bot) = \nabla_i \sqcup \nabla_k$ holds, and we have $C \sqcap \Delta_k \sqsubseteq \nabla_i$.

In either case, $C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \sqsubseteq \nabla_i$ holds for all $j+1 \leq i \leq n$—and this eliminates the remaining Δ_i alternatives.

$$\begin{aligned} U &= \bot \sqcap C \sqcap \Delta_k \sqcap \nabla_{k+1} \sqcap \cdots \sqcap \nabla_m \\ &= \bot \ . \end{aligned}$$

Hence, U is inconsistent and (11.10) holds, which was to be shown. □

Proposition 11.17 combines lemma 11.15 and lemma 11.16. It states that we can select a revision R_k from C by including the change leading up to R_k and excluding the changes leading up to its descendants.

Proposition 11.17 Let C and R be defined as in lemma 11.15 on page 124. For all revisions $R_k \sqsubseteq R_n$ with $0 \leq k \leq n$, we have

$$R_k = C \sqcap \Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l \quad , \tag{11.11}$$

where $R_{k+1}, \ldots, R_m$ are the immediate descendants of R_k.
PROOF. Follows from (11.9) and (11.10) via (8.6). □

Proposition 11.17 implies that the selection $C \sqcap \Delta_i$ returns R_i and all its descendants.

Lemma 11.18 Let C and R be defined as in lemma 11.15 on page 124. For all revisions $R_k \sqsubseteq R_n$ with $1 \leq k \leq n$, we have

$$C \sqcap \Delta_k = R_k \sqcup R_{k+1} \sqcup \cdots \sqcup R_n \quad , \tag{11.12}$$

where $R_{k+1}, \ldots, R_n$ are the descendants of R_k.
PROOF. We prove (11.12) via structural induction. If R_k has no descendants, (11.12) holds because of (11.11). Otherwise, let $R_{k+1}, \ldots, R_m$ be the direct descendants of R_k and let us assume that (11.12) holds for $R_{k+1}, \ldots, R_m$. Starting with (11.11), we obtain

$$R_k = C \sqcap \Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l$$

This can be expanded to

$$R_k \sqcup \left(\bigsqcup_{k+1 \leq l \leq m} C \sqcap \Delta_l \right) = \left(C \sqcap \Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l \right) \sqcup \left(\bigsqcup_{k+1 \leq l \leq m} C \sqcap \Delta_l \right)$$

By applying (11.12) for all $\Delta_{k+1}, \ldots, \Delta_m$ on the left-hand side, we get

$$\begin{aligned} R_k \sqcup \cdots \sqcup R_n &= \left(C \sqcap \Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l \right) \sqcup \left(\bigsqcup_{k+1 \leq l \leq m} C \sqcap \Delta_l \right) \\ &= C \sqcap \left(\left(\Delta_k \sqcap \bigsqcap_{k+1 \leq l \leq m} \nabla_l \right) \sqcup \left(\bigsqcup_{k+1 \leq l \leq m} \Delta_l \right) \right) \quad . \end{aligned}$$

Applying the absorption rule yields

$$R_k \sqcup \cdots \sqcup R_n = C \sqcap \left(\Delta_k \sqcup \bigsqcup_{k+1 \leq l \leq m} \Delta_l \right) \quad .$$

Each R_l in $k+1 \leq l \leq m$ is a descendant of R_k. Because of (11.7), we have $C \sqsubseteq \Delta_l \rightarrow \Delta_k$ and thus $C \sqcap (\Delta_k \sqcup \Delta_l) = C \sqcap \Delta_k$, which results in

$$R_k \sqcup \cdots \sqcup R_n = C \sqcap \Delta_k \quad .$$

We have shown that (11.12) holds for any R_k without descendants, and for any R_k if it holds for its descendants. Hence, (11.12) holds for all R_k. □

Lemma 11.18 on the page before also states that the the two revision set representations are equivalent.

Theorem 11.19 A revision set R can be represented as union of all revisions R_k, as defined in (11.1), or as intersection of revision constraints $C_{i,j}$, as defined in (11.6). Both representations are equivalent:

$$R = \bigsqcup_{0 \leq k \leq n} R_k = \bigsqcap_{\substack{1 \leq i \leq n \\ 1 < j < i}} C_{i,j} \tag{11.13}$$

PROOF. Follows from (11.12): $C = C \sqcap \Delta_0 = R_0 \sqcup R_1 \sqcup \cdots \sqcup R_n = R$. □

11.7 Discussion

In section 6.3, the integration of change-oriented models and version-oriented models turned out as a major SCM research issue. We have seen that feature logic is descriptive enough to model ordinary revision histories, as in the version-oriented models; arbitrary change combinations, as in the change-oriented models are still possible. By submitting changes to revision constraints, the version set model captures the entire range of temporal versioning—from the rigid revision graphs of versions-oriented models to the loosely structured change space of change-oriented models.

Revision constraints are easily constructed from the revision graph. Their intersection is equivalent to the union of all revisions. This equivalence, as stated in theorem 11.19, again shows the expressive power of feature logic: besides simple constraints such as stating unique feature values, we can express implications

between features that are sufficiently complex to model entire revision graphs. In chapter 12, we further discuss revision constraints, especially their maintenance in repositories and their integration with variants and configurations.

There was general agreement
that end-users of applications were not interested in a version model
but were interested in the changes made in previous versions
and, at a more abstract level,
in the features offered by different versions of the system.

— IAN SOMMERVILLE, Sixth International Workshop
on Software Configuration Management

Chapter 12

Constraints and Repositories

Having considered the static aspects of revision graphs, we now turn to some dynamic *aspects, answering questions like: How does a repository representation change when a new revision is added? How does it change, should an old revision be removed? As illustrated in this chapter, maintaining the revision constraints is no more complicated than in "classical" SCM systems. Furthermore, we discuss the integration of revision constraints with variants and configurations.*

12.1 Creating Revisions with a Single Origin

We begin with the problem of adding a single revision to a repository. According to (9.1) and (11.1), adding a new revision R_i to a repository R results in tagging the old revisions with ∇_i and adding the new revision R_i, resulting in a new repository R':

$$R' = (R \sqcap \nabla_i) \sqcup R_i \tag{12.1}$$

But this straight-forward approach again leads us to an inefficient representation of R', since the constraint form of R is lost and the term R_i can be quite long. The constraint structure of R can be conserved, however, if we know the revisions $R_j, \ldots, R_k$ from which R_i originates; or, in other words, which changes $\delta_j, \ldots, \delta_k$ are implied by δ_i.

As a simple example, consider adding a new revision R_7 to the revision graph shown in figure 11.1 on page 114. In our setting, revision R_7 is a descendant of R_6; the resulting new revision graph is shown in figure 12.1 on the next page.

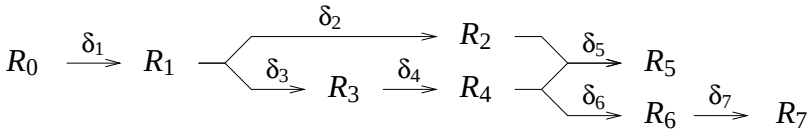

Figure 12.1: Adding a revision R_7 with a single origin R_6

Starting from (12.1), we have

$$\begin{aligned}
R' &= (R \sqcap \nabla_7) \sqcup R_7 \\
&= (R \sqcap \nabla_7) \sqcup (R_6 \sqcap \Delta_7) \\
&= (R \sqcap \nabla_7) \sqcup (R \sqcap \nabla_2 \sqcap \Delta_6 \sqcap \Delta_7) \\
&= R \sqcap \big(\nabla_7 \sqcup (\nabla_2 \sqcap \Delta_6 \sqcap \Delta_7)\big) \\
&= R \sqcap (\nabla_7 \sqcup \nabla_2) \sqcap (\nabla_7 \sqcup \Delta_6) \sqcap (\nabla_7 \sqcup \Delta_7) \\
&= R \sqcap (\nabla_7 \sqcup \nabla_2) \sqcap (\nabla_7 \sqcup \Delta_6) \sqcap \top \\
&= R \sqcap (\nabla_7 \sqcup \Delta_6) \sqcap (\nabla_7 \sqcup \nabla_2) \\
&= R \sqcap (\Delta_7 \to \Delta_6) \sqcap (\Delta_2 \sqcap \Delta_7 \to \bot) \ ,
\end{aligned}$$

which is exactly the constraint form we should expect from theorem 11.19 on page 128.

We conclude that an SCM system adding a new revision R_i with one single origin R_j need do no more than to add one additional constraint $(\Delta_i \to \Delta_j)$ to the representation of the repository term. If the integration of R_i and some other revision does not exist, the appropriate constraints must be updated as well—as in our example, where the old constraint $(\Delta_2 \sqcap \Delta_6 \to \bot)$ is subsumed by the new constraint $(\Delta_2 \sqcap \Delta_7 \to \bot)$.

12.2 Adding Revisions with Multiple Origins

Adding a revision R_i with multiple origins $R_j, \ldots, R_k$ is more complicated, as it results in the *removal* of constraints, namely those constraints which previously prohibited the integration of the changes $\delta_j, \ldots, \delta_k$.

As an example, assume revision R_7 were based on R_5 as well as R_6, as illustrated in figure 12.2 on the next page. In this case, the term R' can no more contain the constraint $(\Delta_2 \sqcap \Delta_6 \to \bot)$. Let R'' hold all other constraints from R such that $R = R'' \sqcap (\Delta_2 \sqcap \Delta_6 \to \bot)$. Adding revision R_7 to the repository R would

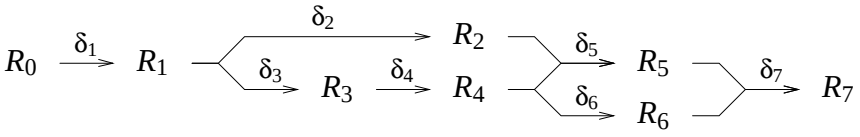

Figure 12.2: Adding a revision R_7 with two origins R_5 and R_6

then result in

$$\begin{aligned}
R' &= (R \sqcap \nabla_7) \sqcup R_7 \\
&= (R \sqcap \nabla_7) \sqcup (R'' \sqcap \Delta_5 \sqcap \Delta_6 \sqcap \Delta_7) \\
&= (R'' \sqcap (\nabla_2 \sqcup \nabla_6) \sqcap \nabla_7) \sqcup (R'' \sqcap \Delta_5 \sqcap \Delta_6 \sqcap \Delta_7) \\
&= R'' \sqcap \big((\nabla_2 \sqcup \nabla_6) \sqcap \nabla_7\big) \sqcup (\Delta_5 \sqcap \Delta_6 \sqcap \Delta_7)\big) \\
&= R'' \sqcap \big((\nabla_2 \sqcup \nabla_6) \sqcup (\Delta_5 \sqcap \Delta_6)\big) \sqcap \big((\nabla_2 \sqcup \nabla_6) \sqcup \Delta_7\big) \sqcap \big(\nabla_7 \sqcup (\Delta_5 \sqcap \Delta_6)\big) \\
&= R'' \sqcap (\nabla_2 \sqcup \nabla_6 \sqcup \Delta_7) \sqcap \big(\nabla_7 \sqcup (\Delta_5 \sqcap \Delta_6)\big) \\
&= R'' \sqcap (\Delta_2 \sqcap \Delta_6 \to \Delta_7) \sqcap (\Delta_7 \to \Delta_5 \sqcap \Delta_6) \\
&= R'' \sqcap (\Delta_2 \sqcap \Delta_6 \to \Delta_7) \sqcap (\Delta_7 \to \Delta_5) \sqcap (\Delta_7 \to \Delta_6) \quad ,
\end{aligned}$$

that is, the old constraint $(\Delta_2 \sqcap \Delta_6 \to \bot)$ is replaced by $(\Delta_2 \sqcap \Delta_6 \to \Delta_7)$ and two new constraints $(\Delta_7 \to \Delta_5)$ and $(\Delta_7 \to \Delta_6)$ are added.

We deduce a general scheme for the incremental maintenance of revision constraints:

Proposition 12.1 (Incremental maintenance of revision constraints)
The general scheme for adding a revision R_i to a repository R is:

1. For each origin R_j, add a constraint $C_{i,j} = \Delta_j \to \Delta_i$ to R.
2. For each pair j,k of ancestor revisions R_j, R_k, replace the old constraint $\Delta_j \sqcap \Delta_k \to \bot$ in R by $\Delta_j \sqcap \Delta_k \to \Delta_i$.
3. For each ancestor R_j and any non-integrating revision R_m, replace the constraint $\Delta_j \sqcap \Delta_m \to \bot$ in R by $\Delta_i \sqcap \Delta_m \to \bot$.

PROOF. *Added constraints*: The new constraints added are an immediate consequence of theorem 11.19 on page 128: step 1 adds the constraints for R_i and its origins, step 2 adds constraints for R_i as new descendant, and step 3 adds constraints for R_i and non-integrating revisions. All remaining revisions R_k are non-origin ancestors of R_i; the constraints $C_{i,k}$ are subsumed by $C_{i,j}$ added in step 1 and the unchanged constraint $C_{j,k}$.

Removed constraints: The constraints removed in step 2 are no more adequate, as the descendant of R_j and R_k now exists as R_i. The constraints removed in step 3 are subsumed by their replacement constraint in conjunction with the implications added in step 1. □

We see that maintaining the revision constraints is no more difficult than maintaining a revision graph in "classical" SCM systems.

12.3 Removing Revisions

Although removal of revisions is seldom desirable in an SCM context, it constitutes another example on the usage of revision constraints. The straight-forward approach to removing a revision R_i from a repository R is to intersect R with the complement of R_i, resulting in a new repository R':

$$R' = R \sqcap \sim R_i \tag{12.2}$$

From (12.2), we can immediately deduce an appropriate constraint representation. Remember that R_i is selected from R by specifying the change Δ_i and excluding all later changes $\Delta_j, \ldots, \Delta_k$ leading up to the immediate descendants of R_i, namely $R_j, \ldots, R_k$. Hence, we can transform (12.2) to:

$$\begin{aligned} R' &= R \sqcap \sim (R \sqcap \Delta_i \sqcap \nabla_j \sqcap \cdots \sqcap \nabla_k) \\ &= R \sqcap (\sim R \sqcup \nabla_i \sqcup \Delta_j \sqcup \cdots \sqcup \Delta_k) \\ &= R \sqcap (\Delta_i \to \Delta_j \sqcup \cdots \sqcup \Delta_k) \ , \end{aligned}$$

that is, we simply add a new constraint stating that, whenever the change Δ_i is to be included, one of the later changes $\Delta_j, \ldots, \Delta_k$ is to be included as well.

As an example, reconsider figure 11.1 on page 114. Removing revision R_3 from the repository R would result in a new repository R' with an additional constraint

$$R' = R \sqcap (\Delta_3 \to \Delta_4) \ ,$$

ensuring that, whenever the δ_3 change is requested, the δ_4 change is included as well—and vice versa, as the old constraint $(\Delta_4 \to \Delta_3)$ is still part of R.

12.4 Orthogonal Changes

So far, we have only considered "classical" version graphs, where changes imply each other more or less rigidly. In the Change-Oriented Model, this setting is

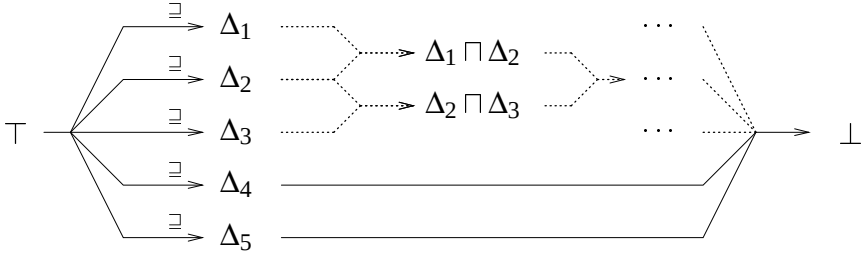

Figure 12.3: Orthogonal changes

different: arbitrary combinations of changes are allowed, unless they result in a conflict. We call these changes *orthogonal,* since they are independent from each other.

The versions resulting from the possible change combinations cannot all be depicted, due to their large number. In figure 12.3, we see five changes $\delta_1, \ldots, \delta_5$, each resulting in a set $\Delta_1, \ldots, \Delta_5$ where this change has been applied. All versions resulting from change combinations are identified by an enumeration of included and excluded changes. The only restriction in our example is that the changes δ_4 and δ_5 cannot be integrated; hence, Δ_4 and Δ_5 are not orthogonal, but disjoint.

How is such a version set represented? Following (11.6), each constraint $C_{i,j}$ with $1 \leq i \leq 5$ and $i < j < 5$ is

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

but since $\Delta_{\overline{i,j}} = \Delta_i \sqcup \Delta_j$ and $\Delta_{\underline{i,j}} = \Delta_i \sqcap \Delta_j$ holds for all i, j in R, we have

$$C_{i,j} = \top$$

except for $C_{4,5}$, which is

$$\begin{aligned} C_{4,5} &= (\Delta_4 \sqcap \Delta_5 \to \bot) \\ &= \nabla_4 \sqcup \nabla_5 \end{aligned}$$

Hence, $R = C_{4,5} = \nabla_4 \sqcup \nabla_5$ is the only constraint required to represent the set of all versions as shown in figure 12.3.

Obviously, a repository dealing with orthogonal changes would use dynamic version creation, as discussed in section 9.4. The repository would create versions

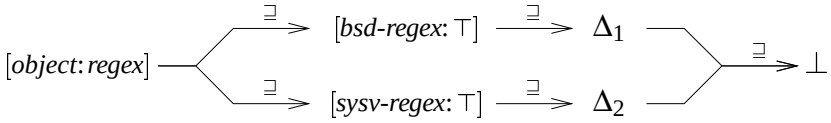

Figure 12.4: Combining delta features and variant features

as required by integrating the requested changes and applying them to a baseline; this is also the way RCS repositories are organized internally.

We see that revision constraints can be used to model both the safe, but rigid version-oriented models as well as the flexible, but unsafe change-oriented models. The higher the number of specified constraints, the lower the number of remaining change combinations—from the flexibility of the Change-Oriented Model with virtually no constraints to the rigidness of the version-oriented models with a small set of versions easily enumerated and tested. In our model, both models are just two extremes in a wide range between safety and flexibility.

12.5 Changes and Other Features

So far, our examples have only covered delta features, modeling historical versioning through changes. How would other features modeling relationships, variance, or workspaces be integrated? The answer is simple: they are used just like delta features, with implications expressing constraints—but not only implications between changes, but between values of arbitrary versioning dimensions.

As an example, consider the GNU REGEX library used for compiling and searching regular expressions. The initial versions of GNU REGEX came with a BSD UNIX interface, while the later versions came with a POSIX UNIX interface. Unfortunately, there is no version supporting both interfaces, and both are maintained individually, as illustrated in figure 12.4—the change δ_1 applies to the BSD version $bsd = [object: regex, bsd\text{-}regex: \top]$ only, while the change δ_2 applies to the POSIX version $posix = [object: regex, posix\text{-}regex: \top]$ only.

Obviously, Δ_1 and Δ_2 are disjoint, as are their respective supersets *bsd-regex* and *posix-regex*. According to theorem 11.19 on page 128, this disjointness can be expressed through the constraints C with

$$\begin{aligned} C &= (\Delta_1 \to bsd) \sqcap (\Delta_2 \to posix) \sqcap (bsd \sqcap posix \to \bot) \\ &= [object: regex] \sqcap \big(\Delta_1 \to [bsd\text{-}regex: \top]\big) \sqcap \big(\Delta_2 \to [posix\text{-}regex: \top]\big) \\ &\qquad \sqcap \big([bsd\text{-}regex\uparrow] \sqcup [posix\text{-}regex\uparrow]\big) \\ &= [object: regex, bsd\text{-}regex: \top, posix\text{-}regex\uparrow, \nabla_1, \nabla_2] \end{aligned}$$

$$\sqcup\,[\mathit{object}\colon \mathit{regex}, \mathit{bsd\text{-}regex}\colon \top, \mathit{posix\text{-}regex}\uparrow, \Delta_1, \nabla_2]$$
$$\sqcup\,[\mathit{object}\colon \mathit{regex}, \mathit{bsd\text{-}regex}\uparrow, \mathit{posix\text{-}regex}\colon \top, \nabla_1, \nabla_2]$$
$$\sqcup\,[\mathit{object}\colon \mathit{regex}, \mathit{bsd\text{-}regex}\uparrow, \mathit{posix\text{-}regex}\colon \top, \nabla_1, \Delta_2] \quad ,$$

that is, $[\mathit{object}\colon \mathit{regex}]$ comes in four variants, depending on whether the BSD or POSIX interface is chosen and whether the respective change has been applied or not.

We now create a REGEX version supporting both the POSIX and the BSD interface. To do so, we apply a change δ_3 to both *bsd-regex* and *posix-regex*, integrating both versions. The resulting subsumption lattice is identical to figure 12.4; only $\bot$ is replaced by Δ_3. The version set itself is described as

$$C = (\Delta_1 \to \mathit{bsd}) \sqcap (\Delta_2 \to \mathit{posix}) \sqcap (\mathit{bsd} \sqcap \mathit{posix} \to \Delta_3)$$
$$\sqcap\,(\Delta_3 \to \mathit{bsd}) \sqcap (\Delta_3 \to \mathit{posix}) \quad .$$

12.6 Changes and Configurations

Just as ordinary variant features can be used instead of delta features in subsumption lattices, delta features can be used instead of ordinary variant features to express the features of configurations—notably, ambiguity in configurations.

As an example, re-consider the discussion on configurations and ambiguity in section 10.6, where the libc library came in two variants: the static variant implied that the *strerror* object be contained in the configuration, while the dynamic variant implied that *strerror* not be contained.

Instead of having two libc variants distinguished by different values of the *linkage* feature, we might as well have two libc revisions distinguished by a change application δ_1. Then, the *strerror* component would only be contained in the configuration if δ_1 had not been applied:

$$C = [\mathit{object}\colon \mathit{program}] \boxplus_I (\nabla_1 \to \mathit{object}\colon \mathit{strerror})$$

We may also use both the *linkage* and delta features to describe the configuration. For instance, if δ_1 had changed the linkage of libc from static to dynamic, we may write

$$C = [\mathit{object}\colon \mathit{program}] \boxplus_I (\mathit{linkage}\colon \mathit{static} \to \mathit{object}\colon \mathit{strerror})$$
$$\sqcap\,(\Delta_1 \to \mathit{linkage}\colon \mathit{dynamic})$$

which leaves the *linkage* feature in the configuration term and makes the nature of the change δ_1 explicit.

To conclude, we see that it makes no difference whether we identify versions and configurations by the applied changes or by other features. We can thus generalize revision constraints to *configuration constraints,* allowing us to express implications between arbitrary versioning dimensions.

12.7 Maintaining Configuration Constraints

We close this chapter by discussing some useful techniques involving revisions and changes.

12.7.1 Revision Tagging

Rather than having changes imply features, one may also have features implying certain changes. In section 2.3.2, for instance, we discussed the CLEARCASE identification scheme: Users can assign names to edges in the version graph and select revisions through a disjunction of name patterns. Such naming of changes is easily expressed through an implication between the name and the respective delta feature, as discussed in section 9.5.5.

As an example, tagging can be used for classifying versions by their *status.* For instance, we may wish to classify versions in three categories *experimental*, *proposed*, and *published*. An implication like $\big([\mathit{status}{:}\mathit{proposed}] \to (\Delta_5 \sqcap \nabla_6)\big)$ as configuration constraint can then ensure that whenever a *proposed* version is required, R_5 is returned.

12.7.2 Maintaining Currency

Tagging is also useful for maintaining *currency.* In our model, we cannot simply devise some revision as "current", because currency may differ across variants; currency constitutes a part of the SCM protocol, expressed through means of the SCM primitives layer—that is, using features. A simple scheme to denote the current versions is to use a set $[\mathit{current}{:}\top]$ that contains the current variants by implying certain revisions. An implication $\big([\mathit{current}{:}\top, \mathit{os}{:}\mathit{unix}] \to [\Delta_2, \nabla_5]\big)$ ensures that whenever the current *unix* variant is requested, the revision R_2 is returned. In section 13.1.3, we give an example of using currency in workspaces.

12.7.3 Extrinsic and Intrinsic Changes

Regarding our discussion of extrinsic and intrinsic features in section 10.1, the question may arise whether delta features are intrinsic or extrinsic features. The answer is: if the change affects other components, it is extrinsic, and so is the delta feature; if the change does not, it is intrinsic, and so is the delta feature as well.

12.8 Conclusion

The maintenance of revision constraints in a repository is no more difficult than maintaining a "classical" revision graph: for any new revision, a simple constraint is added just as a new edge is added to the revision graph. Orthogonal changes impose no special problems.

Revision constraints can be generalized to configuration constraints, expressing implications between arbitrary versioning dimensions. The role of configuration constraints in structuring the configuration space cannot be over-emphasized. Through configuration constraints, we can identify, select, and revise arbitrary configurations, regardless of their specific versioning dimensions, and ensure their consistency with respect to the configuration constraints. In chapter 13, we show how configuration constraints are used to model cooperation techniques.

If a program is useful, it will have to be changed.

— LAWS OF COMPUTER PROGRAMMING, III

Chapter 13

Cooperation Techniques

Having discussed the concepts of logical and historical versioning, as modeled through feature logic, we now examine the third and last versioning dimension, which is cooperative versioning. *We introduce the notion of a* workspace, *confining all user operations to a specific configuration and isolating users from each other's changes. Through dedicated workspaces, users can publish and propagate their changes. Using two cooperation scenarios, optimistic and conservative, we demonstrate how changes propagate across workspaces and show how workspaces integrate with the versioning concepts discussed so far.*

13.1 Working in Workspaces

13.1.1 Context and Confinement

In the context of SCM, the work of an individual developer can be described as a series of *operations*—operations like *reading,* that is, examining components, or *writing,* that is, changing components. Each operation affects a specific configuration of component versions. Often, many subsequent operations affect the same configuration. Hence, it is desirable to specify this common configuration only once and to *confine* all subsequent operations to that configuration:

Definition 13.1 (Context, Confinement) An operation is *confined* to a configuration C (called *operation context* or simply *context*) if it only affects a subset of C. □

Formally, such a confinement can be enforced as follows: Given a context C, an operation on a component K is confined to the set $K \sqcap C$. That is, if $K \sqsubseteq$

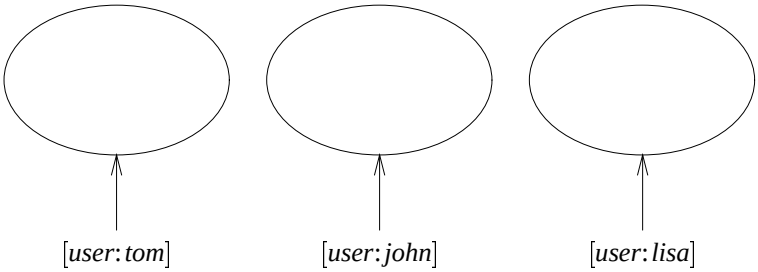

Figure 13.1: Disjoint write contexts

C holds, the operation will succeed; if $K \sqsubseteq \sim C$ holds, the operation will fail; otherwise, only the subset $K \sqcap C$ will be affected by the operation.

In practice, different operations can be confined by different contexts imposed by the SCM system, realizing *access control.* For instance, a system-imposed *read context* may define the component versions a user can examine, while the *write context* defines the component versions a user can change. By assigning each developer an individual write context disjoint from other write contexts, the SCM system can ensure that changes made by one developer do not interfere with changes made by another developer.

The easiest way to realize disjoint write contexts is to use some common feature with a different value for each user, and to make the write context a subset of this feature term. For instance, we may use a *user* feature having the user identification as value: formally, each write context W of a user U is a context $W \sqsubseteq [user{:}U]$, where U is some feature term uniquely identifying the user. Since the *user* feature may have only one value, all write contexts are disjoint, as illustrated in figure 13.1.

Making write contexts disjoint is a necessity for keeping individual changes apart. In practice, users may also choose to keep their read contexts disjoint such that they do not see the changes made by others. Likewise, users may wish to work on a specific configuration only, confining their changes to that configuration. We thus introduce the notion of a user-definable working context or *workspace* confining *all* user operations in addition to the read and write contexts imposed by the SCM system.

Definition 13.2 (Workspace) A *workspace* is a user-definable context confining all user operations. □

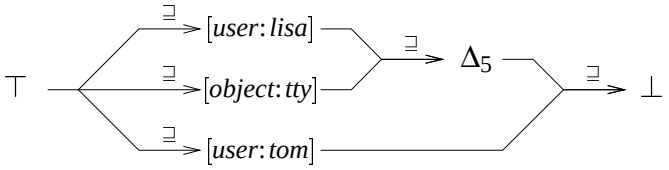

Figure 13.2: Changes and workspaces

For instance, let us assume Lisa has chosen her write context $[user{:}\,lisa]$ as workspace. If Lisa applies a change δ_5 to the *tty* object, this change is confined to her workspace. That is, Δ_5 is subsumed by $[user{:}\,lisa]$; the *tty* component is identified by the additional configuration constraint $\big(\Delta_5 \to [user{:}\,lisa]\big)$, as illustrated in figure 13.2.

Let us assume that Tom also has chosen his write context $[user{:}\,tom]$ as workspace. In this case, Tom cannot access Lisa's change as his view is subsumed by $[user{:}\,tom]$; formally, $user{:}\,tom \sqsubseteq {\sim}[user{:}\,lisa] \sqsubseteq \nabla_5$ holds. Hence, both Tom and Lisa can operate without interfering with each other—until their changes are integrated into some production version.

The confinements imposed by the read and write contexts still apply, regardless of the workspace choice. Hence, if each user U has a write context of $[user{:}\,U]$ and a read context of $\top$, Tom can set his workspace to $\top$ and thus examine Lisa's current work; but his write context keeps him from changing them.

13.1.2 Operations in Workspaces

By adding additional constraints to their workspaces, users can choose to confine their work to specific configurations only. In figure 13.3 on the following page, Tom has chosen his workspace as $[user{:}\,tom, os{:}\,mac]$. Let us choose this example to illustrate the effects of operations in his workspace:

Reading versions. *Reading a component version K in a workspace W returns $K \sqcap W$ only.*

Tom does not see the non-*mac* versions (like $[os{:}\,windows]$ or $[os{:}\,plan\text{-}9]$) nor does he see the changes of other users (like $[user{:}\,lisa]$). Components whose *user* or *os* feature is unspecified are included nonetheless in Tom's view because the components are the same across all *user* or *os* values.

What we have criticized in section 9.2 now comes out as a virtue: overspecialization or orthogonal features in the workspace do not hinder version selection.

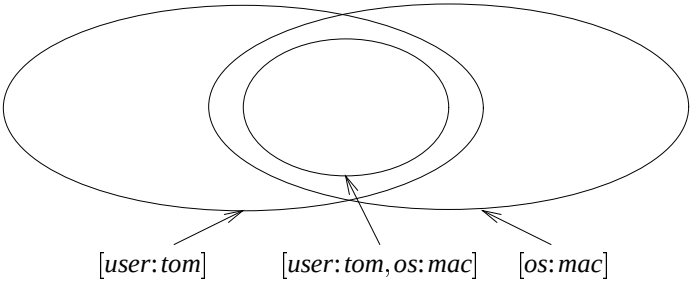

Figure 13.3: Workspaces and configurations

Writing versions. *Writing a component version K in a workspace W changes $K \sqcap W$ only.*

All changes Tom makes in his workspace are automatically confined to the $[user: tom, os: mac]$ variants of the components. The features of the components stay the same, but the $[user: tom, os: mac]$ variant will incorporate Tom's changes, while the $\sim[user: tom, os: mac]$ variant seen by the other users will not incorporate Tom's changes.

Creating versions. *Creating a component version K in a workspace W creates $K \sqcap W$ only.*

Since K must not be visible outside of W, the component version $K \sqcap \sim W$ does not exist; this is expressed by constraining the features of the component to $\sim(K \sqcap \sim W) = \sim K \sqcup W = (K \to W)$, which expresses that K is a subset of W.

If Tom creates a new component in his workspace, this component must remain unaccessible to other users. Hence, any such component inherits the features of Tom's workspace. If Tom creates an *mac*-specific component *active-help*, it will be identified as

$$[object: active\text{-}help, user: tom, os: mac] \ .$$

The additional constraint

$$\big([object: active\text{-}help] \to [user: tom, os: mac]\big)$$

ensures that the *active-help* will not be visible to other users (formally, $\sim[user{:}\,tom] \sqsubseteq \sim[object{:}\,active\text{-}help]$ holds) or be included in other operating systems ($\sim[os{:}\,mac] \sqsubseteq \sim[object{:}\,active\text{-}help]$).

Removing versions. *Removing a component version K in a workspace W removes $K \sqcap W$ only.*

If Tom deletes a component in his workspace, this component must remain accessible to others. Consequently, a deleted components is assigned with an additional feature, namely the *complement* $\sim W$ of Tom's workspace W.

Let us assume that all users see the same version of the $[object{:}\,keyboard]$ component. If Tom deletes the *keyboard* component from the *mac* version, the *keyboard* component will be identified as

$$\big[object{:}\,keyboard, \sim[user{:}\,tom, os{:}\,mac]\big] \ ,$$

such that it will be no more visible in Tom's workspace. Outside of Tom's workspace, the *keyboard* component will still be visible.

13.1.3 Maintaining Currency

Even when their individual workspace is confined to a specific configuration or revision, users may find it convenient to distinguish versions in "current" and "non-current" (i.e. *outdated*) versions, as discussed in section 12.7. Outdated versions may be identified by $[current{\uparrow}]$, for instance, and hidden by making the selection $[current{:}\,\top]$ part of the workspace. Rather than re-setting the workspace to the latest version after every change, users could then simply tag outdated components with $[current{\uparrow}]$ and access only the most recent version.

Definition 13.3 (Outdating) To make the change δ_i current within the workspace $[user{:}\,U]$, and to outdate all versions that where the change δ_i has not been applied, make the set $[user{:}\,U, current{:}\,\top]$ a subset of Δ_i. □

Using the constraint representation to express subsumption relations, this means replacing any constraint

$$\big([user{:}\,U, current{:}\,\top] \to S\big)$$

by

$$\big([user{:}\,U, current{:}\,\top] \to \Delta_i\big) \ .$$

Here is an example, illustrated in figure 13.4 on the next page. In Lisa's workspace, revision Δ_5 is the current revision, which is expressed by a constraint

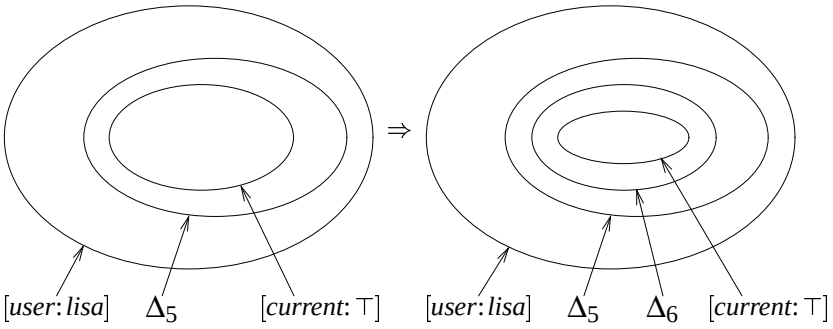

Figure 13.4: Changing currency in a workspace

$\big(([\mathit{current}\colon \top] \to \Delta_5) \sqcap (\Delta_5 \to [\mathit{user}\colon \mathit{lisa}])\big)$ in all components changed by Lisa. After applying a change δ_6, Lisa decides to make the Δ_6 components current; this is done by adding another constraint $\big([\mathit{current}\colon \top] \to \Delta_6\big)$ to the components where the δ_6 change was applied.

Lisa's workspace always remains the same, namely $(\mathit{user}\colon \mathit{lisa}, \mathit{current}\colon \top)$; rather than changing her workspace, she changes the features of the components such that she always sees the current versions. None of these constraints is visible outside of Lisa's workspace, as they are all subsumed by $[\mathit{user}\colon \mathit{lisa}]$.

13.1.4 Working in Teams

Just as a *user* feature is useful to keep user workspaces disjoint, other features can be appropriate to confine changes within larger entities.

Multiple teams. Besides the *user* feature, a *team* feature may be appropriate to organize several people working on one task. For instance, all users in the $[\mathit{team}\colon \mathit{microkids}]$ workspace could work on the soul of a new machine, allowing each other to access their changes; but users working in the $[\mathit{team}\colon \mathit{hardyboys}]$ workspace would not see their changes and vice versa. Sub- or superteams can be modeled likewise.

Multiple projects. Besides teams, users may work in different projects, which could be kept disjoint as well by introducing a *project* feature. For instance, in the setting illustrated in figure 13.5 on the facing page, user Kidder is assigned to two projects *eclipse* and *nova*, which is expressed by setting Kidder's workspace to $\big[\mathit{user}\colon \mathit{kidder}, \mathit{project}\colon \{\mathit{eclipse}, \mathit{nova}\}\big]$. Kidder may refine his workspace to one of these projects and switch workspaces as needed.

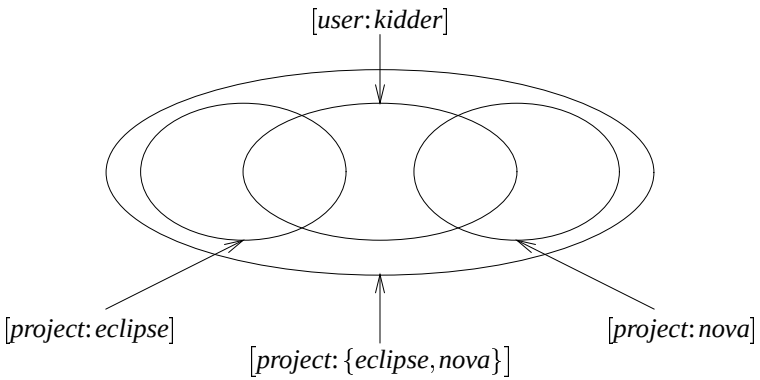

Figure 13.5: Users and projects

Multiple sites. In section 5.7, we discussed techniques for realizing development in multiple sites. If a distributed repository like NUCM is not available, distribution can be made explicit by assigning each development site a specific value of a *site* feature. Just as with teams, users, and projects, users at a particular site can only change the local components. However, read access to the changes made at other sites can be realized by regular updates as realized in the MULTISITE tool.

13.2 Conservative Cooperation Techniques

In section 5.5, we have discussed cooperation strategies that prevent against accidental loss of changes. In this section, we discuss the first group of these strategies, namely *conservative cooperation strategies* that prevent against parallel changes through a *locking* mechanism.

13.2.1 Locking Versions

In a conservative cooperation strategy, a user can change a component if and only if it has not been locked by another user; before changing the component, the user must explicitly lock it.

Using feature logic, we can distinguish locked from unlocked versions using an additional *locked* feature and the *tagging* technique introduced in section 12.7. For a component K, each version $V \sqsubseteq K$ locked by a user U is expressed through a *locking constraint*

$$K \sqsubseteq \Big(\big(V \sqcap [\mathit{locked}\colon \top]\big) \rightarrow [\mathit{user}\colon U] \Big) \tag{13.1}$$

The SCM system must ensure that only locked versions may be changed—for instance, by setting the write context to a subset of $[\mathit{locked}\colon \top]$.

As a simple example, assume that Tom has locked revision Δ_{25} of a *screen* component. The *screen* component then has the features

$$\mathit{screen} \sqsubseteq \big((\Delta_{25} \sqcap [\mathit{locked}\colon \top]) \to [\mathit{user}\colon \mathit{tom}]\big)$$

If Lisa wishes to access a locked revision Δ_{25} of *screen* for writing, this will fail:

$$\mathit{screen} \sqcap \Delta_{25} \sqcap [\mathit{locked}\colon \top] \sqcap [\mathit{user}\colon \mathit{lisa}] = \bot \ ,$$

since $\Delta_{25} \sqcap [\mathit{locked}\colon \top]$ implies $[\mathit{user}\colon \mathit{tom}]$.

Lisa may access an unlocked version for reading, however:

$$\mathit{screen} \sqcap \Delta_{25} \sqcap [\mathit{user}\colon \mathit{lisa}] = \mathit{screen} \sqcap \Delta_{25} \sqcap [\mathit{user}\colon \mathit{lisa}] \sqcap [\mathit{locked}\colon \uparrow]$$

since (13.1) can also be formulated as

$$K \sqsubseteq \Big(\sim[\mathit{user}\colon U] \to \big(\sim V \sqcup [\mathit{locked}\uparrow]\big)\Big) \ ;$$

consequently, $[\mathit{user}\colon \mathit{lisa}] \sqcap \Delta_{25}$ implies $[\mathit{locked}\colon \uparrow]$.

We deduce two operations for locking and unlocking component versions:

Definition 13.4 (Locking) To lock a version set V for a user U, make V a subset of $\big((V \sqcap [\mathit{locked}\colon \top]) \to [\mathit{user}\colon U]\big)$. To unlock V, make V a subset of $[\mathit{locked}\uparrow]$. □

The SCM system must ensure that a version set V can only be locked when it was previously unlocked and vice versa.

13.2.2 Propagating Changes

While the locking mechanism prevents users from making parallel changes to a version set, we need an additional *propagation* mechanism that propagates changes across workspaces.

As an example of propagation, reconsider figure 13.2 on page 143, where Lisa has applied a change δ_5 to the *tty* object in her workspace. Tom wishes to propagate this change to his workspace as well. He invokes the SCM system such that Lisa's version $[\mathit{object}\colon \mathit{tty}, \mathit{user}\colon \mathit{lisa}, \Delta_5]$ is copied into a new version of *tty* named $[\mathit{object}\colon \mathit{tty}, \mathit{user}\colon \mathit{tom}, \Delta_5]$. As illustrated in figure 13.6 on the facing page,

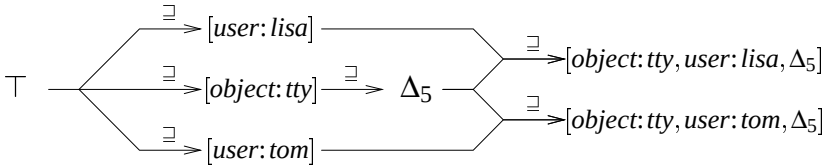

Figure 13.6: Propagating changes across workspaces

this makes Δ_5 a subset of both Tom's and Lisa's workspaces; the features of the *tty* component become

$$tty \sqsubseteq [object{:}\, tty] \sqcap \Big(\Delta_5 \rightarrow \big[user{:}\, \{tom, lisa\}\big]\Big) \quad .$$

Tom may now make the Δ_5 version current and thus determine how Lisa's change affects his current work. Any changes Tom makes in his workspace are still invisible to Lisa—unless she propagates them into her workspace.

We conclude with a general definition of a *propagate* operation that propagates changes across workspaces:

Definition 13.5 (Propagate) Let δ_i be a change. To propagate δ_i from a workspace $[user{:}\, U]$ to a workspace $[user{:}\, U']$, make Δ_i a subset of $[user{:}\, U']$ as well as of $[user{:}\, U]$. □

Using the constraint representation to express subsumption relations, this means replacing the constraint $\big(\Delta_i \rightarrow [user{:}\, U]\big)$ by $\big(\Delta_i \rightarrow [user{:}\, \{U, U'\}]\big)$.

13.2.3 Controlling Change Propagation

Propagating changes across workspaces helps individual users to *synchronize* their work, that is, to make their workspaces identical (or at least, less divergent). To keep divergence small is an important issue in SCM, because the more workspaces diverge, the more likely changes are to conflict with each other, making the construction of the final product a difficult task.

For several users, change propagation must be organized in a special way to ensure that all workspaces are synchronized with each other. A simple way to ensure synchronization is to establish a notion of a common *main development line,* representing the published or end user's view of a product; workspaces are temporary variants of this main development line, as discussed in section 5.5.2. Before publishing changes, users must synchronize their own workspace with the main development line. Hence, this scheme prohibits excessive divergence of user workspaces and encourages frequent synchronization.

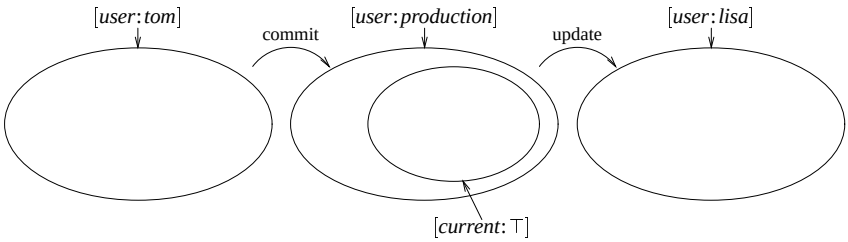

Figure 13.7: Propagating changes through a production workspace

In our model, such a main development line can be realized as follows. To keep the main development line isolated from other's changes, it must be disjoint from all user workspaces. Hence, we can establish the main development line as a dedicated workspace, called *production workspace,* which represents the published view of the product and which is disjoint from all user workspaces.

In this setting, users are discouraged from propagating changes between user workspaces. Instead, changes are propagated from the production workspace to user workspaces, and vice versa, using two operations *update* and *commit*. As illustrated in figure 13.7, the *update* operation propagates the current changes from the production workspace to the user's workspace, and the *commit* operation propagates the current changes from the user's workspace to the production workspace. Both operations also make the propagated changes current in the destination workspace.

Before defining the *update* and *commit* operations, we define a more general *propagate-current* operation which propagates the current changes between workspaces and makes them current in the destination workspace.

Definition 13.6 (Propagate-current) To propagate the current changes from the workspace $[user{:}U]$ to the workspace $[user{:}U']$, propagate all changes subsuming $[current{:}\top]$ in $[user{:}U]$ to $[user{:}U']$, and make them current in $[user{:}U']$. □

In the constraint representation, propagating the current changes means the following: For each change δ_i such that $\Delta_i \sqsupseteq [user{:}U, current{:}\top]$ holds, replace the constraint $\big(\Delta_i \to [user{:}U]\big)$ by $\big(\Delta_i \to [user{:}\{U,U'\}]\big)$ and add a new constraint $\big([user{:}U', current{:}\top]\big) \to \Delta_i$.

Both *update* and *commit* can now be defined using *propagate-current*:

Definition 13.7 (Update) To update a user workspace $[user{:}U]$, propagate the current changes from $[user{:}production]$ to $[user{:}U]$. □

Definition 13.8 (Commit) To commit the current changes from a user workspace $[user{:}U]$, propagate the current changes from $[user{:}U]$ to $[user{:}production]$.

□

13.2.4 A Conservative Scenario

As an example of change propagation through a production workspace, we have illustrated a simple scenario in this section. In figure 13.8, we see a production workspace $[user{:}production]$ containing the end user's view of some product. The product comes in two variants, a demonstration variant $[demo{:}\top]$ and a full-fledged variant $\sim[demo{:}\top] = [demo{\uparrow}]$. The set $[current{:}\top]$ encompasses the current versions of both variants.

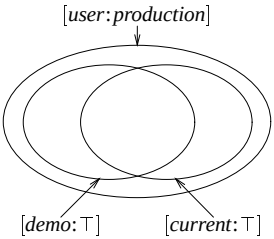

Figure 13.8: A production workspace

Both users Tom and Lisa have established their workspaces $[user{:}tom]$ and $[user{:}lisa]$ as temporary variants of the current production workspace; as illustrated in figure 13.9, each of them can access both the demonstration and the full-fledged variant.

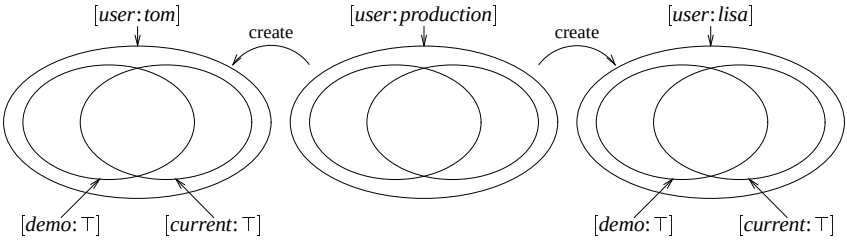

Figure 13.9: Creating user workspaces

Tom wishes to apply a change to the current version. He locks the current version, making $[current{:}\top]$ a subset of $[locked{:}\top]$. Lisa cannot access the locked versions, since $[locked{:}\top] \sqsubseteq [user{:}tom]$ and thus $[user{:}lisa, locked{:}\top] = \bot$ holds, as shown in figure 13.10 on the following page.

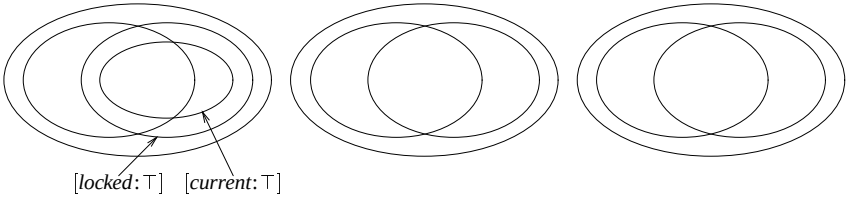

Figure 13.10: Locking the current version

Tom applies his change δ_1 to the current version. Both product variants are affected by the change; Δ_1 is thus orthogonal to $[demo: \top]$. After testing his change, Tom makes Δ_1 current—that is, $[current: \top]$ is now a subset of Δ_1, illustrated in figure 13.11. Still, Δ_1 is locked, as it is a subset of $[locked: \top]$.

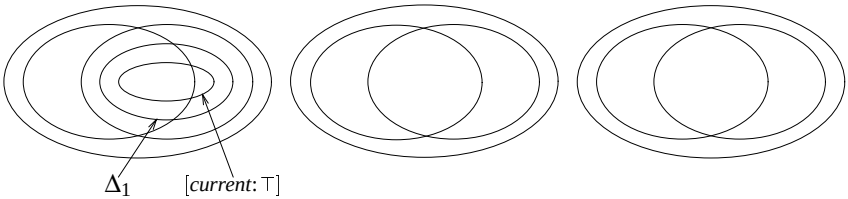

Figure 13.11: Changing a locked version

Tom's work is done; he releases his lock and commits his change δ_1 to the production workspace, making it current there as well. The workspace state is shown in figure 13.12.

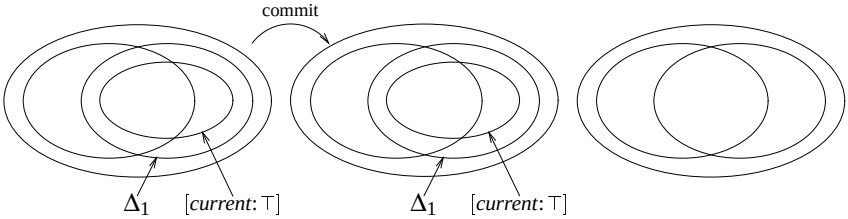

Figure 13.12: Committing changes to the production workspace

Now is the time for Lisa to make her changes. First, Lisa updates her workspace with Tom's changes, as shown in figure 13.13 on the next page. Tom's change δ_1 is now current in Lisa's workspace as well.

Lisa works on the demonstration variant only; she locks the current version, making $[locked: \top]$ a subset of $[demo: \top, user: lisa]$. Selecting the current demon-

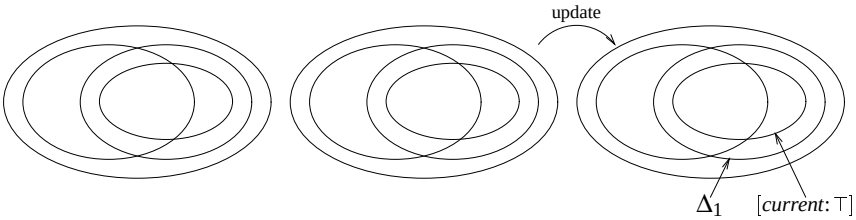

Figure 13.13: Updating a workspace from the production workspace

stration variant now implies that the locked version be selected, as shown in figure 13.14.

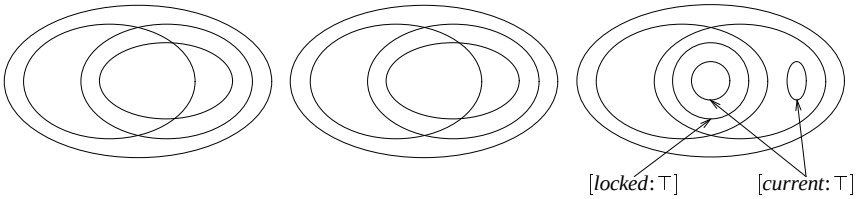

Figure 13.14: Locking a variant

Since the demonstration variant is locked by Lisa, other users can no more lock and change it. Its complement, the non-demonstration variant, is still unlocked and may be locked and changed by other users. Just like Tom, Lisa performs a change δ_2 on the demonstration variant. The Δ_2 set is now current, i.e. a subset of $[current: \top]$, as shown in figure 13.15.

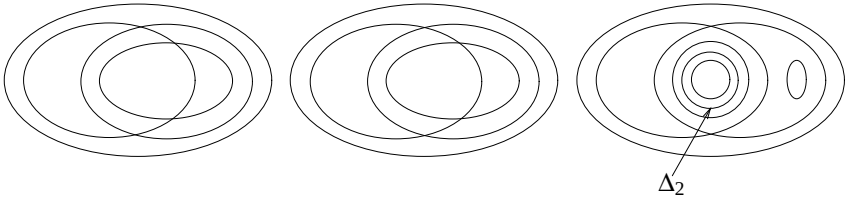

Figure 13.15: Changing a variant

As final step, Lisa commits her change to the production workspace, releasing her lock. This final state is illustrated in figure 13.16 on the following page: In the production workspace, both Tom's change δ_1 and Lisa's change δ_2 have been applied and are both included in the current version.

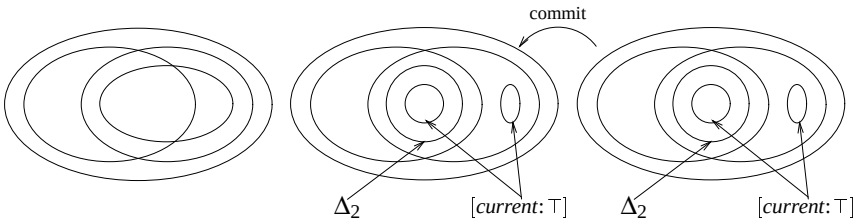

Figure 13.16: Committing variant changes

13.3 Optimistic Cooperation Techniques

13.3.1 Synchronizing Workspaces

Conservative cooperation strategies, as illustrated in section 13.2, have both the advantage and disadvantage that only one developer at a time can work on a particular version of a component. Using an optimistic cooperation strategy, as discussed in section 5.5.2, users are allowed to work in parallel, each on a temporary variant. Here, it is essential that developers *synchronize* their workspaces frequently—that is, catch up with other changes such that the individual workspace is more similar to other workspaces. For this purpose, the changes of other users must first be made visible in the workspace, and then be *merged* with the individual changes.

As an example of merging, consider figure 13.17, where Tom has applied a change δ_1 in his workspace $[user{:}tom]$. Before committing that change back to the production workspace, he updates his workspace by making the parallel change δ_2 available. The change δ_2 is then merged into his current version, creating a merged version $\Delta_1 \sqcap \Delta_2$. This combined change may now be committed to the production workspace.

The versions to be merged can easily be determined automatically. As discussed in section 5.6, automated merging of two versions relies on knowing their common *base version.* Using version sets, the common base version V_0 of two versions V_1 and V_2 is the lowest common ancestor in the subsumption lattice, ex-

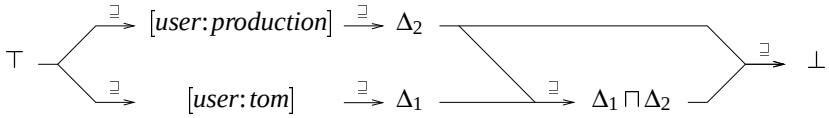

Figure 13.17: Merging changes from the production workspace

cluding any changes leading up to V_1 or V_2. In our example, $V_1 = \Delta_1$ and $V_2 = \Delta_2$ hold; the common base version V_0 is determined as $V_0 = \nabla_1 \sqcap \nabla_2$; that is, the version excluding both changes.

Definition 13.9 (Synchronize) To synchronize a user workspace $[user{:}U]$ with the production workspace $[user{:}production]$, perform the following two steps:

1. Update $[user{:}U]$ from $[user{:}production]$, making the versions $\Delta_1, \ldots, \Delta_n$ accessible in $[user{:}U]$ (but not yet current).

2. In $[user{:}U]$, merge the versions $\Delta_1, \ldots, \Delta_n$ with $[current{:}\top]$, where the base version is the lowest common ancestor in the subsumption lattice, excluding any later changes. The resulting merged version is identified as $[user{:}U, current{:}\top] \sqcap \Delta_1 \sqcap \cdots \sqcap \Delta_n$. □

In a third step, the merged version may now be committed to the production workspace, making the individual changes available to other users. As in the conservative scenario, no changes get lost—provided that the merged version is carefully checked.

13.3.2 Identifying Merged Versions

When the versions to be merged are identified by features other than delta features, special care must be taken when identifying the merged version: As merging has no semantics in terms of feature logic, the features of the merged version cannot be determined automatically.

To illustrate this problem, consider the merge of two versions identified by $[os{:}dos]$ and $[os{:}windows]$. The features of the merged version are dependent on the nature of the merge: if the merged version is system-independent, its *os* feature will be unspecified; if the merged version runs on DOS as well as on WINDOWS, its features are $[os{:}\{dos, windows\}]$, if it does not run on UNIX, its features are $[os{:}\sim unix]$, and so on.

Here are some guidelines in identifying merged versions:

Delta features accumulate. As shown in chapter 11, each revision R_i inherits the delta features of its ancestor revisions $R_j, \ldots, R_k$. Hence, the merge of $R_i \sqsubseteq \Delta_i$ and $R_j \sqsubseteq \Delta_j$ will result in a revision $R_d \sqsubseteq \Delta_d \sqsubseteq \Delta_i \sqcap \Delta_j$. In figure 13.17 on the facing page, the merged version inherits both the Δ_1 and Δ_2 delta features.

Workspace features are ignored. Workspace features are volatile; they should not be considered while merging. Rather, the merged version should inherit the features of the workspace it is created in, like any other new version created. In figure 13.17 on page 154, the merged version is created in $[user:tom]$ and thus a subset thereof.

Other features must be determined again. Features identifying neither workspaces nor changes cannot be inferred from the originating features.

We see that there are few differences between assigning features to a merged version and between specifying the features of a newly created version. Parts that can be automated are the accumulation of delta features and the assignment of workspace features.

13.3.3 An Optimistic Scenario

To conclude, we give another example of using production workspaces, but this time mimicking the optimistic cooperation strategy of the CVS system.

The initial setting of our scenario is shown in figure 13.18. It is the same initial setting as in the conservative scenario from section 13.2.4. Users Lisa and Tom have established their workspaces as temporary variants of the current production workspace; they can access both the demonstration and the full-fledged product variant.

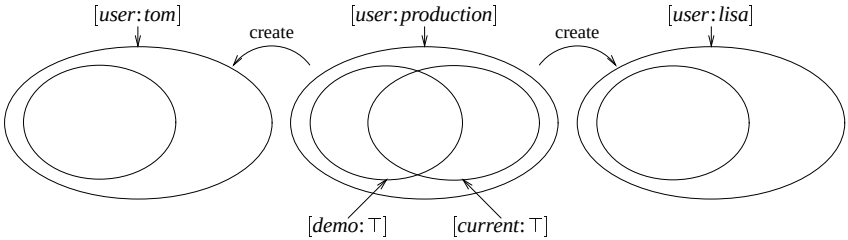

Figure 13.18: A production workspace and two user workspaces

The optimistic scenario does not prevent parallel changes. Hence, both Tom and Lisa can apply changes to the product. Tom's change δ_1 affects both variants at once, while Lisa's change δ_2 affects the demonstration variant only. Neither change is visible outside the respective user workspace, as shown in figure 13.19.[1]

In figure 13.20 on the next page, Lisa commits her change to the production workspace. The merge of her workspace and the production workspace is trivial,

[1] For clarity, we show the current versions $[current:\top]$ in the production workspace only.

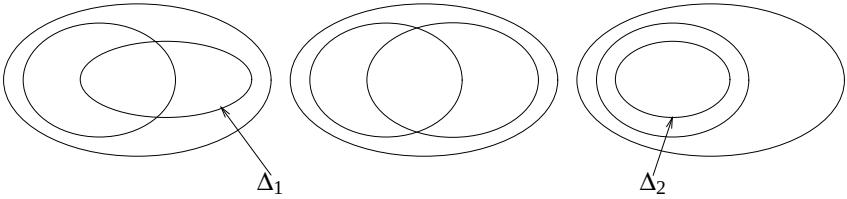

Figure 13.19: Changes in user workspaces

because the base version is identical to the production workspace; hence, Lisa's changed version is simply copied to the production workspace. This makes the current version of the demonstration variant imply the δ_2 change, or formally, $\big([\mathit{current}\colon \top] \to [\mathit{demo}\uparrow] \sqcup \Delta_2\big)$.

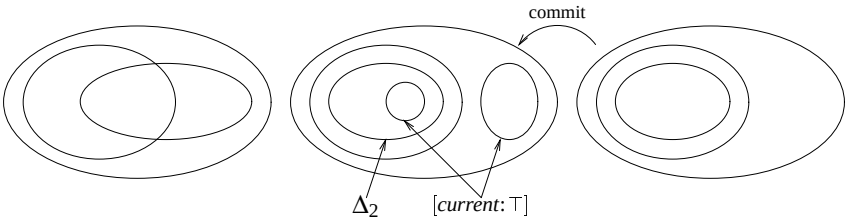

Figure 13.20: Simple synchronization of the production workspace

Tom now wishes to commit his change δ_2. Before doing so, he synchronizes his workspace. The first step is to update his workspace with the current change δ_1 from the production workspace. As shown in figure 13.21, the changes δ_2 and δ_1 are still disjoint.

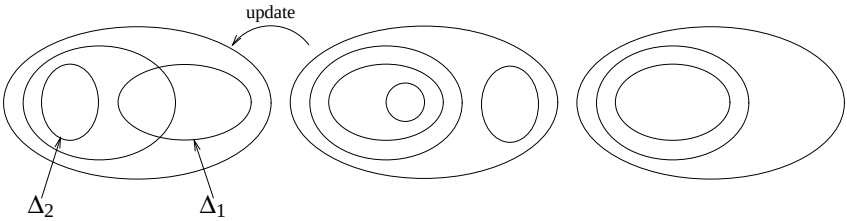

Figure 13.21: Updating a user's workspace

In the second synchronizing step, shown in figure 13.22 on the next page, Tom integrates the two changes δ_1 and δ_2, resulting in the merged version set $\Delta_1 \sqcap \Delta_2$.

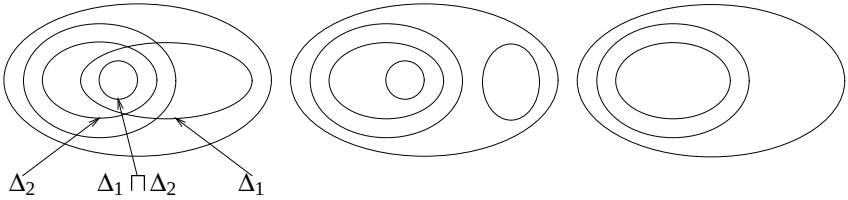

Figure 13.22: Merging in a user's workspace

After removing any conflicts between the changes δ_1 and δ_2, Tom commits his versions back to the production workspace. As shown in figure 13.23, this makes both Δ_1 and Δ_2 current versions in the respective variant.

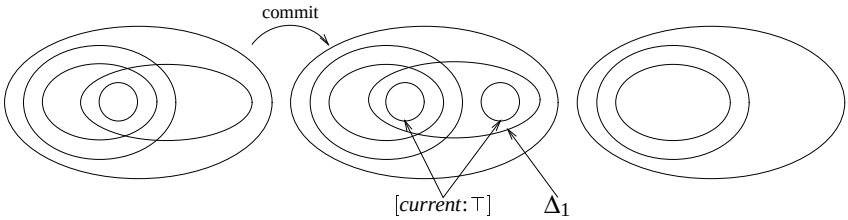

Figure 13.23: Synchronization of the production workspace after merge

In both scenarios, the optimistic scenario presented here and the conservative scenario presented in section 13.2.4, the final current production version includes both Lisa's δ_1 and Tom's δ_2 change; none of their changes is lost. The difference in optimistic cooperation is that changes can be made in parallel and stay orthogonal to each other. In our example, the change δ_2 is orthogonal to the change δ_1; in the conservative scenario, δ_2 implied δ_1, since parallel changes are inhibited.

13.4 Discussion

In this chapter, we have presented some techniques that help organizing the work of several users working on a product by controlling the propagation of changes. We keep changes disjoint by confining them into disjoint user, team, project, and site workspaces. By refining their workspaces, users can decide which versions to work upon without conflicting with other's work. Through a dedicated workspace, users can publish and propagate their changes, using either conservative or optimistic cooperation techniques.

Both the conservative and optimistic scenario presented in this chapter show how the concepts introduced so far integrate—notably, how version sets uni-

formly represent revisions, variants, and workspaces. But the scenarios also show up a deficiency of feature logic. We can easily capture some versioning *state* by means of feature terms and set diagrams. But we cannot express the *transitions* between these states using feature logic—there is no way to express the semantics of an *update* operation in feature logic, for example. This is different from consistency checking and version selection, where we could express all operations in terms of feature logic. The properties of a formalism that allows us to express these transitions, that is, to treat feature terms as first-class objects, remain yet to be discovered.

Der Mensch ist ein zeitliches Wesen,
das nur lebt, indem es seine Welt um sich wandelt.

— KARL JASPERS, Einführung in die Philosophie

Plus ça change, plus c'est la même chose.

— ALPHONSE KARR

Chapter 14

Taming Complexity

For practical systems, a logic foundation alone does not suffice. We also must know whether the central problems are decidable, and if so, at which cost. If these costs are too high, we must identify the circumstances under which the costs can be cut down.

The central problems in feature logic are deciding inconsistency, subsumption, and equivalence. As shown in proposition 8.32 on page 86, all these problems can be reduced to deciding inconsistency. We present Smolka's feature unification algorithm, which decides inconsistency for general quantifier-free feature terms. As deciding inconsistency in general is co-NP-complete, Smolka's algorithm is of exponential time complexity. This makes practical applications unable to scale up beyond a certain problem size. As a solution, we present some specialized procedures that break down complex SCM problems into manageable pieces and discuss the conditions for efficient realization of SCM operations.

14.1 Deciding Inconsistency for Simple Feature Terms

We begin with a discussion of the basic mechanisms to deduce consistency of feature terms—that is, *feature unification.* In [Smo92], Smolka presents a *constraint system* that can be used to decide about the inconsistency of feature terms. The basic idea is to convert a simple feature term into a set of *feature constraints,* The inconsistency of the constraint set can be decided in quadratic time.

Proposition 14.1 Deciding inconsistency of simple feature terms is of quadratic time complexity.

PROOF. Smolka's algorithm for solving feature clauses decides inconsistency of simple feature terms in quadratic time [Smo92]. □

Under certain circumstances, subsumption can also be decided in quadratic time.

Corollary 14.2 Deciding the subsumption $S \sqsubseteq T$ is of quadratic time complexity, if the basic forms of S and $\sim T$ are simple.
PROOF. Deciding whether $S \sqsubseteq T$ holds is equivalent to deciding whether $S \sqcap \sim T$ is inconsistent (proposition 8.32 on page 86). Both S and $\sim T$ can be converted in linear time into basic form (proposition 8.22 on page 84). If the basic forms of S and $\sim T$ are simple, proposition 14.1 on the preceding page applies. □

As term equivalence $S = T$ is reducible to mutual subsumption (8.6), a similar shortcut exists only if the basic forms of S, $\sim S$, T, and $\sim T$ are simple, which is only true for trivial feature terms.

14.2 Deciding Inconsistency for General Feature Terms

For general feature terms including quantifiers and unions, inconsistency, subsumption, or equivalence are undecidable problems.

Proposition 14.3 Inconsistency, subsumption, and equivalence of general feature terms are undecidable problems.
PROOF. In [Smo92]; the proof follows from the word problem of Thue systems being undecidable. □

The problems are decidable, however, for quantifier-free terms.

Proposition 14.4 Deciding inconsistency, subsumption, and the equivalence of quantifier-free feature terms are co-NP-complete problems.
PROOF. In [Smo92]; the proof follows from the satisfiability problem of propositional logic being NP-complete. □

Inconsistency, subsumption, and equivalence being co-NP-complete problems implies that time complexity of decision is exponential.

For arbitrary quantifier-free feature terms, Smolka has presented an algorithm called *feature unification* to decide inconsistency [Smo92]. The basic idea is to convert the feature term into basic form and then into DNF. Since each conjunct of the DNF is simple, inconsistency of each conjunct can be decided in quadratic time, as discussed in proposition 14.1 on the page before. Transformation into

DNF, however, is of exponential time complexity, resulting in exponential time complexity of feature unification.

14.3 A Unification Example

We do not give a complete description of Smolka's algorithm here—the interested reader may refer to [Smo92] for details. Instead, we illustrate feature unification through an example. Let S and T denote the features of two components, where $S = [\textit{host-arch}\colon \{\textit{pentium}, \textit{power-pc}\}, \textit{host-arch} \downarrow \textit{target-arch}]$ and $T = [\textit{target-arch}\colon \sim \textit{power-pc}]$ holds. We use feature unification to determine whether S and T are consistent with each other, or whether $S \sqcap T = \bot$ holds.

1. We determine

$$U = S \sqcap T = \left[\begin{array}{l} \textit{host-arch}\colon \{\textit{pentium}, \textit{power-pc}\}, \\ \textit{host-arch} \downarrow \textit{target-arch}, \\ \textit{target-arch}\colon \sim \textit{pentium} \end{array} \right]$$

2. U is already in basic form. The transformation to disjunctive normal form yields $U = U' \sqcup U''$ with

$$U' = \left[\begin{array}{l} \textit{host-arch}\colon \textit{pentium}, \\ \textit{host-arch} \downarrow \textit{target-arch}, \\ \textit{target-arch}\colon \sim \textit{pentium} \end{array} \right]$$

$$U'' = \left[\begin{array}{l} \textit{host-arch}\colon \textit{power-pc}, \\ \textit{host-arch} \downarrow \textit{target-arch}, \\ \textit{target-arch}\colon \sim \textit{pentium} \end{array} \right]$$

3. Smolka's algorithm processes each conjunct separately. It first transforms U' into a basic set of constraints, introducing temporary variables x and y to express agreement.

$$\begin{aligned} \textit{host-arch} &\doteq \textit{pentium} \\ \textit{host-arch} &\doteq x \\ \textit{target-arch} &\doteq y \\ x &\doteq y \\ \textit{target-arch} &\doteq \neg \textit{pentium} \end{aligned}$$

4. The basic set of constraints is solved by instantiating the variables x and y:

$$\begin{aligned} \textit{host-arch} &\doteq \textit{pentium} \\ \textit{target-arch} &\doteq \textit{pentium} \\ x &\doteq \textit{pentium} \\ y &\doteq \textit{pentium} \\ \textit{target-arch} &\doteq \neg\textit{pentium} \end{aligned}$$

   As *target-arch* is both *pentium* and $\neg$*pentium*, unification fails: $U' = \bot$.

5. Now comes the time for the second conjunct. U'' is also transformed into a set of constraints. After instantiation, we have:

$$\begin{aligned} \textit{host-arch} &\doteq \textit{power-pc} \\ \textit{target-arch} &\doteq \textit{power-pc} \\ x &\doteq \textit{power-pc} \\ y &\doteq \textit{power-pc} \end{aligned}$$

   resulting in the term $U'' = [\textit{host-arch}{:}\,\textit{power-pc}, \textit{target-arch}{:}\,\textit{power-pc}]$.

6. The result of the unification problem is $S \sqcap T = U' \sqcup U'' = \bot \sqcup U'' = U'' = [\textit{host-arch}{:}\,\textit{power-pc}, \textit{target-arch}{:}\,\textit{power-pc}]$.

14.4 Reduction of Feature Terms

As a consequence of feature unification being of exponential complexity, we determine possible optimizations that reduce complexity in practical applications. The field of automated theorem proving (ATP) has determined several *reduction mechanisms* that can be applied before the general decision algorithm. Generally, a reduction satisfies the following properties [Bib92]:

- A reduction truly reduces the size of an ATP problem.
- Validity of the reduced problem implies validity of the original problem (and possibly vice versa).
- Whether the reduction mechanism is applicable can be decided in polynomial time.
- The reduction mechanism itself requires polynomial time.

Since reduction is much more efficient than feature unification, it is worth exploring whether the reduction techniques established in ATP can be applied to feature terms as well. In [Bib92], Bibel gives an overview of existing reduction mechanisms in the context of propositional logic. At least three of these mechanisms, whose validity is shown in [Bib87], can also be applied to general feature terms.

Reduction of Multiple Occurrences (MULT) If a feature term S occurs multiple times in a union or intersection, the term can be reduced to one occurrence only:

$$S \sqcap S = S \tag{14.1}$$

$$S \sqcup S = S \tag{14.2}$$

MULT reduction is easily implemented by sorting the subterms in each union or intersection and removing duplicates. Sorting has a time complexity of $O(n \cdot \log n)$; MULT reduction is thus of linear-logarithmic time complexity.

Reduction of Tautologies (TAUT) If both a feature term S and its complement $\sim S$ occur in a union or intersection, they can be replaced by $\top$ and $\bot$, respectively:

$$S \sqcap \sim S = \bot \tag{14.3}$$

$$S \sqcup \sim S = \top \tag{14.4}$$

Just as MULT reduction, TAUT reduction is implemented by sorting the subterms in each union or intersection, but ignoring outer-level complement signs in the sort comparison. TAUT reduction is also of linear-logarithmic time complexity and can be combined with MULT reduction.

Reduction of Subsumed Terms (SUBS) Let S be a feature term and $S' \sqsubseteq S$ be a subset of S. If both S and S' occur in a intersection or union, only one occurrence remains:

$$S \sqcap S' = S' \tag{14.5}$$

$$S \sqcup S' = S \tag{14.6}$$

Simple subsumption can often be determined on the syntactic level—for instance, if $S' = S \sqcap T$ holds for some feature term T. Again, such a condition can be decided in linear-logarithmic time, by comparing the subterms of S and S'.

14.5 A Divide-and-Conquer Approach

By imposing certain conditions upon feature terms, time complexity of feature unification can be dramatically reduced. The most important condition is *orthogonality*: If deciding inconsistency of a feature term $U = S \sqcap T$ can be divided into deciding inconsistency of S and T separately, the terms S and T are orthogonal.

Definition 14.5 (Orthogonality) Two feature terms S and T are called *orthogonal* if

$$S \sqcap T \text{ inconsistent} \Rightarrow S \text{ inconsistent} \vee T \text{ inconsistent} \tag{14.7}$$

holds. □

An efficient procedure that determines orthogonality would be most useful, because definition 14.5 implies the following corollary:

Corollary 14.6 Let $U = S \sqcap T$ be the intersection of two consistent and orthogonal feature terms S and T. Then, U is consistent.
PROOF. Follows from S consistent $\wedge$ T consistent $\Rightarrow$ $S \sqcap T$ consistent holds, which is the negated form of definition 14.5. □

Fortunately, there is a simple sufficient condition for orthogonality: if S and T have no common features or variables, they are orthogonal.

Proposition 14.7 Two consistent, non-atom feature terms S and T are orthogonal if they have no common features or variables.
PROOF. We show that S consistent $\wedge$ T consistent $\Rightarrow$ $S \sqcap T$ consistent holds, which is the negated form of definition 14.5.

S is consistent. According to definition 8.29 on page 86, there is a feature algebra $I_S = (\mathbf{D}^{I_S}, \cdot^{I_S})$ and an I_S-assignment α_S such that $S^{I_S}_{\alpha_S} \neq \emptyset$ holds. Likewise, since T is consistent, there is a feature algebra $I_T = (\mathbf{D}^{I_T}, \cdot^{I_T})$ and an I_T-assignment α_T such that $T^{I_T}_{\alpha_T} \neq \emptyset$ holds.

Let $\mathbf{D}^I = \mathbf{D}^{I_S} \times \mathbf{D}^{I_T}$ be a domain. Let α be a mapping from the set of all variables to $\mathbf{D}^I$, defined as

$$\alpha(x) = \begin{cases} \alpha_S(x) & \text{if } x \text{ occurs in } S \\ \alpha_T(x) & \text{if } x \text{ occurs in } T \end{cases}$$

and let $\cdot^I \subseteq \mathbf{D}^I \times \mathbf{D}^I$ be an interpretation function defined for all features f as

$$f^I = \begin{cases} f^{I_S} \times T_{\alpha_T}^{I_T} & \text{if } f \text{ occurs in } S \\ S_{\alpha_S}^{I_S} \times f^{I_T} & \text{if } f \text{ occurs in } T \end{cases}$$

and for all atoms a as

$$a^I = a^{I_S} \times a^{I_T} \quad .$$

Both mappings are unambiguous since S and T have disjoint sets of variables and features.

Let $I = (\mathbf{D}^I, \cdot^I)$ be a pair of $\mathbf{D}^I$ and $\cdot^I$. I is a feature algebra—all features are functional, all names are unique, and atoms are still primitive.

Let us now consider the term $S \sqcap T$. Its interpretation results in $(S \sqcap T)_\alpha^I = S_\alpha^I \cap T_\alpha^I$. I interprets all features and variables in S like I_S; consequently, we have $S_\alpha^I = (S_{\alpha_S}^{I_S} \times T_{\alpha_T}^{I_T})$. Likewise, I interprets all features and variables in T like I_T, resulting in $T_\alpha^I = (S_{\alpha_S}^{I_S} \times T_{\alpha_T}^{I_T})$. From the equivalence $S_\alpha^I = T_\alpha^I = (S_{\alpha_S}^{I_S} \times T_{\alpha_T}^{I_T})$, we deduce $S_\alpha^I \cap T_\alpha^I = (S_{\alpha_S}^{I_S} \times T_{\alpha_T}^{I_T})$. Since S and T are consistent, both $S_{\alpha_S}^{I_S}$ and $T_{\alpha_T}^{I_T}$ are nonempty; $(S_{\alpha_S}^{I_S} \times T_{\alpha_T}^{I_T}) \neq \emptyset$ follows. Consistency of $S \sqcap T$ results from definition 8.29. □

Comparing the sets of features and variables occurring in S and T can be done in linear time, such that the conditions for proposition 14.7 on the facing page are easily verified. Consequently, a term $T = T_1 \sqcap T_2 \sqcap \cdots \sqcap T_n$ can be divided into m orthogonal subterms in quadratic time, simply by checking orthogonality for each pair T_i and T_j out of T. Each subterm can then be checked individually for consistency—for example, by using Smolka's feature unification.

14.6 Fast Consistency Checking for Simple Terms

Even if S and T are not orthogonal, their consistency can be checked in quasi-linear time if both are simple, consistent, and variable-free.

Proposition 14.8 Let S and T be simple, consistent, and variable-free feature terms; let neither S nor T contain agreements or disagreements. Consistency of $S \sqcap T$ can then be decided in quasi-linear time.

PROOF. Since T is simple, T can be decomposed into n subterms $T = T_1 \sqcap \cdots \sqcap T_n$, each of the form $f^*{:}T'$, where f^* is a *feature path* of zero or more features $f_1{:}\, f_2{:}\ldots f_m{:}T'$, and where T' is either $\top$ or an atom a or a negated atom $\sim a$ or a divergence $f\uparrow$.

For each pair T_i, T_j of subterms, $T_i \sqcap T_j$ is consistent because T is consistent. Moreover, T_i and T_j are orthogonal in any case:

1. T_i and T_j are equal. Hence, T_i and T_j are orthogonal according to definition 14.5 on page 166.

2. T_i and T_j have different feature paths or are different divergences. Then, T_i and T_j are orthogonal according to proposition 14.7 on page 166.

3. Both T_i, T_j, have the same feature path f^*—that is, $T_i = f^*{:}T_i'$ and $T_j = f^*{:}T_j'$ holds. Then, we have three cases:

   (a) $T_i' = a$ and $T_j' = \top$,
   (b) $T_i' = a$ and $T_j' = \sim b$,
   (c) $T_i' = \sim b$ and $T_j' = \top$,

   where a and b are some atoms. In all cases, $T_i' \sqsubseteq T_j'$ holds and definition 14.5 on page 166 applies. The symmetric cases lead to $T_i' \sqsupseteq T_j'$ and thus to orthogonality as well.

Since every pair of subterms T_i, T_j is orthogonal, deciding whether $S \sqcap T$ is consistent can be broken down in n subproblems:

$$S \sqcap T \text{ consistent} \;\Leftrightarrow\; S \sqcap T_1 \text{ consistent} \wedge \cdots \wedge S \sqcap T_n \text{ consistent} \tag{14.8}$$

Since S is simple as well, the same decomposition applies to the subterms S_i of $S = S_1 \sqcap \cdots \sqcap S_m$. Like the subterms T_i of T, above, each pair S_i, S_j of subterms of S is orthogonal. Hence, we can determine consistency of $S \sqcap T$ simply by determining consistency of each subterm S_i of S and each subterm T_i of T:

$$S \sqcap T \text{ consistent} \;\Leftrightarrow\; S_1 \sqcap T \text{ consistent} \wedge \cdots \wedge S_n \sqcap T \text{ consistent} \tag{14.9}$$

The combination of (14.8) and (14.9) leads to

$$S \sqcap T \text{ consistent} \;\Leftrightarrow\; \bigwedge_{\substack{1 \le i \le n \\ i < j \le m}} (S_i \sqcap T_j \text{ consistent}) \tag{14.10}$$

The subterms S_i and T_j are simple enough such that consistency of any $S_i \sqcap T_j$ can be decided in constant time. To determine the consistency of a single S_i with all T_j, it suffices to consider the term T_j with identical feature path. For a given feature path, it is possible to determine T_j in quasi-constant time using an appropriate data structure—for instance, using a hash table with an entry for each feature path. This is reasonable, since the number of features is small in practice, and so is the data structure. The remaining traversal of S requires linear time again. Overall complexity is thus of quasi-linear time, which was to be shown.

□

14.7 Integrating Reduction and Fast Consistency Checking

The proof of proposition 14.8 on page 167 leads to the construction of an algorithm that integrates consistency checking for simple feature terms with term reduction for arbitrary feature terms.

The basic idea is the principle of *partial evaluation.* In the domain of arithmetic expressions, partial evaluation means to replace known variables by their values and to evaluate resulting constant sub-expressions. This procedure is also applicable to feature terms: In a term $S \sqcap T$, every occurrence of T in S can be replaced by $\top$, since T must be satisfied anyway. Likewise, any subterm in S that is inconsistent with T can be replaced by $\bot$, since it cannot be satisfied.

Here is a simple example of partial evaluation. Consider the term

$$\begin{aligned} U &= S \sqcap T \\ &= [os\colon \sim unix, user\colon \{tom, lisa\}] \sqcap [os\colon dos, user\colon \sim tom] \ . \end{aligned}$$

We have $T \sqsubseteq [os\colon dos]$. Consequently, we can replace $[os\colon \sim unix]$ in S by $\top$, since $[os\colon \sim unix] \sqcap T = \top \sqcap T = T$ holds. Likewise, we can replace $[user\colon tom]$ by $\bot$, since $[user\colon tom] \sqcap T = \bot \sqcap T = \bot$ holds. We obtain

$$\begin{aligned} U &= S \sqcap T \\ &= [\top, user\colon lisa] \sqcap [os\colon dos, user\colon \sim tom] \end{aligned}$$

which feature unification simplifies to

$$= [os\colon dos, user\colon lisa] \ .$$

As stated in proposition 14.8 on page 167, partial evaluation replacement always leads to a full consistency check in quasi-linear time if both S and T are simple; for all other cases, the term S can be reduced in size, simplifying a later consistency check through feature unification (as in our example).

We now present the formal definition of *reduce*, a function integrating partial evaluation and fast consistency checking. First, we define a *simplify* function required by *reduce* to propagate new $\top$ and $\bot$ values.

Definition 14.9 (Simplify) Let $simplify(S)$ be a function mapping a feature term to a feature term such that the following holds:

$$\begin{array}{lll} simplify(\top \sqcap S) = S & simplify(\top \sqcup S) = \top & simplify(\sim\top) = \bot \\ simplify(S \sqcap \top) = S & simplify(S \sqcup \top) = \top & simplify(\sim\bot) = \top \\ simplify(\bot \sqcap S) = \bot & simplify(\bot \sqcup S) = S & \\ simplify(S \sqcap \bot) = \bot & simplify(S \sqcup \bot) = S & \end{array} \tag{14.11}$$

and, for all other cases,

$$simplify(S) = S \ . \tag{14.12}$$

□

The *reduce* function performs the actual replacement, following the proof of proposition 14.8 on page 167.

Definition 14.10 (Reduce) Let $reduce(S,T)$ be a function mapping two feature terms S and T to another feature term such that the following holds:

$$reduce(S, T_1 \sqcap T_2) = reduce\big(reduce(S,T_1), T_2\big) \tag{14.13}$$
$$reduce(S_1 \sqcap S_2, T) = simplify\big(reduce(S_1,T) \sqcap reduce(S_2,T)\big) \tag{14.14}$$
$$reduce(S_1 \sqcup S_2, T) = simplify\big(reduce(S_1,T) \sqcup reduce(S_2,T)\big) \tag{14.15}$$
$$reduce(\sim S, T) = simplify\big(\sim reduce(S,T)\big) \tag{14.16}$$
$$reduce(f{:}S, f{:}T) = f{:}simplify\big(reduce(S,T)\big) \tag{14.17}$$

as well as

$$\begin{array}{lll} reduce(S,S) = \top & reduce(f\!\uparrow, a) = \top & reduce(f{:}S, a) = \bot \\ reduce(a, f{:}T) = \bot & reduce(f\!\uparrow, f{:}T) = \bot & reduce(f{:}S, f\!\uparrow) = \bot \\ reduce(a,b) = \bot & reduce(a, \sim a) = \bot & \end{array} \tag{14.18}$$

and, for all other cases,

$$reduce(S,T) = S \ . \tag{14.19}$$

□

In definition 14.10, (14.13) and (14.14) reflect the recursive descent of (14.8) and (14.9), respectively. Equations (14.15), (14.16) and (14.17) descend along unions, complements and (common) feature paths. The remaining equations in (14.18) either determine inconsistencies for non-composed cases, resulting in $\bot$, or simplify subterms of S by replacing them with $\top$.

Obviously, the term computed by $reduce(S,T)$ is not larger than S. *reduce* may thus be used as general reduction step before using feature unification. In an intersection $S \sqcap T$, we can replace S by $reduce(S,T)$ while preserving validity:

Proposition 14.11 For any two feature terms S and T, the equation

$$S \sqcap T = \mathit{reduce}(S,T) \sqcap T \tag{14.20}$$

holds.

PROOF. We show that (14.20) holds via structural induction. We begin with the non-composed cases in (14.18) and (14.19); assuming that these hold, we continue with the composed cases. Without loss of generality, we use a simpler definition of *simplify*, namely $\mathit{simplify}(S) = S$.

1. We show that (14.20) holds for the non-composed cases by showing that both $S \sqcap T \sqsubseteq \mathit{reduce}(S,T) \sqcap T$ and $S \sqcap T \sqsupseteq \mathit{reduce}(S,T) \sqcap T$ hold.

   (a) We begin with $S \sqcap T \sqsubseteq \mathit{reduce}(S,T) \sqcap T$. Due to (8.4), this is equivalent to $(S \sqcap T) \sqcap {\sim}\big(\mathit{reduce}(S,T) \sqcap T\big) = \bot$. Now let U be defined as $U = (S \sqcap T) \sqcap {\sim}\big(\mathit{reduce}(S,T) \sqcap T\big) = S \sqcap T \sqcap \big({\sim}\mathit{reduce}(S,T) \sqcup {\sim}T\big) = S \sqcap T \sqcap {\sim}\mathit{reduce}(S,T)$. For the cases in (14.18) and (14.19), showing that $U = \bot$ holds is trivial.

   (b) The next step is to show that $S \sqcap T \sqsupseteq \mathit{reduce}(S,T) \sqcap T$ holds. Due to (8.4), this is equivalent to ${\sim}(S \sqcap T) \sqcap \mathit{reduce}(S,T) \sqcap T = \bot$. This time, let U be defined as $U = {\sim}(S \sqcap T) \sqcap \mathit{reduce}(S,T) \sqcap T = ({\sim}S \sqcup {\sim}T) \sqcap \mathit{reduce}(S,T) \sqcap T = {\sim}S \sqcap \mathit{reduce}(S,T) \sqcap T$. Again, $U = \bot$ holds for all cases in (14.18) and (14.19).

2. We continue with the composed cases. Assume that (14.20) holds for some feature terms S, T_1, and T_2. Let $T = T_1 \sqcap T_2$. Then, using (14.13), we obtain $S \sqcap T = S \sqcap (T_1 \sqcap T_2) = (S \sqcap T_1) \sqcap T_2 = \big(\mathit{reduce}(S,T_1) \sqcap T_1\big) \sqcap T_2 = \big(\mathit{reduce}(S,T_1) \sqcap T_2\big) \sqcap T_1 = \mathit{reduce}\big(\mathit{reduce}(S,T_1),T_2\big) \sqcap T_2 \sqcap T_1 = \mathit{reduce}(S, T_1 \sqcap T_2) \sqcap T_1 \sqcap T_2 = \mathit{reduce}(S,T) \sqcap T$. It follows that (14.20) holds for $T = T_1 \sqcap T_2$ as well.

3. Assume that (14.20) holds for some feature terms S_1, S_2, and T. Let $S = S_1 \sqcap S_2$. Then, using (14.14), we have $S \sqcap T = S_1 \sqcap S_2 \sqcap T = (S_1 \sqcap T) \sqcap (S_2 \sqcap T) = \big(\mathit{reduce}(S_1,T) \sqcap T\big) \sqcap \big(\mathit{reduce}(S_2,T) \sqcap T\big) = \big(\mathit{reduce}(S_1,T) \sqcap \mathit{reduce}(S_2,T)\big) \sqcap T = \mathit{reduce}(S_1 \sqcap S_2,T) \sqcap T = \mathit{reduce}(S,T) \sqcap T$. Consequently, (14.20) holds for $S = S_1 \sqcap S_2$ as well.

4. Assume that (14.20) holds for some feature terms S_1, S_2, and T. Let $S = S_1 \sqcup S_2$. Then, using (14.15), we have $S \sqcap T = (S_1 \sqcup S_2) \sqcap T = (S_1 \sqcup T) \sqcap (S_2 \sqcup T) = \big(\mathit{reduce}(S_1,T) \sqcap T\big) \sqcup \big(\mathit{reduce}(S_2,T) \sqcap T\big) = \big(\mathit{reduce}(S_1,T) \sqcup \mathit{reduce}(S_2,T)\big) \sqcap T = \mathit{reduce}(S_1 \sqcup S_2,T) \sqcap T = \mathit{reduce}(S,T) \sqcap T$. Consequently, (14.20) holds for $S = S_1 \sqcup S_2$ as well.

5. Assume that (14.20) holds for some feature terms S' and T. Let $S = {\sim}S'$. Then, using (14.16), we have $S \sqcap T = {\sim}S' \sqcap T = ({\sim}S' \sqcup {\sim}T) \sqcap T = {\sim}(S' \sqcap T) \sqcap T = {\sim}\big(\mathit{reduce}(S', T) \sqcap T\big) \sqcap T = \big({\sim}\mathit{reduce}(S', T) \sqcup {\sim}T\big) \sqcap T = {\sim}\mathit{reduce}(S', T) \sqcap T = \mathit{reduce}({\sim}S', T) \sqcap T = \mathit{reduce}(S, T) \sqcap T$. It follows that (14.20) holds for $S = {\sim}S'$ as well.

6. Assume that (14.20) holds for some feature terms S' and T'. Let f be some feature and let $S = f{:}S'$ and $T = f{:}T'$. Then, using (14.17), we have $S \sqcap T = f{:}S' \sqcap f{:}T' = f{:}(S' \sqcap T') = f{:}\big(\mathit{reduce}(S', T') \sqcap T'\big) = \big(f{:}\mathit{reduce}(S', T')\big) \sqcap (f{:}T') = \mathit{reduce}(f{:}S', f{:}T') \sqcap (f{:}T') = \mathit{reduce}(S, T) \sqcap T$. Consequently, (14.20) holds for $S = f{:}S'$ and $T = f{:}T'$ as well.

Since (14.20) holds for all non-composed feature terms as well as for all composed feature terms, it holds for all feature terms, which was to be shown. □

As a result of proposition 14.11, we can apply *reduce* as a reduction step before any feature unification. Moreover, if the conditions of proposition 14.8 on page 167 are met, *reduce* determines consistency of $S \sqcap T$ in quasi-linear time:

Corollary 14.12 Let S and T be simple, consistent, and variable-free feature terms; let neither S nor T contain agreements or disagreements. Then,

1. $S \sqcap T$ is consistent iff $\mathit{reduce}(S, T)$ is consistent; and

2. $\mathit{reduce}(S, T)$ requires quasi-linear time.

PROOF. The terms S and T meet the conditions of proposition 14.8 on page 167. Hence, consistency of S and T can be decided in quasi-linear time. Applying *reduce* compares each pair of subterms S_i and T_j, as specified in (14.10); through the propagation of $\bot$ values in *simplify*, the result of *reduce* is consistent iff $S \sqcap T$ is consistent. No further time complexity is added by *reduce*. □

14.8 Two Reduction Examples

All of the strategies presented in this chapter can be combined into one single procedure, choosing the least cost method wherever appropriate. As an example, reconsider the editor example from figure 10.1 on page 104. The features of the entire configuration are described as

$$\mathit{editor} = \mathit{os} \sqcap \mathit{screen\text{-}type} \sqcap \mathit{screen\text{-}device} \ ,$$

where *os*, *screen-type*, and *screen-device* are defined as

$$\begin{aligned} \mathit{os} = {} & \big[\mathit{os}{:}\,\mathit{dos}, \mathit{screen\text{-}type}{:}\,\{\mathit{ega}, \mathit{tty}\}, \mathit{concurrent}{:}\,\mathit{false}\big] \\ & \sqcup \big[\mathit{os}{:}\,\mathit{unix}, \mathit{screen\text{-}type}{:}\,\{\mathit{x11}, \mathit{news}, \mathit{tty}\}\big] \end{aligned}$$

$$\begin{aligned} \mathit{screen\text{-}type} = {} & \big[\mathit{screen\text{-}type}{:}\,\mathit{ega}, \mathit{screen\text{-}data}{:}\,\mathit{bitmap}\big] \\ & \sqcup \big[\mathit{screen\text{-}type}{:}\,\mathit{tty}, \mathit{screen\text{-}data}{:}\,\mathit{ascii}\big] \\ & \sqcup \big[\mathit{screen\text{-}type}{:}\,\mathit{x11}, \mathit{screen\text{-}data}{:}\,\mathit{bitmap}\big] \\ & \sqcup \big[\mathit{screen\text{-}type}{:}\,\mathit{news}, \mathit{screen\text{-}data}{:}\,\{\mathit{postscript}, \mathit{bitmap}\}\big] \end{aligned}$$

$$\begin{aligned} \mathit{screen\text{-}device} = {} & \big[\mathit{screen\text{-}device}{:}\,\mathit{dumb}, \mathit{data}{:}\,D, \mathit{screen\text{-}data}{:}\,D\big] \\ & \sqcup \big[\mathit{screen\text{-}device}{:}\,\mathit{ghostscript}, \mathit{data}{:}\,\mathit{postscript}, \\ & \quad \mathit{screen\text{-}data}{:}\,\mathit{bitmap}, \mathit{concurrent}{:}\,\mathit{true}\big] \quad . \end{aligned}$$

Let us identify the configurations in $T = [\mathit{os}{:}\,\mathit{unix}, \mathit{screen\text{-}type}{:}\,\mathit{x11}]$. For this purpose, we create a subset of *editor*, namely $\mathit{editor} \sqcap T$. Applying Smolka's feature unification alone, as discussed in section 14.1, requires *editor* to be transformed into DNF form. Since *os* comes in five variants, *screen-type* in four variants, and *screen-device* in two variants, this means a term with $5 \times 4 \times 2 = 40$ conjuncts, which would again be multiplied with each alternative in T. Due to the procedures discussed in the previous sections, much fewer steps are required. First, we decompose the problem $\mathit{editor} \sqcap T$ into three subproblems $\mathit{editor} \sqcap T = (\mathit{os} \sqcap T) \sqcap (\mathit{screen\text{-}type} \sqcap T) \sqcap (\mathit{screen\text{-}device} \sqcap T)$.

1. The selection $\mathit{os} \sqcap T$ can be done by reduction:

$$\begin{aligned} \mathit{os} \sqcap T &= \mathit{reduce}(\mathit{os}, T) \sqcap T \\ &= \mathit{reduce}\big(\mathit{reduce}\big(\mathit{os}, [\mathit{os}{:}\,\mathit{unix}]\big), [\mathit{screen\text{-}type}{:}\,\mathit{x11}]\big) \sqcap T \end{aligned}$$

   Evaluating $\mathit{reduce}\big(\mathit{os}, [\mathit{os}{:}\,\mathit{unix}\big)$ yields

$$\begin{aligned} \mathit{reduce}\big(\mathit{os}, [\mathit{os}{:}\,\mathit{unix}]\big) = {} & \mathit{reduce}\big([\mathit{os}{:}\,\mathit{dos}, \mathit{screen\text{-}type}{:}\,\{\mathit{ega}, \mathit{tty}\}, \\ & \qquad \mathit{concurrent}{:}\,\mathit{false}], [\mathit{os}{:}\,\mathit{unix}]\big) \\ & \sqcup \mathit{reduce}\big([\mathit{os}{:}\,\mathit{unix}, \\ & \qquad \mathit{screen\text{-}type}{:}\,\{\mathit{x11}, \mathit{news}, \mathit{tty}\}], \end{aligned}$$

$$\begin{aligned} & [os\colon unix]\big) \\ &= \bot \sqcup \big[screen\text{-}type\colon \{x11, news, tty\}\big] \\ &= \big[screen\text{-}type\colon \{x11, news, tty\}\big] \end{aligned}$$

Reducing each of the remaining alternatives yields

$$\begin{aligned} os \sqcap T &= (\top \sqcup \bot \sqcup \bot) \sqcap T \\ &= [os\colon unix, screen\text{-}type\colon x11] \ . \end{aligned}$$

2. The selection $screen\text{-}type \sqcap T$ is also done by reduction. Since the *os* feature does not occur in *screen-type*, it suffices to perform the reduction $reduce\big(screen\text{-}type, [screen\text{-}type\colon x11]\big)$. Reducing each of the four alternatives leaves only

$$\begin{aligned} screen\text{-}type \sqcap T &= (\bot \sqcup \bot \sqcup [screen\text{-}data\colon bitmap] \sqcup \bot) \sqcap T \\ &= [os\colon unix, screen\text{-}type\colon x11, screen\text{-}data\colon bitmap] \ . \end{aligned}$$

3. The selection $screen\text{-}device \sqcap T$ is trivial, since *screen-device* and T have no common features and are thus orthogonal:

$$screen\text{-}device \sqcap T = screen\text{-}device \sqcap T$$

4. We now compute $os \sqcap T$, $screen\text{-}type \sqcap T$, and $screen\text{-}device \sqcap T$. The intersection of $os \sqcap T$ and $screen\text{-}type \sqcap T$ can be trivially computed by reduction: $reduce(os \sqcap T, screen\text{-}type \sqcap T) = \top$ holds and thus

$$\begin{aligned} (os \sqcap T) \sqcap (screen\text{-}type \sqcap T) &= \top \sqcap (screen\text{-}type \sqcap T) \\ &= (screen\text{-}type \sqcap T) \ . \end{aligned}$$

5. The final step is the intersection of $(screen\text{-}type \sqcap T)$ and $(screen\text{-}device \sqcap T)$. Since one of the alternatives of *screen-type* contains variables, we cannot use reduction for this alternative: full-fledged feature unification is required, instantiating the variable *D* to *bitmap*.

$$\begin{aligned} & (screen\text{-}type \sqcap T) \sqcap (screen\text{-}device \sqcap T) \\ & \quad = [os\colon unix, screen\text{-}type\colon x11, screen\text{-}device\colon dumb, \\ & \qquad data\colon bitmap, screen\text{-}data\colon bitmap] \\ & \qquad \sqcup [os\colon unix, screen\text{-}type\colon x11, screen\text{-}device\colon ghostscript, \\ & \qquad data\colon postscript, screen\text{-}data\colon bitmap, concurrent\colon true] \end{aligned}$$

This final term also identifies the entire configuration *editor* $\sqcap T$. Rather than invoking feature unification for 40 conjuncts, it sufficed to invoke it for one single conjunct. The entire selection, including the consistency check of the resulting configuration, required only one reduction call for each component version, as well as two reduction calls for determining consistency.

As another example, consider the revision graph in figure 11.1 on page 114. As stated in (11.3), the revision graph is expressed by

$$\begin{aligned} R = {} & (\nabla_2 \sqcup \Delta_1) \sqcap (\nabla_3 \sqcup \Delta_1) \sqcap (\nabla_4 \sqcup \Delta_3) \sqcap (\nabla_5 \sqcup \Delta_2) \sqcap (\nabla_5 \sqcup \Delta_4) \\ & \sqcap (\nabla_6 \sqcup \Delta_4) \sqcap (\nabla_2 \sqcup \nabla_3 \sqcup \Delta_5) \sqcap (\nabla_2 \sqcup \nabla_6) \quad , \end{aligned}$$

where we use $\sqcup$ instead of $\rightarrow$ to express implications.

Let us assume we wish to retrieve the revision R_3, identified by a selection term $S = \Delta_3 \sqcap \nabla_4$. We determine the selection $R_3 = R \sqcap S$. Invoking $\mathit{reduce}(R,S)$ yields

$$\mathit{reduce}(R,S) = (\nabla_2 \sqcup \Delta_1) \sqcap \Delta_1 \sqcap \top \sqcap \nabla_5 \sqcap \nabla_6 \sqcap (\nabla_2 \sqcup \Delta_5) \sqcap (\nabla_2 \sqcup \nabla_6) \quad .$$

which is already a lot smaller than R. Resolving the intersections in $\mathit{reduce}(R,S)$ by calling *reduce* with Δ_1, ∇_5, and ∇_6, respectively, yields

$$\mathit{reduce}(R,S) = \Delta_1 \sqcap \nabla_5 \sqcap \nabla_6 \sqcap \nabla_2$$

which completes the term R_3 to

$$\begin{aligned} R_3 &= R \sqcap S = \mathit{reduce}(R,S) \sqcap S \\ &= \Delta_1 \sqcap \nabla_2 \sqcap \Delta_3 \sqcap \nabla_4 \sqcap \nabla_5 \sqcap \nabla_6 \end{aligned}$$

Again, had we used feature unification alone, converting R into DNF would have given us a term with $2^7 \times 3 = 384$ conjuncts. Instead, four applications of *reduce*, each with quasi-linear time complexity, sufficed to determine R_3.

14.9 Conclusion

Deciding inconsistency of feature terms is NP-complete. This implies that the following problems are NP-complete, too:

- Is a version part of a specific selection set?
- Is a configuration consistent with respect to the features of its components?

In this chapter, we have presented specialized *deductive shortcuts* exist that show much better complexity for special cases. The problem of deciding consistency can be broken down in smaller subproblems if the feature term breaks down into *orthogonal parts,* that is, parts without common features or variables. The technique of *partial evaluation* leads to efficient decision of consistency for simple feature terms.

While orthogonality is an important property for the separation of concerns, partial evaluation is an important shortcut for version selection. In fact, the common SCM version selection schemes discussed in section 7.3 can all be implemented in quasi-linear time complexity:

Simple selection terms. If the version selection term is simple, consistent, and variable-free, consistency checking and thus version selection has quasi-linear time complexity. This is the "strong identification, weak selection" scheme, as realized in CPP.

Simple version identification terms. If the version identification terms are simple, consistent, and variable-free, consistency checking and thus version selection also has quasi-linear time complexity. This is the "strong selection, weak identification" scheme, as realized in JASON and other attribute-oriented SCM systems.

We see that despite the generality of version sets and feature unification, common SCM versioning schemes can still be realized efficiently. But to be absolutely convincing, this claim requires more than a proof—it requires a working prototype. This is what we have built, and this is what we present in part four.

We remark that certain worst-case complexity results
are not considered to be a problem,
because the examples are pathological
and do not arise in practice.

— ALEX BORGIDA, Description Logics are not just for the `Flightless-Birds`

Part Four

Applications

Chapter 15

A SCM Environment

In software engineering, proposing a new design alone does not suffice. As Lukowicz et al. *state in* [LHPT95]*,*

> *Such designs must be judged by whether they increase our knowledge about what are useful and cost-effective problem solutions. In most cases, objective judgement can only be achieved on the basis of reproducible experiments.*

For this purpose, we have implemented the version set model in an experimental SCM system, called ICE for Incremental Configuration Environment. *This chapter gives a general overview about the architecture and components of ICE.*

15.1 The Properties of ICE

The basic properties of ICE are those of the version set model; notably, ICE supports the integration of versioning dimensions, consistency checking in abstract configurations, and tolerates ambiguities at all SCM levels. Other features of ICE include:

Version sets as first-class objects. In ICE, every component and every configuration is treated as set of possible versions, where an unambiguous item is just the special case of a singleton set. Version sets are represented as individual entities and can be examined and manipulated as a whole, using the well-known CPP representation as discussed in section 2.6.1; likewise, all version specifications are given as CPP expressions—that is, boolean C expressions.

Transparent version set access. For integration into common software development environments, ICE makes version sets accessible through a virtual file system called FFS for *featured file system*. Version sets are accessed explicitly by appending a version specification to file and directory names. Implicit version set access is realized by changing the current directory version.

Incremental version selection. Many software development tools require that items be unambiguous. ICE provides incremental and interactive disambiguating facilities, allowing users to explore the version space. For each configuration, ICE lists possible features and values that constrain the version space while keeping consistency. Users can select these feature values and refine their selection incrementally until the selection is unambiguous.

Intensional system construction. ICE realizes a MAKE tool that acts like an ordinary MAKE, but with built-in version set support. ICE MAKE deduces the features of derived components and tolerates ambiguity in dependency descriptions, such that entire systems can be built and configured just by stating a few target features. As described in section 4.6, ICE MAKE determines whether required components have been built identically in another configuration and reuses them across versions wherever possible. A full description of ICE MAKE can be found in [Bra96].

Revision and workspace management. At the protocol layer, ICE provides facilities to create revisions and to propagate changes, realizing the optimistic cooperation strategy as discussed in section 13.3.3. A textual merging algorithm enhanced for version sets realizes change integration for arbitrary version sets. The resulting TWICE tool is specified in [Men96].

ICE is part of the inference-based software development environment NORA[1]. NORA aims at utilizing inference technology in software tools; concepts and preliminary results can be found in [FKS95, KS94, Lin95, Sne96].

15.2 Using Industry Standards

In section 15.1, we have seen that ICE relies on existing industry standards wherever possible: component versions are accessed as files, multiple versions are represented in CPP format, the system model comes as an ordinary MAKE file. The choice to use existing representations instead of designing own, maybe better, representations, were made for three reasons.

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

Economy in use. Using industry standards allows for smooth integration of ICE into real-world software development environments. Existing documents, such as MAKE files or CPP-maintained source files, can be reused. End users familiar with MAKE and CPP need not learn new paradigms or representations, just some bits of additional functionality. Users can switch back to their original tools if ICE does not satisfy them.

Economy in development. Using industry standards facilitates the development of ICE. Syntax and semantics of MAKE, CPP, or file systems are well-documented and well-understood among developers. Rather than to coordinate, document, implement, and debug basic SCM functionality as realized in these tools, developers can focus upon the new functionality. More even, mature implementations are available that can be reused and extended.

Economy in concepts. As an SCM foundation, the version set model should integrate and unify existing SCM concepts, rather than introducing new ones. Hence, ICE need not rely on new representations for new concepts, but rather demonstrate how existing representations are interpreted and reused under the version set model.

15.3 A Layered Architecture

As discussed in section 6.6, future SCM systems should be decomposed into three layers—primitives, protocol, and policy—, each providing a specific set of SCM services. The architecture of ICE can be divided into these three layers; an additional *foundation layer* realizes primitives for handling version sets, as discussed in part three.

Foundation layer. The foundation layer is not accessible to end users. It provides the basic functionality useful for realizing user-level SCM services. This includes support for maintaining feature terms, access to the inference engine, and facilities for reading, writing, and manipulating simple version sets.

Primitives layer. The primitives layer embeds ICE into software development environments. The FFS is part of the primitives layer, allowing users and user tools to access and refine version sets. Versions are identified by arbitrary feature terms; feature names have no specific meaning. The FFS realizes access control by maintaining access rights for individual file versions.

Figure 15.1: The ICE service layers

Protocol layer. The protocol layer gives meaning to specific features and provides support for specific SCM tasks and procedures. Revisions and workspaces are handled at this layer, accessing version sets through the FFS. Locking is also handled here, in contrast to [BDFW91], where locking is a service of the primitives layer. Other SCM tools working on version sets can be located at this layer, such as software construction or interactive version selection.

Policy layer. The policy layer uses the services provided at the SCM protocol layer to encode procedures specific to an organization. ICE does not yet provide facilities at this layer.

In the following chapters, we discuss the individual components of ICE, starting with the ICE foundations.

By three methods we may learn wisdom:
First, by reflection, which is noblest;
Second, by imitation, which is easiest;
and third by experience, which is the bitterest.

— CONFUCIUS

Chapter 16

Representing Version Sets

Upon designing ICE, the first problem that arose was the representation and efficient storage of version sets at the SCM primitives 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. Our choice fell on the well-established CPP format, discussed in section 2.6.1. We show how to represent feature terms as CPP expressions, providing users with a familiar syntax to denote version sets.

16.1 A Multi-Version Representation

Upon designing ICE, it was our aim to make version sets transparently accessible to developers, such that they could manipulate several versions at once. We consider a document as a set of related items, where each item is versioned separately. For ordinary text documents, organized as a list of lines, this results in each line being tagged with a feature term S indicating the document version(s) it belongs to.

In figure 16.1 on the following page, we have illustrated such a versioned text. The lines tagged with $\top$ occur in every version of the text. The lines tagged with $[\mathit{author}{:}\mathit{tichy}]$ belong to the *tichy* version only, while the lines tagged with $[\mathit{author}{:}\mathit{dart}]$ are part of the *dart* version. Upon selecting a version S of the document, only those lines are included whose feature term T is consistent with S—that is, where $T \sqcap S$ is consistent.

Since ICE was designed to work with ordinary files, we had to design some representation for tagging lines with feature terms. The most frequently used multi-version representation for ordinary files is the CPP format, as discussed in section 2.6.1. Using CPP-like directives, but with feature terms, we could have

Line	Features
`Configuration`	$\top$
`management is the`	$[author{:}tichy]$
`management is a`	$[author{:}dart]$
`discipline`	$\top$
`of organizing and`	$[author{:}tichy]$
`controlling evolving`	$[author{:}tichy]$
`for controlling`	$[author{:}dart]$
`the evolution of`	$[author{:}dart]$
`systems.`	$\top$

Figure 16.1: Tagging lines with feature terms

used *feature directives* like `#if` ... `#endif` to specify the feature term applying to the enclosed lines. An example is shown in figure 16.2 on the next page on the left side.

But, since we're already using a CPP-like representation, why not use CPP expressions as well? Feature terms and CPP expressions are quite similar: Both support boolean equations and equality, and features in feature term can easily be expressed by CPP variables, which also can have only one value. Also, allowing ICE to read and write CPP files offers the possibility to re-use existing CPP representations and to interact with tools requiring CPP representation.

Consequently, we chose the CPP representation as standard representation for version sets in ICE. The resulting file is shown on the right side of figure 16.2 on the facing page.

16.2 Representing Feature Terms

ICE allows users that know feature logic to enter feature terms directly, using a straight-forward ASCII representation. But usually, users are expected to use the more familiar, well-understood CPP representation. In the following, we discuss the mapping of CPP expressions to feature terms and vice versa, as summarized in table 16.1 on page 186.

Set Operations. ICE uses CPP boolean operators for the set operations of feature logic—that is, `&&` for the intersection ($\sqcap$), `||` for union ($\sqcup$), and `!` for complement ($\sim$).

Lines with feature directives

```
Configuration
#if [author:  tichy]
management is the
#endif
#if [author:  dart]
management is a
#endif
discipline
#if [author:  tichy]
of organizing and
controlling evolving
#endif
#if [author:  dart]
for controlling
the evolution of
#endif
systems.
```

Lines with CPP directives

```
Configuration
#if author == tichy
management is the
#endif
#if author == dart
management is a
#endif
discipline
#if author == tichy
of organizing and
controlling evolving
#endif
#if author == dart
for controlling
the evolution of
#endif
systems.
```

Figure 16.2: Multiple versions in one file with feature and CPP directives

Selection. A selection is represented by the CPP operator ==; that is, the feature term $f{:}S$ becomes, as CPP expression, `f == S`. The CPP operator `!=` is used for negated feature values; `author != lisa` stands for the feature term *author*:$\sim$*lisa*.

Atoms. Besides identifiers like `lisa`, ICE allows arbitrary C literals as atoms—that is, strings (`"lisa"`), characters (`'l'`), integers (`42`) and floating point numbers (`4.711e+3`), following the C standard [ISO90].

Agreement. Disagreement. The == and != operators can also be used for agreements and disagreements. This introduces an ambiguity in CPP expressions, because identifiers may be interpreted as features or atoms. To distinguish between selection and agreement or disagreement, and arithmetic expressions involving equality, the following rules are used. In an expression S `==` T (S `!=` T),

1. the expression is an agreement (disagreement) if

Abstract syntax	ASCII representation	CPP representation
$\top$ (also $[]$)	`[]`	`!0`
$\bot$ (also $\{\}$)	$\{\}$	`0`
a	`a`	`a`
x	`X`	(see section 16.2)
$f{:}S$	f`:` S	f `==` S
$f{:}\sim S$	f`: ~`S	f `!=` S
$f{:}\sim 0$	f`: ~0`	f
$f{:}\top$	f`: []`	`defined` f
$f\uparrow$	f `^`	`!defined` f
$f\downarrow g$	f `=` g	f `==` g
$f\uparrow g$	f `^` g	f `!=` g
$\sim S$	`~`S	`!`S
$S\sqcap T$ (also $[S,T]$)	`[`S`, `T`]`	S `&&` T
$S\sqcup T$ (also $\{S,T\}$)	$\{S$`, `$T\}$	S `||` T
$S\to T$	(see section 16.2)	(see section 16.2)
$S\leftrightarrow T$	(see section 16.2)	(see section 16.2)
$\exists x(S)$	(see section 16.2)	(see section 16.2)

Table 16.1: Representing feature terms in ASCII and as CPP expressions

   (a) S and T are identifiers,
   (b) T begins with an upper-case letter.

2. Otherwise, the expression is a selection with T ($\sim T$) as value if S is an identifier.

3. Otherwise, the expression is an arithmetic expression.

Multiple Selections. Variables. To avoid further ambiguities, feature terms with multiple selections like $f{:}g{:}S$ and variables like X cannot be mapped to CPP expressions. Such feature terms can be embedded in the CPP representation by enclosing their ASCII representation in square brackets. For example, the term $[f{:}a, g{:}h{:}X, i{:}X]$ becomes, in CPP representation,

```
f == a && [g: h: X] && i == [X]
```

Embedding of CPP expressions in the ASCII representation is not supported.

Top and Bottom. CPP expressions, like C expressions, are arithmetic by nature: the boolean values of true and false are expressed by zero and non-zero values, respectively. Consequently, we use `0` to express the feature term $\bot$ and `!0` to express $\sim\bot$ or $\top$.

To avoid ambiguities between representing $\top$ and the negated atom 0, we use the CPP expression `defined` f for the feature term $f\colon\top$. When $\top$ does not occur as feature value, it can usually be eliminated from set expressions. Divergence $f\uparrow$ becomes `!defined` f.

In CPP expressions, a single identifier `x` occurring in a boolean formula is interpreted like `(x != 0)`; ICE reflects this interpretation by mapping the feature term $f\colon\sim 0$ to the CPP expression f.

Implications. Implications $S \to T$ do not have an equivalent in the ASCII or the CPP representation of feature terms. They can be represented using the alternate forms $\sim S \sqcup T$—that is, {`~`S`, `T`}` in the ASCII representation and `!`S` || `T in the CPP representation.

Equivalences. Like implications, equivalences $S \leftrightarrow T$ must be represented using an alternate form. Since $S \leftrightarrow T = (S \sqcap T) \sqcup (\sim S \sqcap \sim T)$ holds, the form {`[`S`, `T`], [~`S`, ~`T`]`} is a possible ASCII representation; the CPP representation becomes `(`S` && `T`) || (!`S` && !`T`)`.

Quantifiers. Quantifiers $\exists x(S)$ are not supported by ICE. They have neither an ASCII representation nor a CPP representation.

ISO keywords. In compliance with the forthcoming C++ standard [Str94], ICE recognizes the keywords `and`, `or`, `not`, and `not_eq` instead of `&&`, `||`, `!`, and `!=`. ICE may also be instructed to generate these keywords.

Other CPP expressions. All CPP expressions that cannot be converted into a feature term using the rules above, are treated by ICE as a single atom. We call these expressions *arithmetic expressions.*

16.3 Syntax and Semantics of CPP Directives

16.3.1 Specifying Line Features

Besides the simple `#if` ... `#endif` construct, ICE handles all CPP directives related to conditional inclusion, improving the readability of multi-version files.

Each block of the text is read within a certain *context,* a feature term that determines the features of the line; one also says that the context *governs* the line. CPP directives like `#if` may be used to narrow this context for the enclosed lines.

`#if` ... `#elif` ... `#else` ... `#endif`. The `#if` directive occurs in the general form

```
#if S_0
t_0
#elif S_1
t_1
#elif S_2
t_2
⋮
#elif S_n
t_n
#else
t_{n+1}
#endif
```

where the `#elif` and `#else` directives and the following text blocks t_i are optional. Let C be the context of the entire `#if` ... `#endif` form. Each text block t_i is then interpreted with the context T_i defined as

$$\begin{aligned} T_i &= C \sqcap {\sim}S_0 \sqcap {\sim}S_1 \sqcap \cdots \sqcap {\sim}S_{i-1} \sqcap S_i \\ &= C \sqcap \prod_{0 \le j < i} {\sim}S_j \sqcap S_i \ , \end{aligned} \tag{16.1}$$

where S_{n+1} is defined as $S_{n+1} = \top$.

Figure 16.3 on the facing page gives an example of using `#if` ... `#endif`.

`#ifdef`. A control line of the form

```
#ifdef f
```

is equivalent to

```
#if defined f
```

`#ifndef`. A control line of the form

```
#ifndef f
```

Line	Features
`// Init random seed`	$\top$
`#if HAVE_SRAND`	
`// srand() available`	$[\mathit{have\text{-}srand}: \sim 0]$
`#if defined USE_SRAND`	
`srand(time);`	$[\mathit{have\text{-}srand}: \sim 0, \mathit{use\text{-}srand}: \top]$
`#else`	
`// No srand()`	$[\mathit{have\text{-}srand}: \sim 0, \mathit{use\text{-}srand}\uparrow]$
`#endif`	
`#elif HAVE_SRANDOM`	
`srandom(time);`	$[\sim\mathit{have\text{-}srand}: \sim 0, \mathit{have\text{-}srandom}: \sim 0]$
`#else`	
`// No random seed`	$[\sim\mathit{have\text{-}srand}: \sim 0, \sim\mathit{have\text{-}srandom}: \sim 0]$
`#endif`	

Figure 16.3: Interpretation of `#if` directives

is equivalent to

```
#if !defined f
```

16.3.2 Specifying File Features

CPP directives may also be used to specify non-existent versions, and thus to define the features of the entire file. For instance, by stating that the version subset $[\mathit{tested}: \top]$ does not exist, the features of the entire file become $\sim[\mathit{tested}: \top] = [\mathit{tested}\uparrow]$.

`#error`. A control line of the form

```
#error token-string
```

in a context C expresses that the file does not exist in the context C; in other words, the features of the file are a subset of $\sim C$.

`#error` directives are useful for specifying the features of a file explicitly. As an example, the CPP directives

```
#if !(SCREEN_TYPE == ega) \
    || !(SCREEN_DATA == bitmap)
```

```
#error
#endif
```

specify the features of the file as $[\mathit{screen\text{-}type}\colon \mathit{ega}, \mathit{screen\text{-}data}\colon \mathit{bitmap}]$. No subset of $\sim[\mathit{screen\text{-}type}\colon \mathit{ega}] \sqcup \sim[\mathit{screen\text{-}data}\colon \mathit{bitmap}]$ exists.

`#define`. Using C encoding, `#define` may be used to specify the features of a file. ICE can be instructed to interpret a control line of the form

```
#define f
```

as

```
#if !defined f
#error
#endif
```

and to interpret a control line of the form

```
#define f T
```

as

```
#if !(f == T)
#error
#endif
```

By default, ICE ignores `#define` directives.

`#undef`. Using C encoding, `#undef` may be used to specify the features of a file. ICE can be instructed to interpret a control line of the form

```
#undef f
```

as

```
#if defined f
#error
#endif
```

By default, ICE ignores `#undef` directives.

16.3.3 Miscellaneous Directives

ICE also recognizes the CPP `#line` directive, which is useful for diagnostics. The `#pragma` directive, followed by the keyword `ice`, is used by ICE-specific extensions to the CPP representation.

`#line`. A `#line` directive in the form

```
#line constant
```

or

```
#line constant "filename"
```

sets the current line number to *constant*, for the purpose of error diagnostics. If present, the name of the current file is set to *filename*.

`#pragma`. A `#pragma` directive followed by the token `ice` is recognized as ICE-specific directive. Any `#pragma` directives not followed by `ice` are ignored.

ICE recognizes the following `#pragma ice` directives:

`#pragma ice config`. A control line of the form

```
#pragma ice config S
```

is equivalent to

```
#if !S
#error
#endif
```

`#pragma ice config` is obsolete; `#error` should be used instead.

`#pragma ice encoding`. A control line of the form

```
#pragma ice encoding e
```

sets the subsequent encoding of the file to the encoding specified by *e* (see section 16.4 for details on file encodings). Possible values of *e* and the resulting encodings are shown in table 16.2 on the next page.

Token	Encoding	Token	Encoding
`asis`	As-is	`text`	Text
`c` or `C`	C	`binary`	Binary

Table 16.2: Encoding tokens

16.4 File Encodings

The CPP format, as defined in [ISO90], was designed for C and C++ programs. Using the CPP format for arbitrary files requires some slight changes to the CPP encoding, depending on the file to be processed. ICE knows four file encodings: C encoding, Text encoding, Binary encoding, and "As-is" encoding.

C Encoding. In C encoding, CPP directives are read and interpreted according to the ISO C standard [ISO90]. There may be arbitrary white space before and after the # character; and the # character may also be replaced by the ISO C *trigraph sequence* `??=` or by the *digraph sequence* `%:` from the proposed C++ standard [Str94]. CPP directives enclosed by C comments `/* ... */` are ignored. CPP directives may extend across multiple lines: the character \ followed by a newline is ignored, allowing for *continuation lines.* C and C++ comments (`//` to the end of a line) are recognized.

C encoding is useful for processing CPP-managed source files. Figure 16.4 gives an example of a file in C encoding.

```
#if HAVE_ATHENA_WIDGETS
  #if HAVE_X11_XAW_FORM_H
    #include <X11/Xaw/Form.h>
  #endif
#endif
```

Figure 16.4: A program file in C encoding

The second line in figure 16.4 is interpreted as CPP directive although preceded by white space.

Text Encoding. Text encoding is a restricted form of C encoding. The # character must be the first in the line; no white space before or after the # character is allowed. The # character may not be replaced by a trigraph or digraph; C comments around directives are ignored. Continuation lines are still allowed; C and C++ comments may be used within a CPP directive.

Text encoding is useful for general text files. In figure 16.5, we see an example of a multi-version Makefile in text encoding.

```
# Sample Makefile
# if we're using GCC, use the -O2 flag
#if CC == gcc
CFLAGS = -O2
#else
CFLAGS = -O
#endif
```

Figure 16.5: A Makefile in text encoding

Using text encoding, the second line is treated as ordinary text as intended. With C encoding, the second line would flag an error, since it would be interpreted as an `#if` directive followed by an invalid CPP expression.

Binary Encoding. In binary encoding, CPP directives are enclosed in square brackets. They may occur anywhere in a file, making this encoding suitable for arbitrary files. Continuation lines and C++ comments are not allowed; C comments may be used.

Figure 16.6 gives an example of a multi-version C++ program in binary encoding.

```
// Initialize [#if d1]PTY[#else]TTY[#endif]
#if USE_[#if d1]PTY[#else]TTY[#endif]
int open_[#if d1]pty[#else]tty[#endif]();
#endif // USE_[#if d1]PTY[#else]TTY[#endif]
```

Figure 16.6: A C++ program file in binary encoding

Obviously, the change δ_1 in figure 16.6 consisted in changing all occurrences of `tty` to `pty`. Note that the CPP directive on the second line is treated as ordinary text, since it is not preceded by a `[` character.

As illustrated in figure 16.6, binary encoding can be used for fine-grained differences in files. The placement of directives influences both size and readability of the text. Instead of placing directives on word boundaries, as in the example, we could also have placed directives on letter boundaries, resulting in the representation shown in figure 16.7 on the next page; the file is smaller, but even less legible.

```
// Initialize [#if d1]P[#else]T[#endif]TY
#if USE_[#if d1]P[#else]T[#endif]TY
int open_[#if d1]p[#else]t[#endif]ty();
#endif // USE_[#if d1]P[#else]T[#endif]TY
```

Figure 16.7: Binary encoding with character boundaries

Placing directives on line boundaries, makes the file larger, but improves readability, as illustrated in figure 16.8.

```
[#if d1]// Initialize PTY
#if USE_PTY
int open_pty();
#endif // USE_PTY
[#else]// Initialize TTY
#if USE_TTY
int open_tty();
#endif // USE_TTY[#endif]
```

Figure 16.8: Binary encoding with line boundaries

Since the character sequence `[#` starts a directive, the special sequence `[##` is used to encode the sequence `[#` itself.

As-is Encoding. This is a simple one: The entire file is read "as is" as one single version, without any encoding.

ICE can also be instructed to determine the encoding of a file dynamically, using a simple heuristic:

1. If the file begins with the character sequence `[#`, binary encoding is used.
2. If the file ends in a newline character and does not contain control characters besides newline and tab characters, text encoding is used.
3. Otherwise, as-is encoding is used.

Using the first alternative, the encoding can be specified explicitly at the beginning of the file, using a `#pragma encoding` directive. For instance, the sequence `[#pragma encoding text]` at the beginning of the file enforces text encoding in the remainder of the file.

If any syntax errors occur during the interpretation of CPP directives, ICE gives a diagnostic and reads the file again, using as-is encoding.

16.5 Implementation Notes

Feature terms are implemented as abstract syntax trees using the *composite* pattern [GHJV94]; each operator in the abstract syntax is represented by a separate class. To minimize the effort for copying feature terms, subtrees are shared wherever possible. Feature terms are accessed through *smart references,* an instance of the *proxy* pattern [GHJV94] implementing a simple reference-count mechanism deleting unreferenced feature trees.

The scanner for CPP files, distinguishing CPP directives from ordinary text, was written directly in C++. CPP expressions are processed by a scanner/parser automatically generated from a LEX token specification and a YACC grammar specification. The 145 rules of the YACC grammar handle both feature terms and CPP expressions.

A processed CPP file is represented internally as a list of text blocks, where each block contains a sequence of characters and the associated feature term. Each block is also associated with lexical details about the indentation, any comments found on CPP directives, whether ISO keywords are used, etc., such that subsequent writing does not change these details. The internal CPP file representation was realized by Lars Düning [Dün94].

16.6 Conclusion

ICE uses CPP expressions to represent feature terms and files enriched with CPP directives to represent version sets. The intent is to give users a familiar, well-understood representation of multiple items in one representation. By supporting various encodings, ICE can interpret existing CPP-managed files (especially C and C++ program files) and represent binary files using a CPP-like encoding.

The primary advantage of the CPP format is that only the differences between versions are specified. An increasing number of differences between versions also implies a larger number of directives. While this is no problem for ICE, it makes the resulting files hard to read for humans. In the following chapter, we discuss techniques to select and change arbitrary version subsets out of a CPP representation, such that users can work upon singleton version sets without any CPP directives.

I didn't like CPP at all, and I still don't like it.

— BJARNE STROUSTRUP, The Design and Evolution of C++

Chapter 17

Handling Version Sets

Having discussed the CPP representation of version sets, we demonstrate how version subsets are selected from CPP files, realizing reading of arbitrary version sets. These subsets can also be changed and merged back into the original file, using a DIFF algorithm to determine a compact representation. Through selection and changing, we can define the effects of usual file operations (read, write, create, remove) on version sets in CPP representation.

17.1 Selecting Version Sets

We show how arbitrary version subsets can be accessed from a version set in CPP file representation. Let F be a CPP file representing all source code versions. 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, its governing feature term C is intersected with the selection term S. If $C \sqcap S$ is inconsistent, the code piece is removed from the selection. If $C \sqcap S = S$, the `#if` directive is removed, because $S \sqsubseteq C$ holds. Otherwise, C is simplified respective to S, using partial evaluation as discussed in section 14.7. The new (smaller) CPP representation can be characterized by S and is written $F[S] = F \sqcap S$ (obviously, $F = F[\top]$ holds).

17.1.1 A Variant Example

Figure 17.1 on the next page shows three subset selections from the source code of `xload`, a tool to display the current system load. `xload` is available for several architectures, each with a different method to determine the system load. Consequently, each architecture is identified by an individual CPP variable.

From top to bottom, figure 17.1 shows

xload$[os: unix]$

```
InitLoadPoint()
{
  extern void nlist();
#if defined(AIXV3) && !defined(hcx)
  knlist(namelist, 1, sizeof(struct nlist));
#else
  nlist(KERNEL_FILE, namelist);
#endif
#ifdef hcx
  if (namelist[LOADAV].n_type == 0 &&
#else
  if (namelist[LOADAV].n_type == 0 ||
#endif
    namelist[LOADAV].n_value == 0) {
    xload_error("cannot get name list from", KERNEL_FILE);
    exit(-1);
    }
```

xload$[os: unix, hcx\uparrow]$

```
InitLoadPoint()
{
  extern void nlist();
#ifdef AIXV3
  knlist(namelist, 1, sizeof(struct nlist));
#else
  nlist(KERNEL_FILE, namelist);
#endif
  if (namelist[LOADAV].n_type == 0 ||
    namelist[LOADAV].n_value == 0) {
    xload_error("cannot get name list from", KERNEL_FILE);
    exit(-1);
    }
```

xload$[os: unix, hcx: \top]$

```
InitLoadPoint()
{
  extern void nlist();
  nlist(KERNEL_FILE, namelist);
  if (namelist[LOADAV].n_type == 0 &&
    namelist[LOADAV].n_value == 0) {
    xload_error("cannot get name list from", KERNEL_FILE);
    exit(-1);
    }
```

Figure 17.1: Three version selections from a CPP file

- the original selection $xload[os{:}\,unix]$;
- a *hcx* version $xload[os{:}\,unix][hcx{:}\,\top] = xload[os{:}\,unix, hcx{:}\,\top]$;
- a non-*hcx* version $xload[os{:}\,unix][hcx\uparrow] = xload[os{:}\,unix, hcx\uparrow]$.

Each selection reduces the number of governing CPP expressions and simplifies the remaining ones. In the case of $xload[os{:}\,unix, hcx{:}\,\top]$, no CPP expressions are left—the version set is unambiguous.

17.1.2 A Revision Example

Another example is shown in figure 17.2. We use delta features to identify revisions, as discussed in chapter 11. The CPP expression `d`*i* stands for the feature terms Δ_i; likewise, `!d`*i* stands for $\nabla_i = \sim\Delta_i$.

The file *Text* comes in three revisions identified by $R_0 = \nabla_1 \sqcap \nabla_2$, $R_1 = \Delta_1 \sqcap \nabla_2$, and $R_2 = \Delta_1 \sqcap \Delta_2$. According to (11.6), the features of *Text* are $\Delta_2 \to \Delta_1 = \nabla_2 \sqcup \Delta_1$. These features are encoded in the CPP representation of *Text*, using an `#error` directive with the feature complement $\sim\nabla_2 \sqcup \Delta_1 = \Delta_2 \sqcap \nabla_1$ as context.

The overall structure of the *Text* file is as follows: The word `explain` occurs in R_0 only. In R_1, it was changed to `demonstrate`, and again changed in R_2 to `show`. Note how the feature implications and the `#elif` directive keep the actual

Text

```
#if d2 && !d1
#error
#endif
We
#if d2
show
#elif d1
demonstrate
#else
explain
#endif
the encoding
of revisions.
```

Text$[\Delta_2]$

```
#if !d1
#error
#endif
We
show
the encoding
of revisions.
```

Text$[\nabla_2]$

```
We
#if d1
demonstrate
#else
explain
#endif
the encoding
of revisions.
```

Figure 17.2: Selecting revisions from a CPP file

expressions small—instead of `d2 && d1`, only `d2` is required, since `d2` implies `d1`, and instead of `!d2 && d1`, only `d1` is required, due to usage of the `#elif` directive.

On the right-hand side of figure 17.2, we see two subset selections of *Text*. The selection $\mathit{Text}[\Delta_2]$ has the δ_2 change applied; the `#error` directive states that the ∇_1 subset does not exist. In the selection $\mathit{Text}[\nabla_2]$, no `#error` directive is required, because all its subsets exist; the word `demonstrate` is part of the subset $\mathit{Text}[\nabla_2 \sqcap \Delta_1]$, and the word `explain` belongs to the subset $\mathit{Text}[\nabla_2 \sqcap \nabla_1]$. We see how the complexity of CPP directives decreases as we narrow the version set by specifying more features.

If the selection term S is simple, like in our examples, subset selection is very efficient, since nearly all consistency checking can be done via reduction and orthogonality checking. Unless non-simple terms are used, subset selection in ICE takes no more time than a SCCS checkout or a CPP run without macro expansion.

17.2 Changing Version Sets

Having shown how version subsets are selected, we show how the original CPP file can be reconstructed after a change in a subset. Let us assume we want to change a version subset $F[S]$ in a file F to $F'[S]$. What we need now is a mechanism to construct the file F' from F and $F'[S]$—or, more specifically, from $F[\sim S] \mathit{and} F'[S]$, since $F[S]$ is to be overwritten by $F'[S]$. This is the general problem of *uniting* two version sets represented as CPP files; in our case, we want to construct $F' = F'[S] \sqcup F[\sim S]$.

A trivial mechanism to generate F' from $F'[S]$ and $F[\sim S]$ is to concatenate $F'[S]$ and $F[\sim S]$, each in its specific context. The file F' would then have the structure:

```
#if S
  ... contents of F'[S] ...
#else
  ... contents of F[~ S] ...
#endif
```

The advantage of this mechanism is its simplicity. Its disadvantage is that each version is stored separately, wasting space. What we would prefer is a representation where only the *differences* between $F'[S]$ and $F[\sim S]$ are enclosed by `#if S ... #endif`. For this purpose, we need a mechanism that generates a compact representation by determining the differences between versions, respecting CPP directives.

In this section, we present an algorithm that generates the union of two version sets $F[S]$ and $F[T]$, where S and T are disjoint—that is, $S \sqcap T = \bot$, $S \sqsubseteq \sim T$, and $T \sqsubseteq \sim S$ hold. The basic idea is to compare the two files textually, using a DIFF algorithm ignoring all CPP directives. In the resulting union $F[S \sqcup T]$, text parts occurring only in $F[S]$ or $F[T]$ are governed by S or T, respectively; common parts are governed by $S \sqcup T$. The more similar $F[S]$ and $F[T]$ are, the more commonalities will be detected by DIFF, and the smaller the representation of $F[S \sqcup T]$ will be.

As an example of how this works, consider the *Text* example from figure 17.2 on page 199. Let us assume we change the word `encoding` in $\mathit{Text}[\Delta_2]$ to `usage`, giving $\mathit{Text}'[\Delta_2]$ as shown in figure 17.3.

Text$'[\Delta_2]$

```
#if !d1
#error
#endif
We
show
the usage
of revisions.
```

Text$[\nabla_2]$

```
We
#if d1
demonstrate
#else
explain
#endif
the encoding
of revisions.
```

Figure 17.3: Changing a version subset

For the DIFF run, we use the internal representation without CPP directives, where each line is tagged with its features, shown in figure 17.4.

Text$'[\Delta_2]$

Line	Features
`We`	$\Delta_2 \sqcap \Delta_1$
`show`	$\Delta_2 \sqcap \Delta_1$
`the usage`	$\Delta_2 \sqcap \Delta_1$
`of revisions.`	$\Delta_2 \sqcap \Delta_1$

Text$[\nabla_2]$

Line	Features
`We`	∇_2
`demonstrate`	$\nabla_2 \sqcap \Delta_1$
`explain`	$\nabla_2 \sqcap \nabla_1$
`the encoding`	∇_2
`of revisions.`	∇_2

Figure 17.4: Version subsets in internal representation

The DIFF algorithm runs on the lines of $F[S]$ and $F[T]$ alone, ignoring the respective features. For each line, DIFF determines whether it occurs in $F[S]$, in

$F[T]$, or in both. The line features are obtained according to definition 17.1:

Definition 17.1 (DIFF line features) In a representation of $F[S \sqcup T]$ generated from $F[S]$ and $F[T]$, where S and T are disjoint, the features of each line are determined as follows.

1. Let $S' \sqsubseteq S$ be the features of the line in $F[S]$. If the line does not occur in $F[S]$, let $S' = \bot$.
2. Likewise, let $T' \sqsubseteq T$ be the features of the line in $F[T]$. If the line does not occur in $F[T]$, let $T' = \bot$.
3. The new features of the line are determined as $S' \sqcup T'$.

□

All lines originally contained in $F[S]$ only are thus governed with $S' \sqsubseteq S$; likewise, lines originally contained in $F[S]$ only are governed by $T' \sqsubseteq T$. The following proposition ensures that the representation given by definition 17.1 is correct.

Proposition 17.2 Let $F' = F[S \sqcup T]$ be a representation for the union of two version sets $F[S]$ and $F[T]$, as described above, and where S and T are disjoint. Then,

$$F'[S] = F[S] \qquad F'[\sim S] = F[T] \qquad F'[T] = F[T] \qquad F'[\sim T] = F[S]$$

hold.

PROOF. Without loss of generality, we show that $F'[S] = F'[\sim T] = F[S]$ holds. Let $U = S' \sqcup T'$ be the features of a line contained in F'. Both $S' \sqsubseteq S \sqsubseteq \sim T$ and $T' \sqsubseteq T \sqsubseteq \sim S$ are formed according to definition 17.1. The term S' represents the original features of the line in $F[S]$; if the line did not occur in $F[S]$, we have $S' = \bot$.

1. The selection $F'[S]$ determines the new features of this line as $U \sqcap S = (S' \sqcap S) \sqcup (T' \sqcap S) = S' \sqcup \bot = S'$.
2. The selection $F'[\sim T]$ returns the new features $U \sqcap \sim T = (S' \sqcap \sim T) \sqcup (T' \sqcap \sim T) = S' \sqcup \bot = S'$.

We see that the original line features S' remain unchanged; the line is contained in either all of $F[S]$, $F'[S]$, and $F'[\sim T]$ (if $S' \neq \bot$ holds) or in none of them (if $S' = \bot$ holds). Hence, $F[S] = F'[S] = F'[\sim T]$ holds, which was to be shown. □

$\mathbf{Text}'[\Delta_2] \sqcup \mathbf{Text}[\nabla_2]$		
Line	Features	
	Original	Reduced
`We`	$\nabla_2 \sqcup (\Delta_2 \sqcap \Delta_1)$	$\top$
`show`	$\Delta_2 \sqcap \Delta_1$	Δ_2
`the usage`	$\Delta_2 \sqcap \Delta_1$	Δ_2
`demonstrate`	$\nabla_2 \sqcap \Delta_1$	$\nabla_2 \sqcap \Delta_1$
`explain`	$\nabla_2 \sqcap \nabla_1$	∇_1
`the encoding`	∇_2	∇_2
`of revisions.`	$\nabla_2 \sqcup (\Delta_2 \sqcap \Delta_1)$	$\top$

Figure 17.5: Determining new line features

In our example, running DIFF and applying definition 17.1 on the facing page yields the output shown in figure 17.5. The central column shows the features determined according to the rules above.

The feature terms of the individual lines can be simplified with respect to the features of the entire file. In our case, the features of the file are $(\Delta_2 \sqcap \Delta_1) \sqcup \nabla_2 = (\Delta_2 \sqcup \nabla_2) \sqcap (\Delta_1 \sqcup \nabla_2) = \top \sqcap (\Delta_1 \sqcup \nabla_2) = \Delta_2 \to \Delta_1$. The simplifications for the line features follow the general scheme

$$(S \sqcap T) \sqcap (S \to T) = S \sqcap (S \to T) \tag{17.1}$$

$$(S \sqcup T) \sqcap (S \to T) = T \sqcap (S \to T) \quad , \tag{17.2}$$

leading to the simplified feature terms shown in the right column of figure 17.5.

The resulting CPP representation of $\mathit{Text}' = \mathit{Text}'[\Delta_2] \sqcup \mathit{Text}[\nabla_2]$ is shown in figure 17.6 on the following page, together with its two sources $\mathit{Text}'[\Delta_2]$ and $\mathit{Text}[\nabla_2]$.

17.3 Creating a CPP Representation

The CPP representation of the re-united version set, as shown in figure 17.6 on the next page, is not the only possible one. By interchanging text blocks and using other CPP directives, a multitude of representations is possible. This is illustrated in figure 17.7 on page 205: we see three alternate CPP representations for the version set in figure 17.6.

Since the text blocks can be rearranged in an arbitrary manner, there is no canonical CPP representation. Moreover, determining the smallest possible CPP representation is probably NP-complete, as it is closely related to finding the smallest possible representation of a formula in first-order logic.

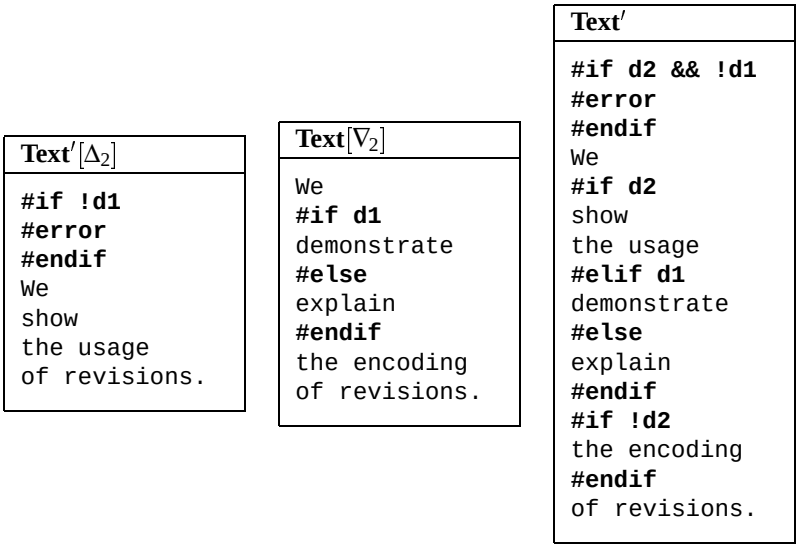

Figure 17.6: CPP representation after a subset change

For generating the CPP representation in ICE, we have chosen not to determine the smallest possible representation. Instead, ICE attempts to generate appropriate CPP directives by comparing the feature terms of subsequent text blocks.

17.3.1 An Algorithm to Create Nested CPP Directives

The easiest algorithm to create a CPP representation is to enclose each text block governed by a feature term T in `#if` T ... `#endif`. The first refinement of this representation is to generating *nested* CPP directives by maintaining a stack of feature terms where we save the current *contexts*—that is, the currently governing feature terms. Here is a simple algorithm realizing this approach.

Algorithm 17.3 (Creating nested CPP directives) To write a version set in CPP format, using CPP `#if` directives, use the following algorithm.

The algorithm consists of three pieces. The used variables are declared in ⟨Declarations⟩ and initialized in ⟨Initialization⟩. The CPP representation is written in ⟨Write body⟩.

⟨Algorithm 17.3⟩ ≡
    ⟨Declarations⟩

Text′

```
#if d2 && !d1
#error
#endif
We
#if d2
show
the usage
#elif d1
demonstrate
#else
explain
#endif
#if !d2
the encoding
#endif
of revisions.
```

Text′

```
#if d2 && !d1
#error
#endif
We
#if d2
show
the usage
#else
#if d1
demonstrate
#else
explain
#endif
the encoding
#endif
of revisions.
```

Text′

```
#if d2 && !d1
#error
#endif
We
#if !d2
#if !d1
explain
#else
demonstrate
#endif
the encoding
#else
show
the usage
#endif
of revisions.
```

Figure 17.7: Alternate CPP representations

⟨Initialization⟩
⟨Write body⟩

The algorithm requires two variables.

⟨Declarations⟩ ≡
Let the feature term *C* be the current context.
Let *CC* be a stack of contexts.

These variables are initialized as follows.

⟨Initialization⟩ ≡
Initialize $C := \top$.
Initialize *CC* with the empty stack.

The file body is written via a loop across all text blocks.

⟨Write body⟩ ≡
for all text blocks **do**
⟨Write block⟩
od
⟨Close body⟩

For each text block to be written, let T be its governing feature term. We must now generate CPP directives that change the context from C to T.

- If $T = C$ holds, the text is simply written.
- Otherwise, if $T \sqsubseteq C$ holds, save the current context on the stack, and write an `#if` S directive such that $T = C \sqcap S$ holds.
- Otherwise ($T \not\sqsubseteq C$), write an `#endif` directive, restore the context C from CC, and retry writing the text block with the new context.

In a more structured way, this is expressed as follows:

⟨Write block⟩ ≡
    Let T be the governing feature term of the current text block.
    while $T \not\sqsubseteq C$ **do**
        ⟨Write #endif⟩
    od
    if $T \neq C \wedge T \sqsubseteq C$ **then**
        ⟨Write #if⟩
    fi
    ⟨Write text⟩

If $T = C$ holds, write no CPP directive at all.

⟨Write text⟩ ≡
    Write the text block without any directive.

Otherwise, if $T \sqsubseteq C$ holds, we write an `#if` directive. The old context is saved on the stack CC.

⟨Write #if⟩ ≡
    Let $S \sqsupseteq T$ be a feature term such that $T = C \sqcap S$ holds.
    Write `#if` S.
    Save C on CC.
    Set the context to $C := C \sqcap S$.

Otherwise, we must use an `#endif` directive to exit the current context. This is done until we reach a suitable context. Since the outermost context is $\top$, such a context is always reached.

⟨Write #endif⟩ ≡
    Write `#endif`.
    Restore C from CC, discarding it.

Eventually, an `#endif` is written for each stacked context.

⟨Close body⟩ ≡
    while *CC* is non-empty **do**
        ⟨Write #endif⟩

□

17.3.2 Generating `#else` and `#elif` Directives

The actual algorithm used in ICE is somewhat more complex: it also generates `#else` and `#elif` directives. For this purpose, the algorithm maintains a current else-expression E as well as a stack EE of else-expressions. Another refinement found in this algorithm is the handling of overall file features F.

Algorithm 17.4 (Creating full CPP directives) To create a CPP representation of a version set, using the full set of CPP directives, use the following algorithm.

The algorithm consists of four pieces. The used variables are declared in ⟨Declarations⟩ and initialized in ⟨Initialization⟩. The CPP representation is written in ⟨Write header⟩ and ⟨Write body⟩.

⟨Algorithm 17.4⟩ ≡
    ⟨Declarations⟩
    ⟨Initialization⟩
    ⟨Write header⟩
    ⟨Write body⟩

The algorithm requires five variables.

⟨Declarations⟩ ≡
    Let the feature term F be the features of the file.
    Let the feature term C be the current context.
    Let the feature term E be the current else-expression.
    Let CC be a stack of contexts.
    Let EE be a stack of else-expressions.

These variables are initialized as follows.

⟨Initialization⟩ ≡
    Initialize F with the features of the file.
    Initialize $C := F$.
    Initialize $E := \bot$.
    Initialize CC and EE with the empty stack.

The file version is identified using an `#error` directive.

⟨Write header⟩ ≡
if $F \neq \top$ **then**
Write `#if` $\sim F$.
Write `#error`.
Write `#endif`.
fi

The file body is written via a loop across all text blocks.

⟨Write body⟩ ≡
for all text blocks **do**
⟨Write block⟩
od
⟨Close body⟩

The variable T holds the feature term of the current block; the variable C holds the current context. Before writing the block, we insert appropriate CPP directives such that the new context becomes T.

⟨Write block⟩ ≡
Let T' be the governing feature term of the current text block.
Let $T = T' \sqcap F$.
Let C' be the top element of CC, or $\bot$ if CC is empty.
while $T \not\sqsubseteq C \wedge T \not\sqsubseteq C' \sqcap E$ **do**
⟨Write #endif⟩
od
if $T \neq C$ **then**
if $T \sqsubseteq C$ **then**
⟨Write #if⟩
elsif $T = C' \sqcap E$ **then**
⟨Write #else⟩
elsif $T \sqsubseteq C' \sqcap E$ **then**
⟨Write #elif⟩
fi
fi
⟨Write text⟩

If $T = C$ holds, we do not need any CPP directive.

⟨Write text⟩ ≡
Write the text block without any directive.

Otherwise, if $T \sqsubseteq C$ holds, we write an `#if` directive. The old context and else-expressions are saved on the stack; the else-expression is the complement of the `#if`-expression.

⟨Write #if⟩ ≡
  Let $S \sqsupseteq T$ be a feature term such that $T = C \sqcap S$ holds.
  Write `#if` S.
  Save C on CC.
  Save E on EE.
  Set the context to $C := C \sqcap S$.
  Set the else-expression to $E := \sim S$.

Otherwise, if $T = C' \sqcap E$ holds, we can write an `#else` directive. we prohibit multiple `#else` directives by setting E to $\bot$,

⟨Write #else⟩ ≡
  Write `#else`.
  Set the context to $C := C' \sqcap E$.
  Set the else-expression to $E := \bot$.

Otherwise, if $T \sqsubseteq C' \sqcap E$ holds, we write an `#elif` directive.

⟨Write #elif⟩ ≡
  Let $S \sqsupseteq T$ be a feature term such that $T = C' \sqcap E \sqcap S$ holds.
  Write `#elif` S.
  Set the context to $C := C' \sqcap E \sqcap S$.
  Set the else-expression to $E := E \sqcap \sim S$.

Otherwise, we must use an `#endif` directive to exit the current context. This is done until we reach a suitable context.

⟨Write #endif⟩ ≡
  Write `#endif`.
  Restore E from EE, discarding it.
  Restore C from CC, discarding it.

When the last text block is processed, we must write an `#endif` for each remaining `#if`.

⟨Close body⟩ ≡
  while CC is non-empty **do**
    ⟨Write #endif⟩

□

17.3.3 An Example Run

We illustrate the use of algorithm 17.4 by applying it to the version set shown in figure 17.5 on page 203.

1. (Initialization) The features of the file are $F = \Delta_2 \to \Delta_1$.
   - The context is initialized to $C := F = \Delta_2 \to \Delta_1$.
   - The else-expression is initialized to $E := \bot$.
   - CC and EE are initialized with the empty stack.
2. (Write header) $F \neq \top$ holds. The complement of F is $\sim F = \sim(\nabla_2 \sqcup \Delta_1) = \Delta_2 \sqcap \nabla_1$.
   - `#if d2 && !d1` is written.
   - `#error` is written.
   - `#endif` is written.
3. (Write block) The text is `We`; $T = \top \sqcap F = \Delta_2 \to \Delta_1$ holds.
4. (Try equality) $T = C$ holds.
   - `We` is written.
5. (Write block) The text is `show`; $T = \Delta_2 \sqcap F = \Delta_2 \sqcap \Delta_1$ holds.
6. (Try equality) $T = C$ does not hold.
7. (Try #if) $T \sqsubseteq C$ holds; $S = \Delta_2$.
   - `#if d2` is written.
   - `show` is written.
   - The context $C = \Delta_2 \to \Delta_1$ is saved on CC.
   - The else-expression $E = \bot$ is saved on EE.
   - The context becomes $C := C \sqcap S = \Delta_2 \sqcap \Delta_1$.
   - The else-expression becomes $E := \sim S = \nabla_2$.
8. (Write block) The text is `the usage`; $T = \Delta_2 \sqcap F = \Delta_2 \sqcap \Delta_1$ holds.
9. (Try #endif) $T \not\sqsubseteq C$ does not hold.
10. (Try equality) $T = C$ holds.
    - `the usage` is written.
11. (Write block) The text is `demonstrate`; $T = \nabla_2 \sqcap \Delta_1 \sqcap F = \nabla_2 \sqcap \Delta_1$ holds. The outer context is $C' = \Delta_2 \to \Delta_1$.
12. (Try #endif) $T \not\sqsubseteq C' \sqcap E$ does not hold.
13. (Try #if) $T \sqsubseteq C$ does not hold.
14. (Try #else) $T = C' \sqcap E$ does not hold.
15. (Try #elif) $T \sqsubseteq C' \sqcap E = \nabla_2$ holds; $S = \Delta_1$.
    - `#elif d1` is written.
    - `demonstrate` is written.
    - The context becomes $C := C' \sqcap E \sqcap S = \nabla_2 \sqcap \Delta_1$.
    - The else-expression becomes $E := E \sqcap \sim S = \nabla_2 \sqcap \nabla_1$.
16. (Write block) The text is `explain`; $T = \nabla_1 \sqcap F = \nabla_1 \sqcap \nabla_2$ holds. The outer context is $C' = \Delta_2 \to \Delta_1$.
17. (Try #endif) $T \not\sqsubseteq C' \sqcap E$ does not hold.
18. (Try equality) $T = C$ does not hold.
19. (Try #if) $T \sqsubseteq C$ does not hold.
20. (Try #else) $T = C' \sqcap E = \nabla_2 \sqcap \nabla_1$ holds.
    - `#else` is written.
    - `explain` is written.
    - The context becomes $C = \nabla_2 \sqcap \nabla_1$.
    - The else-expression becomes $E := \bot$.
21. (Write block) The text is `the encoding`; $T = \nabla_2 \sqcap F = \nabla_2$ holds.

22. (Try #endif) $T \not\sqsubseteq C$ holds and $T \not\sqsubseteq C' \sqcap E$ holds.
    - `#endif` is written.
    - E is restored to $E := \bot$.
    - C is restored to $C := \Delta_2 \to \Delta_1$.
    - CC and EE become the empty stack again.
23. (Try #if) $T \sqsubseteq C$ holds; $S = \nabla_2$.
    - `#if !d2` is written.
    - `the encoding` is written.
    - The context $C = \Delta_2 \to \Delta_1$ is saved on CC.
    - The else-expression $E = \bot$ is saved on EE.
    - The context becomes $C := C \sqcap S = \nabla_2$.
    - The else-expression becomes $E := {\sim}S = \Delta_2$.
24. (Write block) The text is `of revisions.`; $T = \top \sqcap F = \Delta_2 \to \Delta_1$ holds. The outer context is $C' = \Delta_2 \to \Delta_1$.
25. (Try #endif) $T \not\sqsubseteq C$ holds and $T \not\sqsubseteq C' \sqcap E$ holds.
    - `#endif` is written.
    - E is restored to $E := \bot$.
    - C is restored to $C := \Delta_2 \to \Delta_1$.
    - CC and EE become the empty stack again.
26. (Try equality) $T = C$ holds.
    - `of revisions.` is written.
27. (Close body) The stack CC is empty; no more `#endif` directives need to be written.

The complete output is shown on the right side of figure 17.6 on page 204.

17.3.4 Efficiency

Algorithm 17.4 requires some deduction steps, notably the decision of subsumption. This can be done efficiently using *reduce*. Using (8.4), (14.20) and (14.16), we have:

$$\begin{aligned} T \sqsubseteq U &\Leftrightarrow {\sim}U \sqcap T = \bot \\ &\Leftrightarrow \mathit{reduce}({\sim}U, T) \sqcap T = \bot \\ &\Leftrightarrow {\sim}\mathit{reduce}(U, T) \sqcap T = \bot \end{aligned} \tag{17.3}$$

If U and T are simple, this problem is equivalent to

$$T \sqsubseteq U \Leftrightarrow \mathit{reduce}(U, T) = \top \ , \tag{17.4}$$

which requires quasi-linear time, according to corollary 14.12 on page 172.

The feature term $S \sqsupseteq T$ required in ⟨Write #if⟩ and ⟨Write #elif⟩ can also be obtained via *reduce*. In ⟨Write #if⟩, we have $T \sqsubseteq C$, and S must satisfy $T = C \sqcap S$. (Note that $S = T$ is a trivial choice for S). Since $T \sqsubseteq C$ holds, we have $T = C \sqcap T = C \sqcap \mathit{reduce}(T, C)$, following (14.20). Hence, $S = \mathit{reduce}(T, C)$ is a valid choice for S. The same applies to ⟨Write #elif⟩, where we obtain $S = \mathit{reduce}(T, C' \sqcap E)$.

Even better performance is achieved by saving the values of S across selections and unions. In ICE, each text block is associated with a set of CPP directives and possible values for S. Upon parsing, this set is initialized to contain the CPP directive separating this text block from its predecessor. Uniting version sets unites the two sets of CPP directives for each text block; upon selection, the terms S are reduced according to the selection term.

When writing a version set in CPP representation, ICE first determines whether using one of the saved CPP directives leads to the desired governing expression; if yes, the CPP directive is written and the remaining set members are discarded. Besides a maximum of performance, notably with orthogonal selection terms, this helps maintaining the structure of the original CPP file as much as possible.

The CPP directives generated by ICE are to be read and understood by humans. Beyond a certain term complexity, the effort for deducing an easily readable representation is wasted. Hence, ICE can be instructed to disable the generation of special CPP directives as soon as the terms exceed a specific length. Instead, ICE uses `#if` ... `#endif` directives only, without `#else`, `#elif`, and further nested directives. Writing this format does not require any deduction steps, and is easily processed again by ICE.

17.4 File Operations on Version Sets

Based on the selection and changing of version sets, we can now summarize the effects of file operations on version sets.

Read. Read access to $F[S]$ is accomplished by selecting S from F, as discussed in section 17.1.

Write. Write access to $F[S]$—that is, changing $F[S]$ to $F'[S]$—is implemented by generating $F' = F[\sim S] \sqcup F'[S]$, as shown in section 17.2.

Create. Creating $F[S]$, where F was non-existent before, creates F containing an `#error` directive governed by $\sim S$, such that $F[\sim S]$ is non-accessible.

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

We see that the CPP file representation of version sets allows users to create, read, change, and remove version sets just like ordinary files (that is, singleton version sets), while still only the differences between versions are stored.

17.5 Implementation Notes

The creation of compact CPP representations, as discussed in section 17.2, was realized by Lars Düning [Dün94], using the freely available GNU DIFF implementation. For maximum performance, the DIFF program is not invoked as a separate process, but directly linked within ICE.

Writing of version sets is based on algorithm 17.4 on page 207, extended with some additional optimizations not discussed here. ICE provides an interface for developers wishing to control the CPP output format.

The inference engine used in ICE implements Smolka's feature unification algorithm. It realizes all of the optimization methods discussed in chapter 14, as well as the implication reductions (17.1) and (17.2).

The inference engine provides two entry points. $\mathit{reduce}(S,T)$ realizes the *reduce* function from definition 14.10 on page 170; this assumes that S and T are already consistent. $\mathit{solve}(T)$ determines consistency of T, using Smolka's feature unification. Both rely on each other: *solve* uses *reduce* to reduce the size of subproblems; *reduce* calls *solve* to determine the consistency of non-simple subexpressions. For best performance, the inference engine caches deduction results such that frequent problems are solved only once.

Smolka's feature unification, as described in [Smo92], was implemented by Marc Ziehmann [Zie93].

17.6 Conclusion

ICE provides mechanisms to select and change arbitrary version subsets, using the CPP representation. Version sets can be accessed and manipulated like ordinary files, making version sets first-class objects in an SCM-aware environment, while still only the differences between versions are computed and stored.

Whoever shouted the loudest about their particular feature would usually get it in.
If the feature was some new 3-D chart or some very 'cool' thing, that would get in.
And if it wasn't cool but certainly was important, nobody would rally behind it ...
So it was working out not to be a process we felt very comfortable about
for designing our new versions.
So we decided, "Well, let's kind of invert the process a little bit.
Let's not even think about features."

— MIKE CONTE
in: MICHAEL A. CUSUMANO and RICHARD W. SELBY, Microsoft Secrets

Chapter 18

A Shell for Version Set Access

Based on the file operations, as discussed in section 17.4, we have implemented a library called LIBICE *that realizes file operations on version sets in* CPP *representation; arbitrary version sets can be created, read, written, and removed. To experiment with these mechanisms, we have realized a simple* command shell *on top of* LIBICE *that simulates transparent version set access for arbitrary files. The name of the shell is* ICICLE *(for* ICE *integrated command line engine).*

18.1 Reading Version Sets

Basically, ICICLE is a command shell roughly complying to the POSIX shell standard. Users can invoke programs by entering the program name, possibly followed by program arguments:

```
(icicle) more sample.txt
#if SAMPLE
This is a sample text.
#else
This is a simple text.
#endif
```

Here, `(icicle)` is the ICICLE prompt, `more sample.txt` is the user input, and `#ifdef` ... `#endif` is the output of the `more` command. The `more` command was invoked with `sample.txt` as argument; it simply prints the file given as argument on standard output.

The special feature of ICICLE is that it allows transparent version set access. To access a file F in the version S, users write $F[S]$, using the CPP representation for feature terms. Hence, users can access the `SAMPLE` version of `sample.txt`:

```
(icicle) more sample.txt[SAMPLE]
This is a sample text.
```

as well as its complement:

```
(icicle) more sample.txt[!SAMPLE]
This is a simple text.
```

This transparent access is realized as follows:

1. For each word $F[S]$, where F is a file name and S is a valid CPP expression, create a file named $F[S]$ containing the selection S of the file F.
2. Run the specified command.
3. Remove all files $F[S]$.

Hence, in our example, two temporary files named `sample.txt[SAMPLE]` and `sample.txt[!SAMPLE]` are created before `more` is invoked. After `more` has finished, they are removed.

18.2 Writing Version Sets

Besides reading of version sets, ICICLE also allows to change version sets, as discussed in section 17.2. Here is an example:

```
(icicle) cat > sample.txt[SAMPLE]
This is a text sample.
^D
(icicle) more sample.txt[SAMPLE]
This is a text sample.
(icicle) more sample.txt
#if SAMPLE
This is a text sample.
#else
This is a simple text.
#endif
```

The `cat` command copies the standard input to standard output; the `>` character redirects this output to the given file. The standard input is typed in by the user and finished using an end-of-input character (`^D`, Control-D). The contents of `sample.txt[SAMPLE]` become `This is a text sample.`

Writing version sets is realized by extending transparent access as follows:

1. For each word $F[S]$, where F is a file name and S is a valid CPP expression, create a file named $F[S]$ containing the selection S of the file F.

2. Run the specified command.

3. If one of the files $F[S]$ has changed to $F'[S]$, change F to $F' = F[\sim S] \sqcup F'[S]$.

4. Remove all files $F[S]$.

18.3 Removing Version Sets

ICICLE also allows to remove version sets. Here is another example:

```
(icicle) more sample.txt
#if SAMPLE
This is a text sample.
#else
This is a simple text.
#endif
(icicle) rm sample.txt[SAMPLE]
(icicle) more sample.txt[SAMPLE]
sample.txt[SAMPLE]: No such file or directory
(icicle) more sample.txt
#if !SAMPLE
#error
#else
This is a text sample.
#endif
```

The `rm` command removes the file given as its argument. Consequently, the `more` command cannot find the file and issues an error message. We see that issuing the `rm` command in ICICLE causes an `#error` directive to be inserted into `sample.txt`, identifying the non-existent versions.

Removing version sets is realized by extending transparent access as follows:

1. For each word $F[S]$, where F is a file name and S is a valid CPP expression, create a file named $F[S]$ containing the selection S of the file F.

2. If the selection S does not exist, do not create the file.

3. Run the specified command.

4. If one of the files $F[S]$ has changed to $F'[S]$, change F to $F' = F[\sim S] \sqcup F'[S]$.

5. If one of the files $F[S]$ has been removed, change F to $F' = F[\sim S]$.

6. Remove all files $F[S]$.

18.4 Multi-Version Merging

The CPP representation used in ICE also inspired a simple textual merging algorithm that merges an arbitrary number of versions. Let T be a version set with $T_1 \sqsubseteq T, T_2 \sqsubseteq T, \ldots, T_n \sqsubseteq T$ being n version subsets to be merged. Let us assume that all T_i were created independently from T such that all T_i are pairwise disjoint, i.e. $\forall i, j \in \{1, \ldots, n\}(T_i \sqcap T_j = \bot)$ holds.[1]

To generate a merged version from the CPP representation of T, we proceed as follows. The merged version T', denoted as $T' = T_1 \bowtie T_2 \bowtie \cdots \bowtie T_n$, must include code pieces that were added in any T_i and exclude code pieces that were deleted in any T_i. Each code piece governed by a CPP expression C is included if $C \sqsubseteq T_1 \sqcup T_2 \sqcup \cdots \sqcup T_n$ holds; in T', the governing expression is simplified (partially evaluated) respective to all T_i. Otherwise, if $\exists i(C \sqsubseteq \sim T_i)$ holds, the code piece governed by C was deleted in at least one T_i (and unchanged in all T_j with $j \neq i$) and thus is not included in T'. Everything else stays unchanged.

A minimum distance between parallel changes must be preserved in order to identify merging conflicts. Between any two code pieces governed by C' and C'' both being a subset of different T_i sets, a separating code piece governed by D must reside such that the following holds. Formally, let $T_{i'}$ be the unique element from $\{T_1, \ldots, T_n\}$ such that $C' \sqsubseteq T_{i'}$; similarly, $T_{i''}$ is the unique element from $\{T_1, \ldots, T_n\}$ such that $C'' \sqsubseteq T_{i''}$. Then, $D \not\sqsubseteq T_{i'} \wedge D \not\sqsubseteq T_{i''}$ must hold. If such a D does not exist, or if the length of D is below a certain minimal distance, C' and C'' are in conflict with each other.

As an example, consider the *tty.c* file in figure 18.1, where the version subsets $T_1 = [user{:}lisa]$ and $T_2 = [user{:}tom]$ are merged. Code piece A' is included, because its governing expression $[user{:}lisa]$ is equal to T_1; code piece A is excluded because its governing expression is equal to $\sim T_1$. Code piece C' would be included, as it is in T_1; but as it is immediately followed by C'', whose governing expression is equal to the different T_2 subset, the two changes are in conflict with each other.

For convenience, ICE flags this section still being a subset of a T_i with a "`// >< CONFLICT`" comment; only the code piece C can safely be removed as it is a subset of both $\sim T_1$ and $\sim T_2$. At the end, the code piece E is included, since

[1] Otherwise, replace the non-disjoint pair T_i, T_j by $T_k = T_i \sqcap T_j$.

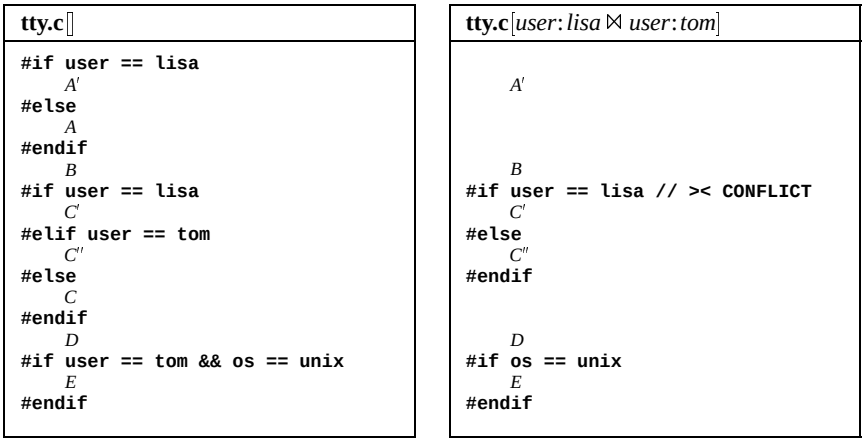

Figure 18.1: Merging of version sets

it is separated from the conflict by code piece *D*; the expression governing code piece *E* is simplified respective to T_2.

The ICICLE shell provides transparent access to merged version sets; the ⋈ operator is represented by the special CPP operator `><`. As an example, the ICICLE command

```
(icicle) more tty.c[user == lisa >< user == tom]
```

displays the merge of `tty.c[user == lisa]` and `tty.c[user == tom]` on standard output.

18.5 Handling Arithmetic Constraints

To provide some basic support for arithmetic CPP expressions, ICE realizes *partial evaluation* of CPP expressions.

Using arithmetic constraints for both selection and identification leads to undecidability, as discussed in section 7.3. Some special cases may be recognized, though:

Partial evaluation of arithmetic expressions. Constant CPP arithmetic expressions are evaluated according to their C semantics [ISO90] and replaced by the resulting value. Arithmetic expressions involving identifiers are replaced by feature values, if applicable. For instance, a CPP expression like

```
T == 200 && (T >= 100) || C == T)
```

evaluates to

```
T == 200
```

since `T >= 100` evaluates to non-zero—that is, $\top$.

Solving inequalities. The ICE inference engine contains an arithmetic constraint checker using the Simplex Method. The simplex method allows the ICE inference engine to recognize inconsistencies in a conjunction of simple inequalities. For instance, the arithmetic expression

```
T < 200 && T - 1 > 199
```

can be recognized as inconsistent by the ICE inference engine.

Partial evaluation of arithmetic expressions as well as arithmetic constraint solving allow ICE to handle an important subset of arithmetic constraints. Both mechanisms are implemented within the solving of feature clauses in Smolka's feature unification; all three methods are applied in turn on the constraint set until the constraint set is unchanged.

18.6 More ICICLE Features

Besides basic shell functions and transparent version access, ICICLE supports more than 250 commands to control ICE functionality. ICICLE also contains facilities to define new commands as scripts of other commands. All common shell mechanisms like variables and control structures are available, including an interactive line editor with completion of file names and CPP expressions. However, by far most of these facilities are used for testing and debugging, and are not intended for end users.

18.7 Implementation Notes

Multi-version merging was implemented by Andreas Mende [Men96]. Arithmetic constraint solving was realized by Christina Trenkner [Tre96].

18.8 Conclusion

On version sets represented as CPP files, all elementary file operations like reading, writing, creation, or removal are defined. These basic access methods are

available in LIBICE, the ICE library; the ICICLE command shell simulates transparent version set access through temporary files. Version sets can be merged using a simple textual algorithm, integrating changes in multi-version representations. These elementary file operations, as realized in LIBICE and ICICLE, constitute the base of an entire virtual file system, as discussed in chapter 19.

Feature: n. 1. A good property or behavior.
2. An intended property or behavior.
3. A surprising property or behavior.
4. A property or behavior that is gratuitous or unnecessary.
5. A property or behavior that was put in to help someone else but that happens to be in your way.
6. A bug that has been documented.

— ERIC RAYMOND, The Jargon File

Chapter 19

The Featured File System

The featured file system (FFS) realizes transparent version set access in arbitrary environments. In addition to versioned file access, as demonstrated in chapter 17, it supports versioned directories and thus versioning of entire file systems. Directory versions confine the versions of the contained files and subdirectories. Directory versions can thus be used as workspaces; users can change workspaces like they change directories. Additional facilities like virtual subdirectories facilitate the interactive and incremental exploration of the configuration space, as implemented in the SKATE configuration browser.

19.1 A SCM Primitives Layer

The *featured file system* (FFS) is a virtual file system that realizes the ICE primitives layer—that is, access to version sets and integration into software development environments. Compared to ICICLE and LIBICE, the FFS has the following advantages:

Version set access. Besides versioned file access, as realized in ICICLE and LIBICE, the FFS provides versioned access to *directories.* Directory versions confine the versions of all contained files, and may thus be used to realize *workspaces*; users can change their workspace just like changing directories. All file system operations, including the creation of directories, permission changes, and file mode changes, are versioned.

Exploration of the configuration space. The FFS represents non-singleton version sets as *directories* containing the individual versions. This provides accidental access to non-singleton version sets. By adding or removing

more version specifications, users can explore the configuration space interactively.

Environment integration. The FFS is realized as a true file system, accessed through the operating system interface. Existing programs need neither be changed, nor must they be invoked in a special manner, nor must they be linked with a special library.

19.2 Versioned Directories

Versioned files and versioned directories, as supported by the FFS, cover the state and changes of the entire file system—that is, the whole configuration universe. Basically, a versioned directory is stored and accessed like ordinary versioned files are, using the CPP representation. As an example, figure 19.1 shows a user-readable representation of a versioned directory. (The FFS itself uses a more efficient binary format.) We see that the `icicle` and `libice` directories were added in a change δ_1, which also removed the `lib` directory.

```
-rw-r--r--  1 zeller       1462 Jun 22  1995 host.h.in
#if d1
drwxr-sr-x  4 zeller       4096 Jun 10 15:15 icicle
#endif
drwxr-sr-x  3 zeller       1536 Apr 16 15:19 info
-rwxr-xr-x  1 zeller       4771 Feb  9  1995 install
#if d2
drwxrwsr-x  3 zeller       7168 Jun 10 15:15 libice
#elif d1
drwxr-sr-x  3 zeller       7168 Jun 10 15:15 libice
#else
drwxr-sr-x  4 zeller        512 Mar  3 11:35 lib
#endif
```

Figure 19.1: A versioned directory

The later change δ_2 is more subtle: the access mode of the `libice` directory was changed from `rwxr-sr-x` to `rwxrwsr-x`, making it writable by a group.

Users may now access individual versions of this directory, by appending a version specification $[S]$ to the directory name, just as with ordinary files. A typical interaction is shown in figure 19.2 on the next page. The `$` character is the UNIX shell prompt. The UNIX command `ls -l` lists the contents of the directories given as its arguments. The single dot "`.`" stands for the current directory; the directory name `.[d2]` is the current directory in version Δ_2. To avoid shell-specific interpretation of brackets, we enclose the directory name in quotes.

We see that `ls ".[d2]"` shows the Δ_2 version of the current directory, while `ls ".[!d1]"` shows the ∇_1 version.

```
$ ls -l ".[d2]"
-rw-r--r--  1 zeller       1462 Jun 22  1995 host.h.in
drwxr-sr-x  4 zeller       4096 Jun 10 15:15 icicle
drwxr-sr-x  3 zeller       1536 Apr 16 15:19 info
-rwxr-xr-x  1 zeller       4771 Feb  9  1995 install
drwxrwsr-x  3 zeller       7168 Jun 10 15:15 libice

$ ls -l ".[!d1]"
-rw-r--r--  1 zeller       1462 Jun 22  1995 host.h.in
drwxr-sr-x  3 zeller       1536 Apr 16 15:19 info
-rwxr-xr-x  1 zeller       4771 Feb  9  1995 install
drwxr-sr-x  4 zeller        512 Mar  3 11:35 lib

$ ls -l .
-rw-r--r--  1 zeller       1462 Jun 22  1995 host.h.in
drwxr-sr-x  4 zeller       4096 Jun 10 15:15 icicle
drwxr-sr-x  3 zeller       1536 Apr 16 15:19 info
-rwxr-xr-x  1 zeller       4771 Feb  9  1995 install
drwxr-sr-x  3 zeller       7168 Jun 10 15:15 libice
drwxr-sr-x  4 zeller        512 Mar  3 11:35 lib
```

Figure 19.2: Three views of a versioned directory

The final view, `ls .`, shows all versions of the current directory. Since the `ls` command is not aware of multi-version directories, every existing directory is listed, regardless of its specific version; file modes, sizes, and times are set up appropriately.[1]

Instead of explicit version access, as illustrated in figure 19.2, FFS also supports implicit version access through the current directory. Since `.[d2]` is a directory version as well, users can make it their current directory, using the UNIX `cd` command. Hence, `cd ".[d2]"` followed by `ls .` has the same effect as `ls ".[d2]"`, with the difference that all following commands reference the Δ_2 version of the current directory as well—until the directory is changed again.

[1]Here are the details. The access mode of a version set is the logical AND of the modes of all its individual versions—that is, the least permissive mode. The size of a version set is the maximum size of its individual versions. The owner of a version set is the owner of the individual versions, or `nobody`, if ambiguous. The access time of a version set is the most recent access time of the individual versions.

19.3 Version Confinements

Having a versioned directory in the current path not only affects this particular directory version. A directory version also confines the versions of all files and directories contained within that directory. Formally, 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, after changing to the directory version Δ_2, all files and directories are visible in their Δ_2 version only.

This property is useful for setting up *workspaces,* as discussed in section 13.1. For instance, entering the UNIX command

```
$ cd ".[user == lisa && current]"
```

confines the versions of all files and directories in the current directory and below to $[user: lisa, current: \sim 0]$—that is, the current version of Lisa's workspace. All changes made in this workspace affect only Lisa's current version.

As in ordinary file systems, the directory name "`..`" refers to the enclosing directory—the second last component from the current path. For example, the path `testdir/.[user != tom]/..` is equivalent to `testdir`. Hence, Lisa may issue the UNIX command

```
$ cd ..
```

to exit her workspace again and to see all versions at once.

As illustrated in figure 19.3 on the facing page, such directory changes may be also be performed incrementally, subsequently narrowing the configuration space as more and more features are specified.

For user convenience, the FFS interprets a version specification "`[`*S*`]`" like "`.[`*S*`]`". Hence, entering

```
$ ls -l "[user == lisa]/[tested]"
```

has the same effect as

```
$ ls -l ".[user == lisa]/.[tested]"
```

which in turn is equivalent to

```
$ ls -l ".[user == lisa && tested]"
```

Whenever a change is made within a versioned directory, all rules for operations in workspaces, as defined in section 13.1.2, apply. Hence, no change made within a directory version is visible in the complement of this version.

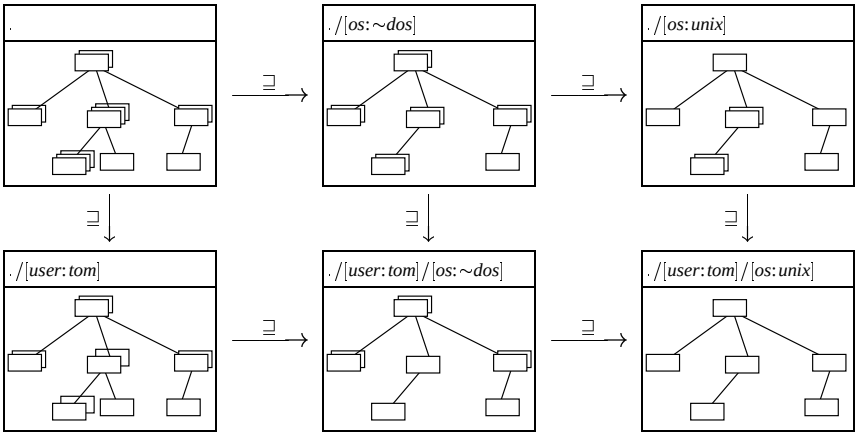

Figure 19.3: Narrowing the configuration space in the FFS

19.4 Version Shortcuts

Specifying the current version as part of the path name has the advantage of supporting both implicit and explicit version access. Arbitrary version sets can be accessed from any program; by changing the directory version, entire file systems can be accessed in a specific version without any additional version specifications. This is superior to approaches where the current version is specified as part of the process environment (since the environment must be interpreted by special runtime libraries) or as part of the user's file system (since this implies a state which must be changed explictly).

The drawback is that the file names used by the FFS are quite uncommon—for operating systems and programs. Users must be aware of possible problems.

Operating system caveats. In the UNIX operating system, the slash `/` is reserved as path separator—one cannot use arithemtic expressions involving integer division. In MacOS, the Macintosh operating system, colons `:` are used instead—one cannot enter feature terms in the ASCII representation. In the DOS operating system, all is lost, as it supports only eleven characters as file names.

Program caveats. In the UNIX command shell, characters like `&`, `|`, `*`, `?`, `!`, or `[` have a special interpretation and must be quoted. Many shell scripts are not

protected against file names containing space and quote characters. Users are frequently unfamiliar with the shell quoting mechanisms.

There are three issues addressing these problems. First, characters like `/` and `:` can easily be avoided. Second, as graphical user interfaces become more and more common, so are the possibilities to specify arbitrary file names. Macintosh users, for instance, have no concept of a command shell and of characters with a specific interpretation. Existing shells can be easily adapted for FFS usage by leaving characters within square brackets uninterpreted, like ICICLE does. Third, the FFS supports *symbolic links* that allow users to specify ordinary directory names for version sets—so-called *version shortcuts.* As an example, consider the setting in figure 19.4:

```
$ ls -l workspaces
drwxr-sr-x  1 lisa         1024 Jun  8  1996
    lisa -> .[user == lisa && current]
drwxr-sr-x  4 tom          1024 Jun 10 15:15
    tom  -> .[user == tom  && current]
drwxr-sr-x  3 john         1024 May  6 15:19
    john -> .[user == john && current]
```

Figure 19.4: Symbolic links to workspaces

A symbolic link $F_1 \rightarrow F_2$ makes F_1 an *alias* for F_2; whenever F_1 is part of a path name, F_2 is substituted. In our case, Lisa can simply enter

```
$ cd workspaces/lisa
```

which is a convenient replacement for

```
$ cd "workspaces[user == lisa && current]"
```

.

The directory name `workspaces/lisa` may even be defined as Lisa's home directory, such that Lisa automatically enters her current workspace upon logging in. No special file names are ever required, unless someone wants to access variants or determine the differences between versions by examining the entire version set. Of course, symbolic links are versioned just like other parts of the file system, such that each user may maintain a set of individual links for frequently accessed versions.

19.5 Exploring the Version Space

19.5.1 Virtual Directories

Since few tools can interpret version sets in CPP representation, the FFS takes precautions against multiple file versions being accessed as single items. The basic idea is to represent non-singleton version sets as *virtual directories* containing the individual versions. These versions are listed as possible version specifications, narrowing the version space. As an example, the file `tasks.txt`, occurring in multiple versions, is listed as

```
$ ls -l .
drwxr-sr-x 12 zeller       1024 Jun 10 14:20 tasks.txt
$ ls tasks.txt
[user == john]          [!(user == john)]
[user == lisa]          [!(user == lisa)]
[user == tom]           [!(user == tom)]
```

where the files

```
tasks.txt/[user == john]
tasks.txt/[user == lisa]
tasks.txt/[user == tom]
```

are the individual versions of `tasks.txt`.

Each complement like `tasks.txt/[!(user == john)]` is again a directory, since two choices remain. For instance, listing the entries of the subdirectory `tasks.txt/[!(user == john)]` yields:

```
$ ls "tasks.txt/[!(user == john)]"
[user == lisa]          [!(user == lisa)]
[user == tom]           [!(user == tom)]
```

where all entries are files, since they are singleton. Note that the file

```
tasks.txt/[!(user == john)]/[user == lisa]
```

is identical to

```
tasks.txt/[!(user == john)]/[!(user == tom)] ,
```

and could also be accessed as

```
tasks.txt[user == lisa] .
```

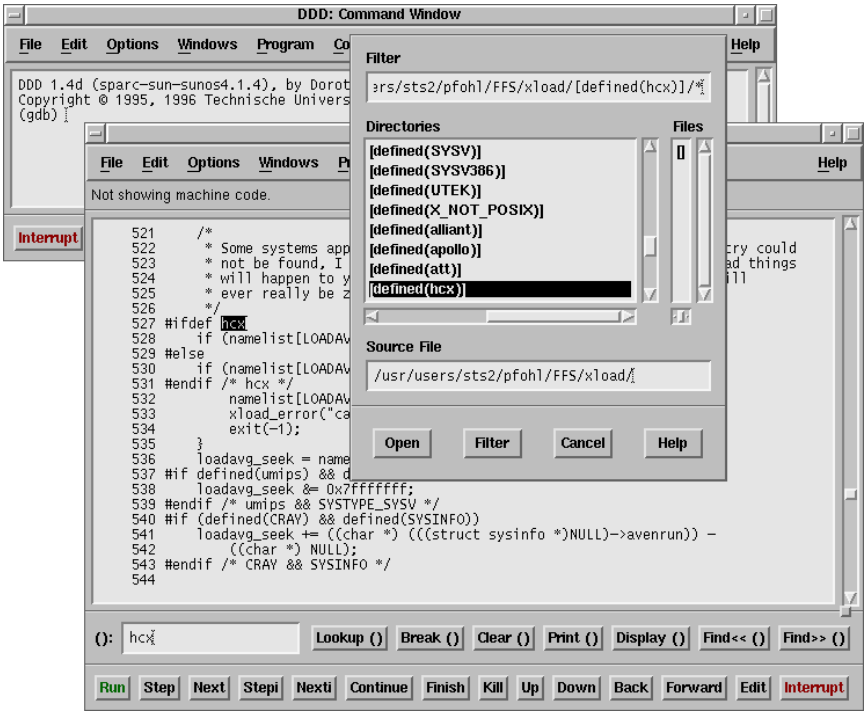

Figure 19.5: Using virtual subdirectories to select configurations

How do we obtain these subdirectories? Let F be a version set in CPP representation to be processed by the FFS server. The CPP server scans F for CPP directives; if none are found, F is singleton and thus presented as file. Otherwise, F is presented as a directory.

Each feature/value combination $T = f{:}S$ or $T = f{\uparrow}$ found in governing CPP expressions results in two entries T and $\sim T$ in the directory F. These entries are again files, if singleton, and virtual directories, otherwise.

In figure 19.5, we see the DDD debugger accessing the versions of the `xload` file discussed in section 17.1.1. The central window is a file system browser allowing the user to choose files and directories. In the upper field, the user has entered a *file filter* specifying the files to be shown; the current pattern `xload/*` shows all files in the `xload` directory. In our case, `xload` is a multi-version

file; the FFS represents it as a virtual directory with possible CPP expressions as entries shown in the list below. Although neither DDD nor its file selection dialog are aware of versions, the user can select an individual version from the virtual file system just by including and excluding options.

The example also illustrates a problem when reusing existing CPP files like `xload`: the knowledge about inconsistencies is not explicitly expressed. For instance, there is no machine in the real world where both `apollo` and `att` are defined. But this mutual exclusion is not specified in `xload`, such that Lisa must specify both explicitly. Having a `manufacturer` feature with values `apollo` and `att` would make version selections much faster; limiting the choice to configurations with syntactically correct programs would also help here.

19.5.2 Feature Completion

A special problem comes up when the workspace is narrowed such that a version set becomes singleton before all its features have been specified. When a file F has the features S, it exists as $F[S]$ only. Let us assume we have narrowed our workspace down to $F[S']$, such that $F[S']$ becomes singleton. In principle, we may list $F[S']$ as an ordinary file, since there is no difference between reading $F[S']$ and reading $F[S]$. With *writing,* this is different—writing $F[S']$ assigns F the features S'; the features S are lost. For this reason, the FFS displays $F[S']$ as a symbolic link *completing* the features of F by pointing to $F[S]$.

As an example, consider the `screen-device` component from the editor example in figure 10.1 on page 104. Listing the `dumb` version yields

```
$ ls screen-device
[Concurrent == true]    [ScreenData == bitmap]
[Data == postscript]    [ScreenDevice == dumb]
[Data == ScreenData]    [ScreenDevice == ghostscript]
```

as well as the respective complements.

The *screen-device:ghostscript* version is already singleton. The remaining features are explicitly completed by the symbolic link:

```
$ ls "screen-device[ScreenDevice == ghostscript]"
screen-device[ScreenDevice == ghostscript] ->
    screen-device[ScreenDevice == ghostscript
                      && Data == postscript
                      && ScreenData == bitmap
                      && Concurrent == true]
```

The user can specify any unambiguous superset of `screen-device` and still access the single existing version for reading and writing; file names need be no longer than required for disambiguation.

The drawback of this FFS feature is that once a version set $F[S]$ has been created, it is impossible to create a superset $F[S']$ with $S' \supseteq S$ except by removing $F[S]$ first. But as the redirection from $F[S']$ to $F[S]$ is shown explicitly, few problems should arise in practice.

19.5.3 Accessing the CPP Representation

Listing possible refinements as version subdirectories not only allows the user to explore the configuration space, but also prohibits accidental processing of version sets in CPP representation. In fact, users need never see the CPP representation, unless maybe to examine differences between versions. In some cases, however, it is desirable to access all versions at once.

The CPP representation of a file F as a whole may be accessed using the special form $F[]$, meaning "all versions". Instead of exploring the configuration space of `tasks.txt`, we may as well open

```
tasks.txt[user == lisa || user == tom][]
```

and thus view and edit both Lisa's and Tom's versions at once; likewise, opening `xload[]` gives us the CPP representation of `xload`. Besides being convenient for developers, this feature is a must for programs that recursively descend the directory tree; such programs would otherwise suffer from the combinatorical explosion of possible configurations if they traversed all possible configurations through virtual directories. The ability to access version sets in the CPP representation is also required for higher-level SCM tools discussed in the next chapters.

Finally, it should be noted that all this version selection is not necessary when working in a sufficently narrow workspace, making every version singleton and unambiguous.

19.6 A Configuration Browser

While the FFS provides some basic facilities to explore the version space, existing applications can be enhanced by making them aware of versions. One such example is the SKATE browser, shown in figure 19.6 on the facing page. The SKATE browser enhances a usual file system browser with the ability to visualize and explore the configuration space. For each possible feature, we generate a menu listing the possible feature values. The subsumption lattice formed by the version sets is shown as a graph, visualizing revision graphs and variant/workspace hierarchies. Through these menus, the user can specify a (possibly incomplete) configuration.

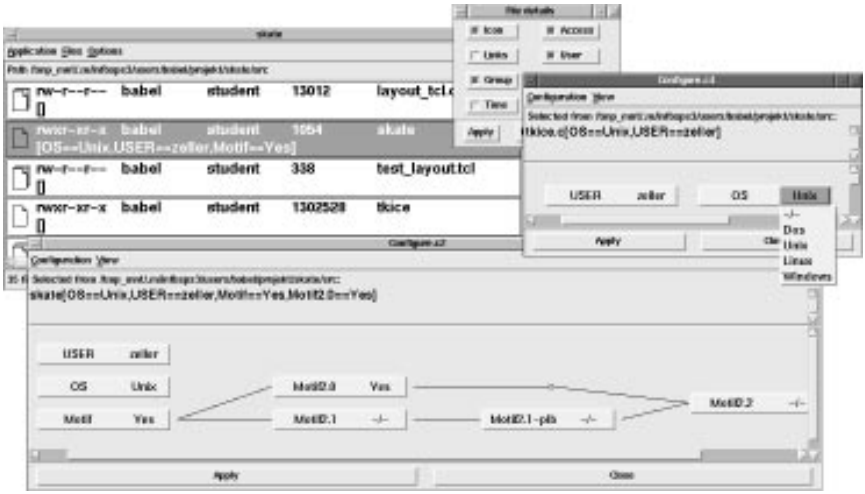

Figure 19.6: Browsing through files and configurations with SKATE

SKATE ensures consistency by making menu items insensitive that would result in an inconsistent or non-existent configuration. In the `OS` menu, for instance, all items are sensitive; there is no choice for `USER == zeller` that makes the selection inconsistent. Now let us assume that user Lisa works on all versions except the `Windows` operating system. This means that the version

```
USER == lisa && OS == Windows
```

does not exist; the directory "`.[USER == lisa && OS == Windows]`" is inaccessible in the FFS. If we set the value of the `USER` feature to, say, `lisa`, the `Windows` value of the `OS` feature would be grayed out, indicating that this selection would lead to an inconsistent configuration. Likewise, selecting `Windows` for `OS` would make the `lisa` entry in the `USER` menu insensitive. As the global effects of choice refining and revoking are immediately visualized in the configuration panels, the user can interactively explore the configuration universe while ICE checks for consistency.

19.7 Implementation Notes

The FFS is realized on top of the popular *network file system* (NFS) [SGK+85]. As discussed in section 5.4.3, this allows arbitrary programs to access the file

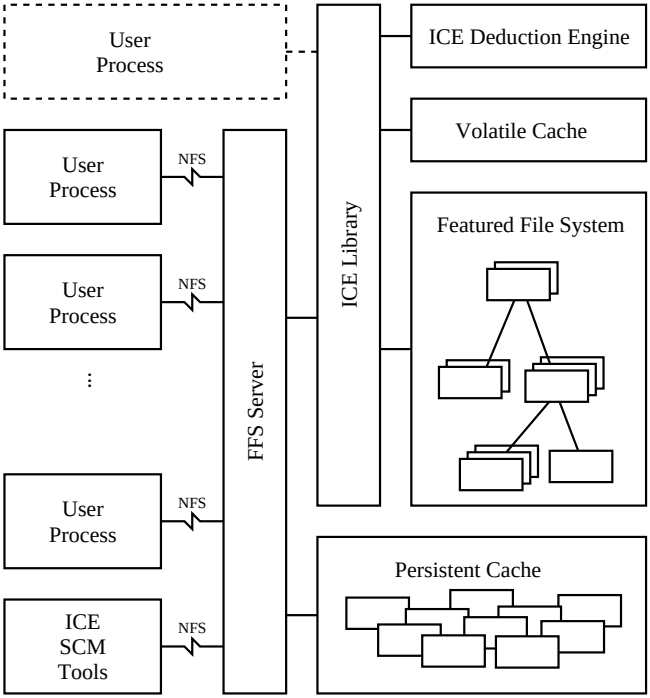

Figure 19.7: Processes accessing the featured file system

system transparently. The FFS server was designed and implemented by Olaf Pfohl [Pfo96], by extending a freely available NFS server (originally designed for the LINUX operating system). The overall architecture is shown in figure 19.7.

To maximize performance, the FFS server maintains a persistent cache, where all version sets once read are stored. Whenever a file $F[S]$ is requested, the FFS server first looks up $F[S]$ in the persistent cache, and scans the CPP representation F only if $F[S]$ was not found in the cache. Hence, F is scanned only at the first access; second and later version set accesses are served in constant time.

When $F[S]$ is written, it is also stored in the persistent cache; the originating version set F is only updated when a superset of S is requested. In practice, this means that once a workspace is entered, the FFS server has the same performance as an ordinary NFS server. But still, all files common to several workspace are cached only once, showing the space-saving effects of the viewpathing techniques

used in *n*-DFS.

To minimize problems with existing multi-version representations, the FFS server uses "as-is" encoding for reading ordinary files; hence, CPP directives in maintained files are left uninterpreted. However, if a multi-version representation is read by the FFS server, using the $F[]$ form, the FFS server uses the dynamic encoding as discussed in section 16.4. Hence, ordinary files `sample.c` are left unprocessed; but renaming the CPP file `sample.c` to `sample.c[]` makes FFS interpret the CPP directives and create the appropriate versions.

The SKATE configuration browser was realized by Dirk Babel [Bab96], using the freely available Tcl/Tk graphical user interface. SKATE runs in two modes. In *remote mode,* the question whether a specific configuration exists is answered by attempting to access this configuration from the FFS server. Since this places a heavy load on the FFS server, an alternative is provided. In *local mode,* SKATE gets the possible configurations from the file directly and uses a local ICE deduction engine to deduce whether a configuration leads to inconsistency.

19.8 Discussion

A virtual file system, as realized in the FFS, is certainly the most convenient way to integrate version access in today's software development environments. There can be no doubt that virtual file systems will constitute the standard for version access in future integrated SCM systems.

Basic read and write access to version sets can only constitute the primitives layer of SCM access. Based on these primitives, specialized SCM tools must exist that organize the SCM protocol and process layers—for instance, workspace management and change propagation as discussed in chapter 13. Such tools are currently in development for ICE, and the problems encountered during their development show that there is still much to do for future SCM researchers.

I would give the spec to marketing and say,
"Please give me your feedback. Is this the right set of features to do?"
And marketing would either read it or not read it,
because it was way too long.
Or, if they did read it, they would get lost in it,
because it's a super-technical thing.
And if they did comment on it, ... they would say,
"Well, we think this dialog box is laid out wrong.
You should really have the check boxes on the left," or something.
It's not the feedback you want as a program manager.

— MIKE CONTE
in: MICHAEL A. CUSUMANO and RICHARD W. SELBY, Microsoft Secrets

Chapter 20

Performance Studies

We present the results of some experiments performed to determine the feasibility of the version set model. We show how ICE can be used to select and change version subsets, how "classical" revision graphs are represented and how the FFS performs in practice. It turns out that all these "classical" tasks can be handled efficiently.

20.1 Working On Variants

As a first case study, we shall use ICICLE to extract and modify version subsets out of an existing CPP representation. The example file we have chosen is the `xload` file discussed in section 17.1.1.[1]

20.1.1 Retrieving Single Variants

We shall retrieve a single `xload` variant using ICICLE and compare it with CPP in terms of performance and flexibility.

Table 20.1 on the following page shows the 26 CPP symbols governing the `xload` source code. Each CPP symbol represents a specific machine architecture (like `sun`, `macII`, or `CRAY`) or feature (like `X_NOT_POSIX` or `__STDC__`). To retrieve a single variant, each of these CPP symbols must either be defined or undefined. Using CPP, this is rather simple, since all symbols not explicitly defined are left undefined; moreover, CPP pre-defines appropriate symbols for the machine it is running on. On a SUN machine, for instance, CPP defines the `sun` symbol and

[1] All data required for repeating these ICICLE experiments is contained in the ICE test suite, which is part of the ICE distribution. See appendix B for details on getting the ICE distribution.

AIXV3	CRAY	KERNEL_FILE
KERNEL_LOAD_VARIABLE	KMEM_FILE	KVM_ROUTINES
LOADSTUB	MOTOROLA	SVR4
SYSV	UTEK	X_NOT_POSIX
__STDC__	alliant	apollo
att	hcx	hpux
i386	macII	mips
sequent	sgi	sony
sun	umips	

Table 20.1: CPP symbols in `xload`

nothing else; it thus suffices to invoke CPP on the `xload` file to get the SUN variant. To measure the efficiency of CPP, we have commented out all `#define` and `#include` directives in `xload` and used CPP to select a version from `#ifdef` directives only. The command

```
$ /lib/cpp xload > /dev/null
```

requires an average running time of 0.08 seconds.[2]

Using ICICLE, we must specify for each single CPP symbol whether it is defined or undefined. This results in the ICICLE command

```
(icicle) system cat xload[sun \
&& !defined(AIXV3) && !defined(CRAY) \
&& defined(KERNEL_FILE) \
&& defined(KERNEL_LOAD_VARIABLE) \
&& defined(KMEM_FILE) && defined(KVM_ROUTINES) \
&& !defined(LOADSTUB) && !defined(MOTOROLA) \
&& !defined(SVR4) && !defined(SYSV) \
&& !defined(UTEK) && !defined(X_NOT_POSIX) \
&& !defined(__STDC__) && !defined(alliant) \
&& !defined(apollo) && !defined(att) \
&& !defined(hcx) && !defined(hpux) && !defined(i386) \
&& !defined(macII) && !defined(mips) \
&& !defined(sequent) && !defined(sgi) \
&& !defined(sony) && !defined(umips)] > /dev/null
```

which requires an average running time of 0.79 seconds—that is, one order of magnitude slower than CPP. Of these 0.79 seconds, 0.37 seconds are spent in

[2] All times measured are times spent in the execution of the command (“user time”) on a 75 MHz SUN SPARCstation 20 running SunOS 4.1.4.

reading `xload` into memory; the next 0.37 seconds are required for creating a temporary working file, and running the `cat` command on the working file requires another 0.05 seconds.

Why is ICICLE ten times slower than CPP? This is not the fault of the deduction engine. In the selection `xload`$[S]$ with

$$S = [\mathit{sun}\colon \top, \mathit{aixv3}\uparrow, \mathit{cray}\uparrow, \mathit{kernel\_file}\colon \top, \ldots, \mathit{umips}\uparrow] \quad , \tag{20.1}$$

ICE represents S as a hash table indexed through the feature name, as discussed in section 14.6. Following the proof of 14.8 on page 167, this leads to quasi-linear time for determining the consistency of $T \sqcap S$ for each governing expression T in `xload`—just like CPP, and this is just what is implemented in ICE. So, the lower performance of ICE does not stem from overall complexity, but rather from the general overhead required for generalized solutions.

20.1.2 Using Configuration Constraints

In practice, the long ICICLE command as shown in section 20.1.1 is quite redundant, since only few of the possible CPP symbol combinations actually make sense—there simply is no configuration with both *sun* and *hpux* defined. Such knowledge can be expressed by configuration constraints like $(\mathit{sun}\colon \top \to \mathit{hpux}\uparrow)$, meaning that if *sun* is defined, than *hpux* is not. We extend `xload` with some constraints applying to the *sun* architecture; these constraints in CPP notation are shown in figure 20.1 on the following page.

With these constraints embedded in `xload`, we can now simply say

```
(icicle) system cat xload[sun]
```

to get the SUN configuration; having *sun* defined implies all other features being either explicitly defined or explicitly undefined. The average running time of this command is 0.81 seconds—that is, slightly larger than the first command. This overhead is due to the processing of `#error` directives.

Again, the deduction engine is as efficient as possible. The `xload` configuration constraints are represented as one single implication

$$C = (\mathit{sun}\colon \top \to [\mathit{cray}\uparrow, \mathit{motorola}\uparrow, \mathit{utek}\uparrow, \mathit{alliant}\uparrow, \ldots, \mathit{x\_not\_posix}\uparrow])$$

in an efficient form using hash tables for the left-hand sides and right-hand sides of an implication. Hence, $C \sqcap [\mathit{sun}\colon \top]$ evaluates to S from (20.1) in quasi-constant time, and the remaining selection is done as in section 20.1.1.

The configuration constraints we introduced for `xload` are still incomplete. A full solution would not only express implications for the *sun* architecture, but

```
#if defined(sun)
// 'sun' excludes all other architectures.
// This should be done for all architectures!
#if defined(CRAY) || defined(MOTOROLA) \
|| defined(UTEK) || defined(alliant) \
|| defined(apollo) || defined(att) \
|| defined(hcx) || defined(hpux) || defined(i386) \
|| defined(macII) || defined(mips) \
|| defined(sequent) || defined(sgi) \
|| defined(sony) || defined(umips)
#error
#endif
// 'sun' also implies SunOS (in our example)
#if defined(AIXV3) || defined(SVR4) || defined(SYSV)
#error
#endif
// Other features implied by the 'sun' architecture.
#if !defined(KERNEL_FILE) || !defined(KVM_ROUTINES) \
|| !defined(KERNEL_LOAD_VARIABLE) \
|| !defined(KMEM_FILE) \
|| defined(LOADSTUB) || defined(X_NOT_POSIX)
#error
#endif
#endif // defined(sun)
```

Figure 20.1: `xload` configuration constraints

for all other architectures as well. For n architectures, we have $n^2/2$ mutual exclusions—and thus $n^2/2$ configuration constraints. This number can be dramatically reduced by expressing architectures through feature values rather than features. A single feature *architecture*:*sun* would automatically exclude all other possible values for *architecture*, reducing the need for explicit configuration constraints. Hence, `xload` also demonstrates the benefits of functional features, and consequently the advantages of feature logic.

20.1.3 Modifying Variants

Of course, the strength of ICE is not to simulate CPP behavior, but rather to go beyond. As an example, we shall use ICICLE to create a user-specific copy of the SUN `xload` variant. The ICICLE command

```
(icicle) vi xload[sun && USER == lisa]
```

requires 0.87 seconds to create a temporary working file containing the selected variant and to invoke the `vi` text editor on it. Lisa may now perform arbitrary changes on the working file.

After having made some changes and leaving the editor, ICICLE opens the version set `xload[sun && USER == lisa]` for writing. Writing the version set is more expensive that reading: ICICLE requires 1.42 seconds to perform the write operation. This time is spent in determining the textual difference between the original and the changed version set, in determining the new features, and in writing an efficient representation.

Why is writing more expensive than reading? The vast majority of time is spent in algorithm 17.4 on page 207, which re-creates the CPP representation even for trivial changes. Storing the original CPP directives and re-using them if applicable, as discussed in section 17.3.4, already shows significant improvements here; but further speed improvements like disabling nested directives would result in files that are barely readable by humans.

20.2 A Revision History

In a second experiment, we have determined how ICE handles configuration constraints in revision histories. As case study, we have chosen the GNU MAKE program, which is publicly available in 17 revisions named 3.55 to 3.74.[3] From the GNU MAKE distribution, we have considered a single file named `commands.c`; this file happened to be modified in each revision.[4]

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 . In the FFS, a new revision *`new`* is created as a subset of an existing revision set *`old`*, using the command sequence

```
$ cd old
$ cat revision > commands.c[new]
```

such that `commands.c[`*`new`*`]` becomes a subset of `commands.c[`*`old`*`]`. Here, *`revision`* is the specific revision of `commands.c`. The ICICLE shell does not support versioned directories, but provides an equivalent short-hand notation:

```
(icicle) cat revision > commands.c[old, new]
```

This ICICLE command was repeated once for each new revision, where the individual changes were identified by `d355` (for the initial revision 3.55) to `d374` (for

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

[4] The revision history of `commands.c` is also part of the ICE distribution.

Revision	ICE	RCS	SCCS	Revision	ICE	RCS	SCCS
1 (3.55)	0.13s	0.03s	0.08s	10 (3.68)	1.05s	0.06s	0.15s
2 (3.56)	0.28s	0.02s	0.06s	11 (3.69)	1.15s	0.06s	0.16s
3 (3.60)	0.35s	0.03s	0.06s	12 (3.70)	1.60s	0.07s	0.16s
4 (3.62)	0.42s	0.05s	0.06s	13 (3.71)	2.44s	0.07s	0.16s
5 (3.63)	0.39s	0.05s	0.12s	14 (3.72)	3.15s	0.04s	0.14s
6 (3.64)	0.46s	0.03s	0.11s	15 (3.72.1)	4.01s	0.03s	0.12s
7 (3.65)	0.57s	0.02s	0.16s	16 (3.73)	3.75s	0.07s	0.15s
8 (3.66)	0.79s	0.05s	0.09s	17 (3.74)	4.40s	0.08s	0.18s
9 (3.67)	0.87s	0.06s	0.11s				

Table 20.2: Revision checkin times for ICICLE, RCS, and SCCS

the final revision 3.74). The resulting execution times for each checkin process in ICICLE, as well as the checkin times for RCS and SCCS, are shown in table 20.2.

We see that the ICE checkin time grows with the revision number, while the RCS and SCCS checkin times remain fairly constant. Could this a negative effect of NP-complete feature unification? The answer is no, because the exponential effect of feature unification looks different. In table 20.3, we have repeated the same experiment with a specially prepared ICE variant that relies on NP-complete feature unification alone—that is, all speed-ups discussed in chapter 14 have been disabled. Already with the 4th revision, execution time grows beyond all limits; we had to abort the operation after five minutes. Table 20.3 thus illustrates the necessity of specific deduction shortcuts for common SCM operations.

Careful analysis of the deduction process shows that only trivial reductions are required in this linear revision history—all that is needed is to add a new revision constraint upon each checkin, as discussed in section 12.1, and to reduce the new governing feature terms according to the revision constraints. Each of these reduction processes takes at most quasi-linear time proportional to the length of the governing feature terms; full-fledged feature unification is never required.

Revision	ICE with reduction	ICE without reduction
1 (3.55)	0.13s	0.12s
2 (3.56)	0.28s	0.77s
3 (3.60)	0.35s	3.87s
4 (3.62)	0.42s	>300.00s

Table 20.3: ICICLE checkin times with and without reduction

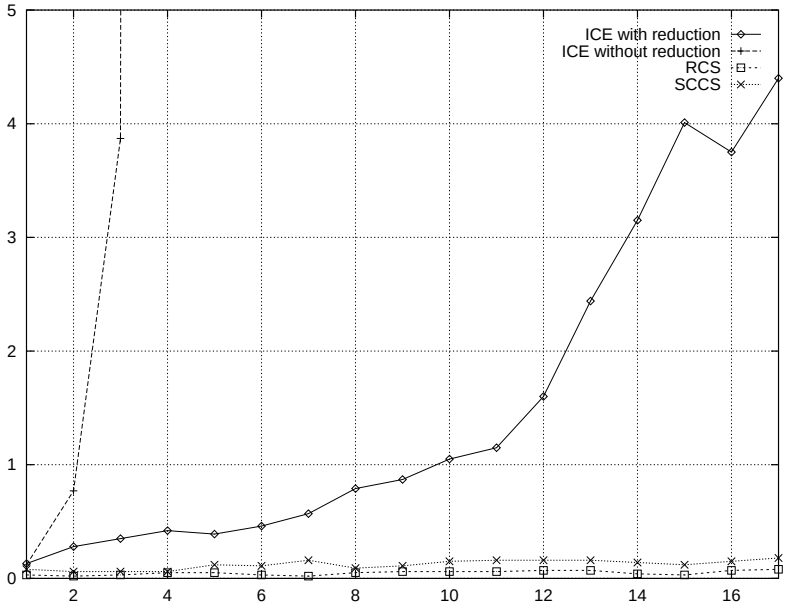

Figure 20.2: Revision checkin times for ICICLE, RCS, and SCCS

So why does the ICE checkin time grow? As discussed in section 17.2, ICE compares entire version sets when determining a new compact representation. In our example, this implies that the new revision is compared with the entire repository; code removed in some earlier revision and re-inserted in some later revision is stored only once. This is in contrast to RCS and SCCS, which compare the new revision with the previous revision only, and where the same code may be stored in multiple places. In ICE, as the repository grows with the number of revisions, so does the time for comparing it with the new revision, as shown in figure 20.2.

In our example, the checkin problem could easily be solved by comparing the latest revisions only; the data above shows that ICE is quite efficient when comparing small revision sets. But if we have multiple variants in multiple revisions, all sharing some common code, which are the "latest" revisions ICE should compare? And to which extent should variants be compared? A possible practical solution might be to flag revisions as "old" and to exclude old revisions from comparison. However, the shortened check-in time might be compensated by a

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

Figure 20.3: A multi-revision file

larger version set representation.

Speaking of version sets, what does the version set `commands.c` actually look like? An excerpt in ordinary CPP representation is shown in figure 20.3. We see that the change `d370` replaced `file->name` by `dep_name(d)` and that change `d374` introduced a parenthesized subexpression. In this excerpt, there is a maximum number of two features that govern code pieces, making the excerpt quite readable. But `commands.c` also contains code pieces governed by four features, which is a little harder to understand—but still an alternative to a set of mutual DIFF runs.

Just like the example in section 20.1, individual revision sets can be retrieved in linear time; the whole revision set can be subject to versioning, even if this requires some time for creating an appropriate CPP representation. However, if only a single revision is subjected to versioning (for instance, if a user works on an individual revision in a workspace), this is equivalent to the creation of a new

Operation	FFS		NFS	CVS
	uncached	cached		
Read (check out) file	11.2s	1.6s	1.5s	5.8s
Write (check in) file	122.0s	57.0s	12.5s	58.8s
Read (check out) project	173.0s	32.4s	28.7s	108.0s
Write (check in) project	not available[5]			

Table 20.4: FFS performance sample

revision—except that it would be identified differently, using `[USER == lisa]` instead of `d375`, for instance. Again, a "classical" SCM task like branching in a revision tree is handled efficiently.

20.3 Caching Effects

While the times shown in sections 20.1 and 20.2 could be acceptable for stand-alone SCM tools, they are totally unacceptable for a virtual file system like the FFS—a read or write operation on a file should not take more than a few milliseconds to complete. In section 19.7, we have discussed the caching mechanisms used by the FFS server; in this experiment, we shall see whether these caching mechanisms bring satisfying performance.

Table 20.4 gives typical performance times of FFS access. We have chosen the following operations: reading and writing a 4.5 MB file (a 40-page article in PostScript format) as well as reading and writing the ICE distribution, release 0.5 (8.5 MB in 1115 files). For CVS, "reading" means checking out each file from its RCS repository, and "writing" means checking in each file back again after a change.

We see that in writing files, the FFS server is four times slower than the vendor NFS server. We assume this is due to deficiencies in the FFS server implementation—for instance, the vendor NFS server is multi-threaded, while our FFS server is (yet) single-threaded. The difference between uncached and cached writing of version sets, however, is the time spent in actual work, rather than file transmission. We see that this time ($122\text{s} - 57\text{s} = 65\text{s}$) is similar to the time required by the RCS check in. This does not surprise, as both do the same work: both run DIFF to determine the differences between origin and new revision; afterwards, both create a new version set representation.

[5]Due to a bug in the current implementation of the FFS directory cache, significant times for writing projects were not available at the time of writing.

Reading version sets shows much better performance. The execution times show that reading uncached version sets is comparable with RCS repository access, while reading cached version sets can compete with the original NFS server. Stil, even better performance is possible. A real-world implementation of the FFS server would probably be multi-threaded, avoiding delays in the deduction engine, and sharing a common version set and deduction caches. Even better, it would be based on a operating system interface for virtual file systems, bypassing the NFS bottleneck for local disk access.

20.4 Conclusion

Using the ICE implementation, we have shown three simple case studies that support the efficiency claims raised in chapter 14. We see that retrieving individual versions from a ICE version set need not be more expensive than a simple CPP run. Since ICE compares entire version sets rather than only the latest revisions, adding revisions to a repository requires more time than RCS or SCCS, but yields a potentially more compact and user-readable representation. Remaining read/write delays can be compensated through caching of version sets. As soon as a version set is cached, the FFS server behaves as fast as an ordinary NFS server.

Our case studies have avoided the use of full-fledged feature unification and relied on trivial reduction schemes that have been optimized for handling standard SCM tasks. But as we know that arbitrary SCM tasks will result in arbitrary complexity, some questions must remain open:

- Which new SCM protocols are feasible on top of version sets?
- Can we find deduction shortcuts that make these SCM protocols efficient?
- If we cannot find such shortcuts, is this due to the problem or due to ourselves?

In chapter 21, we close this work with some general observations on these topics, discussing the conditions for efficient SCM tasks.

On the other hand,
a generalized solution may be more costly,
in terms of speed of execution,
memory requirements, or development time,
than the specialized solution
that is tailored to the original problem.

— CARLO GHEZZI, MEHDI JAZAYERI, DINO MANDRIOLI,
Fundamentals of Software Engineering

Chapter 21

Efficient SCM

The proofs in chapter 14, substantiated by the studies in chapter 20, show that classical SCM tasks such as version selection can be realized efficiently on top of feature logic, by exploiting orthogonality and reduction of feature terms. In this final chapter, we present some strong arguments that wherever there is an efficient implementation of a specific SCM task, there is also an efficient shortcut bypassing the complexity of feature unification; but as soon as deduction facilities are required, complexity becomes exponential. We discuss the conditions under which SCM tasks remain efficient; it turns out that a good software design according to the principles of software engineering principles is a key factor for efficient SCM.

21.1 Version Selection

The version selection mechanisms, as discussed in section 3.3, all rely on a finite set of versions, all identified by the equivalent of a simple feature terms; the selection term can also be expressed as a simple feature term.

According to proposition 14.8 on page 167, the time required for selecting versions using feature logic is proportional to the number of stored versions—just as the time required by existing implementations. As soon as variables, agreements, or disagreements are used in selection terms, checking consistency requires quadratic time for each version—again, just as in existing implementations.

When versions are identified by general feature terms, and the selection is a simple feature term, orthogonality checking and feature term reduction will often reduce the problem size considerably. Every remaining possible version must be

checked for consistency with the selection term. The same applies for identification with simple feature terms and selection with general feature terms. If general non-orthogonal feature terms are used for both identification and selection, every possible alternative must be checked, resulting in exponential complexity.

21.2 Versioning Dimensions

When multiple versioning dimensions can be selected independent from each other, as changes in the Change-Oriented Model, for example, this has no impact on complexity—as should be expected from proposition 14.7 on page 166, since every versioning dimension is represented through another feature.

Complexity becomes an issue as soon as versioning dimensions are no more orthogonal. For instance, maintaining configuration constraints, as discussed in chapter 11, has a serious impact on determining consistency of configurations, since every constraint must be satisfied. In practice, this means that users must first select a small subset of configurations in order to keep the problem size small. On the other hand, a large number of constraints, such as the constraints used for modeling revision graphs, implies a small number of possible versions, reducing the problem size as well.

21.3 Configuration Consistency

Existing SCM systems can only determine consistency of bound configurations—that is, a configuration described by a simple feature term. Even with agreements, disagreements, and variables, consistency checking can be done in quadratic time, as stated in proposition 14.1 on page 161.

When introducing ambiguities in consistency checking, such as allowing multiple versions for each component, the number of possible configurations grows exponentially and so grows the effort for consistency checking—unless orthogonality again reduces the problem size.

21.4 The Benefits of Low Coupling

Having considered these complexity problems, how can we keep SCM efficient and our software maintainable? From the previous sections, we see that orthogonality of feature terms is a major issue in keeping the size of SCM problems small. If the feature terms describing two components A and B are orthogonal, we can select arbitrary versions of A and B without affecting the consistency of their respective configuration. In software engineering, this property is also known as *low coupling* of modules. Coupling is a measure of strength of interconnections

between components; low coupling is a desirable property because it keeps evolving programs manageable—notably, we can make a change (create a new version) of *A* or *B* without affecting the other component. Low coupling is obtained through basic software engineering principles such as abstraction, parameterization, generalization, localization, and, most of all, anticipation of change. Since low coupling implies orthogonality and vice versa, low coupling between components has immediate benefits in simplifying SCM problems and thus keeping evolving software manageable.

21.5 The Benefits of High Cohesion

Another desirable property in software engineering is *high cohesion* within a component. Cohesion is a measure of how well a component fits together; high cohesion expresses that all parts of a component should contribute to its implementation, which also means that a change in a component part implies changes in other parts of the component. In our model, high cohesion between components leads to many configuration constraints between these components, such that the actual number of possible configurations is kept small and thus becomes manageable as well.

21.6 Maintaining Unstructured Software

The problematic cases for automated deduction are exactly those cases that make software difficult to maintain: a number of unstructured constraints involving components all over the system, expressing chaotic interconnections between components. In such cases, the only remedy may be to *restructure* the system such that variance is kept as local as possible, eliminating version dependencies between components and thus reducing complexity. This can be done by examining the configuration space [KS94] and to reengineer it [Sne96] such that dependencies between configuration threads are significantly reduced—as is the complexity of SCM tasks.

Where such a restructuring is not possible, automated deduction like feature unification can help to keep SCM tasks manageable. It may especially be helpful to manage a temporary situation, such as lots of developers creating lots of temporary variants, whose consistency must be checked and expressed. But due to its complexity, automated deduction does not scale up beyond a certain problem size. In the long term, applying the principles of software engineering to avoid permanent, non-orthogonal variance, is the only way to keep SCM tasks manageable and the product maintainable.

21.7 Conclusion

Feature unification is the standard technique for deciding consistency of general feature terms. Feature unification is NP-complete and thus applicable to small problems only. The problem of deciding consistency can be broken down in smaller subproblems if the feature term breaks down into orthogonal parts, that is, parts without common features or variables. The technique of partial evaluation leads to efficient decision of consistency for simple feature terms. Both determining orthogonality and partial evaluation already suffice to realize standard SCM tasks efficiently on top of feature logic.

The most difficult SCM problem is to determine the consistency of abstract configurations, where the feature terms describing the individual components are non-orthogonal. A well-structured configuration space, as obtained through the principles of software engineering, ensures orthogonality of feature terms and thus keeps SCM problems manageable. A small amount of non-orthogonal ambiguity can be tolerated using feature unification—for instance, temporary, non-orthogonal variance as it occurs during parallel development.

Today, the field of ATP has produced several deduction techniques for propositional logic that might turn out useful for feature logic as well. The application of these deduction techniques may raise the amount of ambiguity tolerance in practical SCM systems, maybe even beyond any ambiguity as found in today's software systems. This should allow SCM systems to manage several parallel development paths at once and ensure consistency across all ambiguities. But still, a good software design is the key factor to keep evolving software maintainable.

In skating over thin ice our safety is in our speed.

— RALPH WALDO EMERSON

Part Five

Odds and Ends

Chapter 22

Conclusion

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

In part three, we have shown how version sets capture and integrate the SCM concepts introduced in part one. The covered SCM concepts range from versions to components, from configurations to revisions, from changes to constraints, and from relationships to system modeling. The principal results fulfill the requirements raised in chapter 6:

Unified versioning. Version sets provide one single formalism to express all versioning dimensions as well as constraints on them, integrating SCM concepts like revisions, variants, workspaces, and configurations in one single model. The SCM policy is not constrained by decisions made in lower SCM layers.

Integration of changes and revisions. Configuration constraints, expressed in feature logic, allow us to capture the entire range of temporal versioning—from the rigidness of versions-oriented models to the flexibility of change-oriented models.

Consistency checking under ambiguity. Through feature logic, we deduce the features and the consistency of configurations as well as derived compo-

nents and thus describe how features propagate in the SCM process. Inconsistencies are detected even when the configuration description is incomplete or ambiguous. Ambiguity is not only tolerated in consistency checking; at all SCM layers, sets rather than single items are the primary objects of SCM tasks and procedures.

Our implementation of the version set model in ICE has shown that this foundation has several user-visible benefits. Through the FFS, users can access version sets consisting of arbitrary combinations of revisions, changes, variants, and workspaces. Individual versions are accessed as files; version sets as a whole can be accessed via version directories or through the CPP representation. On top of the FFS, specific SCM protocols are realized efficiently through simple file operations on version sets. These features make ICE a universal platform for individual SCM policies and demonstrate the flexible and unifying nature of the version set model.

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

Beyond feature logic. Feature logic, as defined by Smolka, is not appropriate for all SCM aspects. As discussed in section 10.8, an extension to specify set values would be most helpful to overcome the difficulties in specifying aggregations (section 10.4). Also, feature logic does not distinguish between provided and required features. There is no notion of cardinality and ambiguity; hence, preference and default operators (section 9.3) cannot be defined in feature logic. On the other hand, one must be careful that *ad hoc* extensions for SCM purposes do not endanger the generality of feature logic.

Versioned Relationships. In chapter 3, we have introduced relationships as a means to represent the structure of a system; typical relationships included *is-instance-of* to represent the dependency between source components and derived components, or *is-a-part-of* to model the aggregation of components into systems. Such relationships are subject to versioning just as the individual components are—that is, a system model may occur in several variants or may be revised frequently. We want to model such relationships as features and roles, providing a natural link between object versioning and relation versioning.

Efficient integration of SCM concepts. We have seen that all SCM concepts introduced in part one can be realized on top of the version set model, and

the ICE system already shows how these concepts can be integrated into a single SCM system. We also have identified possible complexity problems with non-orthogonal SCM concepts, especially variance. Based on further experience with the FFS and the underlying deduction engine, we want to investigate how far integration of SCM concepts can go without endangering efficiency. Furthermore, we want to see which other 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.

Support of the SCM process. Our long-term goal is to establish a layered SCM model where each layer is defined in concepts of the next lower layer. In this task, we pursue a bottom-up approach. Having supplied feature logic as an SCM foundation and proposed version sets as SCM primitives, our next step would be to specify the SCM protocol in terms of transitions between version sets, and to specify the SCM process in terms of transitions between SCM protocols. We imagine organizing the SCM process entirely by manipulating component features—changing their state from *proposed* via *tested* to *released*. Eventually, we hope to map the entire SCM process to operations on version sets denoted by feature logic, providing a uniform semantic foundation for all SCM layers.

Although we would have liked to present
the ultimate version management model,
such a model is not likely to exist for some time.

— RANDY H. KATZ, Version Modeling in Engineering Databases

Appendix A

Frequently Asked Questions

In this appendix, we have summarized the most frequently answered questions about the version set model and ICE.

Note: Questions A.3.1 to A.3.9 are taken from [Est95, p. 80]*, reproduced in section 6.4 on page 57.*

A.1 General Questions

A.1.1 What are the main achievements of your work?

There are three of them, as discussed in chapter 22:

- The unification of SCM versioning concepts in one single formalism.
- The integration of change-oriented and version-oriented versioning through configuration constraints.
- The ability to check and propagate consistency even for incomplete or ambiguous configurations.

A.1.2 Why do you neglect SCM process issues?

We believe in the importance of separating SCM issues, such as policy, protocol, and primitives. We also believe that you cannot define a layer without first defining its foundation. In this work, we have provided such a foundation covering the SCM primitives layer, extending a little into the SCM protocol layer. As soon as the SCM protocol layer will be fully understood, we can turn to the policy layer, covering the SCM process.

A.1.3 Why don't you compare the version set model to other integrated SCM models?

To the best of our knowledge, there aren't any.

A.2 Topic: Feature Logic

A.2.1 Why do you use feature logic?

Because it allows us to combine attribute descriptions with boolean operations. Both play a central role in SCM, notably for identification and retrieval.

A.2.2 But you could also use first-order logic, could you?

In principle, yes—there are few domains, if any, where first-order logic would not suffice. Unfortunately, first-order logic is not attribute-oriented. Modeling the functional nature of attributes or features in first-order logic leads to a large number of explicit constraints, which are difficult to read and to maintain. Feature logic is much more appropriate here.

A.2.3 Why didn't you use some more general description logic?

A description logic more general than feature logic would probably also do the job. However, several intrinsic properties of feature logic would have to be modeled explicitly, such as feature propagation and the functional nature of features. On the other hand, such an explicit modeling allows for alternate feature propagation schemes that may be appropriate in several domains. Try it out.

A.2.4 Why didn't you use an existing theorem prover?

First, existing theorem provers are batch-oriented. This is not appropriate for our model, where thousands of comparatively small problems must be solved in a minimum amount of time. Second, building a theorem prover ourselves allowed us to adapt it to the specific needs of SCM and to develop appropriate deduction techniques.

A.2.5 Is the formalization of SCM concepts necessary at all?

Yes. The SCM community has been inventing and implementing for years, realizing great SCM tools and systems. Now is the time to look back and evaluate what has actually happened, to provide a foundation for yet smarter SCM support.

A.2.6 Do I really have to learn feature logic to solve SCM problems?

No, not at all. A system like ICE shows that all this logic can be hidden behind a set of familiar and well-understood representations.

A.3 Topic: The Version Set Model

A.3.1 Is the versioning model linked to the data model, the product model (schema), the transaction model (uni-version subdatabases), or is it independent?

The version set model is orthogonal to any other software models and independent from a specific representation.

A.3.2 At what granularity are deltas expressed, computed and merged—on the base of whole files, text lines, or syntactical entities?

The model is independent from a specific representation of version sets. Our implementation expresses, computes, and merges deltas on the base of arbitrary sequences of characters, notably text lines.

A.3.3 And how is versioning combined with e.g. inheritance and parameterization?

Inheritance is realized through the subsumption relation; that is, a version set is a subset of another version set, inheriting its features. Parameterization is realized through incremental version selection, constraining version sets through further feature values (or parameters).

A.3.4 Does basic versioning only apply to atomic and textual objects, and not to composites or to the entire database?

Versioning applies to primitives (chapter 9) as well as to arbitrary composites (chapter 10).

A.3.5 How to version relationships, and thus configurations?

Relationships may be modeled as features of the originating version sets. Configurations are independent from relationships, and independently versioned, although an SCM system may enforce the specification of configurations through relationships.

A.3.6 How to express intentional version selection, and how to express constraints, defaults and preferences for such selections?

Defaults and preferences are realized through preference operators (section 3.3.3) The version set model handles ambiguities at all levels; defaults and preferences are thus needed only if an application requires unambiguous configurations in order to proceed.

A.3.7 Is the selection based on symbolic attribute values, that together constitute a version space?

Yes. That's what feature logic is for.

A.3.8 Can the constraints and attribute domains evolve over time?

Yes. Constraints and attributions are subject to versioning as well.

A.3.9 Given a system model with objects and relationships: is the product selection (AND-closure) done before the version selection within each group (OR-choices), or vice versa, or intertwined?

Selection is unconstrained, i.e. intertwined.

A.4 Topic: Complexity

A.4.1 I saw feature unification is NP-complete! How dare you suggest an NP-complete method for practical usage?

Feature unification is NP-complete, yes, leading to exponential complexity in the worst case. The emphasis here is on "worst case"—nearly all examples in this book run very efficiently in practice. The rule of thumb is: if an SCM concept has been implemented efficiently somewhere else, then there is an appropriate deductive shortcut.

A.4.2 And what about the integration of SCM concepts? Can I really combine revisions, variants, and workspaces just as I like?

In principle, yes. But you will probably run into complexity problems—arbitrary combinations means arbitrary feature terms, which leads to arbitrary complexity.

A.4.3 Can I something do to avoid complexity problems?

The general rule is to keep versioning dimensions either very orthogonal or very non-orthogonal, as discussed in sections 21.4 and 21.5. In short: follow a well-established SCM model, like the ones realized in current SCM tools and systems. None of these models imposes any severe complexity problems (otherwise, they would not work efficiently). Follow the principles of software engineering, notably abstraction, generalization and localization.

A.5 Topic: Applications

A.5.1 What are the new features of your application?

Again, there are three of them:

- Handling of multiple versions through a CPP-like representation.
- Incremental configuration refinement through a virtual file system.
- Workspace management through versioned directories.

A.5.2 I saw you use `#ifdef` to represent versions. Isn't `#ifdef` bad SCM practice?

One should be careful not to confound message and messager. It is the CPP tool that causes the problem, since it forces you to maintain all versions at once. Also, `#ifdef` is commonly used for permanent variance, which should also be avoided. In the ICE context, `#ifdef` is just a representation for multiple versions, as is an SCCS repository or an ordinary data base.

A.5.3 Do you really want us to read and write `#ifdef`'ed code?

You don't have to. If you don't see `#ifdef` today, you don't need to see it with ICE either. You see (and possibly write) `#ifdef` as soon as you want to work on several versions at once, or if you want to determine the differences between some versions. You never have to read or write all versions at once, as CPP forces you to.

A.5.4 Wouldn't defining CPP variables like `d1` break my code?

No. ICE does not perform macro expansions or alter the code in any way.

A.5.5 My SCM vendor says we should use databases instead of file systems. So, why do you use a file system instead of a database?

Your vendor is right. Databases are much more secure than file systems. On the other hand, other vendors will tell you that a file system is much better suited for integration into a software development environment. The probably best thing to do is to use a database for version storage and a file system for version access. For ICE, this means future work.

A.5.6 What is the performance of ICE on large-scale projects?

Unfortunately, the current ICE implementation leaves too much to be desired for large-scale projects. Open issues include transaction safety and general robustness, as well as efficient usage of ressources, notably memory requirements. The biggest problem of all is the lack of a higher-level interface. However, there is no reason why the results of chapter 20 should not be applicable to large-scale projects.

A.5.7 Will ICE improve my productivity?

This is a question which should be verified empirically, and thus requires a fully usable ICE system; see question A.5.6. Generally, we think that the best way to improve productivity is a well-understood and well-supported software process. An adaptive environment like ICE can help you implement a process according to your needs, rather than a process enforced by some SCM vendor. Other ICE features such as transparent version set access or the ability to view and change version sets may also improve the individual productivity.

A.5.8 I want to use ICE. Is there anything I can do?

Support this work. Help us designing and building a foundation for better SCM environments.

Alles Wissen und alle Vermehrung unseres Wissens
endet nicht mit einem Schlußpunkt,
sondern mit einem Fragezeichen.

— HERMANN HESSE, Lektüre für Minuten

Some say the world will end in fire,
Some say in ice.

— ROBERT FROST, Poems

Appendix B

Obtaining ICE

A free ICE distribution is available for UNIX systems under the conditions of the GNU general public license. The ICE distribution contains the source code and the documentation of the ICICLE shell as well as the FFS server and the SKATE configuration browser as described in this work.

The ICE distribution and related technical reports are available through the ICE WWW page,

```
http://www.cs.tu-bs.de/softech/ice/
```

and through the ICE FTP site,

```
ftp://ftp.ips.cs.tu-bs.de/pub/local/softech/ice/ .
```

Building ICE requires a C++ compiler such as GNU C++. Running SKATE requires a Tcl/Tk interpreter. Both GNU C++ and Tcl/Tk are freely available from several sources.

The ICE maintainers can also be reached by electronic mail. Send mail to:

`ice-bugs@ips.cs.tu-bs.de`	— for bug reports and suggestions
`ice@ips.cs.tu-bs.de`	— everything else.

Be aware that the ICE maintainers cannot provide full-time technical support, although they will try to help as much as they can.

A drawback of attempting to impose a standard is that it will quickly become outmoded.

— DAVID LEBLANG, The CM Challenge

Acknowledgements

This research owes to the suggestions and assistance of several people, who deserve all my thanks and acknowledgments. First of all comes the software technology department at Braunschweig. Many thanks go to Jens Krinke for careful proofreading. Bernd Fischer, Franz-Josef Grosch, and Christian Lindig were a persevering and helpful audience for any new ideas.

The implementation of ICE was possible only through the contributions of several student workers. Dirk Babel realized the SKATE configuration browser. Michael Brandes conceived ICE MAKE. Lars Düning implemented the CPP representation of version sets. Andreas Mende extended ICICLE with multi-version merging. Olaf Pfohl built the FFS server. Christina Trenkner integrated arithmetic constraint solving in Smolka's feature unification, which was originally coded by Marc Ziehmann. Other parts of ICE were contributed by Ahmad Alsaadi, Thorsten Sommer, Ragnar Stahl, and Rolf Watermann.

ICE itself relies on free software such as GNU DIFF, GNU MAKE, READLINE, AUTOCONF, and the GNU C++ compiler. Many thanks go to the people of the Free Software Foundation for developing and maintaining these bullet-proof products. Free software was also used in typesetting this book, using the great and free TEX/LATEX system from Donald Knuth, Leslie Lamport and others.

Finally, I owe a great deal to my teachers. Wolfgang Bibel taught me the techniques of automated deduction. Gregor Snelting set me on the right track by proposing feature unification as a means to determine configuration consistency. And special thanks go to Petra Funk, for all the moral and technical support she gave me.

Parts of this work have been supported by the Deutsche Forschungsgemeinschaft, grants Sn11/1-1 and Sn11/1-2.

About the Author

Curriculum Vitae

28 October 1965	Born in Hanau, Germany
1970–1973	Elementary school, Großauheim, Germany
1973–1978	Collège André Malraux, Bangui, Central Africa
1978–1984	Karl-Rehbein-Gymnasium, Hanau, Germany
1984	Final examination (Abitur)
1984–1991	Computer science studies, Technical University of Darmstadt, Germany
1991	Computer science diploma (Dipl.-Inform.)
1991–today	Research assistant, Technical University of Braunschweig, Germany

Lebenslauf

28. Oktober 1965	Geboren in Hanau/Main
1970–1973	Grundschule, Großauheim/Main
1973–1978	Collège André Malraux, Bangui, Zentralafrika
1978–1984	Karl-Rehbein-Gymnasium, Hanau
1984	Abitur
1984–1991	Studium der Informatik, Technische Hochschule Darmstadt
1991	Diplom (Dipl.-Inform.)
1991–heute	Wissenschaftlicher Mitarbeiter, Technische Universität Braunschweig

Publications

[1] Gregor Snelting and Andreas Zeller. Inferenzbasierte Werkzeuge in NORA. In *Proc. Softwaretechnik 93*, volume 13(3) of *Softwaretechnik-Trends*, pages 25–32, Dortmund, Germany, November 1993. GI. In German.

[2] Gregor Snelting, Bernd Fischer, Franz-Josef Grosch, Matthias Kievernagel, and Andreas Zeller. Die inferenzbasierte Softwareentwicklungsumgebung NORA. *Informatik—Forschung und Entwicklung*, 9(3):116–131, August 1994. In German.

[3] Andreas Zeller and Gregor Snelting. Handling version sets through feature logic. In Wilhelm Schäfer and Pere Botella, editors, *Proc. 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 191–204, Sitges, Spain, September 1995. Springer-Verlag.

[4] Andreas Zeller. A unified version model for configuration management. In Gail Kaiser, editor, *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 20 (4) of *ACM Software Engineering Notes*, pages 151–160, Washington, DC, October 1995. ACM Press.

[5] Andreas Zeller and Dorothea Lütkehaus. DDD—A free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, January 1996.

[6] Andreas Zeller. Smooth operations with square operators—The version set model in ICE. In Ian Sommerville, editor, *Proc. 6th International Workshop on Software Configuration Management*, volume 1167 of *Lecture Notes in Computer Science*, pages 8–30, Berlin, Germany, March 1996. Springer-Verlag.

[7] Andreas Zeller. Software configuration with feature logic. In Franz Baader, Hans-Jürgen Bürckert, Andreas Günter, and Werner Nutt, editors, *Proc. of the Workshop on Knowledge Representation and Configuration (WRKP'96)*, volume 96-04 of *DFKI-Dokumente*, pages 79–83, Dresden, Germany, September 1996. DFKI, Saarbrücken, Germany.

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

[9] Andreas Zeller and Gregor Snelting. Unified versioning through feature logic. *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997. To appear.

Bibliography

[Abr95] Per Abrahamsen. The CPP-parse-edit mode for EMACS. Part of the EMACS distribution, 1995.

[AFK+95] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In Estublier [Est95], pages 194–214.

[AK86] Hassan Aït-Kaci. An algebraic semantics approach to the effective resolution of type equations. *Theoretical Computer Science*, 45:293–351, 1986.

[AKN86] Hassan Aït-Kaci and Roger Nasr. Login: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 1986(3):186–215, 1986.

[AKP91] Hassan Aït-Kaci and Andreas Podelski. Towards a meaning of LIFE. In J. Maluszyński and M. Wirsing, editors, *Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 255–274, Passau, Germany, August 1991. Springer-Verlag.

[AS95] Paul Adams and Marvin Solomon. An overview of the CAPITL software development environment. In Estublier [Est95], pages 1–34.

[Bab96] Dirk Babel. Ein deduktiver Konfigurationsbrowser für ICE. Master's thesis, Technical University of Braunschweig, Germany, 1996.

[BDFW91] A. Brown, S. Dart, P. Feiler, and K. Wallnau. The state of automated configuration management. Technical Report CMU/SEI-ATR-91, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, September 1991.

[Ber90] Brian Berliner. CVS II: Parallelizing software development. In *Proc. of the 1990 Winter USENIX Conference*, Washington, D.C., 1990.

[Ber94] Valdis Berzins. Software merge: Semantics of combining changes to programs. *ACM Transactions on Software Engineering and Methodology*, 16(6):1875–1903, November 1994.

[BESS96] Naser S. Barghouti, Wolfgang Emmerich, Wilhelm Schäfer, and Andrea Skarra. Information management in process-centered software engineering environments. In Alfonso Fuggetta and Alexander Wolf, editors, *Software Process*, volume 4 of *Trends in Software*, chapter 3, pages 53–87. John Wiley & Sons, Chichester, UK, 1996.

[BFH+94] F. Baader, E. Franconi, B. Hollunder, B. Nebel, and H.-J. Profitlich. An empirical analysis of optimization techniques for terminological systems, or making KRIS a move on. *Journal of Applied Intelligence*, 4:109–132, 1994.

[BH91] Franz Baader and Bernhard Hollunder. KRIS: Knowledge representation and inference system. *ACM SIGART Bulletin*, 2(3):8–14, 1991.

[BHR95] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[Bib87] Wolfgang Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, Wiesbaden, second edition, 1987.

[Bib92] Wolfgang Bibel. *Deduktion: Automatisierung der Logik*, volume 6.2 of *Handbuch der Informatik*. Oldenbourg, München, Wien, 1992. In German.

[BJSS90] Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauß. Unification in boolean rings and abelian groups. In Kirchner [Kir90], pages 267–295.

[BL84] Ronald. J. Brachman and H. J. Levesque. The tractability of subsumption in frame-based description languages. In *Proc. of the 4th National Conference of the American Association for Artificial Intelligence*, pages 34–37, Austin, Texas, August 1984.

[BMPS+91a] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. The CLASSIC knowledge representation system. *ACM SIGART Bulletin*, 2(3):108–113, 1991.

[BMPS+91b] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick, and A. Borgida. Living with CLASSIC: When and how to use a KL-ONE-like language. In J. Sowa, editor, *Principles of Semantic Networks*, pages 401–456. Morgan Kaufmann, San Mateo, California, 1991.

[Boo47] George Boole. *The Mathematical Analysis of Logic, Being an Essay Towards a Calculus of Deductive Reasoning.* Macmillan, Cambridge, 1847. Reprints 1948, 1951 (Blackwell, Oxford).

[Bra96] Michael Brandes. Deduktive Programmkonstruktion auf Basis von MAKE. Master's thesis, Technical University of Braunschweig, Germany, December 1996. In German.

[BS86] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM TOPLAS*, 8(4):547–576, October 1986.

[Buf95] Jim Buffenbarger. Syntactic software merging. In Estublier [Est95], pages 153–172.

[CGS91] R. Cunis, A. Günter, and H. Strecker. *Das PLAKON-Buch.* Number 266 in Informatik-Fachberichte. Springer-Verlag, Berlin, Heidelberg, New York, 1991. In German.

[Cle88] Geoffrey M. Clemm. The Odin specification language. In Winkler [Win88], pages 145–158.

[Cle93] Geoffrey M. Clemm. *The Odin System Reference Manual.* University of Colorado at Boulder, 1993.

[Cou89] William Courington. The Network Software Environment. Technical Report FE 197-0, Sun Microsystems, Inc., February 1989.

[CW96a] Reidar Conradi and Bernhard Westfechtel. Configuring versioned software products. In Sommerville [Som96], pages 88–109.

[CW96b] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. Technical Report AIB 96-10, RWTH Aachen, Germany, October 1996.

[Dar91] Susan Dart. Concepts in configuration management systems. In Feiler [Fei91b], pages 1–18.

[Dit89] K. R. Dittrich. The DAMOKLES database system for design applications: its past, its present, and its future. In K. H. Bennett, editor, *Software Engineering Environments: Research and Practice*, pages 151–171. Ellis Horwood Books, Durhan, UK, 1989.

[Dün94] Lars Düning. Variantenmanagement mit Feature-Termen und dem C-Präprozessor. Project report, Technical University of Braunschweig, Germany, April 1994. In German.

[EC94] Jacky Estublier and Rubby Casallas. The Adele configuration manager. In Tichy [Tic94], chapter 4, pages 99–133.

[EC95] Jacky Estublier and Rubby Casallas. Three dimensional versioning. In Estublier [Est95], pages 118–135.

[EGLT76] K. Eswaran, J. Gray, P. Lorie, and I. Traiger. On the notions of consistency and predicate locks in a database system. *Communications of the ACM*, 9(11):624–633, November 1976.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments—Part 1: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, 1992.

[Est85] Jacky Estublier. A configuration manager: The Adele data base of programs. In *Proc. of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, Harwichport, Ma., June 1985.

[Est88] Jacky Estublier. Configuration management: The notion and the tools. In Winkler [Win88], pages 38–61.

[Est95] Jacky Estublier, editor. *Software Configuration Management: selected papers / ICSE SCM-4 and SCM-5 workshops*, volume 1005 of *Lecture Notes in Computer Science*, Seattle, Washington, October 1995. Springer-Verlag.

[ESW93] Wolfgang Emmerich, Wilhelm Schäfer, and Jim Welsh. Databases for software engineering environments—the goal has not yet been attained. In Ian Sommerville and Manfred Paul, editors, *Proc. 4th European Software Engineering Conference*, volume 717 of *Lecture Notes in Computer Science*, pages 145–162, Garmisch-Partenkirchen, Germany, September 1993. Springer-Verlag.

[FDD88] Peter H. Feiler, Susan Dart, and G. Downey. Evaluation of the Rational environment. Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, July 1988.

[Fei91a] Peter H. Feiler. Configuration management models in commercial environments. Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, March 1991.

[Fei91b] Peter H. Feiler, editor. *Proc. 3rd International Workshop on Software Configuration Management*, Trondheim, Norway, June 1991. ACM Press.

[Fel79] Stuart I. Feldman. Make—A program for maintaining computer programs. *Software—Practice and Experience*, 9:255–265, April 1979.

[Fel93] Stuart Feldman, editor. *Proc. 4th International Workshop on Software Configuration Management (Preprint)*, Baltimore, Maryland, May 1993.

[Fis93] Bernd Fischer. A new feature unification algorithm. Computer Science Report 93-01, Technical University of Braunschweig, Germany, December 1993. Submitted for publication.

[FKR94] Glenn Fowler, David Korn, and Herman Rao. *n*-DFS: The multiple dimensional file system. In Tichy [Tic94], chapter 5, pages 135–154.

[FKS95] Bernd Fischer, Matthias Kievernagel, and Gregor Snelting. Deduction-based software component retrieval. In Köhler et al. [KGGW95], pages 1–5.

[Gad95] Christophe Gadonna. *MISTRAL User Manual V1.* Laboratoire de Génie Informatique, Grenoble, May 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachussetts, 1994.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Inc., 1991.

[GMSW89] W. Morven Gentleman, Steven A. MacKay, Darlene A. Stewart, and Marceli Wein. Commercial realtime software needs different configuration management. In Tichy [Tic89], pages 152–161.

[Gra81] J. Gray. The transaction concept: Virtues and limitations. In *Proc. of the International Conference on Very Large Data Bases*, 1981.

[Gul93] Bjørn Gulla. The constraint diagram: An approach to visualizing the version space. In Feldman [Fel93], pages 112–122.

[Gun96] Carl A. Gunter. Abstracting dependencies between software configuration items. In David Garlan, editor, *Proc. 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 21 (6) of *ACM Software Engineering Notes*, pages 167–178, San Francisco, October 1996. ACM Press.

[Har89] Richard Harter. Version management and change control; systematic approaches to keeping track of source code and support files. *Unix World*, 6(6), June 1989.

[HK92] T. Hung and P. F. Kunz. Unix code management and distribution. Technical Report SLAC-PUB-5923, Stanford Linear Accelerator Center, Stanford, California, September 1992.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[HVT96] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. An empirical study of delta algorithms. In Sommerville [Som96], pages 49–66.

[IEE88] The Institute of Electrical and Electronics Engineers, Inc., New York. *IEEE Guide to Software Configuration Management*, 1988. ANSI/IEEE Standard 1042-1987.

[IEE90] The Institute of Electrical and Electronics Engineers, Inc., New York. *IEEE Guide to Software Configuration Management Plans*, 1990. ANSI/IEEE Standard 828-1990.

[ISO90] The International Organization for Standardization and The International Electrotechnical Commission. *Programming Languages—C*, December 1990. ISO/IEC International Standard 9899:1990 (E).

[Joh88] M. Johnson. Attribute-value logic and the theory of grammar. Technical Report CSLI Lecture Notes 16, Stanford University, Center for the Study of Language and Information, 1988.

[Kat90] Randy H. Katz. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408, December 1990.

[Kay79] M. Kay. Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, 1979.

[Kay84] M. Kay. Functional unification grammar: A formalism for machine translation. In *Proc. 10th International Joint Conference on Artificial Intelligence*, pages 75–78, Stanford, 1984.

[KB82] R. M. Kaplan and J. Bresnan. Lexical-functional grammar: A formal system for grammatical representation. In J. Bresnan, editor, *The Mental Representation of Grammatical Relations*, pages 173–381. MIT Press, Cambridge, Mass., 1982.

[KGGW95] Jana Köhler, Fausto Giunchiglia, Cordell Green, and Christoph Walther, editors. *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, Montréal, August 1995.

[Kie92] Thilo Kielmann. Using PROLOG for software system maintenance. In *Proc. of the First International Conference on the Practical Application of PROLOG*, London, UK, April 1992.

[Kir90] Claude Kirchner, editor. *Unification*. Academic Press, London, 1990.

[KR86] R. T. Kasper and W. C. Rounds. A logical semantics for feature structures. In *Proc. of the 24th Annual Meeting of the ACL*, pages 257–265, Columbia University, New York, 1986.

[KR89] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Carl Hanser, Prentice–Hall International, 2. edition, 1989.

[KS94] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proc. 16th International Conference on Software Engineering*, pages 49–57, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[LCD+89] Anund Lie, Reidar Conradi, Tor M. Didriksen, Even-André Karlsson, Svein O. Hallsteinsen, and Per Holager. Change oriented versioning in a software engineering database. In Tichy [Tic89], pages 56–65.

[LCS88] David B. Leblang, Robert P. Chase, and Howard Spilke. Increasing productivity with a parallel configuration manager. In Winkler [Win88], pages 21–37.

[LDC+89] Anund Lie, Tor M. Didriksen, Reidar Conradi, Even-André Karlsson, Svein O. Hallsteinsen, and Per Holager. Change-oriented versioning. In C. Ghezzi and J. A. McDermid, editors, *Proc. 2nd European Software Engineering Conference*, volume 387 of *Lecture Notes in Computer Science*, pages 191–202, Coventry, September 1989. Springer-Verlag.

[Leb94] David B. Leblang. The CM challenge: Configuration management that works. In Tichy [Tic94], chapter 1, pages 1–37.

[LHPT95] Paul Lukowicz, Ernst A. Heinz, Lutz Prechelt, and Walter F. Tichy. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 18(1):9–18, January 1995.

[Lin95] Christian Lindig. Concept-based component retrieval. In Köhler et al. [KGGW95], pages 21–25.

[LL87] M. Lacroix and P. Lavency. Preferences: Putting more knowledge into queries. In Peter M. Stocker and William Kent, editors, *Proc. of the 13th International Conference on Very Large Data Bases*, pages 217–225, Brighton, 1987.

[LM88] Andreas Lampen and Axel Mahler. An object base for attributed software objects. In *Proc. of the Fall '88 EUUG Conference*, pages 95–105, Cascais, October 1988.

[MA96] Boris Magnusson and Ulf Asklund. Fine grained version control of configurations in COOP/Orm. In Sommerville [Som96], pages 31–48.

[Mac91] Robert MacGregor. Inside the LOOM classifier. *ACM SIGART Bulletin*, 2(3):88–92, 1991.

[Mac94] David MacKenzie. *Autoconf—Creating Automatic Configuration Scripts*. Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA, November 1994. Distributed with Autoconf.

[Mah94] Axel Mahler. Variants: Keeping things together and telling them apart. In Tichy [Tic94], chapter 3, pages 39–69.

[MAM93] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In David Notkin, editor, *Proc. of the first ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 33–41, Los Angeles, December 1993. ACM Press.

[Man94] Suresh Manandhar. An attributive logic of set descriptions and set operations. In *Proc. of the 32nd Annual Meeting of the Association for Computational Linguistics (ACL '94)*, Las Cruces, New Mexico, June 1994.

[MC96] Josephine Micallef and Geoffrey M. Clemm. The Asgard system: Activity-based configuration management. In Sommerville [Som96], pages 175–186.

[McD82] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.

[McD84] J. McDermott. R1 revisited: Four years in the trenches. *AI Magazine*, 5(Fall):21–32, 1984.

[MDW91] E. Mays, R. Dionne, and R. Weida. K-Rep system overview. *ACM SIGART Bulletin*, 2(3):93–97, 1991.

[Men96] Andreas Mende. Verwaltung von Revisionen und Arbeitsbereichen in ICE. Master's thesis, Technical University of Braunschweig, Germany, December 1996. In German.

[MLG+93] Bjørn P. Munch, Jens-Otto Larsen, Bjørn Gulla, Reidar Conradi, and Even Andre Karlsson. Uniform versioning: The change-oriented model. In Feldman [Fel93], pages 188–196.

[MM85] W. Miller and Eugene Myers. A file comparison program. *Software—Practice and Experience*, 15(11):1025, 1985.

[MNR83] D. McLeod, K. Narayanaswamy, and Bapa Rao. An approach to information management for CAD/VLSI applications. In *Proceedings of the SIGMOD Conference on Databases for Engineering Applications*, pages 39–50, San Jose, California, May 1983.

[Mor88] Thomas M. Morgan. Configuration management and version control in the Rational programming environment. In *Proceedings of the Ada-Europe International Conference*, pages 18–28. Cambridge University Press, June 1988.

[MR87] M. Drew Moshier and William C. Rounds. A logic for partially specified data structures. In Steve Muchnik and Mark Wegman, editors, *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, pages 156–167, Munich, January 21-23 1987. ACM Press.

[Mun96] Bjørn P. Munch. HiCoV: Managing the version space. In Sommerville [Som96], pages 110–126.

[Nar89] K. Narayanaswamy. A text-based representation for program variants. In Tichy [Tic89], pages 30–33.

[Neb90] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1990.

[Nic91] Peter Nicklin. Managing multi-variant software configurations. In Feiler [Fei91b], pages 53–57.

[NS89] B. Nebel and G. Smolka. Representation and reasoning with attributive descriptions. In K. H. Bläsius, U. Hedstück, and C.-R. Rollinger, editors, *Sorts and Types in Artificial Intelligence*, volume 256 of *Lecture Notes in Artificial Intelligence*, pages 112–139, Eringerfeld, April 1989. Springer-Verlag.

[OG90] B. O'Donovan and J. B. Grimson. A distributed version control system for wide area networks. *Software Engineering Journal*, September 1990.

[OHPDB92] Eduardo Ostertag, James Hendler, Rubén Prieto-Díaz, and Christine Braun. Computing similarity in a reuse library system: An AI-based approach. *ACM Transactions on Programming Languages and Systems*, 1(3):205–228, July 1992.

[PD87] Rubén Prieto-Díaz. Classifying software for reusability. *IEEE Software*, 4(1), January 1987.

[Pel91] Christof Peltason. The BACK system—an overview. *ACM SIGART Bulletin*, 2(3):114–119, 1991.

[PF89] Erhard Ploedereder and Adel Fergany. The data model of the configuration management assistant. In Tichy [Tic89], pages 5–13.

[Pfo96] Olaf Pfohl. FFS – ein versioniertes Dateisystem auf Basis von Feature-Logik. Master's thesis, Technical University of Braunschweig, Germany, 1996.

[Rei89] Christoph Reichenberger. Orthogonal version management. In Tichy [Tic89], pages 137–140.

[Rei95] Christoph Reichenberger. VOODOO: A tool for orthogonal version management. In Estublier [Est95], pages 61–79.

[Roc75] Marc J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, SE-1(4):364–370, December 1975.

[RS91] Anthony Rich and Marvin Solomon. A logic-based approach to system modelling. In Feiler [Fei91b], pages 84–93.

[SAK90] Gerd Smolka and Hassan Aït-Kaci. Inheritance hierarchies: Semantics and unification. In Kirchner [Kir90], pages 489–516.

[SB95] Wilhelm Schäfer and Pere Botella, editors. *Proc. 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, Sitges, Spain, September 1995. Springer-Verlag.

[SBK88] N. Sarnak, R. Bernstein, and V. Kruskal. Creation and maintenance of multiple versions. In Winkler [Win88], pages 264–275.

[Sch95] Ulrik Schroeder. *Inkrementelle, syntaxbasierte Revisions- und Variantenkontrolle mit interaktiver Konfigurationsunterstützung*. PhD thesis, Technical University of Darmstadt, Germany, 1995. In German.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network filesystem. In *Proc. of the Summer 1985 USENIX conference*, pages 119–130, Portland, Oregon, June 1985.

[SGS91] Gregor Snelting, Franz-Josef Grosch, and Ulrik Schroeder. Inference-based support for programming in the large. In A. van Lamsweerde and A. Fugetta, editors, *Proc. 3rd European Software Engineering Conference*, volume 550 of *Lecture Notes in Computer Science*, pages 396–408, Milano, Italy, October 1991. Springer-Verlag.

[SM95a] Bradley D. Schmerl and Chris D. Marlin. Consistency issues in partially bound dynamically composed systems. Technical report, Department of Computer Science, Flinders University of South Australia, 1995.

[SM95b] Bradley D. Schmerl and Chris D. Marlin. Designing configuration management facilities for dynamically bound systems. In Estublier [Est95], pages 88–100.

[Smo92] Gert Smolka. Feature-constrained logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[Sne91] Gregor Snelting. The calculus of context relations. *Acta Informatica*, 28:411–445, May 1991.

[Sne96] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[Som96] Ian Sommerville, editor. *Proc. 6th International Workshop on Software Configuration Management*, volume 1167 of *Lecture Notes in Computer Science*, Berlin, Germany, March 1996. Springer-Verlag.

[SS95] Sabine Sachweh and Wilhelm Schäfer. Version management for tightly integrated software engineering environments. In *Proc. of the 7th international Conference on Software Engineering Environments*, Noordwijkerhout, Netherlands, April 1995. IEEE Computer Society Press.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Reading, Massachusetts, 1994.

[SUP+83] S. Shieber, H. Uszkorzeit, F. Pereira, J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In J. Bresnan, editor, *Research on Interactive Acquisition and Use of Knowledge*. SRI International, 1983.

[TGC95] Eirik Tryggeseth, Bjørn Gulla, and Reidar Conradi. Modelling systems with variability using the PROTEUS configuration language. In Estublier [Est95], pages 216–240.

[Tic81] Walter F. Tichy. A data model for programming support environments. In *Proceedings of the IFIP WG 8.1 Working Conference on Automated Tools for Information System Design and Development*, October 1981.

[Tic84] Walter F. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, November 1984.

[Tic85] Walter F. Tichy. RCS—A system for version control. *Software—Practice and Experience*, 15(7):637–654, July 1985.

[Tic88] Walter F. Tichy. Tools for software configuration management. In Winkler [Win88], pages 1–20.

[Tic89] Walter F. Tichy, editor. *Proc. 2nd International Workshop on Software Configuration Management*, Princeton, New Jersey, October 1989. ACM Press.

[Tic94] Walter F. Tichy, editor. *Configuration Management*, volume 2 of *Trends in Software*. John Wiley & Sons, Chichester, UK, 1994.

[Tic95] Walter F. Tichy. Software-Konfigurationsmanagement: Wie, wann, was, warum? In *Proc. Softwaretechnik 95*, volume 15(3) of *Softwaretechnik-Trends*, pages 17–23, Braunschweig, Germany, October 1995. GI. In German.

[Tre96] Christina Trenkner. PUCK – Einbettung von arithmetischen Constraints in die Feature-Unifikation. Master's thesis, Technical University of Braunschweig, Germany, 1996.

[vdHHW95] André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. Does configuration management research have a future? In Estublier [Est95], pages 305–310.

[vdHHW96] André van der Hoek, Dennis Heimbigner, and Alexander L. Wolf. A generic, peer-to-peer repository for distributed configuration management. In *Proc. 18th International Conference on Software Engineering*, pages 308–317, Berlin, Germany, March 1996. IEEE Computer Society Press.

[Wes91] Bernhard Westfechtel. Structure-oriented merging of revisions of software documents. In Feiler [Fei91b], pages 86–79.

[WG95] Tim A. Wagner and Susan L. Graham. Dynamic configuration abstraction. In Schäfer and Botella [SB95], pages 205–218.

[Whi91] David Whitgift. *Methods and Tools for Software Configuration Management*. John Wiley & Sons, Chichester, UK, 1991.

[Wie93] Douglas Wiebe. Object-oriented software configuration management. In Feldman [Fel93], pages 241–252.

[Wil95] Andrew Wiles. Modular elliptic curves and Fermat's last theorem. *Annals of Mathematics*, 141(3):443–551, 1995.

[Win87] Jürgen F. H. Winkler. Version control in families of large programs. In E. Riddle, editor, *Proc. 9th International Conference on Software Engineering*, pages 91–105, Monterey, California, March 1987. IEEE Computer Society Press.

[Win88] Jürgen F. H. Winkler, editor. *Proc. of the International Workshop on Software Version and Configuration Control*, Grassau, January 1988. Teubner Verlag, Stuttgart.

[WS95] Ian Warren and Ian Sommerville. Dynamic configuration abstraction. In Schäfer and Botella [SB95], pages 173–190.

[Xcc95] Xcc Software Technology Transfer GmbH, Karlsruhe, Germany. *RCE—Revision Control Engine: Introduction and Reference Manual*, 1995.

[Zie93] Marc Ziehmann. Unification of feature terms. Project report CS680, University of Albany, New York, December 1993.

Abbreviations

API Application programming interface.

AtFS Attributed file system.

ATP Automated theorem proving.

BCT Bound configuration thread.

CAD Computer-aided design.

CASE Computer-aided software engineering.

CCB Configuration control board.

CD-ROM Compact disk read only memory.

CM Configuration management.

CMA Configuration management assistant.

CPP C preprocessor.

CR Change request.

CVS Concurrent versions system.

CoV Change-oriented versioning.

DBMS Database management system.

DCVS Distributed concurrent versions system.

DCS Dynamically composed system.

DFG Deutsche Forschungsgemeinschaft.

DNF Distributive normal form.

DOS Disk operating system.

DRCS Distributed revision control system.

DVI Device-independent file.

EPOS Expert system for program and ("og") system development.

EFS Extensible file system.

EGA Enhanced graphics adapter.

FFS Featured file system.

FTP File transfer protocol.

GCC GNU C compiler.

GNU GNU's not unix.

GUI Graphical user interface.

HiCoV High-level extensions to change-oriented versioning.

IBM International business machines.

ICE Incremental configuration environment.

ICICLE ICE integrated command line engine.

IPSEN Integrated project support environment.

LEX Lexical scanner.

MacOS Macintosh operating system.

***n*-DFS** *n*-dimensional file system.

NUCM Network for unified configuration management.

NFS Network file system.

NORA No real acronym.

NSE Network software environment.

PCL PROTEUS configuration language.

PDG Program dependency graph.

PROLOG Programming in logic.

PSG Programming system generator.

RCE Revision control engine.

RCS Revision control system.

REGEX Regular expression.

SCCS Source code control system.

SCM Software configuration management.

Tcl/Tk Tool command language toolkit.

TTY Teletype terminal.

TWICE Tasks within ICE.

VOODOO Versions of outdated documents organized orthogonally.

VoV Version-oriented versioning.

WWW World wide web.

YACC Yet another compiler compiler.

Index

Symbols

A

B

C

D

E

F

G

H

M

N

O

P

S

T

U

V

W

X

Y

Z