A Reusable, Distributed Repository for Configuration Management Policy Programming

by

Adriaan W. van der Hoek

M.S. & B.S., Erasmus University Rotterdam, 1994

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2000

This thesis entitled:
A Reusable, Distributed Repository for Configuration Management Policy Programming
written by Adriaan W. van der Hoek
has been approved for the Department of Computer Science

_____

Alexander L. Wolf

_____

Dennis M. Heimbigner

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

van der Hoek, Adriaan W. (Ph.D., Computer Science)

A Reusable, Distributed Repository for Configuration Management Policy Programming

Thesis directed by Prof. Alexander L. Wolf

Although the number and variety of available configuration management (CM) systems has grown rapidly in the past few years, the need to construct new CM systems remains. The desire to manage different kinds of artifacts other than source code, situations demanding highly specialized solutions, and the exploration of new research questions all may require the construction of a novel CM system. Unfortunately, in the face of today's move towards distributed projects, this is becoming an increasingly daunting task for which existing CM technology provides little to no support.

This dissertation contributes a novel reusable testbed that supports the rapid development of—potentially distributed—prototype CM systems. The testbed separates CM repositories from CM policies by providing a generic model of a distributed repository and an associated programmatic interface. Together, the repository model and programmatic interface stipulate a precisely defined abstraction layer upon which specific CM policies are built. In particular, CM policies are programmed as unique extensions to the interface, while the underlying distributed repository is reused across different policies. Within the abstraction layer, distribution is isolated. Low-level details of distributed programming are placed within the implementation of the repository model whereas distribution aspects that are controlled at the policy programming level are placed in a separate, orthogonal functional category within the programmatic interface.

Two tangible benefits result from the use of the reusable testbed. First, the effort required in constructing prototype CM systems is reduced significantly because the generic repository is reused and the CM policy is easily implemented. Second, the

rapid exploration of new CM policies is enabled, leading to the creation of unique CM policies that are tailored to specific situations.

The testbed is evaluated abstractly, by mapping ten CM policies onto the repository model and programmatic interface. Additionally, it is evaluated concretely through the use of a prototype, called NUCM, upon which three novel CM policies are implemented. Demonstrating the expressiveness, feasibility, utility, and validity of the testbed, these policies are characterized by their rapid development, ease of change, incremental evolution, and seamless distributed operation.

To my family.

Acknowledgements

Throughout this endeavor, the guidance of my advisor Alexander Wolf has been of tremendous importance. Without him, I would not be where I am now. Thank you for putting up with somebody who likes to rival you in healthy stubbornness, it has been a great experience.

The members of SERL have made my stay at the University of Colorado more than worthwhile. Special thanks goes to Dennis Heimbigner, for quietly sharpening my research skills, and Antonio Carzaniga, for the long discussions that shaped the eventual incarnation of NUCM.

Without friends, it would be impossible to undertake a long and complicated journey such as this one. Thank you Jon Cook, for the countless number of epic DOOM battles just when I needed them; Martin and Esther van Wijngaarden, for always popping their heads up at the right time and asking me how I am doing; the Sicottes, for being my family away from home; Corry Schippers, for all those letters and postcards; Remco van der Kuijp, for the e-mails and visits; and Laura Vidal, for being who you are: a best friend I can always count on.

Finally, special thanks and utmost respect goes to my parents and sister. Their "Grote Smurf" would not have made it this far without their undeniable love and support.

Contents

Chapter

1 Introduction 1

1.1 Problem 1

1.2 Approach 4

1.3 Evaluation 8

1.4 Benefits 10

1.5 Roadmap 10

2 Background 13

2.1 Basic Terminology 14

2.2 Spectrum of Functionality 15

2.3 Representative Work in Configuration Management 18

2.4 Scope 20

3 Generic Repository Model 25

3.1 Storage Model 26

3.2 Distribution Model 31

3.3 Naming Model 35

3.4 Access Model 39

3.5 Attribute Model 45

4 Programmatic Interface 48

4.1 Access Functions 50

4.2 Versioning Functions 52

4.3 Collection Functions 54

4.4 Distribution Functions 58

4.5 Deletion Function 62

4.6 Query Functions 64

4.7 Attribute Functions 68

5 Modeling Example Configuration Management Policies 72

5.1 Modeling Versioning Aspects of Traditional CM Policies 73

5.1.1 Checkout/Checkin 73

5.1.2 Composition 87

5.1.3 Long Transaction 95

5.1.4 Change Set 110

5.2 Modeling Distributed Aspects of Traditional CM Policies 118

5.2.1 Client-Server Workspaces 119

5.2.2 Peer-to-Peer Repositories 121

5.2.3 Distributed Long Transaction 126

5.2.4 Repository Replication 129

5.3 Modeling Non-Traditional CM Policies 137

5.3.1 Movement Upon Checkout 138

5.3.2 Product Family Architectures 145

5.4 Lessons Learned 150

6 Prototype Implementation 153

6.1 High-Level Architecture 154

6.2 Implementation Architecture 156

6.2.1 Persistent Storage 156
6.2.2 Concurrency Control 160
6.2.3 Reference Counting & Storage Reclamation 162
6.2.4 Server 163
6.2.5 Server Communication & Access Control 164
6.2.6 Client Communication 165
6.2.7 Moved Artifact Handling 165
6.2.8 Name Interpretation & Workspace Management 166
6.2.9 Client Interface 167
6.3 Implementing Optimizations 168
6.3.1 Caching 168
6.3.2 Delta Storage 170
6.3.3 Compression 171

7 Experience 174
7.1 DVS 175
7.1.1 Design 176
7.1.2 Implementation 177
7.1.3 Observations 181
7.2 SRM 183
7.2.1 Design 184
7.2.2 Implementation 188
7.2.3 Observations 189
7.3 WebDAV 190
7.3.1 Design 191
7.3.2 Implementation 192
7.3.3 Observations 194

7.4 Summary 195

8 Related Work 196

8.1 Architectural Evolution 197

8.2 Alternative Abstractions 201

8.3 Other Domains 204

9 Conclusions 206

9.1 Strengths 207

9.2 Weaknesses 208

9.3 Future Work 209

Bibliography 213

Appendix

A Details of the Programmatic Interface 222

A.1 Access Functions 222

A.1.1 nc_open 222

A.1.2 nc_close 223

A.2 Versioning Functions 224

A.2.1 nc_initiatechange 224

A.2.2 nc_abortchange 225

A.2.3 nc_commitchange 226

A.2.4 nc_commitchangeandreplace 227

A.3 Collection Functions 228

A.3.1 nc_add 228

A.3.2 nc_remove 229

A.3.3 nc_rename 230
A.3.4 nc_replaceversion 231
A.3.5 nc_copy 232
A.3.6 nc_list 233
A.4 Distribution Functions 234
A.4.1 nc_setmyserver 234
A.4.2 nc_getlocation 235
A.4.3 nc_move 236
A.5 Deletion Function 237
A.5.1 nc_destroyversion 237
A.6 Query Functions 238
A.6.1 nc_gettype 238
A.6.2 nc_version 239
A.6.3 nc_lastversion 240
A.6.4 nc_existsversion 241
A.6.5 nc_isinitiated 242
A.6.6 nc_isopen 243
A.7 Attribute Functions 244
A.7.1 nc_testandsetattribute 244
A.7.2 nc_setattribute 245
A.7.3 nc_getattributevalue 246
A.7.4 nc_removeattribute 247

Figures

Figure

1.1 Conceptual Architecture. 6

2.1 CM Spectrum of Functionality. 17

2.2 Scope of the Abstraction Layer. 24

3.1 Example Repository Contents without Versions. 28

3.2 Example Repository Contents with Versions. 30

3.3 Example Repository Contents of Figure 3.2 as Distributed over Three Different Sites. 34

3.4 Hierarchical Naming. 37

3.5 Hierarchical Naming with Version Qualifiers. 40

3.6 An Instance of the Access Model in which the Workspace Is Transferred to a Different Location in the File System (a), and an Instance of the Access Model in which the Workspace Is Hidden from the User (b). . . . 44

3.7 Example Attributes Associated with the Atom `Spell.c`. 46

4.1 Example Repository Contents after a New Version of the Atom `PushUp.c` Is Stored. 55

4.2 Example Repository Contents after a New Version of the Collection `Menu` Is Stored. 59

4.3 Example Repository Contents after the Physical Repository of Milano Assumes Ownership of the Atoms `PushUp.c` and `PopUp.c`. 61
4.4 Example Repository Contents after Version 1 of Atom `Frame.c` Is Deleted, Version 3 Is Removed from the Collection `GUI-lib`, and Version 2 Is Removed from the Collection `Menu`. 65
4.5 Example Repository Contents after the Storage Space for Version 2 and Version 3 of the Atom `Frame.c` Is Reclaimed. 66

5.1 Example Repository Structure for the Checkout/Checkin Policy (a), and Typical Attributes Associated with Individual Versions of a Content Artifact and its Version Tree (b). 77
5.2 Checking Out a Version of a Content Artifact in the Checkout/Checkin Policy. 80
5.3 Checking In a New Version of a Content Artifact in the Checkout/Checkin Policy. 82
5.4 Example Repository Structure for the Composition Policy (a), and Typical Attributes Associated with Individual Versions of a Component and its Version Tree (b). 90
5.5 Populating a Workspace in the Composition Policy. 93
5.6 Example Repository Structure for the Long Transaction Policy. 99
5.7 Creating a Child Long Transaction in the Long Transaction Policy. . . . 102
5.8 Committing Changes from a Child Long Transaction to a Parent Long Transaction. 104
5.9 Recursively Committing a Collection. 106
5.10 Example Repository Structure for the Change Set Policy. 113
5.11 Populating a Workspace in the Change Set Policy. 116
5.12 Forming a Logical Repository in the Peer-to-Peer Repositories Policy. . 125

5.13 Example Repository Structure for Each of the Replicas in the Repository Replication Policy. 132
5.14 Synchronizing a Branch in a Replica. 135
5.15 Checking Out an Artifact Version while Moving it to Closer Proximity. . 140
5.16 Checking Out an Artifact Version while Conditionally Moving it to Closer Proximity. 143
5.17 Recursively Moving an Artifact and Its Members. 144
5.18 Partial Example Repository Structure for the Product Family Architectures Policy. 148

6.1 High-Level Architecture of NUCM. 155
6.2 Internal Design of NUCM. 157
6.3 Structure of a Physical Repository as Implemented in the File System. . 159

7.1 DVS Functionality. 179
7.2 Federated SRM Repository Before Milano Joins the Federation. 185
7.3 Federated SRM Repository After Milano Joins the Federation. 187
7.4 Architecture of WebDAV Prototype. 193

8.1 Evolution of the Architecture Embedded in CM Systems. 199

Tables

Table

4.1 Programmatic Interface Functions. 49

4.2 Using Attributes to Lock a Version of an Artifact. 71

Chapter 1

Introduction

Configuration management (CM) is the discipline of managing the evolution of large and complex software systems [Tic88]. It encompasses many different activities, including coordinating multiple developers making simultaneous updates to one or more artifacts, archiving historical versions of artifacts, selecting a configuration of artifacts, identifying the artifacts that make up a configuration, and building a derived artifact out of its source artifacts.

Since its beginnings in the early 1970s, the discipline of CM has slowly but surely evolved. The marketplace for CM products is now worth well over one billion dollars per year [BW98]. More than one hundred commercial CM systems, representing a wide range of functionality, are currently available. Some are simple clones of SCCS [Roc75] and RCS [Tic85], while others have pushed the state of the art quite considerably and offer a full spectrum of functionality [Dar91]. Most other CM systems fall somewhere in between, each providing some distinguishing combination of functionality.

1.1 Problem

Despite the variety of available systems, several compelling reasons exist to continue the development of new CM systems. First, it is recognized that the benefits provided by the use of a configuration management system also apply to domains other than the traditional domain of source code. Web sites [Mor96], software ar-

chitectures [CW98a], and legal databases [Leu94] are just a few of the domains in which the need for an adequate configuration management system has been reported. A second reason is that certain situations require highly specialized CM functionality that is currently not provided by CM systems. For instance, required compliance with a set of company standards [Ray95], the desire for an e-mail based capability to synchronize distributed workspaces [Mil97], and the need to trace fine-grained artifacts [LT98] all led to the development of new CM systems that specifically targeted the problem at hand. A third reason is that the field of configuration management continues to evolve through the research and development of new approaches. Some of these approaches, as exemplified by EPOS [GKY91], ICE [ZS97], and the Software Dock [HHW99], break traditional assumptions and require the construction of a new type of CM system.

Although desirable from a reuse point of view, current CM systems cannot be used in the implementation or even the prototyping of new CM systems. Simple modifications to existing functionality, such as changing roles, access rights, or transition conditions of states, already require source code changes or additional scripts that use triggers and event mechanisms [Dar96]. Because the construction of a new (prototype) CM system would require far more complex changes, current CM systems are simply not a suitable platform for such construction. Several reasons can be identified for this inadequacy.

- **CM systems focus on the management of source code.**

  If other types of artifacts need to be managed and configured, only a limited amount of support is available. Consider the build process. Most CM systems incorporate some variant of Make [Fel79]. As recognized and demonstrated by Odin [Cle95], however, Make is rather limited and a more generic solution can be devised that not only supports the build process but also other types of derivations.

- **CM systems are inflexible in terms of the functionality they provide.**

If new approaches are developed that are in conflict with some of the assumptions that underlie this inflexibility, little to no help is available to implement those approaches. For example, underlying the virtual file system of ClearCase [Atr92] is the assumption that workspaces are continuously connected to a central server. Of course, this assumption is in violation of a CM system that desires to provide disconnected operation.

- **CM systems are constructed based on a rigid architecture.**

  If new functionality requires changes to this architecture, the changes tend to be far reaching and difficult to implement. A particularly illuminating example of this problem is provided by the recent advent of distribution. Most CM systems were initially designed with a centralized architecture. Because a complete redesign and reimplementation is far too expensive, support for distributed operation in these systems remains rather crude. Typically, either a simple client-server interface is provided, or users are made primarily responsible for the maintenance of consistency among various sites [vdHHW96].

Not all CM systems possess every single one of these drawbacks, but each CM system suffers from at least one. Compounded, thus, these reasons make it impossible to use an existing CM system as a reusable platform upon which a large variety of new CM systems can be constructed.

At the same time, building a new CM system completely from scratch is also a rather daunting undertaking. It is often unclear what the exact requirements are. Several, usually costly, prototypes have to be constructed first in order to understand the exact desiderata for the eventual implementation of the complete CM system. This process is further compounded by the need to construct and prototype not only an appropriate versioning mechanism with an associated user interface, but also a large amount of infrastructure. This results in a development process that regularly takes

several years to complete.

The process of constructing a (prototype) CM system from scratch is even further complicated by the fact that many of today's projects are carried out in a distributed fashion. In these projects, multiple collaborative participants are physically dispersed over a number of geographical locations, sometimes even belonging to different organizations. Not only does this influence the implementation of a CM system, in that it must operate in the context of a wide-area network, it also influences the basic design of a CM system, in that its built-in processes must be supportive of distributed, and sometimes decentralized, collaboration.

Being unable to easily extend or adapt an existing CM system, combined with being unable to easily build a new CM system from scratch, leads to what can be considered one of the most pertinent problems in the field of configuration management: no suitable platform exists that facilitates the rapid construction of, and experimentation with, new—potentially distributed—CM systems.

1.2 Approach

As a first step towards solving this problem, this dissertation contributes a novel, reusable testbed that can be leveraged to rapidly construct and experiment with prototype CM systems that may potentially be distributed. It is recognized that, to effectively create such a reusable testbed, it is critical to separate CM repositories, which are the stores for versions of software artifacts and information about those artifacts, from CM policies, which are the specific procedures for creating, evolving, and assembling versions of artifacts maintained in the repositories. Key to this architectural separation is the precise definition of a novel **abstraction layer** that consists of a generic model of a distributed CM repository and a programmatic interface for implementing, on top of the repository, specific CM policies. The generic model consists of five parts that cover the major aspects of a configuration management repository, namely storage, distribution,

naming, access, and attributes. Similarly, the programmatic interface consists of seven orthogonal categories of functions. These categories are access, versioning, collection, attribute, removal, distribution, and querying. As illustrated by Figure 1.1, use of the abstraction layer results in an architecture in which CM policies are programmed as unique extensions to the generic interface while the underlying distributed repository is reused across different policies. Structured this way, the testbed supports the rapid construction of, and flexible experimentation with, new CM policies.

In any abstraction layer that separates policy from mechanism, the functionality provided by the mechanism defines which types of policies can be built. In the case of the abstraction layer embedded in the testbed, a particular instance of a CM policy combined with the generic repository still does not yield a complete CM system. Other parts of the CM system, such as process management, a user interface, and build tools, are not supported by the abstraction layer and still need to be implemented through some other means. Although an abstraction layer that provides an all-encompassing solution is certainly desirable, the scope of such a project would simply be too large. Therefore, the goal of the abstraction layer is limited in that it is meant to only support the construction of the core of a CM system, namely its storage, distribution, versioning, and access facilities. Overall, this leads to the following high-level objectives by which the design of the abstraction layer is guided.

- **The abstraction layer should be policy independent.**

  In order for the abstraction layer to support the construction of a wide variety of CM policies, the repository model and programmatic interface should not themselves contain any restrictive policy decisions. For example, if the repository model only provided a facility to store versions of artifacts as a traditional version tree, it would be very difficult to implement the more advanced change set policy [GKY91]. Similarly, if the functions in the programmatic interface au-

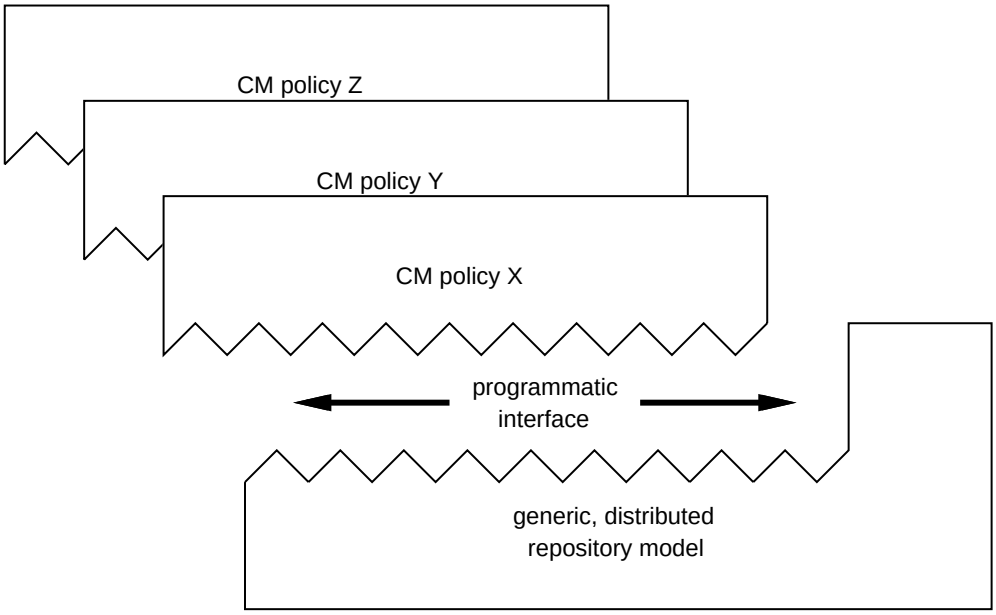


Figure 1.1: Conceptual Architecture.

tomatically created a new version of an artifact whenever one of its constituent parts is modified, CM policies in which the version number of an artifact is explicitly managed by a user would, once again, be difficult—if not impossible—to implement.

- **The abstraction layer should support distributed operation.**

  As proven by the amount of research and development on the issue [AFK+95, CPT97, Con98, HLRT97], providing proper support for the distributed operation of a CM system is a rather complicated task. It is therefore desirable to incorporate support for distribution as an intrinsic property of the repository model and programmatic interface. In particular, the repository model should be able to support a variety of distribution mechanisms (such as peer-to-peer or master-slave), and the programmatic interface should allow for control of the physical placement of artifacts.

  It is important, however, that support for distribution be isolated from other facets of the abstraction. In particular, the low-level details of the distribution aspects of building a CM system (e.g., connection protocols, communication protocols, and time outs) should be isolated from the policy programming aspects by placing those details within the implementation of the repository model. Second, the distribution aspects of relevance to a CM policy (e.g., access to remote repositories and placement of artifacts) should be isolated from the versioning, querying, and other functional categories of CM policy programming. More specifically, the functions in the interface should appear the same regardless of the physical location of the artifacts they manipulate.

- **The abstraction layer should support the management of a wide variety of different kinds of artifacts.**

  As previously mentioned, CM systems are increasingly needed to manage ar-

tifacts other than source code. To allow such specialized CM systems to be prototyped with the abstraction layer, neither the repository model nor the programmatic interface should make assumptions about the kinds of artifacts that are being manipulated. For example, it is well known that certain algorithms for computing the difference between two versions of an artifact work better for textual data, such as documents and program code, than for binary data, such as images or program executables [HVT98]. Incorporating such a biased differencing algorithm into the abstraction layer would violate its ability to properly handle different kinds of artifacts.

- **The abstraction layer should support traditional CM functionality.**

  Even though the abstraction layer is meant to support the construction of new CM policies, it should be obvious that it also must be able to support the construction of existing CM policies. If it could not support the latter, the architectural separation of CM repositories from CM policies results in a loss of functionality, and it would be likely that certain variants of existing CM policies could not be implemented.

1.3 Evaluation

The dissertation evaluates the testbed in two complementary ways. First, the expressiveness of the abstraction layer embedded in the testbed is evaluated by mapping a number of CM policies onto the repository model and programmatic interface. These mappings fall into three categories. The four CM policies that comprise the first category map the versioning aspects of four representative and existing CM policies. The next four CM policies, constituting the second category, show how the distribution aspects of four other CM policies can be mapped. Finally, the two policies in the last category illustrate the simplicity with which novel CM policies can be explored. To-

gether, these ten mappings illustrate that the abstraction layer is capable of supporting the construction of a wide variety of CM policies, whether these CM policies are pre-existing or novel. Moreover, they demonstrate that a variety of distributed CM policies can be constructed and that different kinds of artifacts can be managed.

A testbed like the one developed in this dissertation cannot be evaluated just by mapping CM policies. Additionally, it needs to be evaluated in a concrete setting. Therefore, a prototype implementation of the testbed, called NUCM (Network-Unified Configuration Management), is developed. NUCM consists of two parts: a repository server, which manages the storage of artifacts according to the repository model, and a generic client, which implements the programmatic interface. CM policies are programmed using the interface functions provided by the generic NUCM client. The client, in turn, interacts with sets of geographically distributed repository servers to provide CM policies with access to the artifacts managed.

It is important to realize that, as a prototype, NUCM does not exhibit the robustness or completeness that one would expect from an industrial-strength implementation. Instead, the focus is on the use of NUCM to further evaluate the applicability of the testbed. This evaluation pertains to three critical qualities, namely feasibility, utility, and validity. The first, feasibility, is demonstrated by the implementation of NUCM itself. The simple fact that the prototype exists demonstrates that the abstraction layer, and thus the testbed, can be realized in a concrete setting. The second, utility, is demonstrated by the implementation of three novel CM systems. Two of these, DVS [Car98], a distributed versioning system, and SRM [vdHHHW97], a software release management system, are currently in everyday use. The third is a prototype implementation of an emerging standard, WebDAV [GWF+99]. Following the WebDAV specifications, the prototype extends the HTTP protocol [FGM+98] with authoring and versioning primitives. The last quality evaluated, validity, is demonstrated by the fact that the implemented and mapped CM policies are easily and rapidly constructed, operate in-

herently in a distributed environment, manage different kinds of artifacts, and actually can evolve over time.

1.4 Benefits

In summary, this dissertation contributes a novel reusable testbed that is a first step towards changing the way that CM systems are constructed. Two tangible benefits result from the application of the testbed. First, the testbed reduces the effort required in prototyping new CM systems significantly, because the generic CM repository is reused and the CM policy is easily implemented through the use of the functions in the programmatic interface. Second, the testbed enables the flexible exploration of new CM policies, thereby facilitating the development of CM policies that are tailored to a specific situation. These benefits result from the precise definition of a reusable, generic, and distributed repository that is combined with an associated, equally precisely defined, abstraction layer. Because the underlying repository infrastructure is reused among different CM policies, each policy itself is programmed as a unique extension to the programmatic interface. While still supporting traditional configuration management functionality, new aspects introduced by the testbed are policy independence, intrinsically distributed operation, and management of arbitrary types of artifacts. The power of the testbed, and thus the critical importance of the new aspects, is demonstrated by the implementation of three new CM systems and the mapping of ten other CM policies onto the abstraction layer.

1.5 Roadmap

The remainder of this dissertation elaborates the definition and evaluation of the testbed. It is organized as follows.

- Chapter 2 introduces the topic of configuration management, lays out the spec-

trum of functionality found in configuration management systems, and scopes (within this spectrum) the work presented in this dissertation.

- Chapter 3 describes the generic repository model in detail. Five submodels are discussed: the storage model, which defines the primitive versioning and grouping mechanisms; the distribution model, which defines how artifacts are distributed among separate sites; the naming model, which defines how individual artifacts are identified; the access model, which defines how access to stored artifacts is obtained; and the attribute model, which defines how attributes can be associated with artifacts.

- Chapter 4 discusses the programmatic interface through which artifacts that are stored in the repository are accessed and manipulated. Seven orthogonal categories of functionality are identified: access, versioning, collection, attribute, removal, distribution, and querying. The functions that make up each category are illustrated based on a comprehensive example.

- Chapter 5 demonstrates the expressiveness of the abstraction layer by mapping a number of CM policies onto the repository model and programmatic interface. Four well-known versioning policies, four widely-used distribution policies, and two new versioning and distribution policies are mapped.

- Chapter 6 introduces NUCM, an realization of the abstraction layer into an actual implementation of the testbed. Not only are the particulars of the implementation discussed, but also the optimizations that one would expect to be made in an industrial-strength implementation of the abstraction layer.

- Chapter 7 presents the lessons learned while implementing three novel CM systems with NUCM. Two of those systems are in everyday use, whereas the third is a prototype implementation of an emerging standard in Web versioning.

- Chapter 8 discusses related work. Even though the testbed is unique in terms of the functionality it provides, several other systems are contrasted and compared in order to properly place the contribution of this dissertation within the current body of work in the field of configuration management.

- Chapter 9 summarizes the work presented in this dissertation, reiterates the contributions, and concludes with an outlook at future work that builds upon the results presented.

Chapter 2

Background

Over the years, the field of configuration management has produced a large number of commercial and public-domain CM systems. The functionality provided by these systems varies quite considerably and ranges from simple version control mechanisms to highly advanced process engines. Unfortunately, the full spectrum of functionality of these systems is large and consists of many different dimensions. In addition, some of these dimensions have been addressed by other generic solutions that were specifically designed to address only those dimensions. For these reasons, the work of this dissertation is scoped and exclusively focuses on generically providing storage, distribution, versioning, and access facilities for CM policies.

This chapter defines the scope of the work presented in this dissertation. First, it introduces some basic terminology that is used throughout the remaining chapters. Following that, it discusses a comprehensive spectrum of functionality that accurately characterizes the current state of the art in CM systems. It then briefly presents an overview of some of the more representative work in the field of CM. It concludes by identifying, based on the spectrum and overview, the specific subset of functionality that the testbed is meant to support.

2.1 Basic Terminology

Before discussing the spectrum of functionality and scoping the work of this dissertation, it is necessary to define some of the basic terminology used in the field of configuration management. Based on a coalescence of the definitions of Conradi and Westfechtel [CW98b], Estublier and Casallas [EC95], and Tichy [Tic88], the following terminology is adopted throughout this dissertation.

- **Artifact.** An artifact records the result of a development or maintenance activity. Many different kinds of artifacts, all of differing granularity, are created throughout the software life cycle, including requirement specifications, design documents, source code, test cases, and documentation. Either compound or elementary, artifacts are the basic entities that are stored by a CM system. Because they usually exist in multiple incarnations, artifacts can be versioned.

- **Revision.** A revision is a new incarnation of an artifact that is intended to supersede its predecessor. A series of revisions represents the evolution of an artifact along the time dimension. Bug fixes, enhancements in functionality, and preventive maintenance are some of the typical causes of evolution.

- **Variant.** A variant is a new incarnation of an artifact that is intended to coexist with its predecessor. A group of variants represents, based on some distinguishing property, a set of logical alternatives of an artifact. Variability is often caused by differences in hardware or software platforms, by the existence of multiple performance levels, or by the provision of specialized customizations.

- **Version.** A version is either a revision or a variant of an artifact. Sets of versions are typically organized in a version tree, a directed acyclic graph that captures the evolution and variability of an artifact in a single model.

- **Configuration.** A configuration groups a set of versions of artifacts into a single entity. It provides a shorthand to refer to its contained artifacts via a single designation. Configurations are artifacts themselves and can indeed be versioned as well.

- **Baseline.** A baseline is a specific version of a configuration that is identified as being special. It is typically used to identify the start of a new line of development or to capture the particular configuration that is delivered to a specific customer.

- **CM repository.** A CM repository is that part of a CM system that provides the storage for versions of artifacts and information about those artifacts. Some CM systems use commercial databases (e.g., TrueCHANGE [Sof94b] and Continuus [Con94]), but most are based on proprietary solutions (e.g., the comma-v files of RCS [Tic85] and the hierarchical file system of ShapeTools [ML88]).

- **CM policy.** A CM policy is that part of a CM system that implements the collection of procedures used in creating, evolving, and assembling versions of artifacts that are maintained in an associated CM repository. Two of the better-known CM policies are checkout/checkin, which is based on a pessimistic, transaction oriented style of operation, and change set, which is based on an optimistic, merging oriented style of operation.

2.2 Spectrum of Functionality

The current state of the art in commercially available CM systems is rather stagnant. Research results from the late 1980s and early 1990s have been transferred, and the research conducted in the late 1990s is still too new and immature to be adopted. Moreover, the development that does take place either enhances existing CM systems with simple improvements in usability and available features, or represents the addition

of new entries to an increasingly competitive market. Although certainly important from a commercial point of view, the actual functionality that is provided by these systems is converging towards a stable state.

This stable state is accurately characterized by the spectrum of functionality developed by Dart [Dar90, Dar91]. Illustrated in Figure 2.1, the spectrum identifies eight separate classes of functionality.

(1) **Components.** Functionality in this category is concerned with identifying, classifying, and accessing artifacts. This involves storing versions of artifacts in repositories, distinguishing different kinds of artifacts, grouping artifacts into system components, and recording configurations, baselines, and project contexts.

(2) **Structure.** Functionality in this category is concerned with the consistency of a group of artifacts. This involves modeling the logical and physical structure composed by artifacts, specifying interfaces, identifying and maintaining relationships among artifacts, and selecting a valid and consistent set of versions of artifacts.

(3) **Construction.** Functionality in this category is concerned with the derivation process of a deliverable artifact out of its source artifacts. This involves freezing a set of artifacts, optimizing the derivation process by caching derived artifacts, performing change impact analysis, and regenerating a deliverable artifact out of its frozen source artifacts.

(4) **Auditing.** Functionality in this category is concerned with the creation of an audit trail. This involves keeping a history of all changes to all artifacts, maintaining traceability among all related components and their evolution, and logging details of the work done.

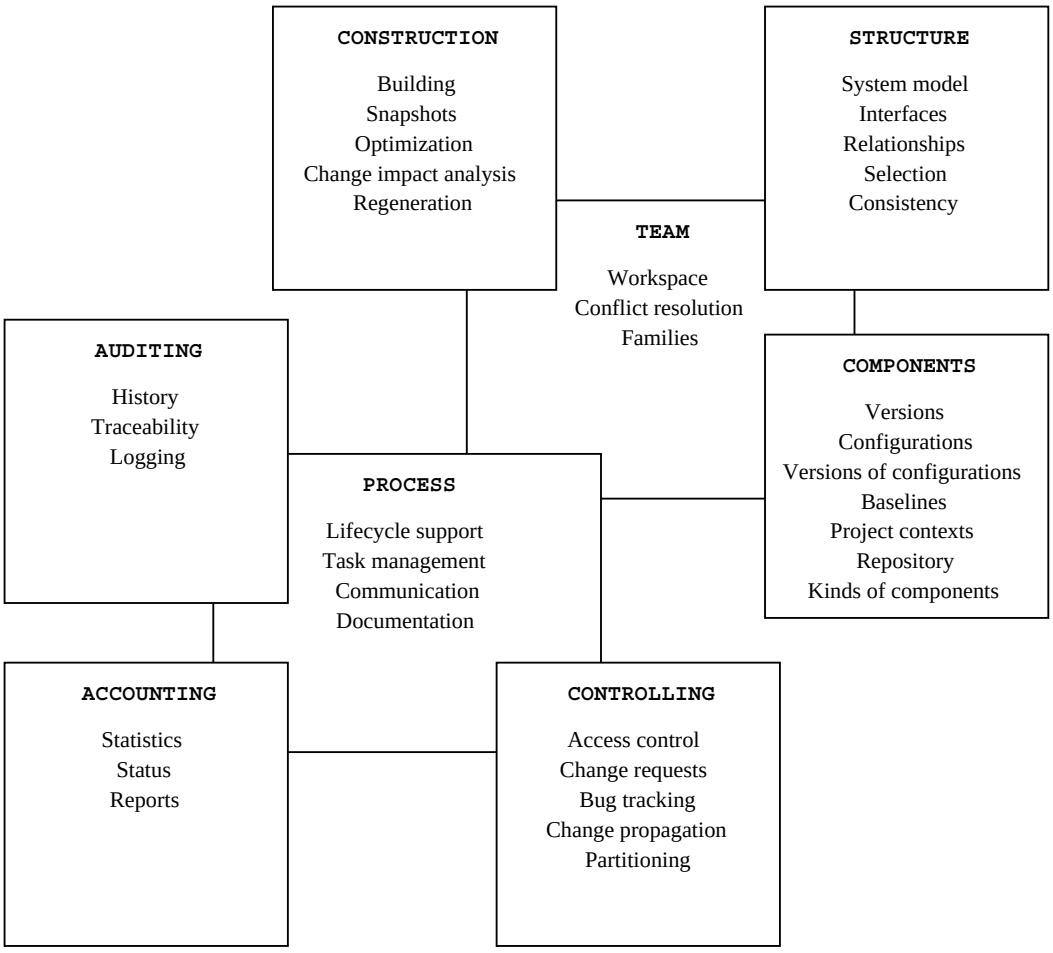


Figure 2.1: CM Spectrum of Functionality.

(5) **Accounting.** Functionality in this category is concerned with the provision of information about the artifacts being maintained. This involves calculating and recording statistics, examining the status of artifacts, and generating reports about all sorts of aspects of the artifacts and process.

(6) **Controlling.** Functionality in this category is concerned with the control over how and when changes are made. This involves controlling access to artifacts, avoiding conflicts, managing change requests, tracking bugs and their impact on artifacts, propagating changes across baselines, and partitioning artifacts to limit the effects of change.

(7) **Process.** Functionality in this category is concerned with the smooth operation of the configuration management process. This involves providing support for managing and enforcing life cycle activities, identifying and assigning tasks, communicating information to relevant parties, and documenting knowledge about the project.

(8) **Team.** Functionality in this category is concerned with team support. This involves managing individual and group workspaces, resolving conflicts among changes to artifacts, and facilitating the maintenance of families of systems.

These eight classes of functionality can roughly be partitioned into three groups. The first primarily addresses management aspects, and consists of `auditing`, `accounting`, and `controlling`. The second primarily addresses development aspects, and consists of `structure`, `components`, and `construction`. The last performs a bridging function between the management and developments aspects, and consists of `process` and `team`.

2.3 Representative Work in Configuration Management

As evidenced by the existence of a workshop series that is exclusively dedicated to the topic of configuration management [Win88, Tic89, Fei91b, Est95, Som96, Con97,

Mag98, Est99], a large body of work has slowly but surely evolved the state of the art towards CM systems that address all of the functionality that is identified in the aforementioned spectrum. In fact, since the invention of SCCS [Roc75] and RCS [Tic85], with their ability to archive and coordinate changes to an artifact, and Make [Fel79] and Build [EP84], with their ability to automate and optimize the software build process, a myriad of CM systems have become available for general use. As each system yields its own strengths and weaknesses, they can be evaluated and compared along many dimensions. Two of the most comprehensive evaluations are provided by Ovum [BW98], which surveys the features that are provided by commercially available CM systems, and Conradi and Westfechtel [CW98b], who analyze, from a modeling point of view, the technology underlying industrial and research CM systems. Regardless of differences in features and technology, though, the underlying goal of all CM systems remains the same: precisely controlling changes to artifacts.

The advances and enhancements that have been made over time can be divided into three groups. Characterizing the first group is the fact that the available functionality is broadened through fundamental changes to the models that underlie CM systems (e.g., change sets to compose new configurations as incremental changes to a baseline [GKY91], object-orientation to ensure interface compatibility in selected configurations [AB89], typed graphs to support a wide variety of derivation processes [Cle95], and feature logic to flexibly integrate a number of versioning styles into a single CM system [ZS97]). The second group has taken existing CM functionality and has focused on enhancing the general applicability of the technology (e.g., Web-based interfaces to provide remote versioning capabilities [HLRT97, Sta96], specialized user interfaces to explicitly manage LaTeX documents [BLNP98], and augmented file systems to provide access to versioned files through "normal" file system paths [LM88]). The last group has made small, but vital, improvements to the models and applicability of CM systems (e.g., platform-independent derivation rules to support shorter and more under-

standable "Makefiles" [Tib96, WS97], inter-file branching to promote variants to easily identified entities [Sei96], better compression techniques to save storage space [HVT98], and syntactic analysis of source code to improve merge results [Buf95]).

Of interest to this dissertation are the advances made in groups one and three. Unlike the advances in group two that change neither the model nor the policy, the advances in groups one and three are directly concerned with the models and policies that are in everyday use. For the advances in group one, the testbed should support the rapid creation of, and experimentation with, new models and policies. For the advances in group three, the testbed should support the adjustment of existing policies to improve their understanding and usability.

2.4 Scope

The testbed separates CM repositories from CM policies. Its focus is on the storage of artifacts and the provision of an interface that facilitates the exact specification of the policy according to which the artifacts are manipulated. This means that certain classes in Dart's spectrum of functionality are out of scope with respect to this dissertation. Of course, the fact that certain classes are in scope and certain classes are out of scope is no coincidence. Rather, the testbed is designed to complement many of the other generic solutions that have already been devised and applied to the field of configuration management. In particular, with respect to the testbed, Dart's classes of `accounting` and `process` are completely out of scope. Although both certainly use the kind of information that is stored in a CM repository, the functionality in these classes is more concerned with supporting the general, overarching software development process rather than configuration management specifically. Moreover, for both classes generic solutions have been devised already that support report generation and process automation, respectively.

Of the remaining classes, the focus of the testbed is on four: `components`, `struc-`

`ture`, `auditing`, and `team`. These classes are clearly within scope: their primary concern is with the storage of artifacts, the maintenance of relationships and structure, and the recording of a history of changes. The functionality in the last two classes, `construction` and `controlling`, is partially supported. Some of it is in scope, such as snapshots and change propagation, and some of it is out of scope, such as change impact analysis and bug tracking. Based on the avoidance of duplication of readily available techniques and the focus of the testbed on storage and policy programming, the following is observed for each (partially) supported class.

- **Components.** The testbed should support the storage of, and access to, multiple versions of artifacts. The testbed should also support the identification and storage of baselines and be able to delineate those artifacts that are part of a specific project. Of course, it should be able to do so in a policy-independent manner.

- **Structure.** The testbed should support the storage and retrieval of relationships among artifacts, including those that model the physical and logical structure of the compound artifacts in the repository. The storage and labeling of consistent sets of artifacts should be supported, but the actual verification of whether a particular set of artifacts is consistent is dependent on the type of artifact being managed and is, thus, outside of the scope.

- **Construction.** The testbed should support the storage of frozen configurations, potentially consisting of both source and derived artifacts. Out of scope, however, is the actual derivation process itself, because independent derivation engines such as Odin [Cle95] and qef [Tib96] can be used as compliments to the functionality provided by the testbed. Change impact analysis is not supported either, since it is heavily dependent on the type of artifact being stored.

- **Auditing.** The testbed should support the creation of CM policies that pre-

cisely track a history of changes, that exactly trace relationships among artifacts (even as both the artifacts and relationships evolve), and that log detailed information about the nature and circumstances of changes.

- **Controlling.** The testbed should support controlled access to artifacts, the propagation of changes across baselines, and the partitioning of artifacts into separate groups. However, advanced infrastructures that specifically support a wide variety of locking policies have been developed [Hei96]. The testbed should not duplicate these efforts and should only provide a basic access control mechanism. Additionally, change request and bug tracking tools are ubiquitous (e.g., Archimedes' BugBase [Arc99] or Tower Concept's Razor [Tow99]) and their functionality is complementary to that of the testbed. Therefore, supporting change requests and bug tracking is also out of scope.

- **Team.** The testbed should support the storage and management of families of compound artifacts as well as the creation of (team) workspaces. Conflict resolution mechanisms are not supported by the testbed, because these mechanisms are dependent on the type of artifact being managed. Moreover, oftentimes these mechanism are based on merge algorithms [Buf95], which are well understood and already developed.

Figure 2.2 presents the resulting scope of the testbed. The amount of coverage of each class with a particular pattern indicates how much of the functionality in that class is in or out of scope. In particular, a striped, slightly darker pattern indicates functionality that is out of scope and already addressed by other solutions. The use of a solid, lighter pattern indicates the functionality that is in scope. The testbed is meant to generically support the construction of those aspects of CM policies. Finally, white indicates the functionality for which no generic solution is available and for which the testbed does not provide any support. These capabilities, which are consistency enforcement and

change impact analysis, are too dependent on the types of artifacts being managed for any solution to address the problem in a generic fashion.

It should be noted that the spectrum of functionality defined by Dart is stated in terms of the management of software. The testbed, of course, should provide the above set of capabilities regardless of what types of artifacts are managed. Additionally, it should be reiterated that the testbed has to function in a distributed setting, and should be policy independent.

Finally, it should follow from the above discussion that the combination of the generic CM repository (as defined by the testbed) with a particular instance of a CM policy (as defined by a CM policy programmer) does not always yield a complete CM system. In some cases, the functionality can be sufficient, but in most, additional pieces of functionality still need to be integrated. Fortunately, in many of these cases the functionality to be integrated is relatively independent and complementary in nature to the functionality that is provided by the testbed. Consider the build process. Given a set of artifacts in a workspace, the actual derivation mechanism can autonomously create the derived artifacts. In other cases in which additional functionality needs to be integrated, that functionality is of a higher-level nature than the testbed and actually builds upon the policies that are implemented. Consider process management. Given a CM policy, a general process engine uses the policy and the information stored in the repository to, for example, assign developers to tasks and communicate the appropriate information to them.

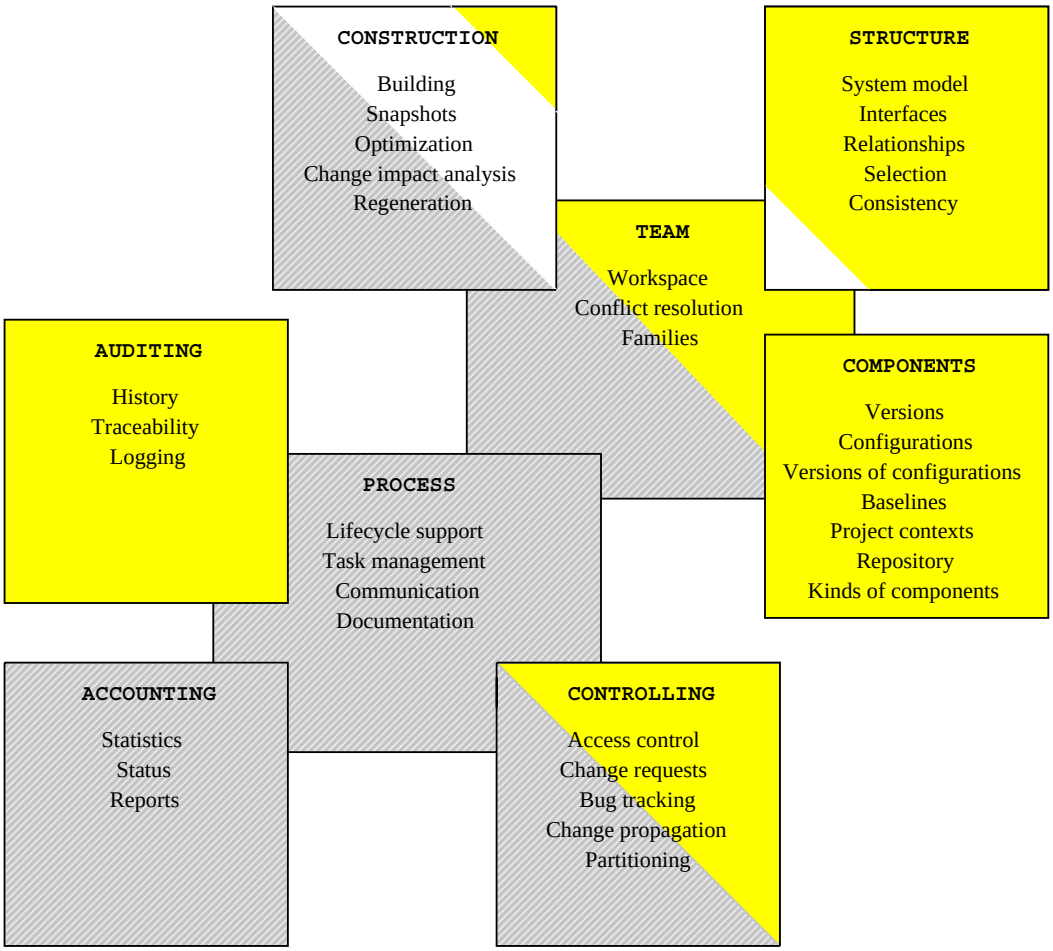


Figure 2.2: Scope of the Abstraction Layer.

Chapter 3

Generic Repository Model

The first component of the abstraction layer defined by the testbed is its generic model of a distributed configuration management repository. The complete model consists of five complementary submodels: the **storage** model, which defines the mechanisms for versioning and grouping artifacts; the **distribution** model, which defines the different ways in which artifacts can be arranged across different sites; the **naming** model, which defines the way individual artifacts can be identified in a distributed repository, the **access** model, which defines the primary method of access to artifacts stored in a distributed repository; and the **attribute** model, which defines how attributes can be used to associate metadata with artifacts.

A key characteristic of the generic repository model is that, even though specifically designed to support the versioning, grouping, distribution, and other aspects of artifacts, it does not enforce any particular policy of doing so. For instance, while the repository model provides the capability of storing multiple versions of an artifact, it does not impose any specific relationships among those versions. Similarly, the repository model facilitates the storage of different artifacts in different repositories, but it does not enforce a particular organization of the artifacts among the different repositories. In both these, and other, cases of separation of CM repository from CM policy, it is up to the CM policy programmer to use the interface functions discussed in the next chapter to manipulate the CM repository into the desired behavior.

In this chapter, each of the submodels of storage, distribution, naming, access, and attribute is discussed in detail.

3.1 Storage Model

The basis for the storage model is a directed graph with two kinds of nodes: **atoms** and **collections.** An atom is a leaf node in a graph and represents a monolithic entity that has no substructure visible to the storage model. Typical atoms include source files or sections of a document. Contrary to atoms, the structure of collections is known to the storage model: collections are the basic mechanism used to group atoms into named sets. For example, a collection might represent a program that consists of a set of source files. Alternatively, a collection could represent a document that is composed out of a number of sections.

Collections can be used recursively and can themselves be part of larger, higher-level collections. For instance, a collection that represents a system release could consist of a collection for the actual source code of the system and a collection for the documentation of the system. Membership of a collection can, of course, be mixed: a single collection can contain both atoms and collections. A collection that represents a document could have as its members short sections that are captured as atoms, as well as longer, further subdivided sections that are captured as collections.

Within the context of configuration management, a directed graph is capable of modeling the required storage needs. In fact, the structure afforded by the directed graph is exactly sufficient and a more restrictive structure would lead to deficiences in modeling capabilities. In particular, imposing a strictly hierarchical graph would result in a structure that prohibits the modeling of shared artifacts (e.g., specialized error handling routines used in all coding projects, or a standard disclaimer included in every document that is made publicly available). Similarly, disallowing the creation of cycles within the graph would result in a structure that prohibits the modeling of mutually

recursive data (e.g., two Web pages that contain links to each other, or collections of clip art that reference each other as containing a related set of illustrations).

Figure 3.1 illustrates the basic concepts of atoms, collections, and their membership relations. The figure shows a portion of a repository for the C source code of two hypothetical software systems, `WordProcessor` and `DrawingEditor`. Collections are shown as ovals, atoms as rectangles, and membership relations as arrows. Both software systems not only contain a separate subsystem, as demonstrated by the collections `SpellChecker` and `Menu`, respectively, but they also share a collection called `GUI-lib`. In turn, these lower-level collections simply contain atoms, which in this example represent the actual source files that implement both systems. It should be noted that one of the atoms is a member of two collections, but is named differently in each one, namely `Frame.c` and `PullDown.c`, respectively.

To simplify capturing multiple versions of a single artifact, the basic directed graph of the storage model is extended to form a versioned directed graph. In this type of graph, versioning is introduced as an orthogonal dimension to both atoms and collections. That is, atoms and collections each can have multiple versions that all exist in parallel. An important consequence of this fact is that membership of collections is on a per-version basis: collections contain specific versions of other artifacts. This scheme allows the membership of collections to evolve in a manner similar to the way the contents of atoms evolve.

In contrast to other approaches, such as CME [HLRT97] or ScmEngine [CPT97], the storage model does not impose any semantic relationship among the versions of an artifact. In particular, the tree-structured revision and variant relationships that are found in many—but by no means all—CM systems is not present in the directed versioned graph. Instead, the graph simply provides a unique number with which to identify each version. This allows a CM system to employ its own type of semantic relationships among versions, and hence increases the generality of the repository.

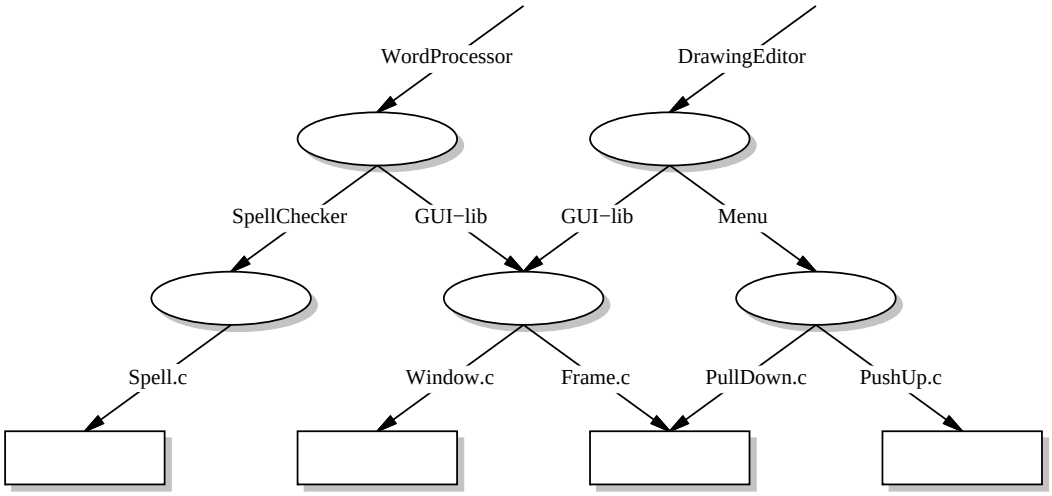


Figure 3.1: Example Repository Contents without Versions.

The decision not to enforce semantic relationships among the versions of an artifact is based on the more general observation that many such relationships exist. Some examples include `derived-from`, `is-composed-of`, `is-part-of`, `depends-on`, and `includes`. Different CM systems support different subsets of these, and other, relationships. Therefore, rather than directly maintaining only an arbitrary subset of relationships, the storage model is generic in that it facilitates the creation and maintenance of arbitrary, policy-programmed relationships. It does so through the use of collections to group artifacts and the use of attributes to label versions of artifacts (see Section 3.5 for a detailed discussion of attributes in the repository model). While that may at first seem inconvenient, since a policy programmer is now expected to implement relationships, the ability to reuse these implementations mitigates the inconvenience. For example, the policy code that defines the version tree relationship of the WebDAV example in Chapter 7 reuses much of the policy code of an earlier CM system. This CM system, an implementation of the checkout/checkin policy discussed in Section 5.1.1, also uses the version tree relationship and was built using the testbed [vdHHW96].

Figure 3.2 shows how the directed graph of artifacts presented in Figure 3.1 is enhanced with versions to form a versioned directed graph. Stacks of ovals and rectangles represent sets of versions of collections and atoms, respectively. Numbers indicate the relative age of versions: the higher the number, the younger the version.[1] Dashed arrows represent the member relationships of older versions of collections. Observe that membership of collections is on a per-version basis. For example, both version 1 and version 2 of the collection `Menu` contain version 1 of the atom `PushUp.c`, but version 2 contains an additional atom, namely version 2 of the atom `PullDown.c`.

[1] As further discussed in Section 4.2, the age of a version merely indicates its time of creation. The actual contents of the version may change over time.

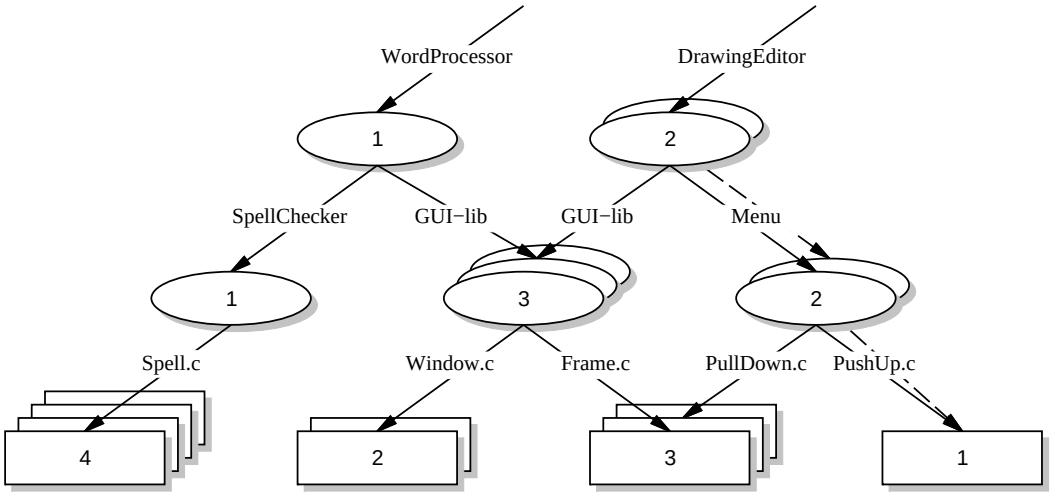


Figure 3.2: Example Repository Contents with Versions.

3.2 Distribution Model

The distribution model of the abstraction layer complements the functionality of the storage model. Whereas the storage model specifies how artifacts can be grouped, versioned, and related through the versioned directed graph, the distribution model precisely defines how the versioned directed graph can be stored in a distributed fashion. In particular, the distribution model defines two types of repositories: a **physical** repository and a **logical** repository. A physical repository is the actual store for some part of a versioned directed graph at a particular site. It contains, for a number of artifacts, the contents of the versions.

A logical repository is a group of one or more physical repositories that together store a complete versioned directed graph of artifacts. Because the distribution model is policy independent, a requirement for a logical repository is that it has to be able to support the modeling of a variety of distributed CM policies. To do so, physical repositories that are part of a logical repository collaborate in a **peer-to-peer** fashion. No centralized "master" repository controls the distribution of artifacts. Instead, distribution is controlled at the individual artifact level: collections not only maintain the names of their member artifacts, they also track the physical repository in which each member artifact is stored. Thus, membership relations span the geographical boundaries that exist among physical repositories.

In contrast to the artifacts in a physical repository, the actual location of artifacts in a logical repository is irrelevant. Artifacts can be obtained from any physical repository that is part of the logical repository, whether the physical repository resides on a local disk, on the local network, or on the other side of the world. Based on the fact that collections keep track of the physical repositories in which their member artifacts reside, requests for member artifacts that are stored at a different physical repository than that of the collection are forwarded. Thus, physical repositories act as both clients

and servers, requesting services from each other and fulfilling service requests for each other.

The level of granularity at which a versioned directed graph is distributed over physical repositories is the artifact. All versions of an artifact are collocated within a single physical repository. The distribution of individual artifact versions over multiple physical repositories is not supported, because such a fine-grained level of distribution would incur a significant communication cost. In particular, when all versions of an artifact are stored within a single physical repository, a storage reclamation mechanism that is based on reference counting does not require any communication among repositories to determine whether the storage space for an artifact can be reclaimed. Similarly, the collocation of all versions of an artifact facilitates a form of navigation in the version space that only requires communication with one physical repository. These and other important optimizations are not possible if individual versions of an artifact can be arbitrarily distributed throughout a logical repository.

A single physical repository can be part of multiple logical repositories. This is an essential aspect of the distribution model, since it allows the sharing of artifacts over otherwise disconnected efforts (e.g., a separately maintained graphics library that is used by a number of coding efforts throughout the organization, or a standard set of company-wide disclaimers and guidelines that are part of every set of documents that is made publicly available). Instead of having to replicate these shared artifacts in each logical repository, the sharing of a physical repository by multiple logical repositories allows a single point of evolution of the shared artifacts.

Figure 3.3 presents these concepts with an example distributed repository. Shown is the repository of Figure 3.2 as distributed over three different sites, namely Boulder, Milano, and Rotterdam. Each of these sites maintains a physical repository with artifacts. The physical repository located in Boulder maintains the collection `WordProcessor`, the physical repository in Milano maintains the collection `Drawing-`

`Editor`, and the physical repository in Rotterdam maintains the collection `GUI-lib`. Because the projects in Boulder and Milano rely on the use of the collection `GUI-lib`, their physical repositories are connected with the physical repository in Rotterdam. Two logical repositories are formed: the physical repositories in Boulder and Rotterdam combine into one logical repository that presents a complete view of the collection `WordProcessor` and its constituent artifacts, and the physical repositories in Milano and Rotterdam combine into one logical repository that manages the complete system `DrawingEditor`.

It is important to note that it is the simple presence of membership relations among artifacts in different physical repositories that creates logical repositories. Without the membership of the collection `GUI-lib` version 2 within the collection `WordProcessor` version 1, for example, the logical repository that is the combination of the physical repositories in Boulder and Rotterdam would not exist. Instead, the physical repository of Boulder would be a logical repository all by itself.

The distribution model is rather versatile. Artifacts can be distributed among physical repositories as desired, a single physical repository can be part of multiple logical repositories, and logical repositories can themselves be part of other logical repositories. This extreme flexibility, combined with a peer-to-peer architecture, allows many different distribution schemes to be mapped onto the distribution model. As demonstrated in Section 5.2, these schemes include the following:

- a single physical repository that is accessed by many CM clients, creating a client-server system like DRCS [OG90];
- several physical repositories that represent a hierarchy of distributed workspaces in which changes in lower level workspaces are gradually promoted up the hierarchy, duplicating the essence of the functionality of such systems as NSE [FD90] and PCMS [SQL98]; and

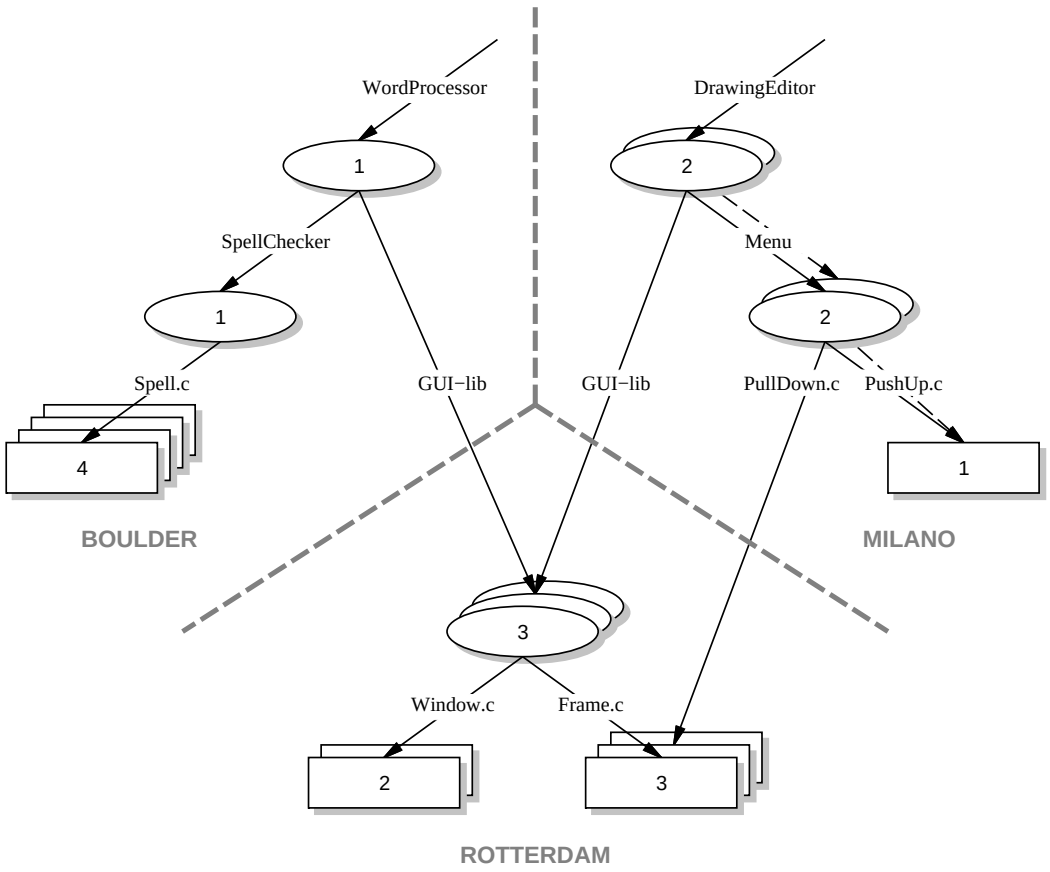


Figure 3.3: Example Repository Contents of Figure 3.2 as Distributed over Three Different Sites.

- a set of physical repositories that act as replicas, in which the contents of the replicas are periodically synchronized by a merging algorithm, a configuration similar to ClearCase Multisite [AFK+95].

These and other approaches to distributed CM can be built using the peer-to-peer architecture. While it is true that a solution based on our generic distribution model might not perform as optimally as a specialized solution for a particular CM policy, the flexibility that the repository model affords allows experimentation with new distribution policies. Once proven to be of use, the implementation of an experimental policy can be optimized for performance.

3.3 Naming Model

An important issue in distributed systems development is naming. Rather than employing a global naming scheme in which each artifact is assigned a single, unique identifier, the naming model is based on a hierarchical naming scheme. The use of hierarchical naming provides three important advantages. Firstly, it naturally fits the hierarchy that is formed by the directed graph of artifacts as defined by the storage model, since each part of a name incrementally indicates which member of a collection is chosen when traversing the directed graph. Secondly, hierarchical naming provides an advantage of scale by avoiding the need for complicated algorithms that create globally unique identifiers. Lastly, it follows the generally accepted practice of decoupling the name of an artifact from its physical location. In particular, since membership relations can span multiple geographical locations, a hierarchical name simply follows these relations without knowing the actual location of the artifact it designates.

By itself, hierarchical naming is not sufficient. Still open is the choice as to whether each part of a hierarchical name is maintained by an artifact or by its containing collection. To allow a single artifact to exist under different names in different collections

(an important facility in current CM systems), the naming model prescribes the latter: names of artifacts are maintained individually by the collections in which the artifact is a member.

Figure 3.4 illustrates the basic hierarchical naming scheme that is used. Highlighted by the bold arrow is the name for version 2 of the artifact `PullDown.c`.

```
DrawingEditor/Menu/PullDown.c
```

Note that even though the name does not contain information about specific locations and originates in the physical repository of Milano, it designates an artifact that actually resides in the physical repository of Rotterdam.

Also illustrated by Figure 3.4 are two deficiences of the basic hierarchical naming scheme when applied to the versioned directed graph as defined by the storage and distribution models. The first deficiency is that the basic hierarchical naming scheme does not operate in the versioning dimension. In particular, because membership in a collection is on a per-version basis, it is possible that certain versions of artifacts (such as version 1 of the atom `PullDown.c`) do not belong to any collection. These versions of artifacts cannot be addressed. The second deficiency is that a hierarchical name does not contain a starting point defining where in a logical repository the interpretation of the name should begin. This means that it is likely a particular name has to be interpreted at multiple physical repositories before the desired artifact is located.

The first deficiency is solved through the use of **version qualifiers**. Version qualifiers are optional extensions that, in each part of a full hierarchical name, can alter the actual version from which interpretation should continue. For example, the name

```
DrawingEditor/GUI-lib:3/Frame.c
```

addresses version 3 of the atom `Frame.c`. This version (in fact the whole artifact `Frame.c`) is not reachable with a regular hierarchical name that includes the collection `GUI-lib`, since the artifact `Frame.c` is only contained by version 3 of the collection

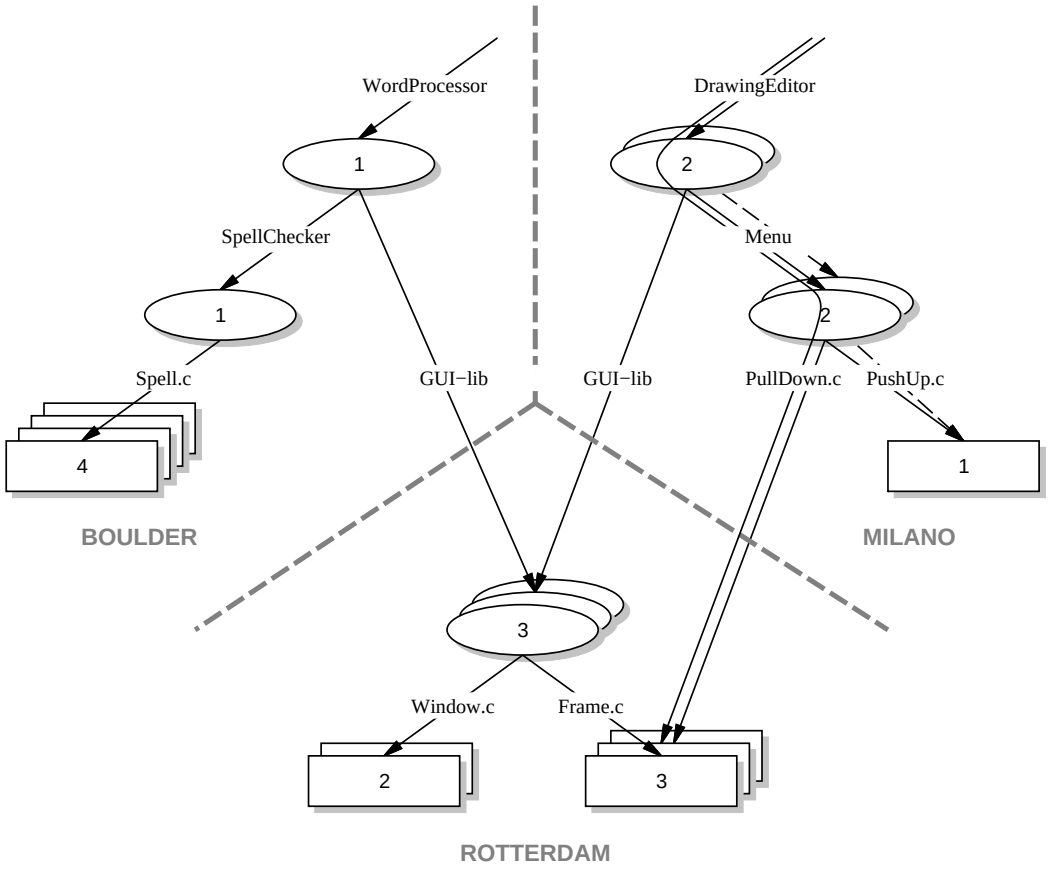


Figure 3.4: Hierarchical Naming.

`GUI-lib` and not by version 2 (and all collections that have as their member a version of the artifact `GUI-lib` only contain version 2). Thus, the version qualifier is needed to navigate from version 2 to version 3 of the collection `GUI-lib`, where the interpretation of the full hierarchical name continues. Note that, in the above name, the collection `DrawingEditor` is not augmented with a version qualifier. By default, the member version of an artifact is chosen if no version qualifier is specified. Of course, version qualifiers can always be used, regardless of whether or not they are needed. Consequently, version 3 of the atom `Frame.c` could also have been identified using version qualifiers at each stage of the hierarchical name.

```
DrawingEditor:2/GUI-lib:3/Frame.c:3
```

The second deficiency, a lack of starting point, is solved by prefixing each hierarchical name with a designation of the physical repository that should interpret the name. This is equivalent to the scheme used for host names on the Internet, where the location of a domain name server must be provided as a starting point for performing name resolution. It is also similar to Universal Resource Locators (URLs), which consist of two parts: a host name (followed by an optional port number) and the hierarchical name that identifies the resource to be located.

With the extensions of version qualifiers and starting points, the full name of an artifact, thus, has to adhere to the following template.

```
//physical-repository/<name[:version]>[</name[:version]>....]
```

For example, assuming the logical repository that is presented in Figure 3.4, the following name is the full name of version 3 of the atom `Frame.c`.

```
//Milano/DrawingEditor/GUI-lib:3/Frame.c
```

Alternatively, it can be named as follows

```
//Boulder/WordProcessor/GUI-lib:3/Frame.c:3
```

or even, as illustrated by the bold arrows in Figure 3.5, as follows.

```
//Milano/DrawingEditor:1/Menu:2/PullDown.c:3
```

A single (version of an) artifact can thus, depending on the actual membership hierarchy of the artifacts in the versioned directed graph, be addressed with a multitude of names that each may originate in a different physical repository.

Note that in an actual implementation of the testbed the identification of a physical repository needs either a private DNS-like service that interprets the starting points, or a mapping of the starting points onto an existing DNS service. The implementation discussed in Chapter 6 uses the latter solution and identifies physical repositories by the host name and port number on which a server is running.

3.4 Access Model

Besides the precise definition of the storage, distribution, and naming of artifacts, the generic repository model needs to specify how access to stored artifacts is obtained. The fact that artifacts reside in a logical repository does not necessarily imply that they are directly manipulated there. In fact, it is common practice to build a CM system around the notion of a **workspace**. A workspace materializes a subset of artifacts in the file system and has three distinct advantages over direct manipulation. First, it provides an insulated work area in which artifacts can be manipulated without being influenced by the work of others. Second, a workspace provides a form of caching: it typically resides much closer in proximity to the originator of changes than the repository. Finally, a workspace is unobtrusive in that it provides existing tools and applications with access to versioned artifacts without the need to modify these tools and applications to understand the details of the storage and versioning mechanisms that are used.

For these reasons the access model prescribes the use of workspaces to access artifacts in a logical repository. Each workspace represents a particular version of a

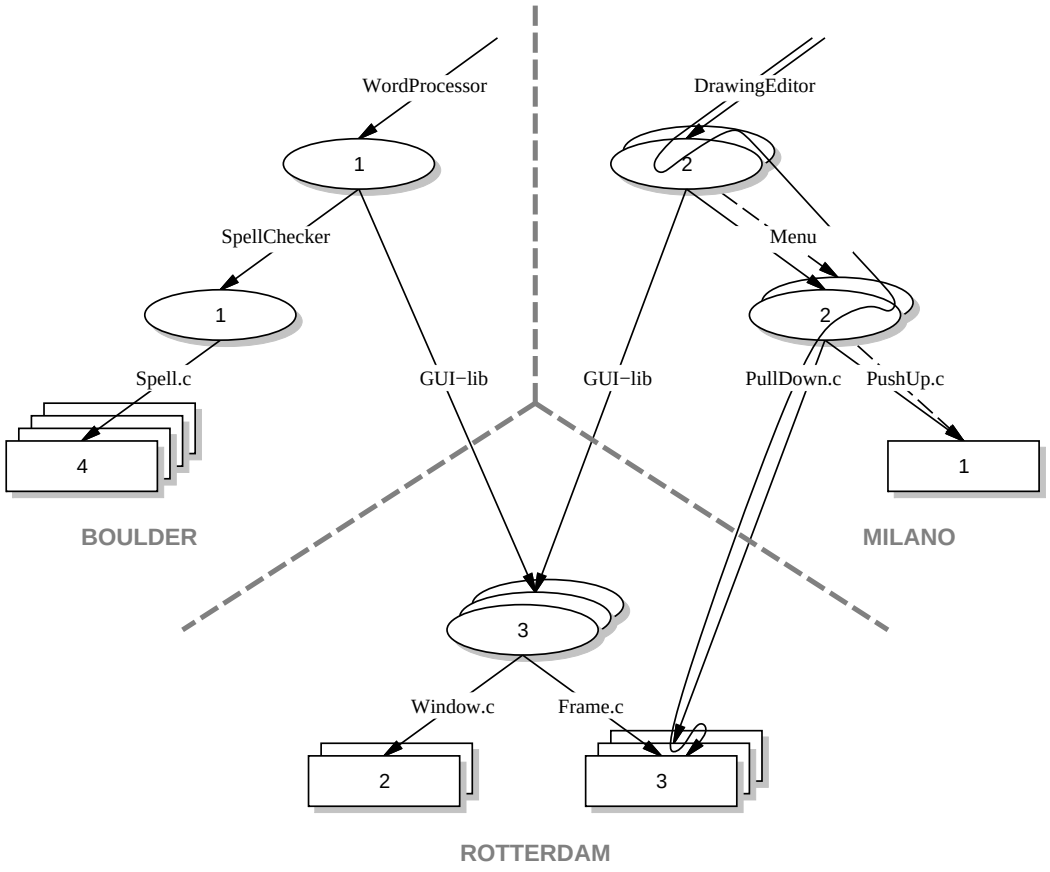


Figure 3.5: Hierarchical Naming with Version Qualifiers.

particular collection. The structure of the workspace follows the structure of the file system. In particular, collections materialize as directories, lower-level collections materialize as sub-directories, and atoms materialize as files. For example, version 2 of the collection `DrawingEditor` as presented in Figure 3.5 would have the following directory structure when materialized into a workspace on a UNIX file system.

```
.../DrawingEditor/GUI-lib/
                  /Menu/PullDown.c
                       /PushUp.c
```

Note that the workspace contains an empty directory called `GUI-lib`. This is due to the fact that the version of the collection `GUI-lib` that is part of the collection `DrawingEditor`, namely version 2, has no members. Note also that, although not illustrated, the workspace is not restricted to contain only the artifacts that are managed by the access model. It also may contain other directories and files, such as the auxiliary and temporary files created by word processors and compilers.

Unlike ClearCase [Atr92], which manages workspaces in the repository by employing a translucent file system in which operating system calls, such as `open`, `read`, and `write`, are trapped and interpreted at the repository, workspaces follow the model that is used by DRCS [OG90] and DCVS [HK92]: materialized artifacts are actual copies (in the file system) of the artifacts in the repository. The advantage is that proprietary replacements for low-level operating system functions do not have to be created (as with ClearCase) and that less network traffic is incurred.

Given that a single artifact can be a member of multiple collections, it is possible that an artifact is copied into a workspace multiple times. For example, if the collection `DrawingEditor` that is displayed in Figure 3.5 is materialized with version 3 of the collection `GUI-lib` rather than version 2 as well as version 3 of the atom `PullDown.c`

rather than version 2, the workspace would be populated as follows.

```
.../DrawingEditor/GUI-lib/Frame.c
                  /Menu/PullDown.c
                       /PushUp.c
```

In this workspace, the files `Frame.c` and `PullDown.c` represent exactly the same version of exactly the same artifact and their contents are, thus, identical. Rather than creating the illusion of a single materialized artifact (e.g., changes made to any of the materialized copies are visible in all copies in a workspace), the access model keeps each copy separate. This solution provides a CM policy with the flexibility to decide whether or not the various copies of an artifact should be kept synchronized, which is an explicit policy decision that should not be made by the access model.

A similar situation arises because of the potentially cyclic nature of the directed versioned graph of artifacts. In the case where no cycles are present, the materialization strategy preserves the hierarchical structure of the storage model and a one-to-one correspondence exists between a "slice" of the directed graph and the directories and files in the file system. However, when cycles are present in the graph, the one-to-one correspondence is broken since the copying strategy instantiates a potentially infinite series of materialized artifacts. Unfortunately, this is a limitation of the use of a file system to represent workspaces: some file systems simply do not have the facilities (e.g., symbolic or hard links) to represent cycles. However, it is pertinent that the access model uses the file system. Therefore, a sacrifice is made in that cycles are an exception with respect to the one-to-one correspondence between a slice of the versioned directed graph and the file system.

In traditional CM systems, the users of a workspace are human. The purpose of the workspace of the access model, though, is to provide CM systems access to artifacts that are stored in a repository. It is expected that these CM systems will manipulate workspace artifacts, providing tailored styles of access to their human users. This is

illustrated in Figure 3.6. Three layers, each containing a different representation of the artifacts, can be identified. The bottom layer is the repository that contains all versions of all artifacts. Some of these artifacts will be materialized into a workspace, which is illustrated by the middle layer. The materialized artifacts, in turn, might be transformed by a CM policy for presentation to a human user or application, resulting in the top layer. Note that the bottom two layers are standard, they are managed by the abstraction layer. The top layer, however, can be of any shape or form because they are policy dependent.

Two basic alternatives exist to representing the contents of a workspace to a user. In the first alternative, the workspace is transformed before its contents are presented to a user. Such a transformation might range from no transformation at all, to a simple restructuring of the artifacts within the workspace (e.g., changing their names or modification rights), to a wholesale restructuring of the artifacts into a separate directory structure. In the latter case, the workspace is solely used as a communication buffer between the CM policy and the repository. Figure 3.6a presents an example of a transformation. CM policy X moves the artifacts in the workspace to a different directory in the file system before giving the user access.

The second alternative is for the developer of a CM policy to provide a specialized browser or editor that hides the details of the workspace from the user. For example, the CM model described by Lin and Reiss [LR96] could use the repository model in such a way that its software units are mapped to atoms, its aggregate modules are expressed as collections, and its specialized browser is used to present the contents of workspaces. As another example, the workspaces used by the WebDAV prototype (see Chapter 7) are never made visible to a user. Instead, the artifacts are transferred to a Web browser that uses a specialized interface to manipulate and version them. Figure 3.6b is similar to the WebDAV prototype: CM policy Y does not allow a user direct access to the artifacts in the workspace and uses a Web browser instead to let a user manipulate

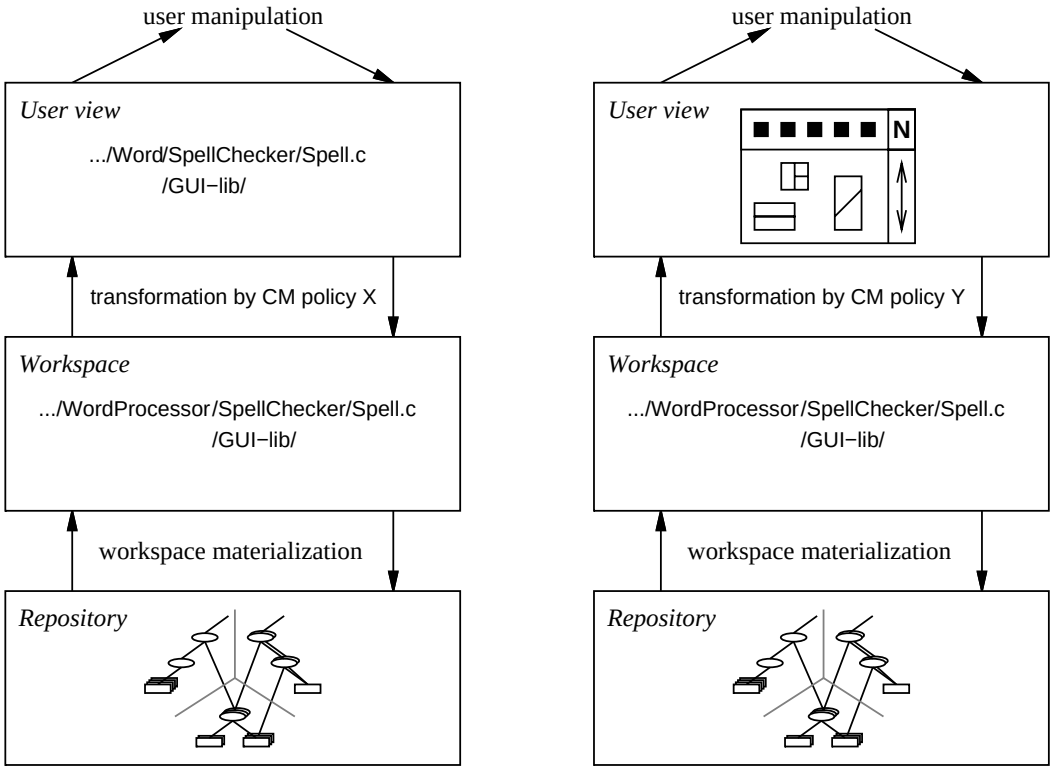


Figure 3.6: An Instance of the Access Model in which the Workspace Is Transferred to a Different Location in the File System (a), and an Instance of the Access Model in which the Workspace Is Hidden from the User (b).

artifacts.

3.5 Attribute Model

To facilitate the storage of metadata in a repository, the repository model allows the association of attributes with individual versions of artifacts. Attributes are untyped name-value pairs. An example of their use is provided in Figure 3.7, which shows the attributes that are associated (by some hypothetical CM policy) with the various versions of the atom `Spell.c`. The CM policy labels all versions of an artifact with the attributes `Author`, `Version`, and `ChangeComment`. Furthermore, if a new version of an artifact addresses a previously identified bug, that version will be labeled with the attribute `BugReport`, which contains the number of the bug report that describes the bug that is resolved. Finally, if a version of an artifact is locked, the attribute `Lock` is set to contain the e-mail address of the person who has placed the lock. Note that some attributes contain values that are assigned by the CM policy itself (e.g., `Author`, `Version`, and `Lock`), whereas other attributes contain values that are supplied by users of the CM policy (e.g., `BugReport` and `ChangeComment`).

Access to attributes is handled differently than access to artifacts. Instead of being materialized in a workspace, attributes are directly accessed and manipulated in a repository. The reason for this distinction is twofold. First, attributes are not the object of change in a CM system, they merely capture auxiliary information about the objects that do change (the artifacts). Therefore, their values tend to be set once rather than continuously changed.

The second reason for directly manipulating attributes in a repository is that they can be used in a supporting role for such process engines as Spade [BFGL94], Oz [BSK94], and Endeavors [BT96]. Whereas these process engines can be used to actually execute and enforce the CM process, attributes can be used for such complementary purposes as setting locks, communicating who is changing a certain artifact,

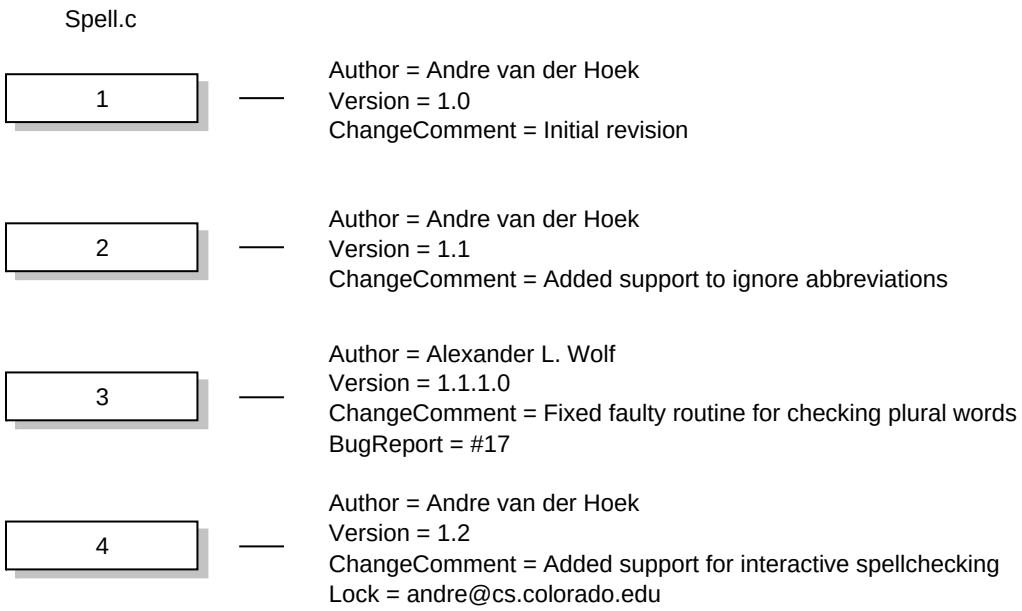


Figure 3.7: Example Attributes Associated with the Atom `Spell.c`.

determining who has ownership of a particular artifact, and other sorts of information that need to be maintained, exchanged, and manipulated by the process engines. For the management of this kind of supporting information it is critical that attributes are directly manipulated in a repository, where changes are immediately visible to all participants, rather than in a workspace, where changes are not visible to others unless they are committed to the repository.

The attribute model is rather basic. More advanced mechanisms, adding inherent support for attribute typing or multi-value attributes, can certainly be adopted. However, these types of functionality represent convenience functions and, in reality, are of a development and not a research nature. Given that the basic attribute mechanism is capable of supporting most CM needs, the addition of these types of extensions is left to future development. The segregation of the attribute model as a separate submodel within the overall generic repository model ensures that such extensions can be made without impacting other areas of functionality.

Chapter 4

Programmatic Interface

The second component of the abstraction layer defined by the testbed is the programmatic interface through which artifacts that are stored in a repository can be manipulated by CM policies. The complete interface consists of seven categories of functionality. These categories, listed in Table 4.1 with the functions they contain, are the following: **access** functions, which provide access to artifacts in a repository by materializing them in a workspace; **versioning** functions, which manage the way artifacts evolve into new versions; **collection** functions, which manage the membership of collections; **distribution** functions, which control the placement of artifacts in specific physical repositories; a **deletion** function, which allows a CM policy to remove artifacts from a repository; **query** functions, which provide a CM policy with various kinds of information about the state of artifacts in a repository or workspace; and **attribute** functions, which manage the association of attributes with versions of artifacts.

A CM policy is built by programming against the interface and using combinations of interface functions to implement the particular functionality needed. Because a wide range of CM policies has to be supported, the interface functions—much like the various submodels in the repository model—do not impose any particular CM policy. Instead, they provide the mechanisms for CM systems to implement specific policies. While the particular semantics of the interface functions might therefore seem odd from the perspective of a human user, those same semantics are invaluable to a CM policy

Table 4.1: Programmatic Interface Functions.

Category	Function	Description
Access	nc_open	Materializes an artifact version into a workspace.
	nc_close	Removes an artifact version from a workspace.
Versioning	nc_initiatechange	Allows an artifact version in a workspace to be modified.
	nc_abortchange	Returns an artifact version in a workspace to the state it was in before it was initiated for change.
	nc_commitchange	Stores a new version of an artifact in a repository.
	nc_commitchangeandreplace	Overwrites the current version of an artifact in a repository.
Collection	nc_add	Adds an artifact version to a collection.
	nc_remove	Removes an artifact version from a collection.
	nc_rename	Renames an artifact within a collection.
	nc_replaceversion	Replaces the version of an artifact within a collection.
	nc_copy	Copies the versions of an artifact and adds a version of the new artifact to a collection.
	nc_list	Determines the member artifact versions of a collection.
Distribution	nc_setmyserver	Sets the default physical repository in which new artifacts will be stored.
	nc_getlocation	Determines the physical repository that contains the versions of an artifact.
	nc_move	Moves an artifact and its versions from one physical repository to another.
Deletion	nc_destroyversion	Physically removes an artifact version from a repository.
Query	nc_gettype	Determines the kind of an artifact.
	nc_version	Determines the current version of an artifact.
	nc_lastversion	Determines the latest version of an artifact in a repository.
	nc_existsversion	Determines whether a version of an artifact exists in a repository.
	nc_isinitiated	Determines whether an artifact version has been initiated for change in a workspace.
	nc_isopen	Determines whether an artifact version has been materialized into a workspace.
Attribute	nc_testandsetattribute	Associates an attribute and its value with an artifact version (if the attribute does not yet exist).
	nc_setattribute	Associates an attribute and its value with an artifact version (whether or not the attribute exists).
	nc_getattributevalue	Determines the value of an attribute of an artifact version.
	nc_removeattribute	Disassociates an attribute from an artifact version.
	nc_selectversions	Determines the set of versions of an artifact for which an attribute has a certain value.

implementor.

An important characteristic of the programmatic interface is the orthogonality among the various functional categories. Specifically, the functions in one category are independent of the functions in the other categories. For example, the distribution functions are the only functions concerned with the distributed nature of a repository. The other functions are not influenced by the fact that artifacts are stored in different locations. Their behavior is the same, regardless of whether the artifacts are managed by a local repository or a remote one. Similarly, the collection functions are the only functions that recognize the special nature of collections. The other functions in the programmatic interface behave the same, irrespective of whether they operate on atoms or collections.

It should be noted that the functionality offered by each individual interface function is rather limited. At first, this seems contradictory to the goal of providing a high-level interface for configuration management policy programming. However, because of the limited functionality, each function can be defined with precise semantics. Not only does that generalize the applicability of the interface functions, it also allows the rapid construction of particular CM policies through the composition of sets of interface functions.

Below we introduce, per category, the individual interface functions that constitute the programmatic interface to the generic repository model. It should be noted that the introduction is relatively informal. The functionality of each function is discussed and examples of their use are given. A detailed definition of each function, as implemented in the NUCM prototype discussed in Chapter 6, is provided in Appendix A.

4.1 Access Functions

Access to the artifacts in a repository is, as discussed in Section 3.4, obtained through a workspace in which artifacts are materialized upon request. The access func-

tions manage these requests, materializing artifacts in a workspace when they are needed and removing artifacts from a workspace when their presence is no longer required. Once artifacts are materialized, other interface functions become available to manipulate them. In particular, versioning functions can be used to create and store new instances of artifacts, and collection functions can be used to manipulate the membership of collections.

The access functions in the programmatic interface are `nc_open` and `nc_close`. The function `nc_open` provides access to a particular version of an artifact by materializing it in a workspace. Atoms are materialized as files, collections as directories. Each use of the function `nc_open` materializes a single version of a single artifact. A workspace, then, has to be constructed in an incremental fashion. This mechanism allows a CM policy to populate a workspace only with the artifacts that it needs, thereby avoiding any overhead of unnecessarily retrieving artifacts that will not be used.

The function `nc_open` can be used in two ways. In the first, an artifact version is directly materialized from a repository and a full name as defined in Section 3.3 is required. In the second, the artifact version to be opened is defined by a regular operating system path name that points to an artifact version that is already open in another workspace; that version of the artifact is retrieved from the repository. Although in both cases the artifact version is retrieved from the repository, the second use allows a CM policy to duplicate an existing workspace without the need to memorize the exact contents of that workspace.

The function `nc_close` is used to remove artifacts from a workspace. The function operates in a recursive manner: when a collection is closed, all the artifacts that it contains are removed from the workspace as well. However, closing a collection only succeeds when neither the collection, nor any of its contained artifacts in the workspace, is in a state that allows them to be changed (see Section 4.2). This forces a CM policy to explicitly commit or abandon any changes, thereby avoiding their unintentional loss.

When an artifact is closed, it is only removed from the workspace. The function `nc_close` has no effect on the contents of the repository.

4.2 Versioning Functions

Once a set of artifacts has been opened in a workspace, the following functions become available to create and store new versions of these artifacts: `nc_initiatechange`, `nc_abortchange`, `nc_commitchange`, and `nc_commitchangeandreplace`.

Through the function `nc_initiatechange`, a CM policy informs a workspace of its intention to make a change to an atom or a collection. In response, permission is granted to change the artifact in the workspace. If the artifact is an atom, it can be manipulated by any user program since its contents are not interpreted by the model or interface. A collection, on the other hand, can only be manipulated through the use of collection functions because those functions preserve its special nature (see Section 4.3).

Permission to change an artifact in one workspace does not preclude that artifact from being changed simultaneously in another workspace. In particular, the function `nc_initiatechange` does not lock an artifact. If a locking protocol is desired, then the attribute functions described in Section 4.7 can be used to construct that protocol. This orthogonality of locking and versioning permits the development of CM policies that range from the optimistic, in which artifacts are not locked and changes are merged when conflicts arise, to the pessimistic, in which artifacts are locked to avoid conflicts.

The function `nc_abortchange` abandons an intended change to an artifact. It reverts the materialized state of the artifact back to the state that it was in before the function `nc_initiatechange` was invoked. Similar to the function `nc_close`, the function `nc_abortchange`, when applied to a collection, can only succeed if no artifacts that are part of the collection are currently in a state that allows them to be changed. Changes to these artifacts need to be either committed or abandoned.

To store the changes that have been made to an artifact, two alternative functions

can be used. The first, `nc_commitchange`, commits the changes by storing a new version of the artifact in the repository. It is the **only** function in the programmatic interface that actually creates new versions of artifacts. None of the other functions have this capability, neither directly nor as a side effect. The second function used to store changes to artifacts is `nc_commitchangeandreplace`. As its name implies, this function is similar in behavior to the function `nc_commitchange`, but instead of creating a new version of the artifact, it overwrites the contents of the version that was initiated for change. Both functions, in addition to storing the new contents of the artifact in the repository, also revoke the permission to make further changes to the artifact in the workspace. But, once again, locking is an orthogonal concern that is managed with a different category of functions. Therefore, neither function releases any locks that may be held.

The availability of these alternative storage functions allows a CM policy programmer to choose whether particular changes lead to new versions of artifacts or not. This is an especially important decision in the case of collections. Whereas some CM policies prescribe that any change to a member artifact leads to a new version of the collection (e.g., Poem [LR96] or CoED [BLNP98]), other CM policies only version collections when the actual structure of the collection (i.e., its artifact membership) has changed (e.g., ShapeTools [ML88] or ClearCase [Atr92]). Since this is a policy decision, the programmatic interface facilitates both cases. To model the first case, the function `nc_commitchange` is used on the collection, whereas the latter case requires the use of the function `nc_commitchangeandreplace`. Given that an artifact can be a member of multiple collections, a CM policy could even choose to use a different approach for each collection.

To illustrate the versioning functions, the example of Figure 3.5 is continued. Assume that, using the access functions, a workspace has been opened that contains a

subset of the artifacts in the repository.

```
.../DrawingEditor/GUI-lib/
                  /Menu/PullDown.c
                       /PushUp.c
```

To be able to modify the atom `PushUp.c`, the function `nc_initiatechange` is invoked. Once the desired changes have been made, the function `nc_commitchange` is used to store a new version of the atom `PushUp.c` in the repository. The result is shown in Figure 4.1. The repository now contains two versions of the atom `PushUp.c`, but note that the collection `Menu` has not changed, since the function `nc_initiatechange` was not invoked on that collection. In particular, the membership of the collection `Menu` is still the same as before the creation of the new version of the atom `PushUp.c`. If, instead of the function `nc_commitchange`, the function `nc_commitchangeandreplace` had been used, no new version would have been created for the artifact `PushUp.c`. In fact, the structure of the repository would still have looked like the one of Figure 3.5, even though the actual contents of version 1 of the atom `PushUp.c` would have changed.

4.3 Collection Functions

Similar to the way an editor or drawing program can be used to change an atom in a workspace, collections need to be changed via some kind of mechanism. But, because collections have special semantics, it would be unwise to allow them to be edited directly. Therefore, the programmatic interface contains a number of functions that preserve the semantics of collections while updating their contents. These functions are the following: `nc_add`, `nc_remove`, `nc_rename`, `nc_replaceversion`, `nc_copy`, and `nc_list`. An important aspect of these functions is that they do not directly modify collections in the repository. Instead, they can only modify collections that have been materialized (and initiated for change) in a workspace. To promote these changes to the repository, the versioning functions described in the previous section must be used.

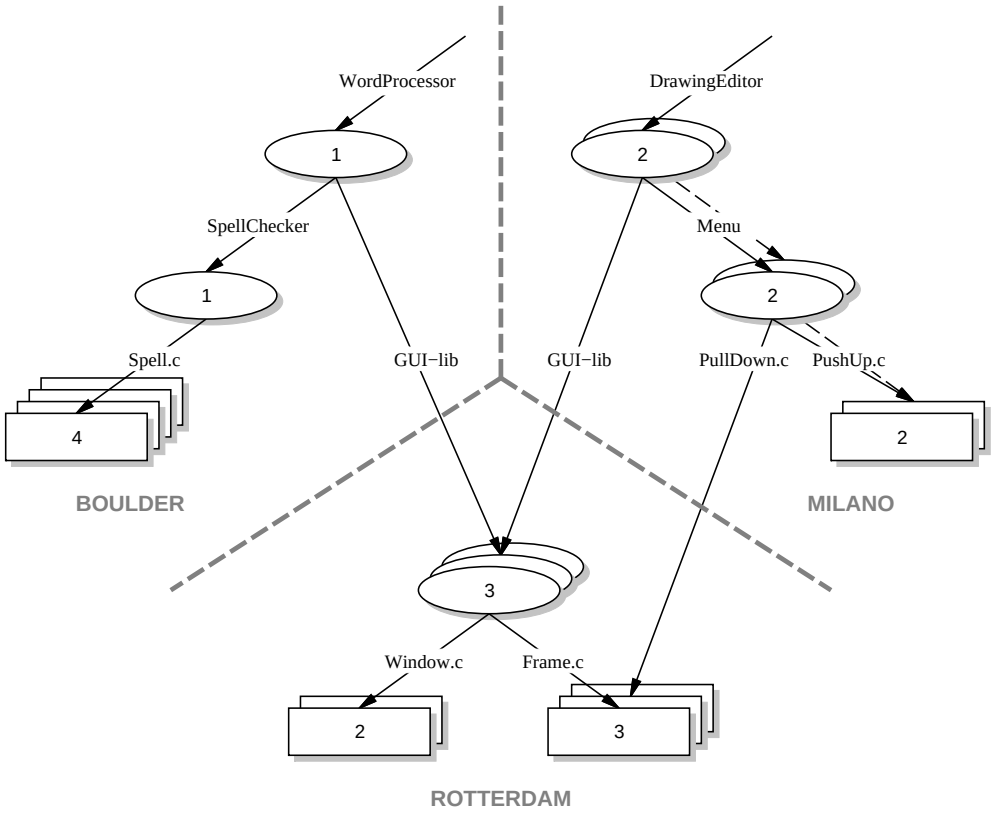


Figure 4.1: Example Repository Contents after a New Version of the Atom `PushUp.c` Is Stored.

This scheme allows many changes to a collection to be grouped into a single change to the repository. As an important side effect of grouping multiple changes into a single change, the version explosion problem is avoided. In particular, if each change to a member artifact leads to a new version of its parent collections, collections at the higher levels in the graph would have to be versioned at a very high rate. This clearly is inefficient and the programmatic interface allows a CM policy to avoid this situation altogether.

The functions `nc_add` and `nc_remove` behave as expected, adding and removing a version of an artifact to and from a collection, respectively. The function `nc_add` can add either a new or an existing artifact to a collection. The addition of a new artifact will simply store its contents in the repository. The addition of an existing artifact, on the other hand, will result in an artifact that is shared by multiple collections and for which a single version history is maintained (such as the collection `GUI-lib` in Figure 4.1, which is shared by the collections `WordProcessor` and `DrawingEditor`). If, instead of a shared version history, a separate version history is desired, then the function `nc_copy` must be used in place of the function `nc_add`. A distinctly new artifact will be created in the repository. Initially this artifact contains the same version history as the artifact that was copied, but the new artifact evolves separately.

A feature that has been difficult to provide in CM systems is the ability to rename artifacts. The testbed solves this problem by providing, directly in its programmatic interface, the function `nc_rename`. Because an artifact is only renamed within a single collection at a time, it is possible for an artifact to exist under different names in different collections. This is an important feature of the programmatic interface, since it allows an artifact to evolve without compromising its naming history (e.g., renaming an artifact in one collection does not change its name in the other collections that contain the artifact).

The function `nc_replaceversion` complements the other collection functions be-

cause it operates in the version dimension as opposed to the naming dimension. Its behavior is simple: it changes the member version of an artifact in a collection to another version. Its purpose is to allow a collection to stay in tune with the ongoing evolution of its member artifacts. However, the function `nc_replaceversion` is not limited to replacing older versions of an artifact with newer ones. Changing the member version of an artifact is also useful in the opposite direction. In particular, for CM policies that would like to provide an "undo" facility, newer versions in a collection can be replaced by older ones.

The function `nc_list` rounds out the collection functions. It returns a list of the names and versions of the artifacts that are members of a collection. This functionality is useful in building a CM policy that, for example, presents a user with the differences between two versions of a collection, recursively opens a workspace, or simply allows a user to dynamically select which artifacts to lock or check out.

The set of collection functions is complete. If we consider the artifacts that are members of a collection to be organized in a two-dimensional space defined by name and version, all primitive functionality is provided. A name-version pair can be added, a name-version pair can be removed, a name is allowed to change, and a version is allowed to change. Despite the rather primitive functionality provided by each individual function, the complete set of collection functions allows for the rapid construction of higher-level, more powerful functions. For example, a function that replaces, under the same name, one atom with another, can be constructed as a sequence of invocations of the functions `nc_remove`, `nc_add`, and `nc_rename`.

The use of the collection functions is illustrated by a continuation of the example of Figure 4.1. Assume that all artifacts are still open in the workspace. To be able to manipulate the membership of the collection `Menu`, the function `nc_initiatechange` is first used to gain proper permission. Then, to update the atom `PushUp.c` to its latest version, the function `nc_replaceversion` is applied. In addition, to provide a

"popup" rather than a "pulldown" menu in the collection `Menu`, the function `nc_remove` is used to remove the atom `PullDown.c` and the function `nc_add` is used to add the newly created atom `PopUp.c`. These changes are transferred to the repository using the function `nc_commitchange`. As a result, the repository looks as shown in Figure 4.2. A new version of the collection `Menu` has been created that reflects the new membership. In addition, because the function `nc_commitchange` was used instead of the function `nc_commitchangeandreplace`, the old version of the collection is still available. This means that if the function `nc_list` is used on version 2 of the collection `Menu`, then version 2 of the atom `PullDown.c` and version 1 of the atom `PushUp.c` are listed as members, whereas if the function `nc_list` is used on version 3 of the collection, then version 2 of the atom `PushUp.c` and version 1 of the atom `PopUp.c` are listed as members.

4.4 Distribution Functions

An important aspect of the distribution model discussed in Section 3.2 is that it isolates distribution. This is reflected in the semantics of the various interface functions, since the functions behave the same whether artifacts are stored locally or remotely. On the other hand, sometimes there is a need for control over the location of artifacts. In general, users of systems that completely hide distribution often encounter performance difficulties related to the physical placement of data. To counter this problem, the programmatic interface contains functions that allow a CM system to determine and change the physical location of artifacts within a logical repository.

The first function, `nc_setmyserver`, specifies the default physical repository to which newly created artifacts are added. New artifacts can be added to any physical repository, since it is not required that they are added to the same physical repository as the one in which their parent collection resides. When a new artifact is added to a different repository, a connection is made between that repository and the repository in which the parent collection is located. This connection is the bridge that forms the

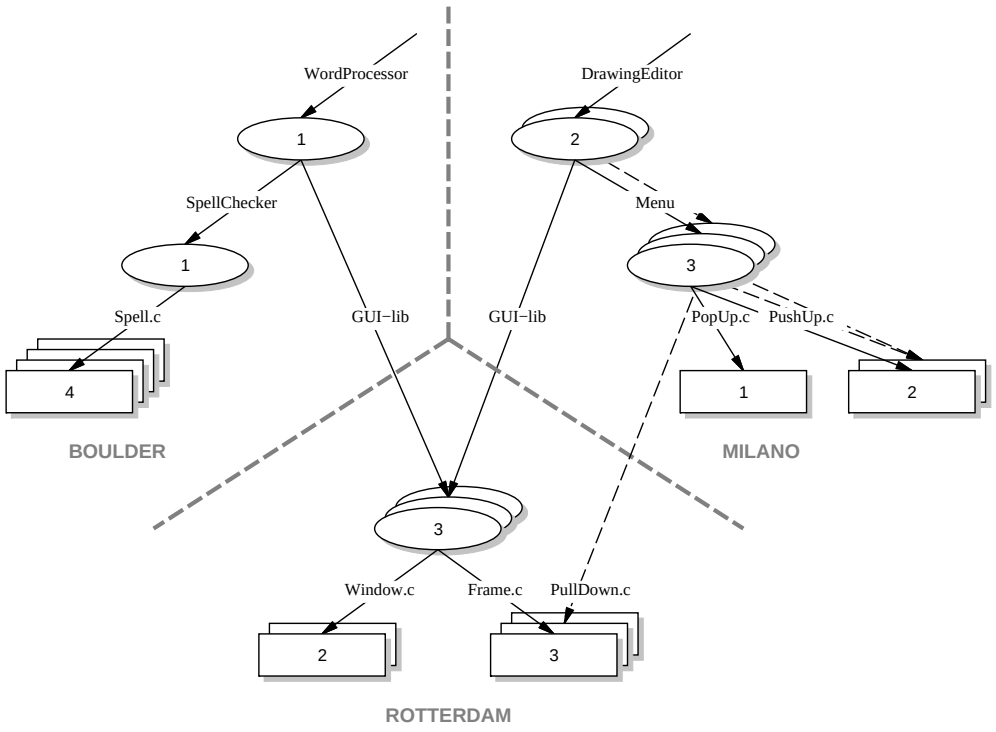


Figure 4.2: Example Repository Contents after a New Version of the Collection `Menu` Is Stored.

logical repository spanning the two physical repositories.

To determine the actual location of an artifact, the function `nc_getlocation` is used. It returns the physical repository in which an artifact is stored. This information can, in turn, be used by the function `nc_move` to collocate artifacts that are regularly used together or to move artifacts to those physical repositories that are closer in proximity to the workspaces in which they are manipulated. To comply with the requirement set forth by the distribution model that all versions of an artifact are located in a single physical repository, the function `nc_move` moves the complete version history of an artifact from one physical repository to another. Note that the movement of artifacts has no effect on the contents of a workspace. In fact, to avoid the opening and closing of artifacts that only need to be moved and do not need to be changed, it is possible to move artifacts directly without first opening them in a workspace.

To demonstrate a typical use of the distribution functions, assume that it is decided that the collection `GUI-lib` of Figure 4.2 should have ownership of the atoms `PopUp.c` and `PushUp.c`. The collection `GUI-lib` is first opened and then initiated for change in a workspace using the functions `nc_open` and `nc_initiatechange`, respectively. Subsequently, the atoms `PopUp.c` and `PushUp.c` are added by using the function `nc_add`, after which the new version of the collection `GUI-lib` is stored using the function `nc_commitchangeandreplace`. At this point, both the atoms `PushUp.c` and `PopUp.c` are still located in the physical repository of Milano, an undesirable situation since they should be collocated with the other members of the collection `GUI-lib`. To remedy this situation, the function `nc_move` is applied in order to move both artifacts to the physical repository of Rotterdam. The result is shown in Figure 4.3. Note that the complete version history of both artifacts has been moved, and not only the directly contained member versions. Note also that the order in which the above operations take place is irrelevant. The artifacts could have been moved before being added to the collection `GUI-lib`.

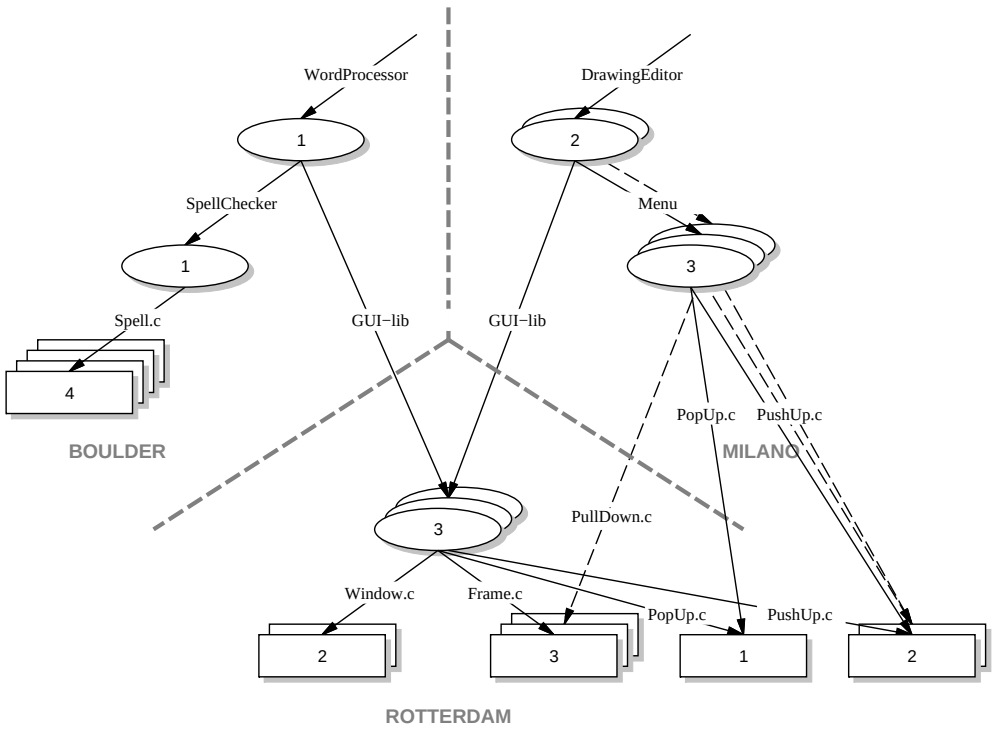


Figure 4.3: Example Repository Contents after the Physical Repository of Milano Assumes Ownership of the Atoms `PushUp.c` and `PopUp.c`.

4.5 Deletion Function

Since it violates the basic premise of always having a precise history of all changes to all artifacts, deleting (versions of) artifacts from a repository is an uncommon practice in the domain of configuration management. Nevertheless, it should still be possible to do so. Therefore, the function `nc_destroyversion` is provided in the programmatic interface to physically delete a particular version of an artifact from a repository. Normally, this function is used to purge obsolete artifact versions from a repository. Consider the example of Figure 4.3. The artifact `spell.c` is present in four different versions. If each of these versions occupies a lot of storage space, a possible scenario involves using the function `nc_destroyversion` to delete some of the earlier versions in order to acquire extra storage space for other artifacts.

A second, less common use of the function involves reverting erroneous checkins. For example, when a new version of `spell.c` is checked in and its contents is just plain wrong, the function `nc_destroyversion` can be used to delete this new version from the repository.

Three observations need to be made about the use of the function `nc_destroyversion`. The first is that the function operates directly on a repository. An artifact version that needs to be deleted does not have to be opened in a workspace. Because the artifact is going to be deleted from the repository, first opening the artifact in a workspace would only incur useless communication overhead without providing any additional benefits.

The second observation regards the versions of an artifact that can be deleted by the function `nc_destroyversion`. A specific rule is enforced: a version of an artifact can only be deleted if it is not a member of a collection. Consider, once again, the example in Figure 4.3. In this example, a CM policy is allowed to delete version 1 of the atom `Frame.c`, since that version is not directly contained by a collection. The

deletion of version 2 and version 3, however, is disallowed because version 2 is a member of version 2 of the collection `Menu.c` and version 3 is a member of version 3 of the collection `GUI-lib`. This restriction preserves the consistency of the repository structure by avoiding "dangling" membership relations. If, for any reason, a CM policy does need to delete a version that is contained by a collection, it first has to remove the containment that prevents the deletion. For example, a CM policy may need to delete version 2 of the atom `Frame.c` from the repository in Figure 4.3. The policy first has to use the function `nc_replaceversion` to change the version that is a member of version 2 of the collection `Menu`. Alternatively, the CM policy may use the function `nc_remove` to remove the atom from the membership of version 2 of the collection `Menu` altogether. After one of those two functions has been properly applied, version 2 of the atom `Frame.c` is no longer contained and can be physically deleted from the repository by using the function `nc_destroyversion`.

The third observation is that the function `nc_destroyversion`, by itself, is not sufficient to be able to delete all artifacts from a repository. A second, implicit form of deletion has to be provided by an implementation of the abstraction layer that complements the explicit use of the function `nc_destroyversion`. The implicit deletion has to take care of two specific cases. First, by allowing artifacts to be removed from a collection with the function `nc_remove`, it is possible that none of the versions of a certain artifact can be addressed. Consider the example in Figure 4.3. If version 1 of the collection `SpellChecker` is removed from version 1 of the collection `WordProcessor`, it is no longer possible to address the single version of the collection `SpellChecker` or the various versions of the atom `Spell.c`. These two artifacts form an isolated part of the graph and the storage space that they occupy needs to be reclaimed.

The second case regards a sequence in which a new artifact is added to a collection in a workspace with the function `nc_add`, but removed from that collection by the function `nc_remove` before a new version of the collection is stored in the repository.

In this case, the function `nc_add` has added a new artifact to the repository that is no longer reachable after the function `nc_remove` has been applied. Once again, its storage space needs to be reclaimed.

As a comprehensive example, consider the contents of Figure 4.3. Assume that version 1 of the atom `Frame.c` is explicitly deleted by using the function `nc_destroy-version`, that version 3 of the collection `GUI-lib` is modified by using the function `nc_remove` to remove version 3 of the atom `Frame.c`, and that version 2 of the collection `Menu` is modified by using the function `nc_remove` to remove version 2 of the atom `PullDown.c`. The repository that results after these changes are made is illustrated in Figure 4.4. Observe that version 2 and version 3 of the atom that was known as `Frame.c` (or `PullDown.c`) are no longer reachable in the graph. Therefore, an implementation of the testbed has to make sure that the storage space occupied by these versions is reclaimed. This results in the repository contents that is shown in Figure 4.5.

4.6 Query Functions

The programmatic interface would not be complete without the ability to examine the state of artifacts. For example, when multiple users share access to an artifact, they should be able to determine whether any new versions of the artifact have been created by another user. As another example, when new artifacts are added to a repository by one user, another user should be able to determine whether the artifact is an atom or a collection. The query functions were designed to provide this type of functionality. Although simple, these functions are essential in the development of CM policies because they provide state information that a CM policy would otherwise have to determine and track itself. The query functions that provide information about the artifacts in a workspace are particularly important in this respect.

Six functions fall into the category of query functions. The first, `nc_gettype`, determines whether an artifact is a collection or an atom. Artifacts do not have to be

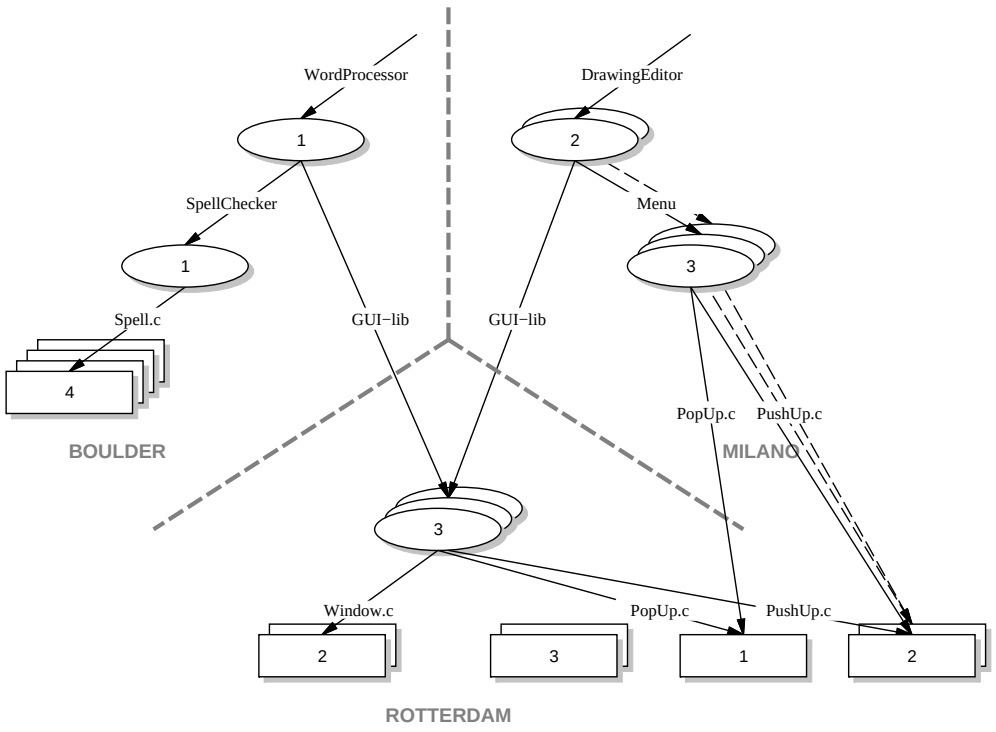


Figure 4.4: Example Repository Contents after Version 1 of Atom `Frame.c` Is Deleted, Version 3 Is Removed from the Collection `GUI-lib`, and Version 2 Is Removed from the Collection `Menu`.

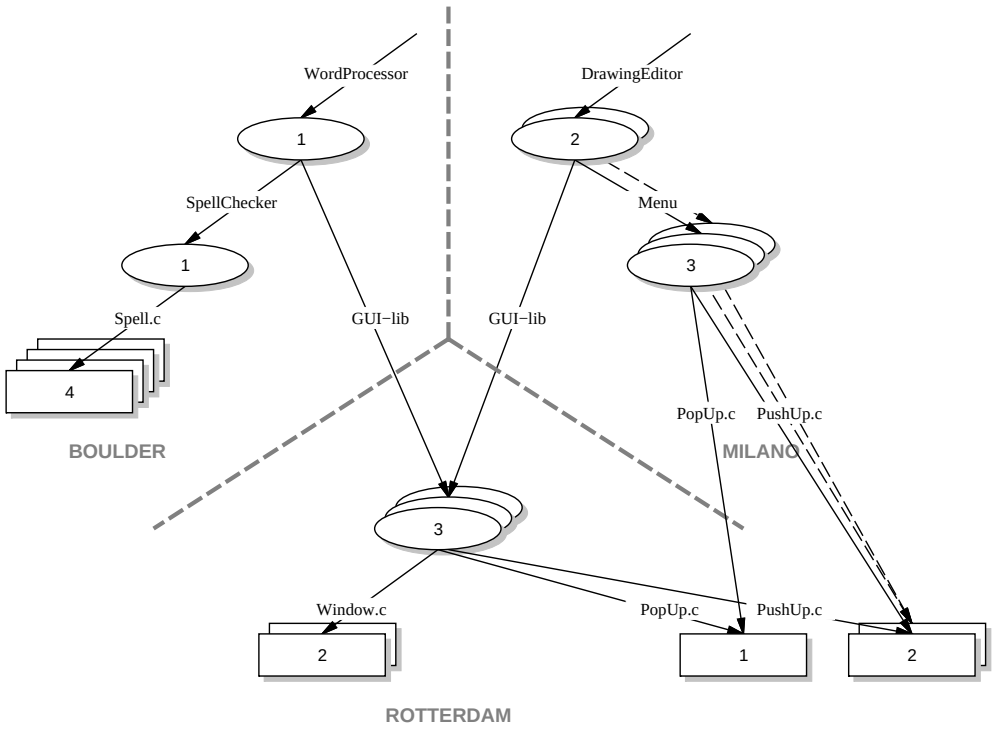


Figure 4.5: Example Repository Contents after the Storage Space for Version 2 and Version 3 of the Atom `Frame.c` Is Reclaimed.

opened in a workspace to use the function `nc_gettype`. Both artifacts in a repository and artifacts in a workspace can be examined for their type. The function is often used when recursively opening a collection and all its contained artifacts in a workspace.

The second function, `nc_version`, determines the specific version of the artifact that a particular name addresses. Once again, artifacts do not have to be opened for their version to be identified. Both the version of an artifact in a repository and the version of an artifact in a workspace can be determined. Typically, the function `nc_version` is used in the management of version relationships, such as a revision history. In particular, when the function `nc_version` is used to determine the version of an artifact before and after the function `nc_commitchange` has been used to store some changes, the predecessor relationship can be tracked and stored in a version tree.

To determine the last version of an artifact that exists in a repository, the function `nc_lastversion` is used. It returns the version number of the last version of the artifact that was added to the repository using the function `nc_commitchange`. Normally, the function `nc_lastversion` is used to check for new versions of an artifact that may have been added by another user.

If some versions of an artifact have been deleted from a repository, a CM policy needs to be able to verify whether a particular version that is requested by a user is still available. The function `nc_existsversion` is designed for exactly this purpose: it returns whether a version of an artifact is present in a repository or not.

The last two functions solely operate on workspaces. The function `nc_isopen` verifies whether an artifact has been opened in a workspace. The function `nc_isinitiated` complements the function `nc_isopen` by determining whether an artifact is in a state that allows a CM policy to modify it (i.e., the function `nc_initiatechange` has been applied to the artifact but neither the function `nc_commitchange` nor the function `nc_commitchangeandreplace` has yet been used to store the changes in the repository). Typically, these functions are used to reacquaint a user with the state of a

workspace after the workspace has not been accessed for some time. In addition, the function `nc_isinitiated` is regularly used to inform a user about the artifacts that can be changed in a workspace.

4.7 Attribute Functions

To facilitate, in accordance with the attribute model, the association of metadata with the artifacts in a repository, the programmatic interface contains a number of primitive functions to manipulate attributes. In particular, it is possible to set the value of an attribute with either the function `nc_setattribute`, which sets the value of an attribute irrespective of whether a value is already set, or the function `nc_testandsetattribute`, which only sets the value of an attribute when the attribute is currently non-existent. To remove an attribute, the function `nc_removeattribute` is used. This function removes both the attribute and its associated value. To search the attributes that may be set on the various versions of an artifact, the function `nc_selectversions` is used: for a particular artifact in the repository and for a desired attribute value, it returns the version number of those versions whose corresponding attribute matches the value.

The attribute functions serve a dual purpose. First, they are used to simply attach metadata to individual versions of an artifact. For example, it is possible to capture such characteristics as the author and creation date of the version, one or more change request identifiers that identify which particular change requests have been incorporated, and a short synopsis of the changes made with respect to the previous version.

The second purpose for which the artifact functions were designed is to support an artifact locking mechanism. In particular, the function `nc_testandsetattribute` only sets the value of an attribute if it does not yet exist. Therefore, the function can be used to create a lock on an artifact by simply setting an attribute that represents the lock. If the artifact had been previously locked (i.e., the attribute is set), then the function, and hence the lock attempt, will fail. If the attribute had not been previously

locked (i.e., the attribute is not set), then the function and lock attempt will succeed. The function `nc_removeattribute` unlocks the artifact by removing the attribute.

Because of their generic nature, the attribute functions do not themselves enforce locks. Any enforcement results from the usage protocol employed by a CM policy. For example, a lock can be "broken" (intentionally or unintentionally) by using the function `nc_setattribute` on an existing lock attribute, since the function will not fail to set the attribute even though the attribute already exists. In a similar vein, the interpretation of a lock on a collection is left to the CM policy: does it mean that only the collection itself is locked or does it mean that anything reachable from the collection is also locked? The usage protocol employed by the CM policy will provide an answer that is consistent with the policy it seeks to implement.

Although using the attribute functions for purposes of artifact locking results in a rather primitive mechanism, it is demonstrated in Section 5.1 that the functions are powerful enough to directly model the locking schemes employed in such existing CM systems as RCS [Tic85], CCC/Harvest [Sof94a], and others. If more sophisticated locking schemes are required, then a separate lock manager, such as Pern [Hei96], can be used instead. This approach is consistent with the desire for locking to be orthogonal to the other functionalities of the interface.

To illustrate both roles of the attribute functions, first revisit Figure 3.7. Shown is a set of attributes and their values as associated with the various versions of the atom `Spell.c`. To change the comment associated with version 4, the function `nc_setattribute` is used. It overwrites the current value of the change comment with the new text. To determine all versions of the atom `Spell.c` that are locked, the function `nc_selectversions` is used. Given the example attributes in Figure 3.7, it returns 4 as the only version that is currently locked.

Table 4.2 demonstrates how the attribute functions can be used to lock individual versions of an artifact. The table contains a sequence of function invocations that are

performed by two different users. Both users want to change the same version of the atom `Spell.c`. However, because the first user is successful in obtaining a lock through the use of the function `nc_testandsetattribute`, the second user cannot set a lock on the artifact. Adhering to the CM policy at hand, the second user refrains from making any modifications, even though the artifact has been opened in the workspace. The first user makes the desired changes and commits those changes to the repository. However, committing changes does not remove any locks, thus the second invocation of the function `nc_testandsetattribute` by the second user still fails. Only after the first user has made more changes, committed those changes, and unlocked the artifact version using the function `nc_removeattribute`, the second user can lock the appropriate version of the atom `Spell.c` and perform the desired changes.

Table 4.2: Using Attributes to Lock a Version of an Artifact.

User 1	**User 2**
nc_open(Spell.c)	
nc_testandsetattribute(Spell.c, LOCK, user1)	
→ (succeed)	
	nc_open(Spell.c)
	nc_testandsetattribute(Spell.c, LOCK, user2)
	→ (fail)
nc_initiatechange(Spell.c)	
. . .	
. . . change Spell.c. . .	
. . .	
nc_commitchange(Spell.c)	
	nc_testandsetattribute(Spell.c, LOCK, user2)
	→ (fail)
nc_initiatechange(Spell.c)	
. . .	
. . . change Spell.c. . .	
. . .	
nc_commitchange(Spell.c)	
nc_removeattribute(Spell.c, LOCK)	
	nc_testandsetattribute(Spell.c, LOCK, user2)
	→ (succeed)
	nc_initiatechange(Spell.c)
	. . .
	. . . change Spell.c. . .
	. . .
	nc_commitchange(Spell.c)
	nc_removeattribute(Spell.c)

Chapter 5

Modeling Example Configuration Management Policies

To understand how the testbed can be used in the creation of CM policies, as well as to demonstrate that the testbed can support the construction of a wide variety of CM policies, this chapter discusses the mapping of ten example policies onto the abstraction layer. For each policy, the storage requirements are mapped onto the repository model, while the essence of the procedures through which the stored artifacts are manipulated is mapped onto the programmatic interface.

Different examples highlight different aspects of the mapping process. The focus of the first four example CM policies is on the versioning aspects of the abstraction layer. These policies, although primarily used to manage source code, are in widespread use in current CM systems and are representative of the breadth of CM policies that have been developed to date. The next four example CM policies illustrate how the isolation of distribution within the abstraction layer can be exploited to model a variety of distributed CM policies. Once again, representative, existing CM systems are used as the basis for these examples. The last two example CM policies are novel, since no existing CM system provides their kind of functionality. In addition to the three CM policies that are presented in Chapter 7, these two policies are provided to demonstrate the ability of the testbed to be applied outside of the traditional CM domain.

The programming language Tcl [Ous94] is used as the specification language in the examples. It should be noted, however, that none of the policies presented in this

section is complete. The goal of the examples is to demonstrate how the functions in the programmatic interface can be combined into useful pieces of functionality, rather than to demonstrate how fully functional policies can be built. Therefore, error handling and irrelevant details are omitted from the presented procedures. The procedures should only be viewed as pseudo code, not as fully functional programs.

5.1 Modeling Versioning Aspects of Traditional CM Policies

Of the many CM systems that have been developed to date, only a handful employ exactly the same policy. The others are all unique in the types of procedures they use to manipulate and create versioned artifacts. However, several core policies have been identified that lie at the heart of most CM systems. These core policies are checkout/checkin, composition, long transaction, and change set [Fei91a]. Actual policies that are used in CM systems are either exact copies or minor variants of these core policies. Below, each core policy is mapped onto the abstraction layer and discussed in detail.

5.1.1 Checkout/Checkin

The first two CM systems, SCCS [Roc75] and RCS [Tic85], pioneered the checkout/checkin policy. It has since been the basis for numerous CM systems, including DSCS [Mil97], Gradient [BKR96], Ode [ABGS91], Sablime [Bel97], ScmEngine [CPT97], and SourceSafe [Mic97]. The policy focuses on providing version support for individual artifacts and is typically based on the combined use of a repository and a file system. Versions of artifacts are stored in the repository, but users do not have direct access to those versions. Instead, they have to **check out** a particular version of an artifact to the file system before being able to access its contents. Artifacts can be checked out for read or write access. Read access simply provides a user with the ability to examine the contents of the artifact. Write access allows a user to actually change the contents

of the artifact. Once the appropriate changes have been made, the user **checks in** the artifact to create and store a new version in the repository.

Typically, a user who checks out a version of an artifact for writing also locks that version in the repository. While other users can check out the same version of the artifact for reading, they cannot check it out for writing as long as it remains locked. This assures that a particular version of an artifact includes all the changes that were made in previous versions. Specifically, new versions cannot, by accident, not include changes, since concurrent work is prohibited.

If it is so desired that concurrent development does take place, branches can be used to allow two or more users to change the same version of an artifact at the same time. The first change is checked in normally, but any subsequent checkin from a branch needs to be merged with the latest version of the artifact, ensuring that all changes are included in the main branch. Depending on the availability of an appropriate merge algorithm (e.g., one that understands the type of artifact being versioned), changes can be propagated among branches by checking out, for writing, a version on one branch, applying a merge algorithm to incorporate changes from another branch, and checking in the resulting artifact. Although the creation of merge algorithms is outside the scope of this dissertation, the checkout/checkin policy does track the fact that a merge has occurred.

Branches serve two other important roles that are of a more permanent nature. First, they are used to represent the start of an independent line of development, such as a line for maintenance or a line for a customer that receives a specialized version of a product. Second, they are used to represent variants of an artifact, such as a variant for Windows and a variant for Unix. In both cases, the branch evolves separately from the main branch. Once again, only if an appropriate merge algorithm is available can changes from one branch be propagated to another branch.

The set of relationships that is created by all the revisions, variants, lines of devel-

opment, and merges is traditionally captured as predecessor and successor relationships in a structure that is called the version tree.[1] Because each edge in the tree is labeled with a set of comments that describe the changes made, the version tree provides a simple model that conveniently aids a user in understanding the variety of artifact versions that are available.

5.1.1.1 Repository Design

Figure 5.1a illustrates one possible mapping of the checkout/checkin policy onto the repository model. Two types of artifacts are stored in the repository, namely **content** artifacts and **version tree** artifacts. Each content artifact has an associated version tree artifact that maintains the version relationships resulting from checking out and checking in the content artifact. Content artifacts are versioned to capture the various revisions and variants that are created over time. Version tree artifacts, on the other hand, only exist in a single version, since they already capture the version relationships of their associated content artifacts.

Note that, to avoid naming conflicts, two separate name spaces exist within the repository. These name spaces are created through the use of two collections that are not versioned, one called `Trees` and one called `Artifacts`. The collection `Trees` has as its members the only version of each version tree artifact and the collection `Artifacts` has as its members the first version of each content artifact. Because the sole purpose of the collection `Artifacts` is to create a separate name space, this scheme suffices and there is no reason to advance or update the specific versions of the content artifacts that are its members.

The schema presented in Figure 5.1a is, by itself, not sufficient to support the full functionality of the checkout/checkin policy. An additional set of attributes is needed. Shown in Figure 5.1b for the content artifact `Artifacts/Window.c` and its associated

[1] Although in reality a version graph is formed, the term version tree is historically used.

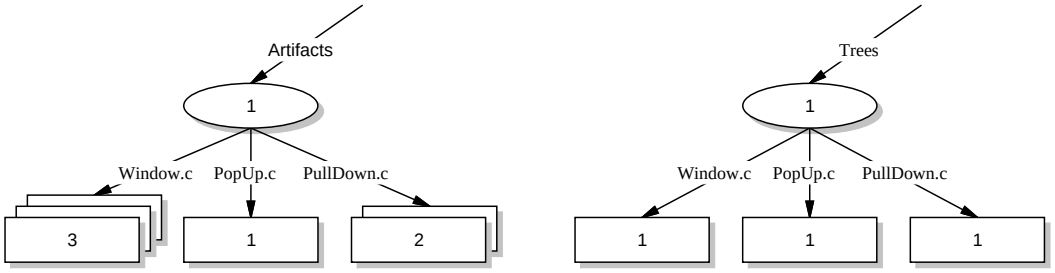
version tree artifact `Trees/Window.c`, these attributes serve four purposes: 1) to relate the version numbers provided by the storage model to the version numbers used by the checkout/checkin policy; 2) to lock individual versions of a content artifact; 3) to capture metadata; and 4) to represent the version tree.

The attribute `Version` performs the first function: attached to each version of a content artifact, it contains a version number that was assigned by the checkout/checkin policy. For example, version 3 of the content artifact `Artifacts/Window.c` has 1.1.1.0 as the value of its attribute `Version`, which means that its contents are presented as version 1.1.1.0 to a user of the checkout/checkin policy.

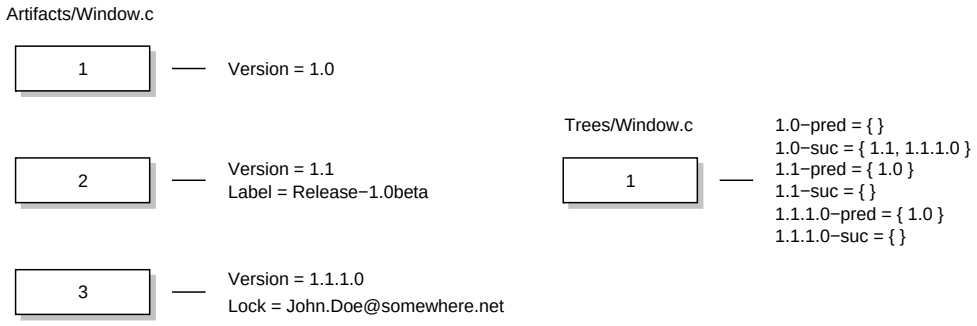
Not only does the attribute `Version` support the conversion of the version numbers used in the storage model to the version numbers used by the policy, the reverse is also supported. In particular, the function `nc_selectversions` can be used for this purpose. Because each version number that is assigned by the checkout/checkin policy is unique, a lookup with the function `nc_selectversions` of a particular version number used by the policy results in a single version number that identifies a specific version of a content artifact in the storage model.

The second and third purposes for which attributes are used are already discussed in previous sections. In particular, locking and the association of metadata in the checkout/checkin policy follow the scheme as laid out in Sections 3.5 and 4.7. Locks are placed on individual versions of a content artifact and identify who owns the lock. Similarly, simple metadata can be associated with each version of a content artifact. In the example of Figure 5.1b, version 3 of the content artifact `Artifacts/Window.c` is locked by the user `John.Doe@somewhere.net` and version 2 is a part of the release 1.0 beta. Note that not all versions of a content artifact have to be part of a release: the attribute `Label` is not present for version 1 and version 3 of the content artifact `Artifacts/Window.c`.

The last purpose of the attributes in the checkout/checkin model is to capture



(a)



(b)

Figure 5.1: Example Repository Structure for the Checkout/Checkin Policy (a), and Typical Attributes Associated with Individual Versions of a Content Artifact and its Version Tree (b).

the predecessor and successor relationships that form the version tree. Two important observations are in place about these attributes. First, unlike the other attributes, they are not attached to the versions of a content artifact. Although that certainly would have been a possibility—with attributes either distributed over the various versions or attached to just its first version—the alternative of associating the attributes with a version tree artifact provides a conceptual separation of concerns and a single location in which all version relations are stored.

The mere existence of the attributes that represent the version tree is the second observation. An alternative design could have represented the version tree as the contents of a version tree artifact instead of as a set of attributes. However, using attributes has the advantage that individual relations can be set and retrieved without opening a version tree artifact in a workspace, making the approach more efficient.

5.1.1.2 Core Policy Design

Figures 5.2 and 5.3 show a mapping of the checkout/checkin policy onto the functions in the programmatic interface. Presented are the two procedures that form the core of the checkout/checkin policy, `checkout` and `checkin`. Based on the repository design of Figure 5.1, these procedures utilize the interface functions to manipulate the various artifacts and attributes into the desired behavior.

The core policy design is based on the use of workspaces. In particular, the procedure `checkout` checks out a particular version of a content artifact into a workspace, making it available for a user to change its contents. The procedure `checkin` complements the procedure `checkout` by storing, as a new version in the repository, the content artifact that has been modified in a workspace. The workspaces as defined by the access model (see Section 3.4) can be used as is, since no special artifact manipulation or presentation is needed to support the checkout/checkin policy.

As implied by its name, the procedure `checkout` checks out a particular version of

a content artifact into a workspace. Its programmatic logic is divided into three parts. The first part determines the values of some useful variables that are used throughout the remainder of the procedure. In particular, the variable `user` is set to the name of the user that employs the procedure; the variable `host` is set to the name of the repository from which artifacts should be retrieved; the variable `artifact` is set to the full name of the content artifact that needs to be checked out; the variable `filename` is set to just the last part of the content artifact name; the variable `wsartifact` is set to the name of the content artifact in the workspace; the variable `storageversion` is, using the function `nc_selectversions`, set to the version of the content artifact in the storage model; and the variable `artifact` is modified by appending a version qualifier to indicate the desired version of the content artifact that needs to be checked out.

By using the function `nc_testandsetattribute`, part two of the procedure attempts to lock the version of the content artifact that needs to be checked out. If the lock is obtained, no further action is taken in this part. If the lock cannot be obtained, the function `nc_getattributevalue` is used to determine which user owns the lock and an appropriate failure message is displayed to inform the user of the procedure.

The final part of the procedure performs the actual checkout. It first opens the desired version of the content artifact in the workspace by using the function `nc_open`. Then, through the function `nc_initiatechange`, it gives the user permission to change the content artifact in the workspace. It concludes by informing the user that the operation has successfully completed. Of note in this part of the procedure is the correspondence between the variable `artifact` and the variable `wsartifact`. Whereas the variable `artifact` designates the content artifact in the repository, the variable `wsartifact` designates the content artifact in the workspace. Because the function `nc_open` preserves the name of an artifact when it opens it in the workspace, the two variables, although different in value, identify the same content artifact.

After a content artifact has been checked out, the user can manipulate it at will.

```
proc checkout { workspace content version }
{
   #
   # Part 1:  set some useful variables.
   #
   set user $env(USER)
   set host $env(REPOSITORYHOME)
   set artifact "//$host/Artifacts/$content"
   set filename [file tail $content]
   set wsartifact "$workspace/$filename"
   set storageversion [lindex [nc_selectversions $artifact "Version" $version] 0]
   set artifact "$artifact:$storageversion"

   #
   # Part 2:  attempt to lock the appropriate artifact version.
   #
   set locked [nc_testandsetattribute $artifact "Lock" $user]
   if { $locked == "false" } {
      set lockuser [nc_getattributevalue $artifact "Lock"]
      if { $lockuser == $user } {
         puts "$artifact $version is already checked out."
         exit
      } else {
         puts "$artifact $version is checked out by $lockuser."
         exit
      }
   }

   #
   # Part 3:  retrieve the locked artifact.
   #
   nc_open $artifact $workspace
   nc_initiatechange $wsartifact
   puts "$artifact $version has been checked out."
}
```

Figure 5.2: Checking Out a Version of a Content Artifact in the Checkout/Checkin Policy.

Once the changes have been finished, though, a new version should be stored in the repository. The procedure `checkin`, shown in Figure 5.3, is used for this purpose. Once again, the procedure is partitioned into a number of parts. Similar to the procedure `checkout`, the first part sets the values of some variables that are used throughout the remainder of the procedure. In particular, the variables `user`, `host`, `filename`, and `wsartifact` are set as in the procedure `checkout`; the variable `oldstorageversion` is, using the function `nc_version`, set to the old version number of the content artifact in the repository; the variable `oldartifact` is set to the name of the old version of the content artifact; the variable `oldversion` is, using the function `nc_getattributevalue`, set to policy version number that the user used to check out the content artifact; and the variable `tree` is set to the full name of the version tree artifact that is associated with the content artifact being checked in. Of note is the variable `oldartifact`, which reflects a name in a workspace to which a version qualifier is attached. When used as a parameter to the function `nc_getattributevalue`, the attribute value is retrieved from the repository, since a workspace does not store attribute values.

The second part of the procedure `checkin` locks the version tree artifact. Note that the procedure repeatedly attempts to lock the version tree artifact if it is already locked. Unlike the locks that are placed by the procedure `checkout`, which may last for a relatively long time, a version tree artifact is only locked to provide for concurrency control while it is being updated during a checkin. Therefore, if a version tree artifact is locked, it suffices to wait and poll until another checkin operation completes and unlocks the version tree artifact.

After the version tree artifact is locked, the actual checkin operation is performed using the function `nc_commitchange`. As a result, a new version of the content artifact is stored in the repository and permission to change the content artifact in the workspace is revoked. The function `nc_commitchange` returns the version number that identifies the new version of the content artifact in the repository. This version number is sub-

```
proc checkin { workspace content }
{
   #
   # Part 1:  set some useful variables.
   #
   set user $env(USER)
   set host $env(REPOSITORYHOME)
   set filename [file tail $content]
   set wsartifact "$workspace/$filename"
   set oldstorageversion [nc_version $wsartifact]
   set oldartifact "$wsartifact:$oldstorageversion"
   set oldversion [nc_getattributevalue $oldartifact "Version"]
   set tree "//$host/Trees/$content"

   #
   # Part 2:  lock the tree to receive exclusive access.
   #
   set treelocked [nc_testandsetattribute $tree "Lock" $user]
   while { $treelocked == "false" } {
      set treelocked [nc_testandsetattribute $tree "Lock" $user]
   }

   #
   # Part 3:  store a new version of the artifact.
   #
   set newstorageversion [nc_commitchange $wsartifact]
   set newartifact "//$host/Artifacts/$content:$newstorageversion"
   set newversion [calcnewversion $oldversion]
   nc_testandsetattribute $newartifact "Version" $newversion

   #
   # Part 4:  update the version tree.
   #
   set successor "$oldversion-suc"
   set successors [nc_getattributevalue $tree $successor]
   lappend successors $newversion
   nc_setattribute $tree $successor $successors
   set newpredecessor "$newversion-pred"
   nc_setattribute $tree $newpredecessor { $oldversion }
   set newsuccessor "$newversion-suc"
   nc_setattribute $tree $newsuccessor {}

   #
   # Part 5:  unlock everything.
   #
   nc_removeattribute $tree "Lock"
   nc_removeattribute $oldartifact "Lock"
   puts "$content $newversion has been checked in."
}
```

Figure 5.3: Checking In a New Version of a Content Artifact in the Checkout/Checkin Policy.

sequently used to create a full name for the new version of the content artifact, as well as to associate a new policy version number as an attribute. Note that the new policy version number is calculated using an external procedure called `calcnewversion`. This procedure advances the policy version number and automatically creates branches using the following algorithm.

```
increment last part of version number by 1
while the version number exists
   append 1.0 to the version number
return version number
```

The version numbers that are calculated by this standard algorithm are unique and allow traceability based on the structure of the version number. Consider, for example, a pre-existing set of versions identified by the numbers 1.0, 1.1, and 1.1.1.0. If version 1.0 is checked out and subsequently checked in, the algorithm iterates over the version numbers 1.1, 1.1.1.0, and 1.1.1.0.1.0 in its calculation of a new version number. Conversely, traceability (in terms of obtaining the parent of a particular version) is achieved by reversing the algorithm: first remove all occurrences of 1.0 at the end of the version number and then subtract 1 from the last part of the resulting version number.

Part four of the procedure modifies the version tree by updating the attributes attached to the version tree artifact. In particular, it updates the successor attribute of the predecessor of the new version, and adds a predecessor and successor attribute for the new version. Note that the function `nc_setattribute` is used in this part of the function. Because the version tree artifact is locked and guarantees exclusive access over time, the use of the function `nc_setattribute` suffices to change some of the existing attributes and to add some new attributes.

The last part of the procedure `checkin` releases the locks that are held. First the version tree artifact is unlocked to allow other users to check in new versions of the content artifact. Subsequently, the appropriate version of the content artifact itself is

unlocked to allow other users to check out that version again in order to make other changes.

5.1.1.3 Variations on the Checkout/Checkin Policy

Several variations of the checkout/checkin policy have been developed over the years. The repository design and associated procedures described above can easily be adapted to support these variations. Below, we briefly describe some of the variations and discuss how the repository design and associated procedures are changed to accommodate each variation.

- **A checkout of a single version of a content artifact locks all versions of the content artifact.**

  Sometimes, exclusive access to all versions of a content artifact is required, without any other user modifying the history of that content artifact at the same time. In such cases, a checkout should lock all versions of the respective content artifact. This policy requires a simple modification to the standard checkout/checkin policy: instead of using multiple instances of the attribute `Lock` (one for each individual version of a content artifact) a single instance is associated with the version tree artifact. Because only one version of the version tree artifact exists, a procedure `checkout` that is modified to attach a lock to the version tree artifact prevents all subsequent invocations from completing. Of course, the lock that is used by the procedure `checkin` of Figure 5.3 to perform concurrency control is now superfluous: the version tree artifact is already locked and the procedure `checkin` does not need to perform any additional locking.

- **The creation of a branch is disallowed.**

  In some policies, it is required that linear evolution takes place [BKR96]. In these policies, branches cannot be created. Two modifications, each leading

to a slightly different behavior, can be made to the standard checkout/checkin policy. In the first, the procedure `checkout` is modified to, a priori, prevent the creation of branches in the procedure `checkin`. With this change, the policy guarantees a linear path of evolution in which each version of a content artifact is based on its immediate prior version. Two modifications are required to attain this behavior. The first modification is based on the recognition that only one change can be made at the same time. The solution to this requirement is provided by the previous variant of the standard checkout/checkin policy: the procedure `checkout` locks all versions of a content artifact by simply placing the lock on the version tree artifact. The second modification obtains the latest version of the content artifact using the function `nc_lastversion`, compares that version against the value of the variable `storageversion`, and only continues the procedure if the values are the same. This modification, executed after the lock on the version tree artifact has been placed, guarantees that the changes to be made are based on the latest version of the content artifact.

The second modification should be made if new versions of a content artifact, even though creating a linear path of storage, can be based on older versions. In this case, the procedure `calcnewversion` needs to be adjusted to return linear version numbers. The simplest way to achieve this behavior is to obtain the value of the attribute `Version` (using the function `nc_getattributevalue`) on the last version of the content artifact (determined by the function `nc_lastversion`) and returning the next version number in the sequence. This change still allows multiple changes to be made concurrently. To avoid such parallelism, the change should be combined with the changes made in the first variation (a checkout of a single version of a content artifact locks all versions of the content artifact).

- **A checkout reserves a revision number.**

  To know up front whether or not a checkout will lead to the creation of a branch, some CM policies employ a scheme in which the procedure `checkout` reserves a revision number [HLRT97]. To modify the standard checkout/checkin policy to attain this behavior, two changes need to be made. The first change concerns the procedure `checkout`, which in effect becomes a mix of the original procedures `checkout` and `checkin`. In particular, the procedure `checkout` needs to lock the version tree artifact for concurrency control, calculate the next version number, update the version tree, and use a sequence of the functions `nc_initiatechange`, `nc_commitchange`, and `nc_initiatechange` to create a new, reserved version of the content artifact in the repository.

  The second change concerns the procedure `checkin`. Because the procedure `checkout` has been modified to perform most of the functions of the original procedure `checkin`, the new procedure `checkin` reduces to using the function `nc_commitchangeandreplace` to store the new version of the content artifact in its pre-allocated slot.

- **Certain branches are represented as a newly named artifact.**

  To avoid the confusion that sometimes arises because branches represent more than one concept (i.e., parallel work, lines of development, and variants), some policies separate these concerns and restrict branches to solely represent parallel work [Sei96]. Any other type of branch is represented in the name space of the repository: a new content artifact is created that, although linked to the original content artifact, evolves separately. To accommodate this change, only the procedure `checkin` needs to be modified. In particular, parts three and four are only executed when regular evolution or parallel work occurs. In other cases, a new content artifact needs to be created by opening the collection `Artifacts`, using

the function `nc_initiatechange` to be able to modify the collection, adding the new content artifact, and using the function `nc_commitchangeandreplace` to store the change in the repository. In a similar fashion a new version tree artifact is created and stored underneath the collection `Trees`. To keep track of the history across content artifacts, appropriate attributes are set on the new content artifact, the new version tree artifact, and the old content artifact.

5.1.2 Composition

The composition policy forms the basis for a large number of well-known CM systems, including Adele [EC94], ClearCase [Atr92], CVS [Ber90], and PVCS [INT98b]. It extends the checkout/checkin policy with a compositional capability that allows individual content artifacts to be grouped into components and components to be grouped into higher-level components. Moreover, just like content artifacts, components can be versioned as well. The resulting graph of versioned components and versioned content artifacts forms a composition that is commonly termed a **system model**. This system model is central to the composition model and provides the basis through which users interact with the repository.

In order to isolate work in progress, users manipulate components in workspaces. These workspaces are populated through the use of version selection rules. Based on the hierarchy of components, these rules typically select a particular version of a component at each level of the hierarchy. Consider, for example, the following set of rules.

```
GUI-lib --> Version == 1.1
GUI-lib/*.c --> Author == John.Doe@somewhere.net
```

These rules specify that a workspace should be populated with version 1.1 of the component `GUI-lib`. By default, the versions of the content artifacts that are contained by version 1.1 of the component `GUI-lib` are placed in the workspace as well, unless the name of a content artifact has the extension ".c". For these content artifacts, the

second selection rule states that it is preferred that the version authored by the user `John.Doe@somewhere.net` is placed in the workspace. If no such version exists, the selection rules revert to the default and populate the workspace with the contained version of the content artifact. It should be noted that the selection rules do not have to address each level of the hierarchy. It is possible, for example, to rely on the default rule for components at the higher levels of the hierarchy while specifying particular versions of components and content artifacts at the lower levels.

The level of sophistication of the selection rules varies within the CM systems that adhere to the composition policy. In some, advanced pattern matching, unification, and compositional capabilities are an integral part of the specification. In others, only one selection rule is used that specifies the version of the top-level component to be included in the workspace. All other artifacts are placed in the workspace based on their (transitive) membership to that version of the top-level component. Despite their differences, all types of selection rules rely on a precise and non-conflicting use of attributes to label particular versions of components and content artifacts. Without proper labeling, ambiguous and incorrect configuration specifications may result and changes may be lost.

To store new versions of components and content artifacts in the repository, a basic checkout/checkin policy is used by the composition policy: before a component or content artifact can be modified in a workspace, it has to be checked out and locked to prevent concurrent modifications. Once an artifact has been modified to satisfaction, a new version of the component or content artifact is stored in the repository. One variable aspect of this scheme is the decision when to store new versions of components. In particular, if contained (versions of) components and content artifacts are changed, some type of rule needs to govern when to create a new version of a component. Most common is to leave this choice up to the user, such that an appropriate aggregate of changes can be checked in as a single modification to a component.

5.1.2.1 Repository Design

Mapping the composition policy onto the repository model requires few modifications with respect to the repository design of the checkout/checkin policy. In fact, many of the unique aspects of the composition policy reside in the complexities of the construction of a workspace, a procedure that is discussed in the next section. The one extension that needs to be made to the mapping presented in Figure 5.1 is the introduction of collections to model components. Similar to the way atoms model content artifacts, these collections are versioned and have an associated version tree artifact. As shown in Figure 5.4a, this leads to the presence of two corresponding hierarchies, one in the name space `Artifacts` and one in the name space `Trees`. The hierarchy in the name space `Artifacts` is versioned to capture the evolution of the components and content artifacts, but the hierarchy in the name space `Trees` is not, since the artifacts in that hierarchy represent version trees that do not need to be versioned.

Attributes serve the same role for the artifacts in the composition policy as they do for the artifacts in the checkout/checkin policy. For both components and content artifacts they are used to: relate version numbers provided by the storage model to the version numbers used by the composition policy; lock individual versions of a component or content artifact; capture metadata; and represent version trees. Of particular importance to the composition policy is their function of capturing metadata. Generally, a specific scheme is set up to which users must adhere when checking in new versions of components or content artifacts. For example, in Figure 5.4b it is illustrated that each version of the component `GUI-lib` has an associated attribute called `Label` that is used in a structured way to identify the release to which the particular version of the component belongs.

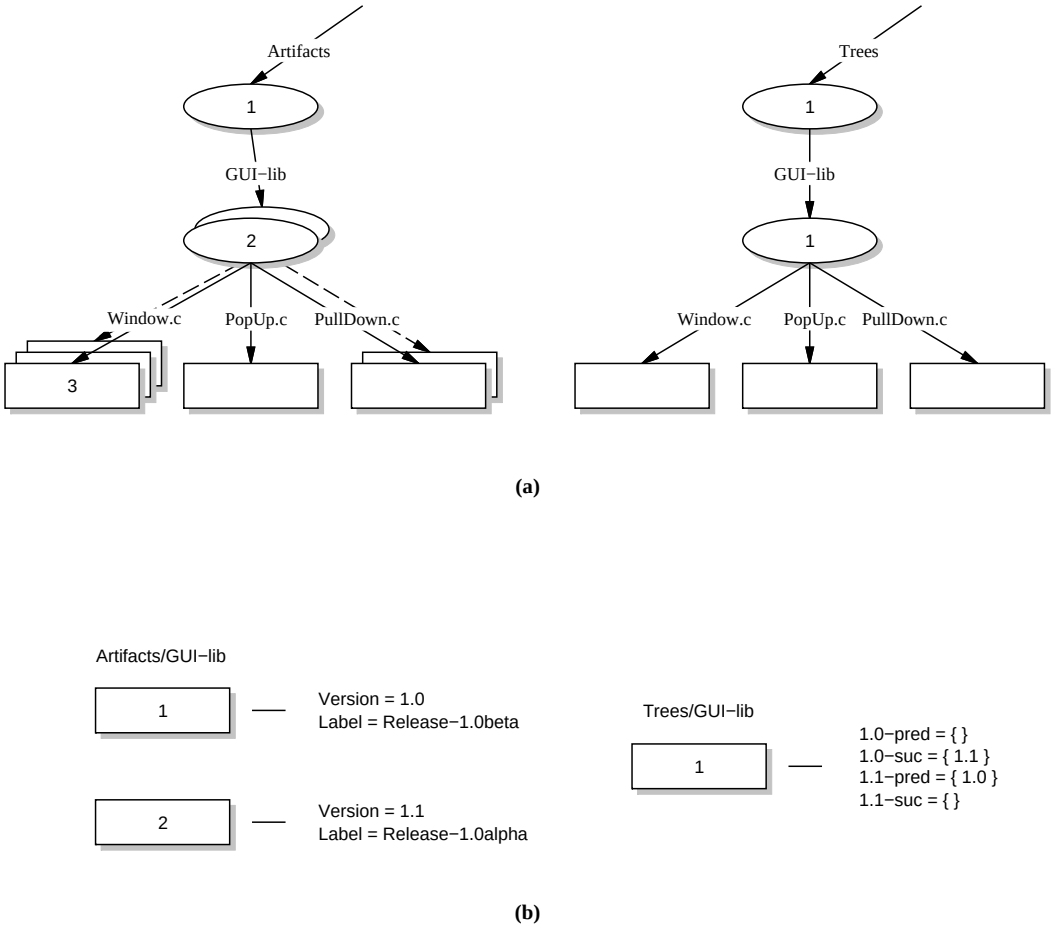


Figure 5.4: Example Repository Structure for the Composition Policy (a), and Typical Attributes Associated with Individual Versions of a Component and its Version Tree (b).

5.1.2.2 Core Policy Design

Based on the repository design of Figure 5.4, Figure 5.5 presents the procedure that forms the core of the composition policy, `populateworkspace`. This procedure has three input parameters: `workspace`, which identifies the location in the file system where the workspace is located; `component`, which determines the component that, in addition to its contained artifacts, needs to be placed in the workspace; and `rules`, which is a set of rules that govern which versions of the component (and contained artifacts) are actually selected to be placed in the workspace. Following the structure of the procedures `checkout` and `checkin` shown in Figures 5.2 and 5.3, respectively, the first part of the procedure `populateworkspace` sets the values of some useful variables for future use. In particular, the variable `host` is set to the name of the repository from which artifacts should be retrieved; the variable `artifact` is set to the name of the component with which the workspace should be populated; the variable `filename` is set to just the last part of the component name; and the variable `wsartifact` is set to the name of the component in the workspace.

Part two of the procedure determines whether the default version of the component needs to be placed in the workspace or whether the rules prescribe a different version than this default. To do so, the procedure `determineversion` is used. In this procedure, the name of an artifact is matched against rules that are of the following generic form.

```
pathname --> attributename == attributevalue
```

If the rules do not contain a pathname that matches the name of the artifact, an empty version is returned to indicate that the default version of the artifact should be used. If the set of rules does contain a matching rule, the function `nc_selectversions` is used to determine the version number of the artifact for which the desired attribute has the associated value. This version number is returned. If none of the artifact versions has

the appropriate attribute value, the function `nc_selectversions` returns an empty set, which is conveniently used by the procedure `determineversion` to indicate that the default version should be opened in the workspace.

Once the version to be opened has been determined, the component is placed in the workspace using the function `nc_open`. After that, the member artifacts of the component are determined using the function `nc_list`. For each of the member artifacts, the procedure `populateworkspace` is then recursively invoked to place the contained artifacts in the workspace as well. Note that the function `nc_list` is applied to the collection as opened in the workspace. This prevents any potential race conditions that might be present if the function is applied, after the collection has been opened in the workspace, to the collection in the repository.

Finally, the procedure concludes by informing the user that the workspace has successfully been populated.

Three observations are in place about the details of the composition policy. First, the procedure `populateworkspace` does not lock any of the artifacts in the workspace. Instead, a simplified version of the procedure `checkout` of the checkout/checkin policy should be used to lock any of the artifacts that are going to be modified. As compared to the original procedure, the simplified version of the procedure does not open the artifact (since it already has been opened in the function `populateworkspace`), but only locks it.

The second observation is that the rules that can be processed by the procedure `populateworkspace` are rather simple. They are based on the equality of a single attribute value. However, this is only a simplification for presentation purposes. More advanced rules can be evaluated as desired by changing the details of the procedure `determineversion`.

The final observation regards the question as to how the membership of a component is updated given that a checkin of a new version of a member artifact does not

```
proc determineversion { artifact component rules }
{
   set i [lsearch -glob $rules "$component*"]
   if { $i == -1 } {
      return {}
   }
   set rule [lindex $rules $i]
   set attrname [lindex $rule 2]
   set attrvalue [lindex $rule 4]
   return [nc_selectversions $artifact $attrname $attrvalue]
}

proc populateworkspace { workspace component rules }
{
   #
   # Part 1:  set some useful variables.
   #
   set host $env(REPOSITORYHOME)
   set artifact "//$host/Artifacts/$component"
   set filename [file tail $component]
   set wsartifact "$workspace/$filename"

   #
   # Part 2:  determine the version to materialize.
   #
   set version [determineversion $artifact $component $rules]
   if { $version != {} } {
      set artifact "$artifact:$version"
   }

   #
   # Part 3:  recursively open the component.
   #
   nc_open $artifact $workspace
   set members [nc_list $wsartifact]
   foreach member $members {
      set subcomponent "$component/$member"
      populateworkspace $workspace $subcomponent $rules
   }

   #
   # Part 4:  done.
   #
   puts "workspace has been populated with component $component."
}
```

Figure 5.5: Populating a Workspace in the Composition Policy.

update the component itself. Therefore, the procedure `checkin` needs to be modified, in case it checks in a component, to first update its membership with the versions of the member artifacts in the workspace before the new version of the component is actually stored to the repository. This allows multiple changes to member artifacts to be grouped into a single change at the component level.

5.1.2.3 Variations on the Composition Policy

Two common variations of the standard composition policy have been developed. The first, recursive propagation, is an attempt to hide the intricacies and complications that arise from versioning a hierarchy of components. The policy automatically creates new versions of components when new versions of their contained artifacts are checked in. Although resulting in a proliferation of the number of component versions at the highest levels in the hierarchy, this policy is used by certain CM systems that manage strictly hierarchical data while relying on linear evolution of the artifacts being versioned (e.g., CoED [BLNP98] or Poem [LR96]). By using an appropriate user interface that hides the details of the versioning policy, these CM systems shield their users from the version explosion problem.

To modify the standard composition policy to provide recursive propagation of changes, the procedure `checkin` needs to be extended. In particular, after the new artifact has been checked in, the procedure needs to recursively move up in the containment hierarchy of components, at each stage opening the latest version of the component using the functions `nc_lastversion` and `nc_open`, updating the membership of the component using the functions `nc_initiatechange` and `nc_replaceversion`, and storing the new version of the component in the repository using the function `nc_commitchange`. Of course, to prevent race conditions, each update of a component should be protected using a simple locking mechanism that is similar to the way the checkout/checkin policy protects the version tree while it is being updated.

The second variant of the standard composition policy is based on an attribute selection mechanism that is not hierarchical in nature and disposes of the guards that are present in each of the rules of the standard composition policy. Instead, the rules solely rely on a pattern matching of attribute values: the attributes associated with each version of an artifact determine the version that is placed in the workspace. Although this policy requires a more stringent use of attributes to properly label artifact versions, it has proven to be more generic than the standard composition policy (which is demonstrated by the flexibility provided by Adele [EC94] or ShapeTools [ML88]).

The plain attribute selection policy can easily be supported by the programmatic logic of the procedure `populateworkspace` of Figure 5.5. The only modification regards the selection process that determines which version of an artifact needs to be opened, i.e., the procedure `determineversion`. First, the procedure needs to be modified to not match pathnames against artifact names, since the selection process is solely based on attributes and their values. Second, the procedure needs to be modified to include a unification process that determines, based on a given set or rules, which (versions of) artifacts are selected to be placed in the workspace. Although at first sight a complicated change, implementations of unification algorithms exist that simply can be used as subroutines by the procedure `determineversion`.

5.1.3 Long Transaction

The long transaction policy was pioneered by NSE [FD90]. In its purest form it has only been used by Vesta [CL93], but two common derivative policies, both of which are discussed in Section 5.1.3.3, are in widespread use. As its name implies, the critical contribution of the long transaction policy is its use of long transactions to encapsulate changes. These long transactions are organized in a hierarchical tree and have versioning capabilities that allow the evolution of artifacts within a long transaction. Artifacts evolve within a long transaction until they have been modified to satisfaction. Only

then are they committed to the long transaction that is the logical parent within the tree of long transactions. This scheme partitions changes among many lower-level long transactions and utilizes higher-level long transactions to aggregate these changes. Of note is the fact that long transactions last a long time as compared to traditional database transactions. In particular, the longevity of the higher-level transactions, which represent an aggregate of lower-level transactions, may be anywhere between a few hours to months at a time.

The hierarchy of long transactions provides a form of isolation in which each long transaction represents a certain stage within the development process. As an example of how this feature of the policy can be used, consider three levels of long transactions. The first and highest level consists of a single long transaction, which is the main repository in which baselines are stored. The second level also consists of a single long transaction, namely the long transaction that is used by a team of developers to collect and group a number of related changes. At the third level, an arbitrary number of development long transactions reside. Each one of those long transactions commits its changes to the team long transaction. Once all desired changes have been collected in the team long transaction, a single change is committed from the team long transaction to the main long transaction in which the change is stored and labeled as a new baseline. This, and other types of hierarchies, are typically created during the use of the long transaction policy.

To support the staging process, the versioning capabilities of a long transaction are essential. In particular, when changes are committed from a child long transaction to its parent long transaction, a new configuration is created in the parent long transaction. This ensures the existence of a precise history of changes. Moreover, it provides users with the ability to undo certain changes by reverting to an older configuration in the parent long transaction. Finally, because users can have private long transactions, the presence of versioning capabilities gives them a checkpointing ability without having to

store the checkpoints in a central storage facility where they are visible to all users.

Just like any other transaction mechanism, rules for resolving conflicts are needed. In the case of the long transaction policy an optimistic approach is taken. It is assumed that changes made to artifacts in different child long transactions do not conflict and that the artifacts can simply be updated in the parent long transaction. If conflicts do occur, a child long transaction is responsible for merging its artifacts with the latest version of the artifacts as stored in the parent long transaction. In this scheme, any long transaction can work on any artifact that is present in its parent long transaction. Moreover, each long transaction does not have to lock any artifacts that it modifies.

It should be noted that, in the same way the composition policy relies on a modified checkout/checkin policy, the long transaction policy uses a modified composition policy to provide parts of its functionality. Selection rules are used by developers to gain access, in a workspace, to those artifacts in the long transaction on which they operate. Furthermore, system models are used to create and version the hierarchy of components that is managed by the long transaction policy.

5.1.3.1 Repository Design

To support the long transaction policy, one possible design represents each long transaction as a separate physical repository. An instance of this design is presented in Figure 5.6. One main long transaction and three child long transactions, all organized in a tree, are shown. The first child long transaction is the team long transaction, which contains a set of baseline artifacts copied from the main long transaction. In the example, no new versions of these artifacts have been stored yet in the team long transaction. Both developer long transactions are also based on the baseline, except that each was originally copied from the team long transaction instead of the main long transaction. Since the creation of both developer long transactions, developer A has modified and stored new versions of only two artifacts, whereas developer B has

modified all artifacts and already stored an intermediate configuration as a checkpoint.

The structure of each physical repository in the long transaction policy is the same as the structure of the repository that supports the composition policy (i.e., the repository structure that is presented in Figure 5.4 is the repository structure used in each of the long transactions shown in Figure 5.6). Although structurally the same, the design of the repository structure for the long transaction policy requires one additional attribute as compared to the design of the repository for the composition policy. This attribute, attached to the top-level artifact in a long transaction, identifies the corresponding artifact in the parent long transaction to which changes are applied when the child long transaction is committed. In essence, the additional attribute builds a bridge between the two, otherwise separate, physical repositories.

Using a multitude of physical repositories instead of a single physical repository to model long transactions has two distinct advantages in supporting the long transaction policy.

- **Clean separation of concerns.**

  Each physical repository represents a particular stage in the development process and only contains the artifacts that are relevant to that stage. If only one physical repository was used to represent all long transactions, that repository needs to track which artifacts belong to which long transaction, hide private "checkpoint" artifacts from other long transactions, and maintain the hierarchy of long transactions as they relate to each other.

- **Load balancing.**

  Instead of using a single physical repository in which all versions of all artifacts are stored, multiple physical repositories are used that each are responsible for managing changes to a part of the versioned graph of artifacts. This scheme distributes the high load that is typically present in case of a single storage

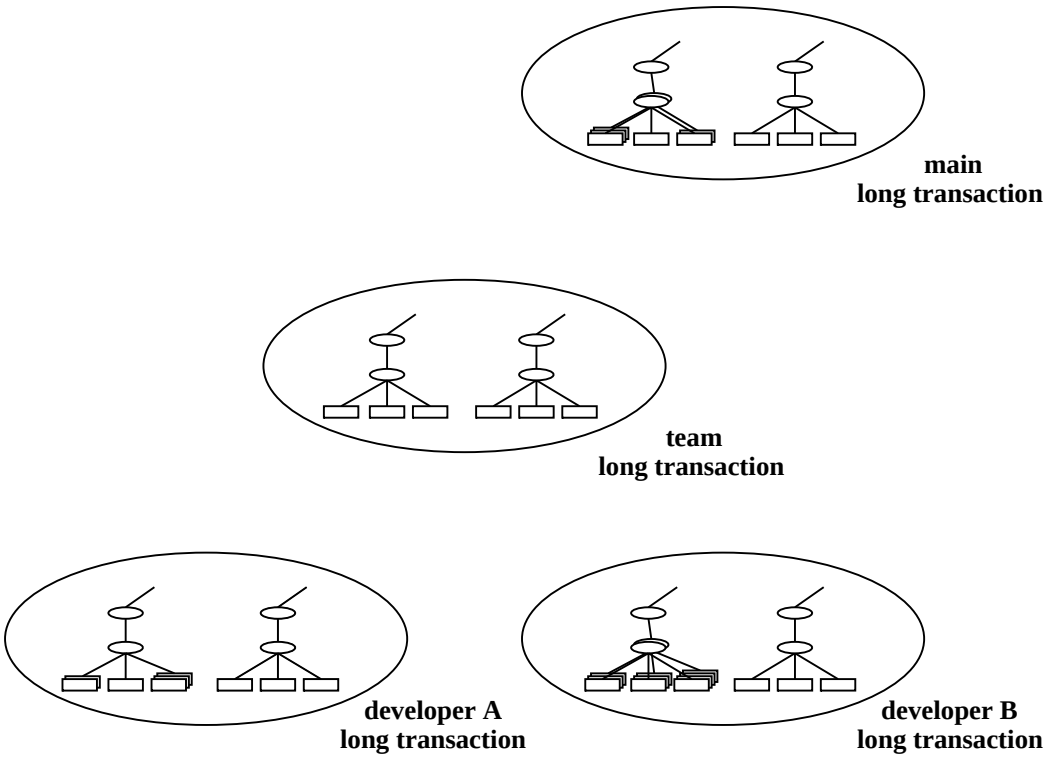


Figure 5.6: Example Repository Structure for the Long Transaction Policy.

point to a multitude of physical repositories.

At the same time, though, the solution is heavy weight. Whereas an implementation that is based on a single repository can take advantage of artifact sharing among long transactions and thus save a lot of storage space, the disadvantage of mapping each long transaction to a physical repository is that it requires copying of artifacts and an considerable amount of extra storage space.

The workspaces as defined by the access model still play a role in the long transaction policy: they are used to gain access to the artifacts that are stored in the physical repositories that represent the long transactions. In particular, both developer A and developer B must use selection rules to gain access, via the file system as defined by the access model, to the artifacts that are stored in a physical repository. The procedure `populateworkspace` as defined in the previous section can be used for this purpose.

5.1.3.2 Core Policy Design

To version artifacts according to the long transaction policy, four procedures are needed. The first, `createlongtransaction`, creates a new child long transaction by deriving it from an existing long transaction. Its details are discussed below. The second and third procedures, `checkout` and `checkin`, respectively, are modified copies of the procedures used by the composition policy. Specifically, both procedures need to be modified to not lock any artifacts, since an optimistic approach underlies the long transaction policy in which it is assumed that conflicts do not occur. The final procedure, `commitlongtransaction`, takes the latest version of the artifacts in a long transaction and updates the parent long transaction with those versions. The details of this procedure are also discussed below.

The programmatic logic of the procedure `createlongtransaction` is illustrated in Figure 5.7. Once again, the first part of the procedure sets the values of some useful variables. First, two workspaces are defined. The first workspace designates the

location in which the artifacts from the parent long transaction are opened, while the other workspace designates the location that is used for the creation of new artifacts that will form the basis for the child long transaction. These workspace are called `parentws` and `childws`, respectively. The variable `parenttransaction` is set to the name of the physical repository in which the parent long transaction resides and the variable `childtransaction` is set to the name of the physical repository that is going to contain the contents of the newly created child long transaction. The last part of the name of the component that forms the basis for the child long transaction is stored in the variable `filename`, and the full name of the component in the parent long transaction is stored in the variable `artifact`.

Part two of the procedure copies the component and its contained artifacts from the physical repository that represents the parent long transaction to the physical repository that represents the child long transaction. To do so, it first uses the function `nc_setmyserver` to make sure that new artifacts are added to the physical repository that represents the child long transaction. Then, the artifacts to be copied are, recursively, placed in the workspace `parentws` by reusing the procedure `populateworkspace` from the composition policy. After that, the artifacts are copied to the workspace `childws` and added to the physical repository that represents the child long transaction by using the procedure `recursivecopyadd`. It should be noted that this procedure physically copies the artifacts from the workspace `parentws` to the workspace `childws`. This form of copying is necessary to remove any connections that the artifacts have with the physical repository in which they are stored. Without the removal of those connections, the use of the function `nc_add` would result in a link to the artifact in the physical repository that contains the parent long transaction rather than the creation of a physically separated artifact that is stored independently.

After the child long transaction has been created, the procedure attaches the attribute `parent` to the copied component to identify the component in the parent

```
proc recursivecopyadd { parentartifact childartifact }
{
   file copy $parentartifact $childartifact
   nc_add $childartifact
   if { [nc_gettype $childartifact] == "collection" } {
      nc_initiatechange $childartifact
      set members [nc_list $childartifact]
      foreach member $members {
         set subparent "$parentartifact/$member"
         set subchild "$childartifact/$member"
         recursivecopyadd $subparent $subchild
      }
      nc_commitchangeandreplace $childartifact
   }
}

proc createlongtransaction { component }
{
   #
   # Part 1:  set some useful variables.
   #
   set parentws "/tmp/parentws"
   set childws "/tmp/childws"
   set parenttransaction "$env(PARENTHOME)"
   set childtransaction "$env(CHILDHOME)"
   set filename [file tail $component]
   set artifact "//$parenttransaction/Artifacts/$component"

   #
   # Part 2:  copy the artifacts.
   #
   nc_setmyserver $childtransaction
   populateworkspace $parentws $artifact {}
   recursivecopyadd "$parentws/$filename" "$childws/$filename
   set newartifact "//$childtransaction/Artifacts/$component"
   nc_setattribute $newartifact "parent" $artifact
   nc_close "$parentws/$filename"
   nc_close "$childws/$filename"

   #
   # Part 3:  done.
   #
   puts "A child long transaction has been created."
}
```

Figure 5.7: Creating a Child Long Transaction in the Long Transaction Policy.

long transaction. Then, both the workspace `parentws` and the workspace `childws` are closed.

The procedure concludes by informing its user that it has completed successfully and that the new child long transaction is available for use.

The procedure `createlongtransaction` as presented in Figure 5.7 is simplified in two ways. First, it assumes that the physical repository that represents the child long transaction has been initialized with the two name spaces, `Artifacts` and `Trees`, respectively. The second simplification regards the creation of the version tree artifacts that are associated with the components and content artifacts that are being copied. Because the creation of a new version tree artifact with a single entry is rather simple, but requires additional workspaces and a slight duplication of code, the appropriate pseudo code is omitted from the figure. Both simplifications are merely omissions for brevity, and their addition to the procedure as shown is straightforward.

Complementary to the functionality of the procedure `createlongtransaction` is the procedure `commitlongtransaction`. This procedure updates a parent long transaction with the latest version of the artifacts that are stored in a child long transaction.

The procedure `commitlongtransaction` is presented in Figure 5.8. The procedure first sets the values of some variables. Two workspaces are defined, namely `childws` and `parentws`. The first workspace is used to materialize the artifacts stored in the physical repository that represents the child long transaction, whereas the second workspace is used to materialize the artifacts stored in the physical repository that represents the parent long transaction. After the workspaces are defined, the variables `childtransaction`, `filename`, and `artifact` are set as in previous procedures. Their values are set to the name of the physical repository that contains the artifacts of the child long transaction, the last part of the name of the component that is being committed, and the full name of the component in the child long transaction, respectively. The variable `artifact` is subsequently modified to contain the full name of the latest

```
proc commitlongtransaction { component }
{
   #
   # Part 1:  set some useful variables.
   #
   set childws "/tmp/childws"
   set parentws "/tmp/parentws"
   set childtransaction "$env(CHILDHOME)"
   set filename [file tail $component]
   set artifact "$childtransaction/Artifacts/$component"
   set version [nc_lastversion $artifact]
   set artifact "$artifact:$version"

   #
   # Part 2:  determine the artifact in the parent transaction.
   #
   set parentartifact [nc_getattributevalue $artifact "parent"]
   set version [nc_lastversion $parentartifact]
   set parentartifact "$parentartifact:$version"

   #
   # Part 3:  commit the workspace.
   #
   nc_setmyserver [nc_getlocation $parentartifact]
   populateworkspace $childws $artifact
   populateworkspace $parentws $parentartifact
   recursivecommit "$childws/$filename" "$parentws/$filename"
   nc_close "$childws/$filename"
   nc_close "$parentws/$filename"

   #
   # Part 4:  done.
   #
   puts "The child long transaction has been committed."
}
```

Figure 5.8: Committing Changes from a Child Long Transaction to a Parent Long Transaction.

version of the artifact in the child long transaction.

The second part of the procedure determines the latest version of the component in the parent long transaction. It does so by first retrieving the value of the attribute `parent`, which contains the full name of the component as it was stored by the procedure `createlongtransaction`. Then the latest version of the component is determined, and the full name of this version is constructed. It is this latest version of the component to which the changes from the child long transaction are applied.

The third part of the procedure is the part that performs the actual update of the physical repository that represent the parent long transaction. Because the child long transaction may have added new artifacts that need to be copied to the parent long transaction, the function `nc_setmyserver` is first used to control the placement of any new artifacts. Then both workspaces are populated, and the procedure `recursivecommit` is used to update the parent long transaction. After that, both workspaces are closed and the user is informed that the update of the parent long transaction has concluded.

The details of the procedure `recursivecommit` are shown in Figure 5.9. The procedure distinguishes collections from atoms. In case a collection is updated, the collection is first initiated for change. Then, the procedure iterates over each of the member artifacts of the collection. If the artifact is open in both the workspace that contains the artifacts from the child long transaction and the workspace that contains the artifacts from the parent long transaction, that artifact may have been changed and the procedure recursively invokes itself to update the artifact.

If the artifact is open only in the workspace that contains the artifacts from the child long transaction, it is a new artifact that has been added by the user of the child long transaction. The new artifact (as well as its member artifacts if the new artifact is a collection) needs to be added to the physical repository that contains the parent long transaction. Depending on the type of artifact, either the procedure `recursiveadd`,

```
proc recursivecommit { childartifact parentartifact }
{
   set version -1
   if { [nc_gettype $childartifact] == "collection" } {
      set changed "false"
      nc_initiatechange $parentartifact
      set members [nc_list $childartifact]
      foreach member $members {
         set subchildartifact "$childartifact/$member"
         set subparentartifact "$parentartifact/$member"
         if { [nc_isopen $subchildartifact] &&
              [nc_isopen $subparentartifact] } {
            set subversion [recursivecommit $subchildartifact $subparentartifact]
            if { $subversion != -1 } {
               nc_replaceversion $subparentartifact $subversion
               set changed "true"
            }
         } elseif { [nc_isopen $subchildartifact] } {
            file copy $subchildartifact $subparentartifact
            if { [nc_gettype $subchildartifact] == "collection" } {
               recursiveadd $subparentartifact
            } else {
               nc_add $subparentartifact
            }
            set changed "true"
         } else {
            nc_remove $subparentartifact
            set changed "true"
         }
      }
      if { $changed == "true" } {
         set version [nc_commitchange $parentartifact]
      } else {
         nc_abortchange $parentartifact
      }
   } else {
      if { [diff $childartifact $parentartifact] != 0] } {
         nc_initiatechange $parentartifact
         file copy $childartifact $parentartifact
         set version [nc_commitchange $parentartifact]
      }
   }
   return $version
}
```

Figure 5.9: Recursively Committing a Collection.

which is not further described, or the interface function `nc_add` is used.

If the artifact is only open in the workspace that contains the artifacts from the parent long transaction, then the artifact has been removed from the collection in the child long transaction. Consequently, the artifact also has to be removed from the collection in the parent long transaction. The function `nc_remove` is used for this purpose. Note that, unlike the addition of new artifacts, the removal does not require recursive processing when operating on a collection. The semantics of the function `nc_remove` is such that once a collection has been removed, its member artifacts are no longer reachable either.

Once the procedure `recursivecommit` has iterated over all members of the collection, it is determined whether the actual membership of the collection has changed. If so, a new version of the collection is stored using the function `nc_commitchange`. Otherwise, the collection is aborted for change.

If the procedure `recursivecommit` operates on an atom, a new version of the atom is stored only if its contents in the workspace of the child long transaction is different than its contents in the workspace of the parent long transaction. The external procedure `diff` is used to determine this. Implementations of this procedure are dependent on the type of contents being versioned. However, at least for textual artifacts, several implementations are readily available.

Three final observations are in place regarding the simplification of the procedures `commitlongtransaction` and `recursivecommit`. First, the procedure `commitlongtransaction` ignores the fact that it may be invoked by multiple child long transactions at the same time. Thus, a simple locking protocol, analogous to the protocol used in the composition policy, should be used to avoid any conflicts. Second, the procedures do not update the version tree artifacts that are associated with the artifacts being managed. Therefore, a full implementation of these procedures needs to include similar version tree manipulations as those used in the procedure `checkin` of the checkout/checkin

policy.

The last observation regards the detection of conflicts. The current procedures assume that no conflicts exist. They simply obtain the latest version of the component in the physical repository of the parent long transaction and apply the changes of the child long transaction. This may lead to problems if that latest version of the component is not the version upon which the child long transaction is based. In particular, it is possible that one long transaction has removed an artifact that the current long transaction has modified. The current routines simply store a new version of this artifact, thereby potentially restoring some problems that were solved by its removal. Unfortunately, automated procedures that resolve these kinds of conflicts are not available, since they are highly dependent on the type of artifact being versioned. Moreover, since certain kinds of conflicts have multiple viable resolutions, users are typically made responsible for resolving the conflicts. Nonetheless, the CM policy should still detect when a conflict occurs. The information necessary to do so is available in the repository model. In particular, the procedure `resursivecommit` can be modified to not only open the latest version of the component being updated, but also the version upon which the child long transaction is based. Depending on the actual differences between these two versions, conflicts can be identified. The example used above can, for example, be detected if the latest version of the component does not contain the artifact, the actual component upon which the child long transaction does include the artifact, and the child long transaction includes the artifact as well.

5.1.3.3 Variations on the Long Transaction Policy

Two common variations of the long transaction policy exist. As opposed to the optimistic approach employed by the standard long transaction policy, the first variation relies on a conservative approach and is based on the explicit use of locking. In particular, each child long transaction workspace locks, in its parent long trans-

action, those artifacts that it intends to change in the child long transaction. This variant can easily be supported by the procedures as described. In fact, the procedure `createlongtransaction` does not have to be changed. Only the procedure `checkout` has to be modified to lock, in the physical repository that represents the parent long transaction, those artifacts that the user intends to change. Repeated use of this modified procedure by various child long transactions results in a partioning scheme in which the hierarchy of artifacts in the parent long transaction is divided among the child long transactions.

Of course, once the artifacts are partitioned and locked, the procedure `commitworkspace` no longer has to perform any conflict detection. It can simply rely on the existence of the locks and update each artifact in the parent long transaction by storing its new version.

The second variation of the long transaction policy is the widely used change package policy (e.g., Continuus [Con94], Perforce [Per98], and NeumaCM+ [Neu98]). This policy relies on a precise and structured labeling of each of the changes that are made such that the versions of the artifacts that are the result of a particular change can be identified with a single name. This name represents the change package. Although strikingly similar to the composition policy, the change package policy provides one additional capability, which is its ability to take a change package from one baseline and merge it into another baseline.

With respect to the procedures that implement the long transaction policy, two changes need to be made to achieve an implementation of the change package policy. The first change is the same change that was previously made: each child long transaction locks in its parent workspace those artifacts that it intends to change by using a modified version of the procedure `checkout`. The second change regards the hierarchy of long transactions. Because the change package policy uses workspaces as in the composition policy, the hierarchy of long transactions that can be constructed by repeatedly em-

ploying the procedure `createlongtransaction` needs to restricted to be just one-level deep. This change requires a simple addition to the procedure `originateworkspace` that uses a standard attribute attached to the collection `Trees` to track the depth of each long transaction in the hierarchy. The needed merge procedure that merges a complete change package into another configuration is a simple routine that iterates over all the members of the change package and applies a standard merge algorithm [Buf95] to each member.

5.1.4 Change Set

The change set policy is an outgrowth of the long transaction policy in which the changes are the central entities. The policy was first implemented in TrueCHANGE (at the time called Aide de Camp [Sof94b]) and EPOS [Mun93]. The critical contribution of the policy is its abandonment of the version tree. As a consequence, the policy does not store complete versions of artifacts. Instead, it stores a set of baselines and a set of changes. A specific configuration is constructed by applying a desired set of changes to a particular baseline. Implementations of the policy are heavily based on automated merge techniques, since a configuration is constructed by merging changes with a baseline.

Underlying the change set policy is the fundamental assumption that the changes that are made by different developers to the same baseline are relatively independent. Shown to be true in practice [Sof94b], this assumption results in a relatively low number of merge conflicts when changes are applied, which is an absolute necessity for the policy to be usable on a day-to-day basis. Still, the frequent creation of new intermediate baselines is a recommended practice to even further reduce the number of merge conflicts.

Because of the assumption of independence of changes, the change set policy is completely free of locks. Developers place a particular baseline in their workspace and

make changes as desired. Upon completion of the changes, the contents of the workspace are committed. However, in contrast to the long transaction policy, which stores new, complete versions of artifacts in the repository, the change set policy calculates and stores the differences between the original artifacts in the workspace and the current artifacts in the workspace.

Even though the creation of merge algorithms is outside the scope of the testbed, the importance of merging in the change set policy warrants that a mapping of the policy onto the abstraction layer makes it easy to insert whatever merge algorithm is desired. An easy integration mechanism that allows different merge algorithms to be inserted for different kinds of artifacts is needed.

5.1.4.1 Repository Design

The mapping of the change set policy onto the repository model is remarkably straightforward. Several reasons can be identified.

- Because change sets are the basis for the policy, they do not themselves need to be versioned.
- Since change sets are logically independent, they can simply be stored under different, unique names.
- Similar to change sets, baselines are the basis for the policy and do not need to be versioned.
- Because baselines are logically independent entities, they can be stored under different, unique names.

This, then, leads to the repository structure that is shown in Figure 5.10. Two name spaces exist, one for baselines and one for change sets. In the name space `baselines`, uniquely named baselines are stored. It is important to realize that, to optimize access,

all artifacts that are part of a baseline are stored in full. An alternative method could have stored just a single baseline that represents the initial state of the repository. All subsequent baselines would then have to be constructed by merging the appropriate change sets each and every time, which is clearly an inefficient solution.

In the name space `csets`, uniquely named change sets are stored. Note that each change set is stored as a single atom, even though it may apply to multiple artifacts. This is due to the fact that change sets are stored in a manner similar to the way patch files are created: they contain sets of differences for a group of artifacts.

5.1.4.2 Core Policy Design

Two procedures implement the essential functionality of the change set policy, `getconfiguration` and `commitchangeset`. The first procedure, `getconfiguration`, is similar in intended functionality to the procedure `populateworkspace` in the composition policy. However, because the repository structure differs quite considerably between the two policies, the details of the procedure `getconfiguration` are rather unique. Shown in Figure 5.11, the procedure has two input parameters, `baseline` and `csets`. The first parameter indicates the baseline to which the change sets that form the list contained in the second parameter need to be applied. Each of the change sets in the list is of the following form.

```
[+|-][change set name]
```

Since each change set is iteratively processed, a series of intermediate configurations is constructed. Depending on whether a change set is preceded by a plus or minus sign, the changes it encapsulates are either added to, or removed from, its preceding intermediate configuration, thereby forming the next intermediate configuration in the series. Once all change sets have been applied, the final configuration is made available to the user.

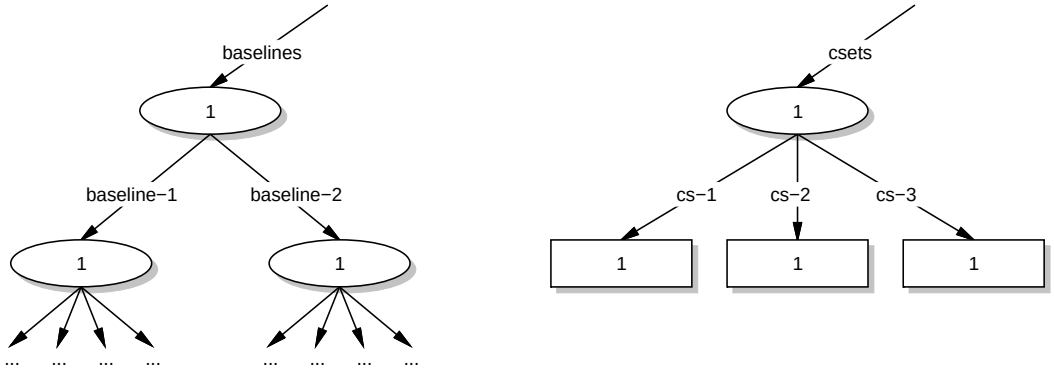


Figure 5.10: Example Repository Structure for the Change Set Policy.

Note that each change set itself also contains additions and subtractions: lines of code as well as artifacts can be added or removed. These changes, however, are encapsulated by the merge algorithm that is applied and are of no further concern of the mapping. It should be noted that preceding, with a minus sign, a change set that removes certain lines of code results in the actual addition of those lines to the intermediate configuration that is being formed.

Both levels of changes are required. Within a change set, the changes captured by the additions and removals define the effects of the application of the change set. At the level of the change set, the plus and minus signs decide whether the changes captured by the change set need to be added to or removed from a baseline. This allows changes that are committed to a baseline to be undone, an important capability in the field of CM.

The procedure `getconfiguration` is structured like any of the other procedures discussed so far. First, the values of a number of variables are defined. The first variable, `baselinews`, is the workspace in which the baseline configuration is opened. The next two variables, `stagingarea` and `finalarea`, are also workspaces, but they are not workspaces such as the ones defined by the abstraction layer. Instead, they are workspaces that are used by the change set policy for two specific purposes: calculating intermediate configurations and providing a user with a changeable copy of the final configuration, respectively.

The next two variables, `host` and `artifact`, are defined as before. They designate the repository in which the baselines and change sets are stored and the full name of the baseline to be opened, respectively.

Part two of the procedure populates the workspace `baselinews` with the desired baseline. To do so, it reuses the procedure `populateworkspace` of the composition policy. After that, a copy of the artifacts in the workspace `baseline` is placed in the staging area so that the change sets can be applied to a fresh copy of the artifacts that

are not restricted by the access model to be solely manipulated through the interface functions. Subsequently, the workspace `baseline` is closed since it is no longer needed in the remainder of the procedure.

Part three of the procedure applies each change set, in order, to the intermediate configuration in the staging area. The procedure `applycset` is used for this purpose. This procedure first opens a change set in its own workspace called `csetws`. Then, based on the type of operation that it needs to perform, it uses yet another procedure, `additivemerge` or `subtractivemerge`, respectively, to apply the change set to the intermediate configuration in the staging area. Because the procedures `additivemerge` and `subtractivemerge` depend on the type of artifact being versioned, and since neither performs any operations that require the repository, their details are omitted here. It should be noted, however, that these two procedures provide the necessary isolation of the merge algorithm that is applied. Specifically, each procedure can be configured to apply a different merge algorithm depending on the type of artifact on which it operates.

The procedure `applycset` concludes by closing the workspace `csetws` after the change set has been applied.

After the last change set has been applied, part four of the procedure `getconfiguration` continues by making a copy of the final intermediate configuration. This copy is essential for future use. In particular, once changes are made and need to be committed to the repository, the procedure `commitchangeset` needs access to the original configuration to calculate a new change set. Instead of reconstructing this original configuration in the procedure `commitchangeset`, an alternative solution is used: the procedure `getconfiguration` simply leaves the intermediate configuration in the staging area where it can be accessed later.

The procedure `getconfiguration` concludes by informing the user that a workspace is available in which the desired configuration has been instantiated.

```
proc applycset { host stagingarea cset }
{
   set csetws "/tmp/csetws"
   set operation [string index $cset 0]
   set csetname [string range $cset 1 end]
   set artifact "//$host/csets/$csetname"
   nc_open $artifact $csetws
   if { $operation == "+" } {
      additivemerge $csetws/$csetname $stagingarea
   } else {
      subtractivemerge $csetws/$csetname $stagingarea
   }
   nc_close $csetws/$csetname
}

proc getconfiguration { baseline csets }
{
   #
   # Part 1:  set some useful variables.
   #
   set baselinews "/tmp/baselinews"
   set stagingarea "/tmp/stagingarea"
   set finalarea "/tmp/finalarea/"
   set host "$env(REPOSITORYHOME)"
   set artifact "//$host/baselines/$baseline"

   #
   # Part 2:  copy the baseline.
   #
   populateworkspace $baselinews $artifact {}
   file copy $baselinews/* $stagingarea
   nc_close $baselinews/$baseline

   #
   # Part 3:  apply the change sets in order.
   #
   foreach cset $csets {
      applycset $host $stagingarea $cset
   }

   #
   # Part 4:  make a separate copy.
   #
   file copy $stagingarea/* $finalarea

   #
   # Part 5:  done.
   #
   puts "The workspace has been populated with the desired baseline."
}
```

Figure 5.11: Populating a Workspace in the Change Set Policy.

The procedure `commitchangeset` is not further detailed here, since its implementation is rather trivial. The procedure uses an external differencing routine to create a new artifact in which the change set is stored and then adds that artifact to the name space `csets` in the repository.

5.1.4.3 Variation on the Change Set Policy

One variation on the change set policy exists. Implemented in the Asgard system [MC96] and called activity-based configuration management, this policy tracks dependencies among change sets. In particular, it operates in such a way that if a change set is included in a configuration, all the change sets upon which it is based are included as well. Consider a change set, `cset-4`, that was created based upon the following configuration.

```
baseline1 + cset-1 + cset-3
```

In addition, consider a new workspace that is created with the following rule.

```
baseline1 + cset-1 + cset-4
```

Asgard ensures that when the new workspace is constructed by iteratively applying the specified change sets to baseline `baseline1`, change set `cset-3`, although not specified, is applied before change set `cset-4` is applied.

This variation is easily accommodated with a few changes to the procedures `getconfiguration` and `commitchangeset`. In fact, only three changes are required. The first change is that the procedure `getconfiguration` has to store the configuration specification in the `stagingarea`. The second change regards the procedure `commitchangeset` which reads the configuration specification from the staging area and attaches it as an attribute to the new change set that is stored. The final change again regards the procedure `getconfiguration`. This procedure needs to be extended with

an additional step before it executes part three. During this step, it reads the configuration specification attribute for each change set, and calculates the order in which the potentially extended group of change sets should be applied.

5.2 Modeling Distributed Aspects of Traditional CM Policies

The testbed is meant to support not only the construction of the versioning aspects of a CM policy, but also the construction of its distribution aspects. This section demonstrates how the isolation of distribution within the abstraction layer aids in the creation of distributed CM policies.

Four canonical distributed CM policies are discussed. Two of those, client-server workspaces and peer-to-peer repositories, rely on a continuous connection among the repositories and workspaces that make up the policy. That is, these policies expect to have available to them a reliable network connection that allows their repositories and workspaces to interoperate at all times. For example, each physical repository in the peer-to-peer repositories policy should be able to request artifacts from other physical repositories at all times.

The other two policies, distributed long transaction and repository replication, support disconnected operation of physical repositories. That is, the physical repositories in which the artifacts are stored only need a network connection among them at certain times. For example, the repository replication policy only needs a network connection when the contents of the repositories are synchronized. At other times, the contents of each repository evolve separately. Note that both the long transaction and replicated repositories policy do require a continuous connection from each workspace to the repository in which the artifacts are stored that are being manipulated in that workspace. It is the particular separation of artifacts among physical repositories that makes these policies operate in a disconnected setting.

Below, each distributed CM policy is mapped onto the repository model and

programmatic interface. Note that the management of the resulting distributed system is ignored in these policies. Typically, this kind of management depends on the type of infrastructure used and is, thus, implementation dependent.

5.2.1 Client-Server Workspaces

The client-server workspaces policy is based on the application of a client-server architecture to a particular CM policy. It was first introduced as an extension to the popular RCS system [Tic85]. The extension, called Distributed RCS (DRCS) [OG90], provides users of the RCS system with the ability to remotely access RCS archives stored at a central server. Since the creation of DRCS, the simplicity with which workspaces can be separated from a central storage facility has made the client-server workspaces policy quite popular with existing CM systems, as evidenced by such distributed CM systems as Distributed CVS [HK92], WWCM [HLRT97], Perforce [Per98], and StarTeam [Sta96].

The critical contribution of the client-server workspaces policy is its departure from the assumption that the repository in which the artifacts are stored is always collocated with its associated workspaces. Instead, the policy allows the physical separation of a central repository (the server) from a multitude of potentially geographically distributed workspaces (the clients). Typically, this separation is achieved through the creation of two collaborating pieces of software: the server that manages the contents of the central repository and the client through which users interact with the CM system. When a user makes requests to the client, the client interacts with the central server to provide the user with access to the artifacts that are stored in the repository.

The client-server workspaces policy can be used in combination with three of the versioning policies discussed in the previous section, namely checkout/checkin, composition, and change set. In such a combination, all (versions of) artifacts, baselines, configurations, change sets, locks, and attributes are stored in a central repository, whereas the user manipulates the artifacts in a workspace that may be located at a

geographically remote site. It is possible to use the client-server workspaces policy in combination with the long transaction policy to allow users to obtain access to artifacts that are managed by a remote long transaction. During the normal use of the long transaction policy, however, each user obtains their own long transaction, that—in a distributed setting—is stored locally. Therefore, the distributed long transaction policy discussed in Section 5.2.3 should be used to create a distributed variant of the long transaction policy.

In the discussion that follows, the composition policy is used as an example of how the client-server workspaces policy can be applied. The conclusions, however, hold for the checkout/checkin and change set policies as well.

5.2.1.1 Repository Design

The repository design presented in Figure 5.4 does not have to be altered to accommodate the physical distribution of the workspaces that is prescribed by the client-server policy. Two reasons can be identified for this rather beneficial fact.

- The client-server workspaces policy relies on the use of a single, central repository.
- Workspaces as defined by the abstraction layer can inherently be located at a different location than the repository in which the artifacts are stored.

As a result, a one-to-one mapping exists between the architecture embedded in the client-server workspaces policy and the actual architecture embedded in the abstraction layer. Consequently, no special provisions have to be included in the repository design of the composition policy to accommodate the client-server workspace policy.

5.2.1.2 Core Policy Design

Similar to the way the client-server workspace policy is intrinsically supported by the repository model of the abstraction layer, the policy is naturally supported by the functions in the programmatic interface as well. Specifically, these functions operate inherently in a distributed setting and interact with a remote physical repository as needed. Therefore, the core policy design of the composition policy does not have to be changed to accommodate the client-server workspaces policy. The pseudo code presented in Figure 5.5 suffices to support the composition policy in operating in a client-server mode.

One observation should be made about the procedure `populateworkspace`. It is parameterized via the environment variable `REPOSITORYHOME`. This environment variable determines the physical repository with which the procedure interacts. Instead of permanently defining that repository inside the procedure, which is a potential alternative, the parameterization makes the procedure more versatile. In particular, a single client can be used to interact with different physical repositories, as long as those repositories use the same versioning policy according to which the artifacts are managed.

5.2.2 Peer-to-Peer Repositories

The peer-to-peer repositories policy is a recent addition to the field of configuration management. It was first employed by ScmEngine [CPT97]. With the exception of DVS [Car98], a CM system that has been implemented using the NUCM prototype implementation of the abstraction layer (see Section 7.1), the policy has not been used since.

Similar to the client-server workspaces policy, the peer-to-peer repositories policy is based on the existence of a multitude of geographically distributed client workspaces that manage the artifacts that are stored in a central server repository. However, con-

trary to the client-server workspaces policy, in which the server repository consists of a single physical repository, the peer-to-peer repositories policy allows for the existence of a logical server repository that consists of multiple, potentially geographically distributed physical repositories. As the name of the policy implies, the physical repositories act as peers to each other and collaborate to provide the workspaces with access to the artifacts that are stored in each of the physical repositories.

A requirement of the peer-to-peer repositories policy is that physical repositories collaborate to provide client workspaces with the illusion that they interact with a single storage location. This requirement of collaborating servers explains why the peer-to-peer repositories policy has hardly been used in current configuration management systems: it leads to fundamental changes at the very core of a CM system, which, because of the typically very tight integration between a CM repository and its associated CM policy, has far-reaching consequences. Therefore, most existing CM systems have resorted to using the client-server workspaces policy as their method of providing distribution.

Similar to the client-server workspaces policy, the peer-to-peer repositories policy can be combined with three of the CM policies discussed in Section 5.1, namely checkout/checkin, composition, and change set. In particular, it is very well suited for (variants of) the composition policy, since the peer-to-peer repositories policy naturally supports the creation of configurations whose member artifacts may span multiple sites. This provides for a partitioning of a set of artifacts among a set of interested parties, thereby allowing each party quick access to the artifacts that are stored in its local repository. The peer-to-peer repositories policy is not applicable to the long transaction policy, since each long transaction is managed by a single physical repository and no need exists to distribute those artifacts over multiple physical repositories.

In the remainder of the discussion the composition policy is used to illustrate the application of the peer-to-peer repositories policy to an existing versioning policy. Once again, however, the conclusions hold for the checkout/checkin and change set policies

as well.

5.2.2.1 Repository Design

When applying the peer-to-peer repositories policy to the composition policy, the repository design presented in Figure 5.4 does not have to be altered to provide client workspaces with the illusion of operating on a single repository. As with the client-server workspaces policy, this is caused by the fact that a one-to-one mapping exists between the architecture embedded in the peer-to-peer repositories policy and the architecture embedded in the abstraction layer. Two reasons can be identified.

- The repository model inherently supports the storage of artifacts in multiple physical repositories.
- Membership relations to collections can span multiple physical repositories.

It should be observed that the peer-to-peer repositories policy utilizes multiple instances of the repository design presented in Figure 5.4. These instances are all active at the same time and collections in one physical repository may contain artifacts in another physical repository.

5.2.2.2 Core Policy Design

In order for the composition policy to operate in a distributed setting and according to the peer-to-peer repositories policy, one procedure needs to be added to the set of procedures that make up the composition policy. This procedure, `createfederation`, builds a connection between two physical repositories. Specifically, it allows the incorporation, in the logical repository with artifacts that are being manipulated, of an artifact from another physical repository. If this physical repository is currently not part of the logical repository, the logical repository is extended. Not only is it extended with the physical repository in which the incorporated artifact resides, but it is also

extended with any other physical repositories that store artifacts transitively accessible from the newly incorporated artifact. If, on the other hand, the physical repository in which the newly incorporated artifact resides is already a part of the logical repository, the set of physical repositories that make up the logical repository remains the same. The connectivity among these repositories, however, is increased.

Figure 5.12 illustrates the details of the procedure `createfederation`. Its behavior is as defined by ScmEngine [CPT97]: the artifact is added to a collection and a new version of that collection is created. The procedure `createfederation` is rather straightforward. The first part sets the values of several variables: `user` is set to the name of the user employing the procedure; `ws` is set to the location in the file system where the artifacts are manipulated; `filename` is set to just the last part of the name of the collection; `artifact` is set to the full name of the collection in the repository; and `wsartifact` is set to the name of the collection in the workspace.

The second part of the procedure borrows from the procedure `checkout` and locks the collection in the repository. The pseudo code is exactly the same as presented in Figure 5.2, and is not further discussed here.

The collection is actually modified in the third part of the procedure. First, it is opened and initiated for change in the workspace. Then, the new artifact is added. After a new version of the collection is stored in the repository, the workspace is closed. It should be noted that the behavior of the function `nc_add` as used in this procedure is different than the behavior that is exploited in the procedure `createlongtransaction` presented in Figure 5.7. A new artifact is added in the procedure `createlongtransaction`, which results in the creation of a new physical artifact in the repository. In the procedure `createfederation`, on the other hand, an existing artifact is added and no new artifact is created in the repository. Instead, a membership relation is put in place that, in this case, crosses the boundary that exists between the physical repository in which the collection is stored and the physical repository in which the artifact is stored. This

```
proc createfederation { myhost collection itshost theartifact }
{
   #
   # Part 1:  set some useful variables.
   #
   set user "$env(USER)"
   set ws "/tmp/workws"
   set filename [file tail $collection]
   set artifact "//$myhost/Artifacts/$collection"
   set wsartifact "$ws/$filename"

   #
   # Part 2:  lock the collection.
   #
   set locked [nc_testandsetattribute $artifact "Lock" $user]
   if { $locked == "false" } {
      set lockuser [nc_getattributevalue $artifact "Lock"]
      if { $lockuser == $user } {
         puts "$artifact $version is already checked out."
         exit
      } else {
         puts "$artifact $version is checked out by $lockuser."
         exit
      }
   }

   #
   # Part 3:  add the artifact to the collection.
   #
   nc_open $artifact $ws
   nc_initiatechange $wsartifact
   nc_add //$itshost/theartifact
   nc_commitchange $wsartifact
   nc_close $wsartifact

   #
   # Part 4:  unlock the collection and inform the user.
   #
   nc_removeattribute $artifact "Lock"
   puts "$theartifact has been added to $collection."
}
```

Figure 5.12: Forming a Logical Repository in the Peer-to-Peer Repositories Policy.

part of the procedure creates the logical repository that is formed by combining the contents of the two participating physical repositories.

The last part of the procedure `createfederation` concludes by unlocking the collection and informing the user that the procedure has finished successfully.

Note that the procedure ignores any modifications that need to be made to the version tree artifact that is associated to the collection that is being changed. Since these modifications are exactly the same as those used by the procedure `checkin` (as presented in Figure 5.3), they are omitted.

An important distinction exists between the core policy design of the client-server workspaces policy and the core policy design of the peer-to-peer repositories policy. Although the pseudo code for the client-server workspaces policy is parameterized, such that it can operate on artifacts stored in different physical repositories, a workspace can only interact with a single physical repository at a time. Therefore, even though a workspace may contain artifacts that stem from different physical repositories, these repositories cannot be integrated into a single logical repository and each physical repository remains independent. Workspaces in the peer-to-peer repositories policy, on the other hand, are not only capable of simultaneously containing artifacts that stem from multiple physical repositories, but can also interact with multiple physical repositories at the same time. Therefore, logical repositories can be formed and artifacts can be related across the physical repositories that form these logical repositories.

5.2.3 Distributed Long Transaction

The distributed long transaction policy is, as its name implies, specifically targeted at making the long transaction policy operate in a distributed setting. Pioneered by NSE [FD90], the distributed long transaction policy has been rarely used since, most notably in PCMS [SQL98]. Compared to the client-server workspaces and peer-to-peer repositories policies, the distributed long transaction is unique in that it facilitates dis-

connected operation. In particular, different long transactions may reside in different locations that are only intermittently connected. The key to this limited connectivity requirement is provided by the fact that the distributed long transaction policy maintains a repository with long transactions in each location. Thus, only when a child long transaction is created that resides at a different location than its parent long transaction, or when a parent long transaction is updated with changes from a child long transaction that resides at a different location, connectivity is required between the repositories residing at each of these locations. At other times, when a long transaction evolves through user updates, no connectivity is required among the various long transactions. Of course, connectivity between a physical repository that contains the artifacts of a long transaction and the workspaces through which the long transaction is advanced, is still needed at all times.

Not surprisingly, the distributed long transaction policy utilizes the same basic principles as the long transaction policy. Child long transactions are created based on existing long transactions, long transactions have versioning capabilities, and new configurations are committed from child long transactions to parent long transactions. Except for the distribution that may exist among different long transactions, the only other difference between the long transaction policy and the distributed long transaction policy is the mechanism used for concurrency control. The long transaction policy can be used in two variants: one that is based on an optimistic, merging-based protocol and one that is based on a pessimistic, locking-based protocol. Since the goal of the distributed long transaction policy is to be able to operate even when long transactions are not connected, only the optimistic, merging-based protocol is supported. The pessimistic, locking-based protocol cannot be supported since it requires a continuous connectivity among long transactions in order to be able to lock artifacts in their parent long transactions.

5.2.3.1 Repository Design

To support the distributed long transaction policy, the same repository design that is used for the long transaction policy can be used. In fact, the repository design that is described in Section 5.1.3.1 and illustrated in Figure 5.6 can be used as is. Key to this reusability are two factors.

- The repository design for the long transaction policy stores each long transaction in a separate physical repository, thereby providing the distributed long transaction policy with the ability to divide the long transactions over various geographically distributed sites.
- Each long transaction in the distributed long transaction policy is self-contained and only needs to interact with other long transactions when changes are committed at the end of a transaction.

Combined with the fact that the abstraction layer isolates the details of low-level distribution, these two important design characteristics allow for the exact reuse of the repository model of the long transaction policy in a distributed setting.

Similar to the long transaction policy, the repository design for the distributed long transaction policy is still a rather heavy-weight solution. Storing each long transaction in a different physical repository typically leads to the creation of more physical repositories than the actual number of physical locations participating in the development effort. Nonetheless, the benefits of a clean separation of concern and a distribution of workload remain.

5.2.3.2 Core Policy Design

Similar to the way the repository design of the long transaction policy is able to support the distributed long transaction policy without any modifications, the core policy design of the long transaction policy can also be reused as is. Because the

design of the repository already partitions different long transactions in different physical repositories, and because the interface functions in the programmatic interface inherently operate in a distributed setting, no changes are necessary to the procedures `createlongtransaction` and `commitlongtransaction` as described in Section 5.1.3.2.

5.2.4 Repository Replication

The final distributed CM policy is called repository replication. It was first used in the domain of configuration management by Adele [EC94], which was enhanced with a tool called Mistral [Gad95], and ClearCase [Atr92], which was enhanced with a tool called MultiSite [AFK+95]. Both enhancements provide users with the ability to keep multiple, replicated repositories synchronized. Second in popularity only to the client-server workspaces policy, the repository replication policy has been used extensively in existing CM systems, as evidenced by Continuus DCM [Con98], PVCS SiteSync [INT98a], NeumaCM+ MultiSite [Neu98], Gradient [BKR96], and DSCS [Mil97].

Similar to the distributed long transaction policy, the repository replication policy supports disconnected operation in that each of the participating repositories has to be connected to the other repositories only at certain times. However, different principles underlie the repository replication policy. Specifically, its key feature is that all versions of all artifacts are independently available in all of the participating repositories. Each repository operates in a stand-alone manner and evolves its artifacts without interaction with the other repositories. Periodically, however, the contents of all repositories are synchronized by propagating new versions of artifacts created in each of the repositories to all of the other participating repositories.

The synchronization process is based on the assumption that the changes that are made in each of the replicated repositories are non-conflicting. Nonetheless, conflicts do occur. To deal with those conflicts, a three-part solution is adopted by the repository replication policy.

(1) Each artifact is assigned a master repository.

(2) Each repository makes its changes on a logically separate branch of artifacts.

(3) All branches are represented in all of the participating repositories.

The synchronization process, then, consists of two separate steps. In the first, each of the branches is synchronized throughout the set of replicated repositories. Because each repository is represented by a unique branch in the other repositories, all this step requires is a simple copying of the new (versions of) artifacts. The second step takes care of resolving potential conflicts. If no conflicts exist, each repository simply copies the changes from each branch to its main branch. If a conflict exists, however, the master repository of the conflicting artifact is responsible for resolving the conflict and rebroadcasting the resulting artifact to each of the replicas. After these two steps have completed, the contents of all replicated repositories are, once again, exactly the same.

To avoid the occurrence of conflicts as much as possible, it is recommended that an appropriate partitioning of the artifacts is created among the users of the various replicas. Users of a particular replica should be responsible for updating and advancing a particular and unique set of artifacts. In addition, it is recommended that synchronizations take place often to help minimize the number of conflicts, as well as to reduce the effort required to resolve the conflicts.

The repository replication policy can be used with either the checkout/checkin or the composition policy. Below, the checkout/checkin policy is used to illustrate the application of the replicated repositories policy to an existing versioning policy.

5.2.4.1 Repository Design

To support the repository replication policy as applied to the checkout/checkin policy, all of the physical repositories in the federation of replicated repositories have the same structure. Specifically, they all have the structure that is presented in Figure 5.13.

This structure is an extension to the checkout/checkin repository design presented in Figure 5.1. It inserts another level in the hierarchy of artifacts. This level partitions each of the participating repositories into several separate name spaces, each of which is responsible for representing the branch of one of the participating replicated repositories. An alternative design could have used a single name space and represented the branches within the regular version trees instead. However, a desire for a clean separation of concerns governs that the solution of separate name spaces is favorable.

As an example, consider a replication architecture with two physical repositories: one in Boulder and one in Rotterdam. Both physical repositories adhere to the same structure, namely the one presented in Figure 5.13. Because of the disconnected operation of the policy, users of the physical repository in Rotterdam locally advance the artifacts in the collection `Rotterdam/Artifacts` and users of the physical repository in Boulder locally advance the artifacts in the collection `Boulder/Artifacts`. During this process, the replicas become out of date.

When the repositories are synchronized, the first step is to propagate the changes from the collection `Rotterdam/Artifacts` located in the physical repository of Rotterdam to the collection `Rotterdam/Artifacts` located in the physical repository of Boulder. Similarly, the collection `Boulder/Artifacts` in Rotterdam is updated with the changes from the collection `Boulder/Artifacts` in Boulder. Once this update has been performed, and the version tree artifacts have been updated in a similar fashion, the contents of all physical repositories are the same. A final step, however, propagates the changes from the various branches within each repository to create an integrated view at the main branch.

5.2.4.2 Core Policy Design

To synchronize the contents of a series of replicated repositories, two procedures are needed. The first procedure, `syncreplica`, updates the contents of a branch at a

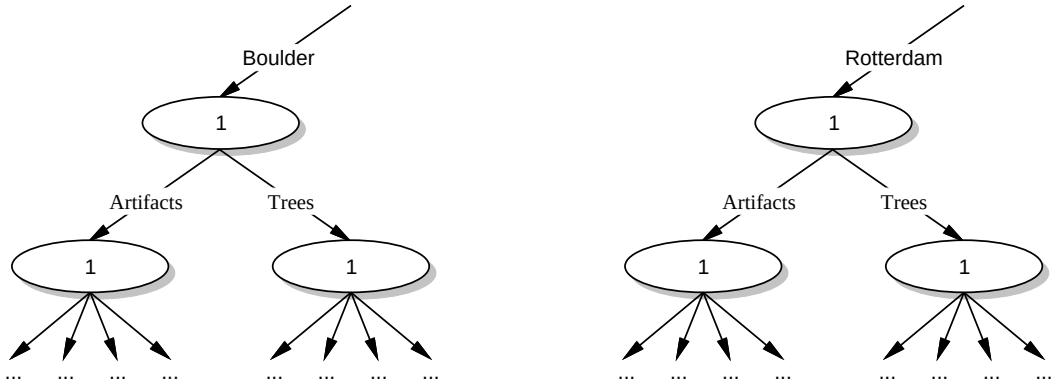


Figure 5.13: Example Repository Structure for Each of the Replicas in the Repository Replication Policy.

particular replica with the changes that have been made at the original repository that "owns" the branch. Although the procedure updates only a single branch in a single replica, when used in a simple nested iteration over all branches within all participating repositories, the contents of all branches in all replicated repositories will be identical. The second procedure takes care of merging the contents of the various branches. The core of this procedure can be reused from the long transaction policy. Specifically, repeatedly applying a modified version of the procedure `commitlongtransaction` (as presented in Figure 5.8) to each of the branches that are stored within a single replica, eventually updates the main branch of that replica. By applying the same kind of iteration to each of the replicated repositories, all changes are merged into all main branches. After this step, the contents of all repositories are synchronized and normal operations may resume.

Since the procedure `commitlongtransaction` is discussed in Section 5.8, the remainder of this section focuses on the details of the procedure `syncreplica`. Shown in Figure 5.14, this procedure is similar in structure to the procedure `recursivecommit` shown in Figure 5.9. First, the values of some variables are set. In particular, `sourcews` is set to the workspace in which the artifacts from the original repository are opened; `destinationws` is set to the workspace in which artifacts from the replica are opened; `sourcecollection` is set to the full name of the collection `Artifacts` in the original physical repository; `destinationcollection` is set to the full name of the collection `Artifacts` in the physical repository that contains the replica; `sourceartifact` is set to the name of the collection `Artifacts` in the workspace `sourcews`; and `destination-artifact` is set to the name of the collection `Artifacts` in the workspace `destination-ws`.

To start the synchronization process, each workspace is populated with the artifacts from their respective physical repository. After that, the complete set of artifacts in the original repository is determined using the function `nc_list`, and each artifact is

individually synchronized.

To synchronize an artifact, it is first determined whether the artifact has been opened in both the workspace `sourcews` and the workspace `destinationws`. If that is the case, only the new versions of the artifact need to be copied to the replica. The procedure `addversions` is used for this purpose. After the range of versions that need to be copied has been determined, this procedure is used to update, version-by-version, the replica.

If the artifact is only open in the workspace `sourcews`, it represents an artifact that has been added to the repository that contains the original copy, but that has not been propagated yet to the physical repository that contains the replica. In this case, the procedure `syncreplica` copies the first version of the artifact to the workspace `destinationws`, then adds that version as a new artifact to the physical repository that contains the replica, and concludes by reusing the procedure `addversions` to add the remaining versions.

Finally, if the artifact is only open in the workspace `destinationws`, it represents an artifact that has been removed from the original physical repository. To synchronize, the procedure `syncreplica` simply removes the artifact from the collection `Artifacts` in the workspace `destinationws`. As a result, the artifact is not only removed from that collection, but also from the physical repository that contains the replica. This last removal is triggered by the storage reclamation mechanism of the abstraction layer: once none of the versions of an artifact are contained, the storage space for all of those versions is reclaimed.

Although straightforward, several important observations need to be made about the repository replication policy and its use of the procedure `syncreplica`.

- **The procedure is independent of the relations among the versions of the artifacts.**

```
proc addversions { startversion endversion sourceartifact destinationartifact sourcews }
{
   set version $startversion
   while { $version <= $endversion } {
      nc_initiatechange $destinationartifact
      nc_close $sourceartifact
      nc_open "$sourceartifact:$version" $sourcews
      file copy $sourceartifact $destinationartifact
      nc_commitchange $destinationartifact
      set version [expr $version + 1]
   }
}

proc syncreplica { destinationhost sourcehost }
{
   set sourcews "/tmp/sourcews"
   set destinationws "/tmp/destinationws"
   set sourcecollection "//$sourcehost/Artifacts"
   set destinationcollection "//$destinationhost/Artifacts"
   set sourceartifact "$sourcews/Artifacts"
   set destinationartifact "$destinationws/Artifacts"
   populateworkspace $sourcews $sourcecollection
   populateworkspace $destinationws $destinationcollection
   set members [nc_list $sourceartifact]
   foreach member $members {
      set subsourceartifact "$sourceartifact/$member"
      set subdestinationartifact "$destinationartifact/$member"
      if { [nc_isopen $subsourceartifact] &&
           [nc_isopen $subdestinationartifact] } {
         set lastsourceversion [nc_lastversion $subsourceartifact]
         set lastdestinationversion [nc_lastversion $subdestinationartifact]
         if { $lastsourceversion > $lastdestinationversion } {
            set startversion [expr $lastdestinationversion + 1]
            addversions $startversion $lastsourceversion $subsourceartifact \
                        $subdestinationartifact $sourcews
         }
      } elseif { [nc_isopen $subsourceartifact] } {
         nc_initiatechange $destinationartifact
         nc_close $subsourceartifact
         nc_open "$subsourceartifact:1" $sourcews
         file copy $subsourceartifact $subdestinationartifact
         nc_setmyserver $destinationhost
         nc_add $subdestinationartifact
         nc_commitchangeandreplace $destinationartifact
         set startversion 2
         set lastversion [nc_lastversion $subsourceartifact]
         addversions $startversion $lastversion $subsourceartifact \
                     $subdestinationartifact $sourcews
      } else {
         nc_initiatechange $destinationartifact
         nc_remove "$subdestinationartifact"
         nc_commitchangeandreplace $destinationartifact
      }
   }
}
```

Figure 5.14: Synchronizing a Branch in a Replica.

In particular, the various artifact versions are copied without the need to pay attention to the predecessor relationships that exist among them. All that matters is the fact that version numbers are preserved during the copying from the original to the replica physical repository. This simplification is a benefit that results from the separation of the storage of the artifact from the storage of its relations in an independent version tree artifact.

- **The procedure could have been written using the function `nc_copy`.**

  Instead of incrementally updating a replica, the function could have been simplified considerably by changing its logic to first remove all artifacts from the collection `Artifacts` in the physical repository that contains the replica, and then using the function `nc_copy` to copy the version history of each artifact from the original physical repository to the replica. Although clearly inefficient, the simplicity of this variant of the procedure can be beneficial if rapid prototyping is needed.

- **The procedure is simplified in that it ignores the synchronization of the attributes and version trees.**

  This is not a problem, since synchronization of attributes and version tree artifacts can easily be added. Attributes are synchronized by using a series of invocations to the functions `nc_getattributevalue` and `nc_setattribute`, and version tree artifacts are synchronized by using a series of invocations to the functions `nc_remove` and `nc_copy`. As opposed to the actual artifacts, the use of the strategy of removing and copying version tree artifacts is reasonable since only a single version of each version tree artifact exists and the contents of that version tends to be small in size.

- **The procedure `syncreplica` is often used by itself for the purpose of**

vendor-code management.

In particular, the procedure can be used to control and manage the evolution of source code libraries that, although an integral part of a system being created, are authored and released by external vendors. Because it is typical that the receiving party does not modify the vendor code, this result in a simple unidirectional replication scenario for which only the procedure `syncreplica` is needed.

- **To apply the replicated repositories policy to the change set policy, a much simpler strategy can be used.**

  Because change sets are independent, they can be propagated among repositories without a potential source of conflict. The only requirement is a naming scheme that uniquely identifies, while avoiding naming conflicts, change sets in different physical repositories. Prefixing each change set with the name of the repository in which it is first stored is sufficient for this purpose. When the contents of physical repositories are synchronized, it is guaranteed that no naming conflicts will occur. The procedure `syncreplica`, modified to operate with the repository structure presented in Figure 5.10, can then be used for the replication of baselines and change sets.

5.3 Modeling Non-Traditional CM Policies

In addition to being able to support the creation of existing CM policies, the testbed is capable of supporting the creation of new CM policies. This section introduces two of those new CM policies. Together with the three new CM policies that are presented in Chapter 7, these two new policies illustrate that the breadth and applicability of the testbed reaches beyond that of traditional CM functionality.

Discussed below, each of the two policies modeled in this section illustrates a

unique aspect of the testbed. The first, movement upon checkout, illustrates how the orthogonality embedded in the abstraction layer can be exploited in adjusting the behavior of an existing distributed CM policy. The second, product family architectures, illustrates how the abstraction layer can be applied to support the management of artifacts other than source code.

5.3.1 Movement Upon Checkout

The movement upon checkout policy is a new CM policy, neither previously modeled nor implemented, that solves a potential weakness of the peer-to-peer repositories policy. This weakness regards the placement of a new artifact in a logical repository. A common practice in the peer-to-peer repositories policy is to store a new artifact in the physical repository that is "owned" by the person or organization that first checks in the new artifact. Typically, this physical repository is also the one that is closest in proximity to that person. Although a straightforward solution, the policy may lead to situations that are undesirable. Consider a collaborative scenario between an organization in the Netherlands and an organization in Colorado. The organization in Colorado has recognized the need for a series of new artifacts and has checked in, as a reminder, stubs for those artifacts. In reality, however, the organization in the Netherlands is responsible for actually creating and maintaining the contents of the artifacts. In the peer-to-peer repositories policy, this leads to rather inefficient access to the new artifacts: a large amount of traffic is induced from the client workspaces in the Netherlands to the physical repository in Colorado.

The movement upon checkout policy aims to remedy this situation. Instead of artifacts being permanently stored in the repository to which they were first added, the policy—as its name implies—moves artifacts from physical repository to physical repository to obtain proximity and speedier access. The policy is based on the assumption that a person who checks out an artifact is likely to access versions of that artifact more

often in the near future. Therefore, it physically moves all of the versions of a remote artifact to a local repository when it is checked out by a user. It is expected that, because of the movement of artifacts to locations of closer proximity, the average access time to artifacts will improve over the peer-to-peer repositories policy. Note, however, that the cost of moving an artifact is dependent upon the length of its version history, since moving an artifact does not involve just moving a single version, but all versions.

5.3.1.1 Repository Design

To modify the peer-to-peer repositories policy to attain movement upon checkout, its repository design does not have to be changed. Due to the orthogonality embedded in the abstraction layer, the actual distribution of the artifacts in the peer-to-peer repositories policy has no influence on its versioning and containment mechanisms. Consequently, an artifact can simply be moved to a different physical repository (as long as, of course, that physical repository adheres to the same repository design).

5.3.1.2 Core Policy Design

The core policy design employed by the peer-to-peer repositories policy also undergoes very little change to support the movement upon checkout policy. In fact, only one procedure, `checkout`, has to be changed to attain movement of artifacts. This procedure, presented in Figure 5.2, needs to be modified into the procedure `movingcheckout` shown in Figure 5.15. As compared to the original version of the procedure `checkout`, the new procedure has only three additional lines of code. The first additional line identifies the version tree artifact that corresponds to the artifact being checked out. The second and third additional lines, respectively, move the artifact and its associated version tree artifact to the repository that is identified by the variable `host`.

Two observations are in place about this modification. First, it should be noted that the procedure contains no explicit reference to the physical repository in which the

```
proc movingcheckout { workspace content version }
{
   #
   # Part 1:  set some useful variables.
   #
   set user $env(USER)
   set host $env(REPOSITORYHOME)
   set artifact "//$host/Artifacts/$content"
   set tree "//$host/Trees/$content"
   set filename [file tail $content]
   set wsartifact "$workspace/$filename"
   set storageversion [lindex [nc_selectversions $artifact "Version" $version] 0]
   set artifact "$artifact:$storageversion"

   #
   # Part 2:  attempt to lock the appropriate artifact version.
   #
   set locked [nc_testandsetattribute $artifact "Lock" $user]
   if { $locked == "false" } {
      set lockuser [nc_getattributevalue $artifact "Lock"]
      if { $lockuser == $user } {
         puts "$artifact $version is already checked out."
         exit
      } else {
         puts "$artifact $version is checked out by $lockuser."
         exit
      }
   }

   #
   # Part 3:  retrieve the locked artifact.
   #
   nc_open $artifact $workspace
   nc_initiatechange $wsartifact
   nc_move $artifact $host
   nc_move $tree $host
   puts "$artifact $version has been checked out."
}
```

Figure 5.15: Checking Out an Artifact Version while Moving it to Closer Proximity.

artifact that is being checked out is stored. This omission is not accidental, since the name of an artifact may cross boundaries among physical repositories. Combined with the fact that the function `nc_move` does not move an artifact if it is already located in the physical repository to which it is supposed to be moved, this powerful naming mechanism results in the simplicity of being able to use the function `nc_move` directly on the full name of the artifact.

The second observation is the fact that, of course, both the artifact and its associated version tree artifact have to be moved. They are intrinsically tied together, and each being located in a different location would not have the same beneficial effect as both being collocated.

5.3.1.3 Variations

Two possible variations on the movement upon checkout policy can be created, namely movement upon multiple checkout and user-controlled movement. The first variation is based on the observation that the movement upon checkout policy does not always provide the most efficient behavior. Consider a setting in which two developers, each located at a different geographical location, collaborate and intermittently check out the same artifact. If first developer A checks out the artifact, then developer B, and then developer A again, the artifact and its associated version tree would be copied three times, incurring far more overhead than if the artifact simply had remained in the physical repository in which it was originally stored. The movement upon multiple checkout variant is designed to avoid such extraneous copying. Instead of always copying an artifact upon checkout, the policy only copies the artifact if it has been checked out by the same developer some consecutive number of times. Although not foolproof with respect to other series of invocations that may lead to other kinds of inefficiencies, the policy is more conservative than the movement upon checkout policy and likely will avoid excessive copying of artifacts.

To support the movement upon multiple checkout policy, the only change with respect to the movement upon checkout policy regards the use of an additional attribute that keeps track of the number of times a user has consecutively checked out a particular artifact. Because this counter is independent of the versions of an artifact, it is attached to the version tree artifact that is associated with each artifact. The procedure `movingcheckout` has to be modified accordingly to decide whether or not an artifact is moved when it is checked out. Illustrated in Figure 5.16, the resulting procedure, `movingmultiplecheckout`, contains an extra part, namely part four. This part first retrieves the value of the attribute and splits it into its two parts: the user who last checked out the artifact and the number of times that the user has consecutively checked out the artifact. If the user stored in the counter is the user of the procedure and the counter has a value greater than the trigger value of three, the artifact is moved. If the user is unequal to the user that previously checked out the artifact, the counter is reset. In both cases, the new value, consisting of the name of the user and the number of consecutive times the user has invoked the procedure to check out the same artifact is stored using the function `nc_setattribute`.

The second variation on the movement upon checkout policy completely abandons automation. Instead, it places the responsibility for artifact movement solely in the hands of a user. To do so, the procedure `checkout` should be restored to its original version as discussed in Figure 5.2. In its place is a single new procedure, `recursivemove`, that recursively moves an artifact and any of its potential member artifacts. Shown in Figure 5.17, the second part of the procedure forms its core. In order to avoid the infinite recursion that may occur because of the potential existence of cycles in the versioned graph, it first determines whether the artifact has been moved already. If so, the procedure does nothing. If the artifact has not yet been moved, it and its associated version tree artifact are moved. Then, in case of collections, the procedure recursively moves each of the member artifacts of the collection. It should be noted that this

```
proc movingmultiplecheckout { workspace content version }
{
   #
   # Part 1:  set some useful variables.
   #
   set user $env(USER)
   set host $env(REPOSITORYHOME)
   set artifact "//$host/Artifacts/$content"
   set tree "//$host/Trees/$content"
   set filename [file tail $content]
   set wsartifact "$workspace/$filename"
   set storageversion [lindex [nc_selectversions $artifact "Version" $version] 0]
   set artifact "$artifact:$storageversion"

   #
   # Part 2:  attempt to lock the appropriate artifact version.
   #
   set locked [nc_testandsetattribute $artifact "Lock" $user]
   if { $locked == "false" } {
      set lockuser [nc_getattributevalue $artifact "Lock"]
      if { $lockuser == $user } {
         puts "$artifact $version is already checked out."
         exit
      } else {
         puts "$artifact $version is checked out by $lockuser."
         exit
      }
   }

   #
   # Part 3:  retrieve the locked artifact.
   #
   nc_open $artifact $workspace
   nc_initiatechange $wsartifact

   #
   # Part 4:  decide whether to move the artifact or not.
   #
   set counter [nc_getattributevalue $tree "COUNTER"]
   set counteruser [lindex [split $counter ":"] 0]
   set counternumber [lindex [split $counter ":"] 1]
   set counternumber [expr $counternumber + 1]
   if { "$user" == "$counteruser" && $counter >= 3 } {
      nc_move $artifact $host
      nc_move $tree $host
   } elseif { "$user" != "$counteruser" } {
      set counternumber 1
   }
   nc_setattribute $tree "COUNTER" "$user:$counternumber"
   puts "$artifact $version has been checked out."
}
```

Figure 5.16: Checking Out an Artifact Version while Conditionally Moving it to Closer Proximity.

```
proc recursivemove { component destinationhost }
{
   #
   # Part 1:  set some useful variables.
   #
   set host $env(REPOSITORYHOME)
   set artifact "//$host/Artifacts/$component"
   set tree "//$host/Trees/$component"

   #
   # Part 2:  prevent infinite recursion.
   #
   set hasmoved [nc_getattributevalue $tree "hasmoved"]
   if { "$hasmoved" == "false" } {
      nc_move $artifact $destinationhost
      nc_move $tree $destinationhost
      if { [nc_gettype $artifact] == "collection" } {
         nc_setattribute $tree "hasmoved" "true"
         set members [nc_list $artifact]
         foreach member $members {
            set subcomponent "$artifact/$member"
            recursivemove $subcomponent $destinationhost
         }
         nc_removeattribute $tree "hasmoved"
      }
   }

   #
   # Part 3:  done.
   #
   puts "$component has been moved."
}
```

Figure 5.17: Recursively Moving an Artifact and Its Members.

recursion is protected by the attribute `hasmoved`, which is used at the beginning of the procedure to check for infinite recursion.

5.3.2 Product Family Architectures

A rather different application of configuration management arises in the field of software architecture [PW92, GS93]. In its simplest form, a software architecture is a specification of the high-level structure of a system in terms of the components and the connections among the components. As an example, consider a word processor that consists of three separate components, a user interface, a storage facility, and a spell checker. To facilitate their interaction, these components are connected via several mechanisms. Specifically, both the user interface and spell checker use a pipe to interact with the storage facility, whereas the user interface uses a software bus to interact with the spell checker.

To precisely capture a software architecture, architecture description languages are used (e.g., UniCon [SDK+95], C2 [TMA+96], Wright [AG97]). These languages typically include a typing mechanism, as well as facilities to describe behaviors and constraints, to ensure that components and connections can only be combined into proper configurations.

Similar to source code, the components and connections that make up a software architecture may exist in multiple revisions and multiple variants. For example, the storage facility may exist in several revisions that each add additional primitives to its interface. Similarly, the spell checker component may exist in multiple variants that each operate on a different language. In addition, certain components or connections may be optional in a software architecture. For instance, the word processor can usefully be applied, regardless of whether or not a spell checker is present. The structure that results when all revisions, variants, and options of all components and connections are put together is called a product family architecture [CW98a].

As of yet, no CM system addresses the management of product family architectures. This is due to the fact that, as compared to the management of source code, a product family architecture has certain characteristics that make it rather unique. Specifically, the structure to be managed includes typing and contains a rather odd composition capability in which types are constructed in terms of specific instances. Therefore, highly specific relationships and semantics have to be put in place by a CM policy that manages a product family architecture.

Although alternative approaches are certainly feasible, the remainder of this section introduces one particular approach to the creation of a CM policy that is specifically designed to manage product family architectures.

5.3.2.1 Repository Design

Product family architectures are closely related to the system model as traditionally used in configuration management by the composition policy [vdHHW98]. Although a system model is actually used to manage source code rather than itself being the objective of the management, its resemblance to a product family architecture indicates that a CM policy that manages a product family architecture can be based on an adaptation of the composition policy. That is the approach taken here.

Of course, to manage a product family architecture, a number of changes are required to the repository design of the composition policy as described in Section 5.1.2.1. The first change concerns the nature of the artifacts being stored. Whereas the composition model only stores a hierarchy of instances, capturing a product family architecture requires the storage of a hierarchy in which types and instances alternate. In particular, types of components (or connections) are composed of instances of other types of components and connections. One solution to this complication represents, in the compositional hierarchy, types as collections and instances as membership relations of collections to other collections. A problem with this kind of solution, however, is that

instances have private data and that the repository model as defined by the abstraction layer does not support the association of attributes with membership relations. Due to this limitation, an alternative solution is applied that, in essence, unrolls the hierarchy. Partly illustrated in Figure 5.18, this solution stores each type as a collection, each instance as an atom, and the relationship from an instance to its type as an attribute to the instance. As a result, the repository design no longer forms an explicit hierarchy such as the one employed by the composition policy. Instead, the hierarchy is implicit in the traversal from collections (types), to atoms (instances), to attributes (instance-type relations), back to collections (types).

Note that the repository design as displayed in Figure 5.18 is not complete. Additional top-level collections are present in the overall design, that capture, among others, connection types, version trees associated with component types, and version trees associated with connection types. Moreover, each collection that represents a (component or connection) type contains three additional atoms: one that captures its behavior, one that captures its constraints, and one that captures its architectural specification in a particular architecture description language.

As an example of how the repository design stores a particular product family architecture, consider the component type `WordProcessor` as shown in Figure 5.18. This component type consists of three component instances, that are represented by the atoms `storage`, `spell`, and `gui`. In addition, it contains three connection instances, that are represented by the atoms `pipe-1`, `pipe-2`, and `bus`. Every component or connection instance has an associated attribute that identifies their type. For example, the atom `spell` is labeled with the attribute `componenttype` that has the value `ComponentTypes/SpellChecker:2` and the atom `bus` is labeled with the attribute `connectiontype` that has the value `ConnectionTypes/Bus:3`.

The repository design represents variants of components (and connections analogously) using a special kind of component, namely the variant component. The member

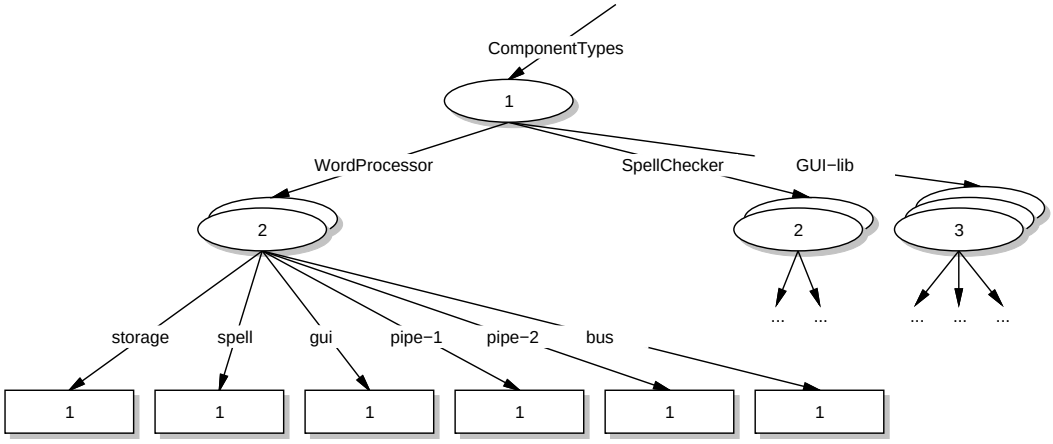


Figure 5.18: Partial Example Repository Structure for the Product Family Architectures Policy.

atoms of a variant component are restricted to only be instances of component types and cannot be instances of connection types. In addition, each atom has an associated attribute that has a unique value as compared to the values associated with the other atoms. This attribute, the variant property, is used in the traversal of the variant component to decide which particular instance is chosen to be part of an architectural configuration.

Options are also represented using properties. Each instance that is optional is guarded by an attribute and an associated value. Only if the property value as assigned by a higher-level component matches the attribute value of the optional instance, that instance is included in the architecture.

5.3.2.2 Core Policy Design

To manage product family architectures, the design of the repository as used in the composition policy was changed significantly. As a result, one would expect a major change in the procedures that form the core of the composition policy for those procedures to function properly in managing a product family architecture. However, an examination of the functionality of Ménage, a graphical environment designed to manage product family architectures [vdHHW99], leads to the result that none of the procedures would need to be changed in order to be used by Ménage.

- The procedure `populateworkspace` does not need to be modified, since a workspace only needs to be populated with a particular type of component (connection) and all of its associated instances. Simply instantiating the procedure with an empty set of rules and the name of a component (connection) type provides the desired result.

- The procedure `checkout` does not need to be modified, since its current programmatic logic simply changes the component (connection) type in the work-

space to allow a user to modify it. Because Ménage only modifies one particular type at a time, and can only change it by adding or removing instances, this is exactly the functionality that is needed.

- The procedure `checkin` does not need to be changed, since its current programmatic logic simply checks in a new version of a component (connection) type, which, once again, is exactly the functionality that is needed by Ménage.

Of course, many other auxiliary procedures are utilized by a CM policy such as the one employed by Ménage. In particular, as new instances are added to a type, atoms need to be created in the workspace to represent those instances. Similarly, when instances are removed from a type, corresponding atoms should be removed from the workspace. Thus, even though the core of the policy remains the same, auxiliary procedures still need to be created to produce a fully functional policy that is capable of managing product family architectures.

5.4 Lessons Learned

Demonstrating the expressiveness of the testbed, this chapter has presented how a large variety of well-known and representative CM policies, ranging from the simple centralized checkout/checkin policy to the more complicated distributed replicated repositories policy, can be mapped onto the repository model and programmatic interface. The ten policies described in detail, as well as all of their variants discussed, show how the four objectives for the abstraction layer are met.

- The fact that the abstraction layer is able to support a wide range of different versioning policies, both new and existing, illustrates that it has satisfied the objective of being policy independent.

- The fact that the abstraction layer is able to support a wide range of different distribution policies, ranging from simple client-server workspaces to peer-to-

peer replicated repositories, illustrates that it has satisfied the objective of being able to support distributed operation.

- The fact that the abstraction layer is able to support the storage of both source code as well as product family architectures, together with the policies discussed in Chapter 7 that store documents, software packages, and Web pages, illustrates that it has satisfied the objective of being able to support the management of a wide variety of different kinds of artifacts.

- The fact that the abstraction layer is able to support the creation of existing CM policies, including the full spectrum identified by Feiler [Fei91a], illustrates that it has satisfied the objective of being able to support traditional CM functionality.

Based on these observations, the definition of the abstraction layer as given in Chapters 3 and 4 can be considered a success: the goals that were set for the abstraction layer are met. However, the benefits provided by the abstraction layer rise beyond just those four goals. In particular, the following two additional benefits are identified.

- **The abstraction layer supports the incremental construction of CM policies.**

  Observe how the long transaction policy builds upon the composition policy and how the composition policy builds upon the checkout/checkin policy. Even the change set policy, which is based on a radically different paradigm, reuses pieces of the core policy design of the composition policy. Another particularly illuminating example is the movement upon checkout policy. It is created by adding three lines of pseudo code to one of the procedures in the composition policy. Similarly, the user-controlled artifact movement policy is created by adding a new, separate procedure to the composition policy. In both cases,

the rest of the core policy design of the composition policy remains the same. These examples illustrate that the abstraction layer supports a powerful model of evolution and experimentation with respect to the creation of CM policies.

- **The abstraction layer supports a great amount of flexibility in the way CM policies use the repository model.**

  Observe how the checkout/checkin policy stores its most important relationships (the version tree) in atoms, how the composition policy captures its most important relationships (the system model) as a compositional hierarchy of collections and atoms, and how the product family architectures policy captures its most important relationships (the type-instance relationships) as attributes. The abstraction layer supports either type of use, and its mechanisms can thus be utilized to match the type of access needed by a CM policy.

Of course, it should also be observed that the policies that are constructed with the abstraction layer are not necessarily as optimized and polished as their counterparts that are developed from scratch. Consider the long transaction policy. Representing its workspaces as physical repositories is a heavy-weight solution that incurs more copying and uses more storage space than an optimized implementation such as NSE [FD90]. Similarly, the repository synchronization routines included in such commercial systems as ClearCase [AFK+95] or Continuus [Con98] are far more efficient than the one developed in this chapter. This is not a serious drawback, since optimality is not a goal of the testbed. Instead, the testbed focuses on facilitating the experimentation with, and creation of new, potentially distributed CM policies. Once a desired policy has been developed, the policy can be reimplemented from scratch if the performance provided by the testbed is not sufficient. In the mean time, though, the availability of the testbed facilitates the rapid exploration and evolution of CM policies that are tailored to a specific situation, a capability that, until now, was non-existent.

Chapter 6

Prototype Implementation

To demonstrate the feasibility of the testbed, this chapter describes how the abstraction layer is realized in a prototype implementation. The prototype, NUCM (Network-Unified Configuration Management), faithfully implements the complete repository model as defined in Chapter 3 and all but one of the functions in the programmatic interface that is defined in Chapter 4.

An important aspect of the prototype is that its internal architecture continues the theme set forth by the abstraction layer: it is based on a separation of concerns that isolates distribution from other aspects of the implementation. This isolation is achieved through an incremental layering of functionality in which a particular layer is responsible for building specific pieces of functionality on top of the layers residing underneath it.

As a prototype, the testbed does not contain all of the functionality that one might expect in an industrial-strength implementation. Missing is such functionality as caching, compression, and delta storage. Still, this omission from the prototype should not prevent an industrial-strength implementation from incorporating those kinds of functionality. Therefore, the last part of this chapter shows that the semantics defined by the abstraction layer does not prevent the inclusion of caching, compression, and delta storage in a full implementation of the testbed.

The remainder of this chapter is organized as follows. First, it discusses a map-

ping of the high-level architecture of the prototype onto the abstraction layer. Then, it introduces the internal architecture of the prototype. The chapter concludes by briefly discussing how the orthogonality underlying the abstraction layer facilitates an implementation that includes caching, compression, and delta storage.

6.1 High-Level Architecture

Figure 6.1 illustrates the high-level architecture of NUCM in terms of an example repository structure. Shown are six types of entities that, combined, implement the abstraction layer as defined in Chapters 3 and 4: artifacts, NUCM servers, a logical repository, NUCM clients, workspaces, and CM policies. Sets of artifacts represent a physical repository. In the figure, three such physical repositories are present. Access to each one of those is provided by an associated NUCM server. Each NUCM server manages a single physical repository, but all NUCM servers collaborate in providing the illusion of a single logical repository. Note that the NUCM servers are not fully connected. This is due to the fact that a logical repository is not defined as a permanent structure. Instead, the contents of the collections that are stored in each physical repository define which NUCM servers interact with which other NUCM servers.

A CM system that uses a NUCM repository consists of three parts: the NUCM client, one or more workspaces, and the specifics of the CM policy that is used by the CM system. The NUCM client implements the programmatic interface and thus is the foundation upon which particular CM policies are implemented. The NUCM client interacts with one or more NUCM servers to provide, via a workspace, a CM policy with access to the artifacts that are stored in a logical repository. The CM policy, in turn, manipulates the artifacts in the workspace and further uses the generic NUCM client to store the modified artifacts back in the logical repository.

All CM policies are built upon the same, generic NUCM client. This is illustrated in Figure 6.1, where two CM policies, namely policy X and policy Y, both use the generic

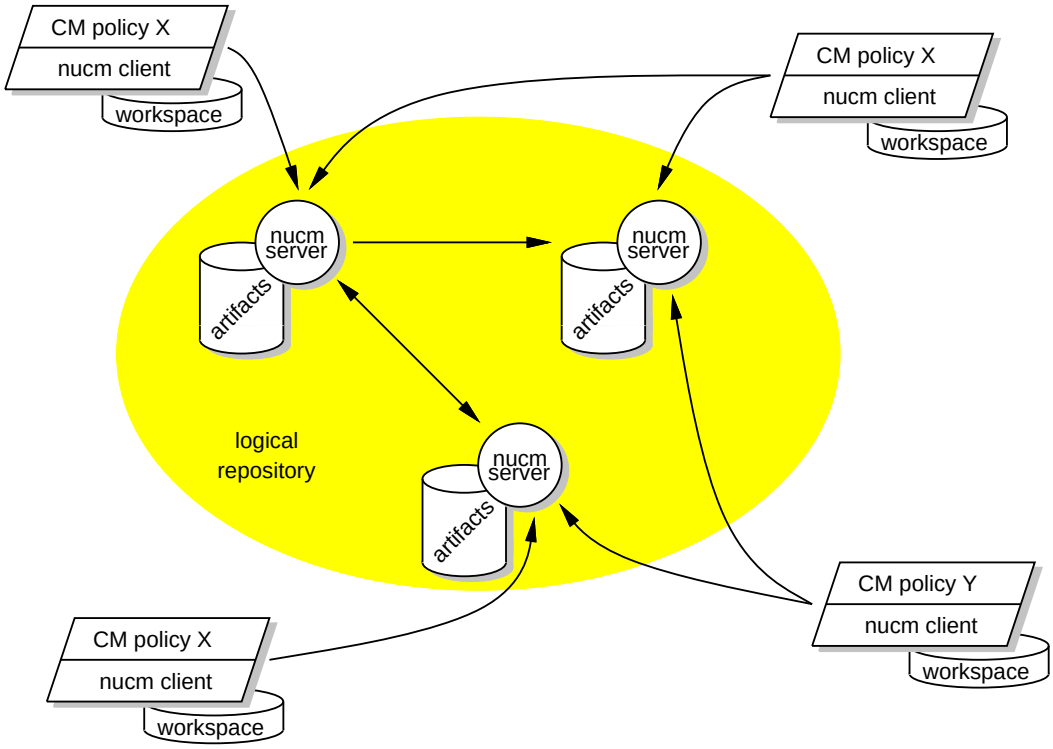


Figure 6.1: High-Level Architecture of NUCM.

NUCM client to store and version the artifacts that they manage. In general, a single (logical or physical) repository can store artifacts that are managed by different CM policies, as long as the policies partition the artifacts in separate name spaces within the repository. If different policies operate on the same artifacts, it is the responsibility of the CM policies to resolve any conflicts.

6.2 Implementation Architecture

Presented in Figure 6.2, the internal design of the prototype implementation of the testbed consists of nine layers. Five of those layers, namely Persistent Storage, Concurrency Control, Reference Counting & Storage Reclamation, Server, and Server Communication & Access Control, constitute the NUCM server. The remaining four layers, which are Client Communication, Moved Artifact Handling, Name Interpretation & Workspace Management, and Client Interface, constitute the generic NUCM client. Communicating via TCP/IP, instances of the NUCM client and server combine to provide a CM policy with access to the artifacts that are stored in a logical repository.

The remainder of this section discusses the functionality that is provided by each of the layers in the implementation architecture. Since the functionality of each layer builds upon the functionality provided by the layers underneath it, the layers are discussed bottom up.

6.2.1 Persistent Storage

The lowest layer in the implementation architecture implements the persistence of artifacts in a physical repository. The layer is responsible for storing different kinds of information, but it does not interpret any of the information. Such interpretation is left to higher-level layers in the architecture. Nonetheless, the persistent storage layer has to define an appropriate schema that can be used by the higher-level layers. Shown in Figure 6.3, this schema is implemented as a hierarchical structure in the file system.

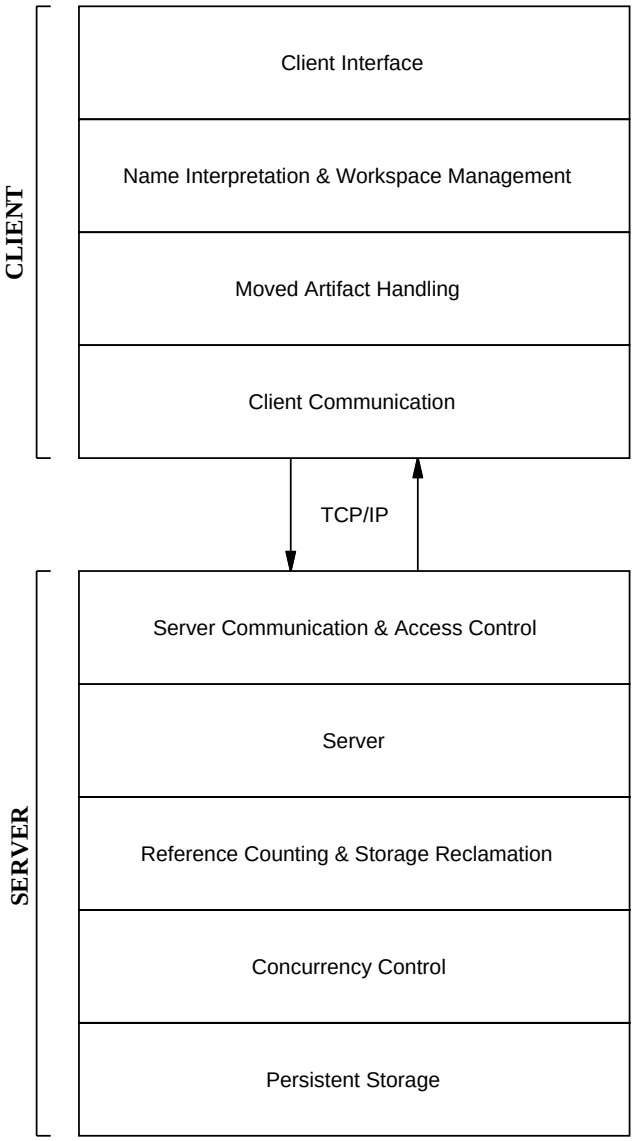


Figure 6.2: Internal Design of NUCM.

The schema is based upon the use of consecutively numbered directories, guaranteeing a unique location each and every time a new (version of an) artifact is stored. To facilitate the creation of uniquely numbered artifacts, a persistent counter is stored in the file `artifactcntr`. This counter tracks the number of artifacts that have been created over time and is updated each time a new directory is created. Located within each artifact directory, the file `versioncntr` contains a similar persistent counter to keep track of the number of versions that have been created.

Two special files reside within each artifact directory. The first file, `forward`, is present when the artifact has been moved to a different physical repository. The contents of this file is a forwarding address (see Section 6.2.4). The other file, `type`, contains a designation as to whether the artifact is a collection or an atom. Regardless of whether a collection or an atom is stored, though, the actual structure of the artifact directory remains the same. It is the interpretation of the stored information at the higher levels in the architecture that distinguishes collections from atoms.

Within each numbered version directory, the actual contents of an artifact are stored in the file `contents`, whereas the file `refcount` contains a counter that keeps track of the number of times the particular version of the artifact is contained by a collection (see Section 6.2.3). Once again, the persistent storage layer does not interpret any of the information and consequently has no knowledge as to whether a collection or an atom is stored in the file `contents`, or as to whether the storage space of an artifact needs to be reclaimed since the artifact is no longer contained.

As defined by the abstraction layer, attributes are specific to a version of an artifact. Because, in addition, the value of an attribute can be of arbitrary length, attributes are stored as a series of files in a separate directory instead of all being stored in a single file. The directory in which the attributes are stored is called `attributes`. Each file in this directory is uniquely named and contains the value of a single attribute. Because attribute names are unique per definition of the abstraction layer, it suffices to

```
/artifacts/artifactcntr
          /artifact_1.lock
          /artifact_1/forward
                     /type
                     /versioncntr
                     /version_1.lock
                     /version_1/contents
                               /contents.lock
                               /refcount
                               /attributes/attribute_A
                                          /attribute_B
                                          /attribute_C
                     /version_2.lock
                     /version_2/contents
                               /contents.lock
                               /refcount
                               /attributes/attribute_A
                                          /attribute_B
                                          /attribute_C
                                          /attribute_D
          /artifact_2.lock
          /artifact_2/forward
                     /type
                     /versioncntr
                     /version_1.lock
                     /version_1/contents
                               /contents.lock
                               /refcount
                               /attributes/attribute_A
                                          /attribute_B
                                          .
                                          .
                                          .
                     .
                     .
                     .
          .
          .
          .
```

Figure 6.3: Structure of a Physical Repository as Implemented in the File System.

name each file after the name of the attribute it contains.

The final part of the physical repository structure is a series of lock files. These files are associated with each artifact, each version of an artifact, and the contents of each version of an artifact. They provide the basics for the higher-level layers in the architecture to perform concurrency control (see Section 6.2.2).

Two observations are in place about the persistent storage layer. First it should be noted that the layer, just like any of the other layers, provides an internal programmatic interface to the higher-level layers. This interface contains functions that, for each type of file in the directory structure, allow its contents to be set and retrieved. In addition, the presence of various directories and files can be tested (e.g., it is possible to verify whether a particular artifact version or attribute exists).

The second observation regards the choice of the file system as the basis for the persistent storage layer. Although sufficient for purposes of prototyping, an industrial-strength implementation should use a database as its fundamental infrastructure to gain such advantages as increased reliability, transaction functionality, and rollback capabilities.

6.2.2 Concurrency Control

Because a NUCM server is capable of handling multiple requests from multiple NUCM clients at the same time, the second layer of the implementation architecture utilizes the locks provided by the persistent storage layer to perform concurrency control. Whereas some operations performed by a NUCM server (see Section 6.2.4) require exclusive access to a whole artifact (e.g., moving an artifact), other operations only require exclusive access to a version of an artifact (e.g., removing a version of an artifact or adding new attributes to an artifact). Yet other operations only require exclusive access to the contents of a version or to an attribute value (e.g., storing new contents for an artifact version or removing an attribute). To support this rather complex style

of concurrency control, an incremental style of locking is provided by the concurrency layer. Based on the use of a combination of exclusive and shared locks, each server operation uses the concurrency layer to prevent any potential conflicts.

In the incremental locking scheme, each server operation first locks the artifact to which it needs access. Depending on whether the operation needs shared or exclusive access, and depending on the type of lock that may already be present, access may or may not be granted. For example, when one NUCM client instructs a NUCM server to move an artifact, the server uses an exclusive lock to prevent any other NUCM client from simultaneously performing an operation on the artifact being moved. On the other hand, if the operation is one that can be shared at the artifact level, such as setting or retrieving an attribute, a shared lock is placed on the artifact. For such a shared lock, a finer-grained exclusive lock is still required. This finer-grained lock is provided in the form of a version lock or, at an even finer grain, a content lock. Similar to the lock at the artifact level, these locks are either exclusive or shared and their use depends on the needs of each particular type of server operation.

As an example of how the incremental locking scheme is used, consider the server operation that sets new contents for a particular version of an artifact. This operation first acquires a shared lock at the artifact level, then a shared lock at the version level, and finally an exclusive lock at the content level. As another example, the operation that removes a complete version of an artifact first acquires a shared lock at the artifact level and then an exclusive lock at the version level.

Like these two examples, all other server operations incrementally use the appropriate set of locks for their needs. This proper use of locking guarantees the serializability of the NUCM server operations.

6.2.3 Reference Counting & Storage Reclamation

To facilitate the reclamation of storage space occupied by artifacts that are no longer contained by any collection, the third layer in the implementation architecture uses a reference counting mechanism to guard all server operations that manipulate collections. In particular, when a new (version of a) collection is added, the functions in the reference counting and storage reclamation layer increment the reference counters of all artifact versions that are a member of the collection. Similarly, when a version of a collection is removed, the reference counters of all member artifact versions are decremented.

If, after being decremented, the reference counter for a particular artifact version has the value of zero, it is attempted to reclaim the storage space of the artifact by verifying whether the reference counters for all of its versions are zero. If that is the case, all versions of the artifact are physically removed from the repository. Of course, if the artifact is a collection, each removal of a version triggers the same process: reference counters for contained artifact versions are decremented. Therefore, it is possible that the removal of a single artifact from a physical repository triggers the recursive removal of a whole tree of artifacts.

Three observations can be made about the mechanism used to reclaim storage space. First, it should be noted that it is necessary to associate multiple reference counters with an artifact. Instead of a single reference counter at the artifact level, the use of a reference counter per version allows the implementation to rapidly verify whether an artifact version is contained or not. This is an essential capability for the interface function `nc_destroyversion`, which is only allowed to remove an artifact version if that version is no longer contained (i.e., the reference counter for that version of the artifact is zero).

The second observation regards the potentially cyclic nature of the versioned

directed graph of artifacts. Specifically, if some number of collections form a cycle, the algorithm as described fails. This is a limitation of the current prototype. However, the isolation of the reference counting mechanism in a single architectural layer facilitates the insertion of more advanced algorithms that do account for cycles.

Finally, it should be noted that, to implement the reference counting mechanism, NUCM servers act as clients of each other. In particular, since membership of a collection can span multiple physical repositories, increments and decrements in reference counters may result in communication among NUCM servers. Although potentially expensive when a collection contains many artifacts that are located in a different physical repository than that of the collection itself, the use of reference counting is required to support storage reclamation as well as to prohibit the removal of artifact versions that are still contained.

6.2.4 Server

Similar to the way the NUCM client provides a programmatic interface upon which CM policies are built, the NUCM server provides a programmatic interface upon which the NUCM client is implemented. This interface is implemented by the server layer and contains a set of functions that represent a middle level between the high-level functions provided by the NUCM client and the low-level functions provided by the persistent storage layer. Forming the core of the NUCM server, the most important functions in the server layer are those that store and retrieve versions of artifacts, manipulate attributes, move and copy artifact histories, and provide information about the type and existence of artifact versions.

In providing its functionality, a NUCM server interacts with other NUCM servers. In particular, when artifacts are moved from one physical repository to another physical repository, the two NUCM servers that manage both of these physical repositories collaborate to transfer the artifact, the versions, and the attributes.

An important consequence of moving an artifact is that all the references stored in collections that contain a version of the artifact become out of date. To avoid a repository-wide search for these references, the NUCM implementation is based on the use of a forwarding address. After an artifact has been moved, a forwarding address is stored in the original repository. When a request is made by a NUCM client to access the moved artifact, the NUCM server does not carry out the request. Instead, it locates the artifact by following its trail of forwarding addresses and returns, to the NUCM client, the physical repository in which the artifact is actually stored. The client, in turn, updates the reference in the collection and continues its operations by interacting with the physical repository that contains the moved artifact (see also Section 6.2.7). Because each of the references to the original location of the moved artifact is updated over time, it can be expected that eventually all old references disappear and all access to the artifact takes place directly via the NUCM server that manages the physical repository in which it is actually stored.

6.2.5 Server Communication & Access Control

The server communication and access control layer is that part of the architecture that makes the NUCM server accessible from remote workspaces. It wraps each of the functions provided by the NUCM server with a TCP/IP-enabled companion function. Each of those companion functions is responsible for parsing incoming requests, invoking the appropriate server function, and transferring the response from that function to the remote NUCM client.

This layer is also responsible for implementing two additional capabilities of the prototype: concurrent access and access control. Concurrent access is provided by processing each request that is made to the server in a separate thread. Since each of the functions in the previous layer utilizes the functions in the concurrency control layer to properly lock those parts of an artifact to which it needs access, simply handing each

incoming request in a separate processing thread suffices.

The second responsibility is access control. To prevent a situation in which any NUCM client can request artifacts from any NUCM server, each NUCM server can be protected by an access control list that specifies those IP addresses from which requests may originate. If a request is made from an IP address that is not in the list, the request is denied and the NUCM client is informed that it is not allowed to access the desired artifact.

6.2.6 Client Communication

The client communication layer of the architecture complements the functionality of the server communication and access control layer. It is the bottom layer of the NUCM client and takes care of establishing a connection with a desired NUCM server, forwarding the request to be made, and receiving and parsing the response from the NUCM server. Aside from communicating with a NUCM server, this layer has no further responsibilities.

6.2.7 Moved Artifact Handling

A NUCM server does not perform a client request when an artifact has moved. Instead, a NUCM server returns the new location of the artifact, forcing a NUCM client to repeat its request to a different NUCM server. It is the responsibility of the functions in the moved artifact handling layer to do so: if an artifact has moved, these functions repeat the particular request to the NUCM server that manages the physical repository to which the artifact has been moved. In this scheme it is possible that, in between the return of the new location and the request to the new physical repository, an artifact moves again. Therefore, the functions in this layer keep repeating a request to different physical repositories until the request completes.

At first sight this solution is inefficient, since it involves more channels of commu-

nication than a solution in which the NUCM servers themselves carry out a request by forwarding and returning the necessary information. However, two reasons exist that justify the use of a solution in which the NUCM client, in essence, "chases" an artifact that moves repeatedly.

(1) The situation in which an artifact continuously moves is unlikely to occur frequently, since the movement of artifacts is a relatively expensive operation that should only be performed when absolutely necessary.

(2) Since a response is only transferred once (e.g., from a physical repository to a NUCM client) rather than multiple times (e.g., repeatedly from one physical repository to another physical repository until the response eventually reaches a NUCM client), it is likely that less overall traffic is incurred. Especially when large artifacts are transferred, the solution in which the client request is repeated is more efficient, since a direct connection is established.

6.2.8 Name Interpretation & Workspace Management

The NUCM prototype uses an internal naming scheme that uniquely identifies each version of an artifact with an associated "NUCM id". It is the responsibility of the first part of the name interpretation and workspace management layer to transform a name that is given to the NUCM client by a user (a name that follows the structure as defined by the naming model in Section 3.3) to an internal NUCM id. Highly specialized name interpretation rules govern this part of the architecture. These rules interpret each part of the name by walking down the versioned directed graph of artifacts. Depending on the nature of the versioned directed graph and its distribution among physical repositories, this may require communication among a NUCM client and several NUCM servers to obtain the NUCM id. Oftentimes, however, name interpretation can be performed locally within a workspace, and no communication overhead

is incurred.

The second part of the name interpretation and workspace management layer is responsible for the management of the contents of a workspace. Special, hidden files are used to track which artifacts have been opened or initiated. In particular, each directory in a workspace contains a file called `.workspace` that maintains, in a concise format, for each contained artifact its name, its type, an indicator as to whether it has been opened, and an indicator as to whether it has been initiated. In addition, a workspace caches the membership relations of a collection to not have to repeatedly request this information from a remote NUCM server. Again, special, hidden files are used for this purpose. In particular, each directory that represents a materialized collection contains a file called `.collection`. This file contains, for each member artifact, its name and its internal NUCM id.

The name interpretation and workspace management tasks are grouped into a single architectural layer on purpose. Both tasks access the same type of information, and their collocation in a single architectural layer facilitates the sharing and caching of this information.

6.2.9 Client Interface

The final layer in the architecture of NUCM is the layer that implements the interface functions that form the NUCM client. Except for the function `nc_selectversions`, all of the functions in the programmatic interface (as described in Chapter 4) are implemented. The only reason the function `nc_selectversions` is not implemented is that it represents a utility function that optimizes network traffic. Its functionality can still be achieved using a combination of the other interface functions that are available. None of the other functions fall into this class. They are all essential and contribute a unique piece of functionality to the programmatic interface. Therefore, all other functions are implemented in the prototype.

The precise details of each function in terms of its parameters, error codes, and return values, are presented in Appendix A.

6.3 Implementing Optimizations

The current NUCM prototype contains a few optimizations for operation in a distributed setting. The most important of those are the fact that collection membership is cached inside a workspace, that NUCM clients follow artifacts that have been moved, and that the membership of collections is updated when artifacts have been moved. An industrial-strength implementation of NUCM, however, should also fully exploit the benefits of other optimization techniques such as caching, delta storage, and compression. Therefore, neither the repository model nor the programmatic interface prohibits incorporation of these techniques in NUCM. The layering that forms the backbone of the implementation architecture of NUCM facilitates their incorporation without the need for widespread changes throughout the whole architecture. In fact, each optimization can be localized in one or two layers, thereby significantly reducing its impact on the implementation. The remainder of this section discusses how the incorporation of caching, delta storage, and compression can be achieved in the NUCM prototype.

6.3.1 Caching

The workspaces defined by the abstraction layer already provide a rudimentary form of caching: artifacts are cached in a workspace while they are being manipulated by a CM policy. Even so, an additional form of caching can—and should—be incorporated in the implementation of the testbed. This form of caching relies on each of the NUCM servers that are part of a logical repository to cache those artifacts that are regularly requested but reside at a different physical repository. In its simplest form, such caching becomes a variant of an automated, read-only replication mechanism. Frequently used artifacts are cached by local NUCM servers to facilitate quick

materialization in a workspace, but changes are still committed directly to the remote physical repositories in which the actual version histories of the cached artifacts are kept.

Due to the layered structure of the NUCM prototype, caching can be added by simply changing one layer in the architecture. Specifically, the server layer can be changed to perform a few additional tasks. These tasks are based on an adaptation of the forwarding mechanism. Specifically, after a NUCM server has decided to cache an artifact, it copies the version history of that artifact to the local physical repository. However, it also attaches a forwarding address to the local copy that points to the remote copy of the artifact. Once this process has completed, a NUCM client interacts with the NUCM server as usual, with the exception that the NUCM server intercepts certain requests. These requests, all of a read-only nature, are satisfied by returning the local copy of the artifact instead of the remote one. When a write request is made, however, the request is not intercepted and the forwarding mechanism is leveraged to make sure that the original copy of the artifact is updated and not the cached version.

Of course, cache consistency needs to be enforced. For this purpose, the server layer should be modified to associate an extra attribute with each version of an artifact. The value of the attribute is a unique hash value that is based on the contents of the artifact. Before a cached copy of an artifact is returned to a NUCM client, the cached attribute value is compared to the attribute value stored at the original physical repository. If both values are equal, the cached copy is returned. Otherwise, the cached copy is inconsistent and needs to be updated before the response to the NUCM client is given.

Two observations can be made about this method of adding caching to the NUCM implementation. First, it should be noted that the mechanism is completely transparent to the NUCM client and only involves modifications to the NUCM server. The second observation is that it is conceivable that a more advanced caching mechanism can be

devised in which NUCM servers detect patterns of access to the artifacts that are stored in a logical repository (e.g., each time the function `nc_getattributevalue` is used on some artifact, it is followed by a use of the function `nc_open` on the same artifact). These patterns can be used to predict and prefetch those artifacts that are expected to be used in the near future.

6.3.2 Delta Storage

An important technique that is used by CM systems to conserve storage space is delta storage. Rather than storing each version of an artifact in its entirety, differences among versions are calculated and stored. When a particular version of an artifact is needed, it is constructed by applying the necessary differences to some base version of the artifact. Some systems use forward deltas, a scheme in which the original version of an artifact is the baseline. All future versions are constructed by applying a series of changes to that baseline [Roc75]. To speed up access to later versions of an artifact, other systems use backward deltas, a scheme in which the latest version of an artifact is the baseline. All previous versions are constructed by undoing a series of changes from that baseline [Tic85].

The addition of a delta storage mechanism to the NUCM prototype is rather straightforward. Once again, only one layer in the architecture is affected, namely the persistent storage layer. In particular, the directory layout utilized by this layer to store versions of artifacts has to change from a series of numbered version directories to a single archive file in which all versions are stored. To manage such an archive file, several reusable libraries are available. For example, vdelta [FKR+95] or bdiff [Tic84] can be used for this purpose.

Because the abstraction layer is based on a linear versioning scheme that does not track the derived-from relationship that is necessary to perform difference calculations, one additional part is required for the solution to be complete. The information that

is kept in a workspace can be leveraged for this purpose. In particular, a workspace keeps track of which version of an artifact is initiated for change. This information can be used by a delta algorithm to keep track of the derived-from relationship when a new version of an artifact is committed to a repository. Of course, to do so, this information needs to be transferred from the name interpretation and workspace management layer to the persistent storage layer, which requires the addition of a parameter to each of the functions in the series that commit a change from a workspace to persistent storage. The isolation of changes that is provided by the architectural layering of functionality is indeed broken by this solution. Yet, each of the intermediate layers is only minimally affected in that a single parameter needs to be passed through a series of functions.

Of course, it is not always desirable that delta storage be applied to artifacts. In fact, it is known that for some artifacts certain delta algorithms produce results that provide hardly any storage improvement despite being computationally expensive [HVT98]. Moreover, the way a CM policy uses the storage model may prohibit the application of a delta algorithm. In general, therefore, a CM policy should decide whether or not a delta algorithm is applied to the artifacts that it manages. Thus, for delta storage to be properly added to the NUCM prototype, the function `nc_add` should be enhanced with a parameter that allows a CM policy to control whether the storage of the artifact being added should use a delta mechanism or not (which, in fact, is similar to the solution being applied in Continuus [Con94]).

6.3.3 Compression

Another important optimization in the presence of wide-area distribution is the use of compression. Rather than shipping, as is, an artifact back and forth over long distances, it is often advantageous to compress an artifact before transmission. Depending on the trade-off between computational expenses and network delay, the use of compression can be beneficial in improving the overall performance of a distributed

system.

To add compression to the NUCM prototype, two layers of the architecture are affected: the server communication and access control layer and the client communication layer. Both of these need to be modified to incorporate a compression and decompression algorithm that is used whenever a large chunk of data needs to be transported. Any of the available compression algorithms can be used for this purpose.

The same argument that is made for delta storage can be made for compression: for certain types of artifacts, the trade-off between computational expense and actual gain is very low. A CM policy must, therefore, indicate whether or not compression should be used. Again, the solution to this problem is to parameterize the function `nc_add` to provide this type of control to a CM policy programmer.

An interesting avenue to explore is the use of deltas as a compression mechanism. In the same way that they can be used to provide efficient storage, the difference between the version of an artifact that is initiated for change in a workspace and the version of the artifact that is committed to a physical repository can be calculated. Rather than transmitting the new artifact in its entirety, NUCM could instead transmit the difference between the original and new version. Combined with compression of the differences, this solution may provide significant performance gains.

Implementing this solution in NUCM is more complicated than incorporating one of the previous optimizations. On top of a change that incorporates delta storage in the primitive storage layer and compression in the two communication layers, this optimization requires another change to be made. Specifically, a workspace has to preserve the original copy when an artifact is initiated for change. Moreover, the namespace interpretation and workspace management layer of the architecture also needs to be enhanced with the same delta algorithm as employed by the persistent storage layer. Overall, this optimization would lead to changes in the name interpretation and workspace management, client communication, server communication and access control, and persistent

storage layers. Although impacting many layers, this optimization is the most advantageous one, and can build upon the solutions created for the previous optimizations.

Chapter 7

Experience

Based on the NUCM prototype, three novel CM systems have been implemented. These three CM systems demonstrate the utility and validity of the testbed. In particular, the utility is demonstrated by the fact that all three systems are actually implemented using the prototype and the validity is demonstrated by the fact that all three systems were constructed relatively quickly and exhibit unique functionality with respect to other CM systems.

Two of the CM systems that have been implemented using the NUCM prototype, namely DVS [Car98] and SRM [vdHHHW97], are presently in everyday use, whereas the third system, WebDAV, represents an experimental implementation of an emerging standard in Web versioning [Whi97]. Characteristics of all three systems are that they have evolved over time, operate in a distributed setting, and manage artifacts of kinds other than traditional source code.

Below, we discuss each system in more detail and use parts of their implementation to illustrate how the NUCM prototype can be used to program particular CM policies. It should, of course, be noted that the policies themselves are not the contribution. Instead, the contribution lies in the ease with which these policies were constructed, in the ability to version different kinds of artifacts, and in the limited amount of effort needed to make them suitable for use in a wide-area setting.

7.1 DVS

DVS (Distributed Versioning System) [Car98] is a versioning system developed by Antonio Carzaniga that is focused on providing a distributed environment in which documents can be authored collaboratively. DVS is centered around the notion of workspaces. Specifically, individual users populate their workspace with the artifacts needed, lock the artifacts they intend to change, modify these artifacts using the appropriate tools, and commit the modified artifacts from the workspace to a storage facility.

DVS is a tool that was developed in response to a need for a distributed group of collaborators to jointly author documents. Combined with the collaborators' familiarity with RCS [Tic85], this domain placed some stringent requirements upon the design and implementation of DVS.

- **DVS should strictly avoid conflicts.**

  Due to the importance of the documents being versioned, each author should be guaranteed exclusive access to the artifacts that they change.

- **DVS should capture the evolution of an artifact in a linear fashion.**

  The nature of the documents being versioned is such that almost all changes are based on the latest version available. Therefore, linear evolution is desired and the creation of branches is not required. Nonetheless, it should still be possible to base changes on a version older than the latest. Even in this case, however, the version that results should supersede the current latest version and form the basis upon which any subsequent changes are made.

- **DVS should support private workspaces.**

  Each user should be able to make their changes in a private workspace, without being disrupted by changes that other users make simultaneously.

- **DVS should shield a user from the potential distributed storage of artifacts.**

  Of course, network delays and other types of inherent network behavior are always visible, but preferably most, if not all, details of distributed operation are hidden from the users of DVS.

- **DVS should be simple to use.**

  Preferably, the operations, flags, and syntax of DVS should be similar to the operations, flags, and syntax used by RCS.

7.1.1 Design

DVS was created before any of the core policy designs discussed in Chapter 5 were developed. Despite this independence, an examination of the similarities between DVS and those policies reveals that DVS can be characterized as combining the peer-to-peer repositories policy with a variation of the composition policy. In particular, the ability of DVS to compose a document out of many parts while allowing those parts to be stored in different physical repositories warrants this characterization.

DVS is normally used in a setting that resembles the client-server workspaces policy: one centralized server stores the documents that are authored by a series of DVS clients. Nonetheless, DVS is based on the peer-to-peer repositories policy in order to support the flexible placement of artifacts. Typically, a group of collocated authors is responsible for the content of certain parts of a document. For performance reasons it makes sense to locate those parts of the document in a physical repository that is close in proximity to the authors. Particularly when the use of a single, central, physical repository leads to unsatisfactory performance, using DVS with a logical repository that consists of two or more physical repositories can be extremely beneficial.

DVS is centered around the composition policy in order for its users to be able to

decompose a single document into multiple parts. Such a decomposition allows many concurrent modifications, each to a different part of the document. Meanwhile, the use of locking ensures that conflicts are avoided.

The standard composition policy discussed in Section 5.1.2 is based on the standard checkout/checkin policy discussed in Section 5.1.1. The composition policy used by DVS, however, uses the variant of the checkout/checkin policy that disallows the creation of branches (see Section 5.1.1.3). This deviation is deliberate in order to support the linear evolution that is desired for both the individual parts of the document as well as the document itself.

7.1.2 Implementation

DVS is fully implemented in roughly 3,000 lines of C source code.[1] Its functionality is shown in Figure 7.1. Thirteen commands are supported. Given the desired similarity of DVS to RCS, it should not be surprising that the core of DVS is formed by the commands `co` and `ci`. These two commands check out and check in, respectively, a (part of a) document. Compared to RCS, these commands are modified in two ways. First, DVS versions collections. Therefore, both the commands `co` and `ci` operate in the context of collections. For example, before a collection is checked in, its membership is updated with any new versions of its member artifacts that may have been created. In addition, the use of the flag `-R` allows a user to recursively check out (or check in) a collection and its member artifacts.

The second modification regards the fact that no version trees are used by DVS. Instead, when a new version of an artifact is checked in, it is automatically checked in as the latest version of the artifact. Of course, when a new version that is being checked in is not based on the latest version of the artifact, DVS issues a warning and requests

[1] In this chapter, all counts of source code lines are total counts. This includes empty lines and comments.

a confirmation to avoid the possibility of inadvertently breaking the linear chain of evolution.

The command `link` allows a user of DVS to add an already existing artifact to a collection that is checked out. Compared to an addition through the use of the command `ci`, which stores a new artifact in the repository, an addition through the use of the command `link` does not create such a new artifact. Instead, the membership of the collection is updated to include the existing artifact. Typically, this command is used to include an artifact that is stored in a remote physical repository. As a result, a logical repository with artifacts is created (or extended).

The next five commands are rather trivial. The command `unlink` removes an artifact from a collection. The command `lock` locks an artifact, while the command `unlock` unlocks an artifact. The command `list` presents an organized view of the membership of a collection. The view can be configured to include auxiliary types of information. Specifically, it is possible to view whether a member artifact is locked and whether a member artifact is a collection or an atom. The command `close`, finally, can be used to remove artifacts from a workspace.

The next two commands, `log` and `setlog`, can be used to manage the audit trail of an artifact that is being versioned by DVS. The command `log` displays the series of comments that have been associated with each subsequent version of an artifact. These comments typically have been associated by the command `ci`, which records the log message that it requests from a user upon checkin. Alternatively, the command `setlog` can be used. After an artifact has been checked in, this command can be used to explicitly attach or update a log message.

The commands `printlocks` and `whatsnew` can be used to obtain information about the artifacts that are being versioned. Specifically, the command `printlocks`, when used on a collection, searches the repository to determine which versions of the member artifacts of the collection are locked. If the flag `-R` is specified, the command

```
ANDRE -- serl 138% dvs
DVS v.  1.3.1
Copyright (c) 1997-1999 Software Engineering Research Laboratory
Department of Computer Science, University of Colorado at Boulder.

Usage:  dvs <command> [options...]
commands and options:
  co           [-R] [-f] [-l] [-last] path [path ...]
  ci           [-R] [-l] [-m message] path [path ...]
  link         [-last] path [path ...]
  unlink       [-f] path [path ...]
  lock         [-last] path [path ...]
  unlock       [-last] [-f] path [path ...]
  list         [-v] [-o <filter>] [path ...]
               <filter>=[cwbo]+ (collection workspace both(w+c) other)
  close        [-f] path [path...]
  log          [-n <num>] path [path ...]
  setlog       [-m message] path [path ...]
  printlocks   [-R] [-v] path [path...]
  whatsnew     [-R] [-v] path [path...]
  sync         [-R] [-f] path [path...]

To specify your NUCM server, set NUCMHOST and NUCMPORT appropriately.
The default values are localhost and 1234 respectively.

You can also tell DVS to redirect its requests through a proxy by setting
DVS_PROXY_MAP to point to a proxy map file.
```

Figure 7.1: DVS Functionality.

operates recursively. This makes it possible for a user to obtain an overview of all the artifacts that are locked, either by the user itself or by any of the other users. The command `whatsnew` is complementary to the command `printlocks`. It provides a user with an overview of those artifacts that are open in the workspace but that are outdated with respect to the current state of the artifacts in the repository. This occurs when another user has checked in a new version of an artifact that has not yet been placed in the workspace.

The last command, `sync`, allows a user to synchronize a workspace with the contents of the repository. Specifically, if new versions of artifacts have been checked in, or artifacts have been added to or deleted from a collection, this command should be used to make sure that the contents of the user workspace are up to date. To avoid the loss of changes, the command does not overwrite artifacts that have been checked out by the user.

The set of thirteen commands satisfies the requirements laid out for DVS in Section 7.1. Conflicts are avoided, since DVS uses locking to guarantee exclusive access to an artifact. Still, parallel work is facilitated, because a document can be partitioned into multiple parts that each can be manipulated and evolved separately. Moreover, the policy of DVS enforces a linear evolution of artifacts while still allowing, as an informed and conscious exception, to base a new version of an artifact on an earlier version than the latest.

By providing each user with a private workspace, users can make changes in isolation. Because of the nature of the workspaces as defined by the abstraction layer, it is possible for workspaces to operate in a disconnected setting. Specifically, a workspace can be placed on a machine that is not continuously connected to the set of machines on which the logical repository with artifacts resides. Only when new versions of an artifact are checked in, or when a workspace is synchronized, is network connectivity required between the machine that contains the workspace and the machines that contain the

logical repository.

DVS shields a user from operation in a distributed environment. With the exception of potential network delays, a user is typically not aware of the physical distribution and location of the artifacts on which it operates. Only two occasions warrant user awareness. First, when a workspace is initially constructed using the command `co`, a user needs to supply the full name of the top-level artifact that populates the workspace. This name includes the physical location of the repository where the artifact resides. The second occasion regards the command `link`. Since it adds an existing artifact, the full name of the artifact to be added is, once again, required. All of the other commands provide transparent distribution to the user. If needed, the commands locate and operate on artifacts in remote physical repositories. No user intervention or direction is necessary.

The final requirement, ease of use, is also satisfied by DVS. Typical use has a user synchronizing their workspace in the morning, checking out and checking in artifacts throughout the day, and repeating the same sequence the next day. With the exception of the synchronization step, this kind of use closely resembles the model provided by RCS.

7.1.3 Observations

DVS has been used on a number of occasions over a time frame of two years. In some cases, it has been used to manage the authoring of grant proposals by collaborators located in Colorado, California, and Hawaii. In other cases, it has been used to manage research papers authored by collaborators located in Italy, California, and Colorado. In fact, on some occasions collaborators have traveled, with their workspace, from location to location, while periodically checking in new artifacts and synchronizing their workspace. Throughout these experiences, DVS has proven to be reliable in use, even while managing thousands of checkins, checkouts, and synchronizations.

DVS possesses some characteristics that illustrate the power of the abstraction layer. First, no special code needed to be developed for DVS to operate in a wide-area setting. To provide for its distribution, DVS relies entirely on the mechanisms defined by the abstraction layer and included in the NUCM client implementation. Specifically, to store different artifacts in different physical locations, DVS fully leverages the peer-to-peer facilities that are provided by NUCM servers.

The second advantage in using the abstraction layer in the construction of DVS shows itself in the number of lines of source code needed to develop DVS. Approximately $3,000$ new lines were needed to create the full functionality of DVS. The newly written source code mostly deals with the text-based user interface, the recursive operations on workspaces, the proper locking of artifacts, and the storage of metadata about the artifacts that are versioned. Other functionality, such as distribution, collections, and versioning, is inherited from the NUCM implementation.

The third and final advantage in using the abstraction layer demonstrated itself in the evolution of the functionality of DVS. The initial set of functionality of DVS did not quite fulfill all desired needs. Therefore, DVS needed to evolve. New commands were added and flags and default behavior of commands were changed. It turned out that these additions and changes could be made easily. No complete redesigns or overhauls were needed. In fact, because of the storage compatibility among versions of DVS, it even turned out that the changes could be deployed in an incremental fashion. While some collaborators still used an older version of DVS, others experimented with the functionality provided by the new version. In this process, the NUCM repository required no downtime and slightly different policies could be used by multiple authors at the same time. Clearly, the separation of storage from policy proved to be invaluable in this situation.

7.2 SRM

SRM (Software Release Manager) [vdHHHW97] is a tool that was developed to address a need for multiple, collaborating organizations to coordinate their release process. Specifically, systems built at one organization depended on other systems built at other organizations. Using the Web or FTP as the release mechanism led to a rather laborious and error-prone release process. Specifically, an inability to properly and consistently track dependencies among systems was the source of a large number of mistakes. SRM addresses this problem. It supports the release of "systems of systems" from multiple distributed sites. In particular, SRM tracks dependency information to automate and optimize the retrieval of system releases. Developers are supported by a simple release process that hides distribution. Users are supported by a simple retrieval process that allows the retrieval, via the Web, of a system of systems in a single step as a single package.

Although SRM is not a traditional CM system that stores and versions source code, it has many similarities to a CM system: it needs to manage multiple (versions of) releases, it needs to manage dependencies among these releases, and it needs to store metadata about the releases. Combined with the need for a distributed repository that allows multiple sites to collaborate in the release process, these similarities led to the choice of NUCM as the platform upon which to build SRM.

Similar to DVS, the creation of SRM was guided by a number of requirements. Specifically, its distributed operation has to adhere to the following.

- **The federation of physical repositories that form the logical repository used by SRM should be easy to change.**

  Since new organizations may join the federation or existing organizations may leave the federation, it should be easy to add or remove a physical repository.

- **SRM should hide distribution.**

Specifically, SRM should hide distribution from developers when they specify dependencies and it should hide distribution from interested parties when they retrieve a system release and its dependencies.

- **SRM should store the systems that are released by a particular organization at the physical repository maintained by that organization.**

  Specifically, since the storage space required by certain system releases may be rather large, and since it is likely that a system released by a particular organization is most often retrieved via the Web pages of that organization, a system release should be managed and stored in the physical repository that belongs to the organization.

7.2.1 Design

To address these requirements, a rather unique design is employed by SRM. A part of this design is illustrated in Figure 7.2. Shown are three sites, namely Boulder, Rotterdam, and Milano. Only the sites in Boulder and Rotterdam are currently part of the SRM repository. Key to the design is the presence of two types of collections. The first type is called `my_releases` and is present in each physical repository. In this type of collection, local releases are stored. For example, the site at Boulder has released a system called `WordProcessor`, which is stored in the collection `my_releases` at that site. Similarly, the site at Milano has released a system called `GUI-lib` that is stored in its local collection `my_releases`. This kind of use of a local collection addresses the last requirement of SRM, which regards the placement of system releases in the appropriate physical repositories.

The second type of collection addresses the second requirement for SRM. This collection, called `all_releases`, is stored in the master repository of the SRM federation and has as its members all system releases at all participating sites. The availability of

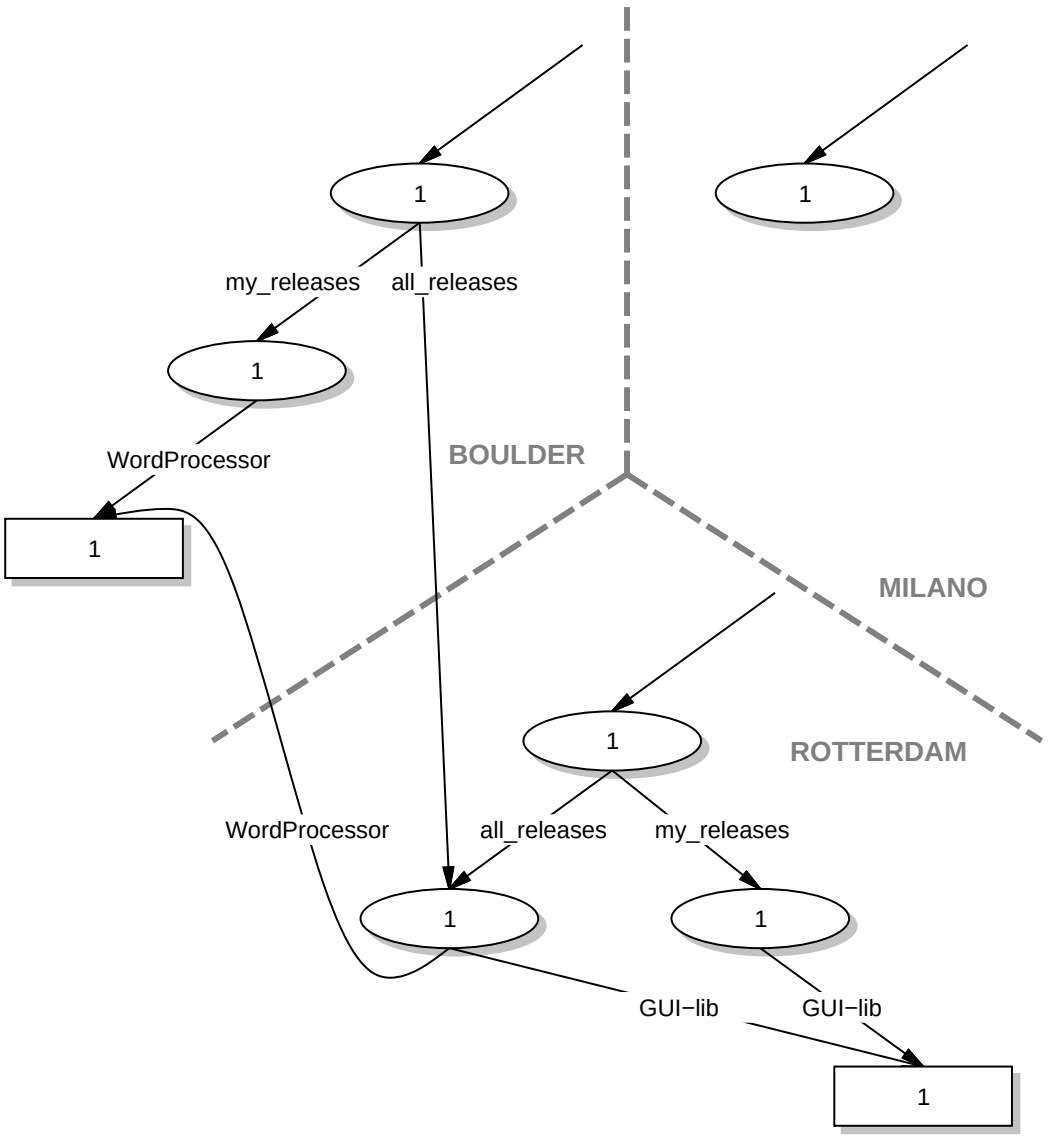


Figure 7.2: Federated SRM Repository Before Milano Joins the Federation.

this collection provides each user of SRM with an overview of, and access to, all systems that have been released without having to know where these systems reside.

The combination of the collections `my_releases` and the collection `all_releases` addresses the first requirement. Specifically, when a new organization or site joins the SRM federation, all it needs to do is create a local collection `my_releases` and link to the global collection `all_releases`. The result of these actions is shown in Figure 7.3 after the repository in Milano has joined the federation. Dashed lines indicate the new relationships and artifacts that have been put in place as compared to the repository presented in Figure 7.2.

To leave a federation, once again two steps are required. First, an organization has to remove its system releases from the collection `all_releases`. Subsequently, the collection `all_releases` itself has to be removed from the physical repository that leaves the federation. Of course, consistency constraints must be enforced in this process. In particular, dependencies cannot be broken when a site leaves the federation. Therefore, the full design of the repository used by SRM maintains dependency counters to enforce a consistent state. The details of this full design, which includes many other features not presented here, are discussed by Smith [Smi99].

Note that the essence of the repository design of SRM is similar to the creation of a federation in the peer-to-peer repositories policy (discussed in Section 5.2.2). One important difference exists. Whereas the peer-to-peer repositories policy allows any collection in any repository to contain any artifact in any other repository, the repository design of SRM is based on the creation of a very specific federation in which connectivity is limited to a few places in the federation of repositories. In effect, the repository design of SRM can be viewed as a special case of the peer-to-peer repositories policy in which a number of rules define a precise schema. This schema restricts which kinds of artifacts may be stored, how these artifacts may relate, and how they can evolve. The rules, of course, are encoded in the policy that is implemented by SRM.

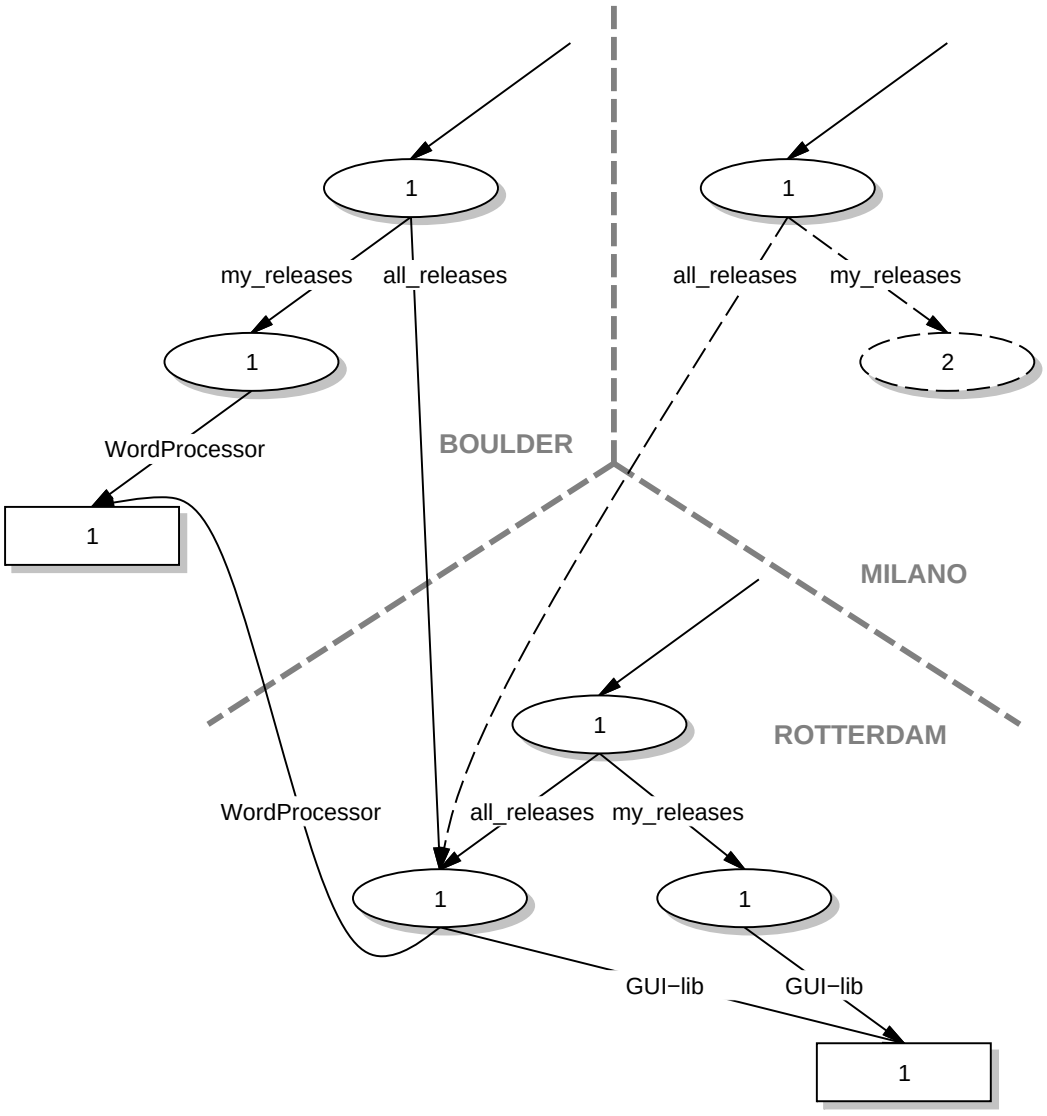


Figure 7.3: Federated SRM Repository After Milano Joins the Federation.

7.2.2 Implementation

The implementation of SRM hides all details of the presented repository design and its associated versioning policy. In fact, SRM operates in such a manner that all its user actions are based on simple forms. Specifically, developers fill out a form that describes the system to be released. On this form, system releases are identified by a name and release number. This release number is mapped, by SRM, onto a version number in the repository.

The fact that dependencies may span multiple sites is also hidden from the developers. They simply choose the desired set of dependencies from the list of systems that have been released so far. SRM obtains this list using the function `nc_list` on the collection `all_releases` in the logical repository.

User interested in obtaining a system of systems are similarly unaware of the physical distribution of artifacts. They are presented with a Web page from which they can select the system in which they are interested. SRM then takes care of retrieving the system and all of its transitively dependent systems from the appropriate physical repositories, packs up the system of systems in a single archive, and hands the archive to the user.

To achieve the high-level interface, the architecture of SRM is based on an internal storage layer that translates such entities as system releases and dependencies (as well as many others defined by the full design of SRM [Smi99]) into collections, atoms, and attributes. This storage layer consists of about 10 percent of the approximately 17,000 lines of source code that constitute SRM. Of this 10 percent, only about 200 lines explicitly manage the distributed nature of an SRM repository. Specifically, these lines implement the joining and leaving of a physical repository. The rest of the code implementing the storage layer constitute the rules that govern the repository contents. The remaining 90 percent of the source code of SRM is concerned with the implementation

of an appropriate user interface.

7.2.3 Observations

SRM has been in actual use since 1997. Its primary use has been as the software release manager for the software created by the Software Engineering Research Laboratory of the University of Colorado. This represents a local use of SRM, since all participants reside at the same site. In an alternative setting, SRM has served for over a year as the central release site for the participants in the DARPA EDCS (Evolutionary Design of Complex Software) program. In this setting, participants from all over the United States have released artifacts to a single, central SRM repository located at the Software Engineering Institute. Although neither use takes advantage of the full power of the design of SRM, both have shown the applicability of SRM and the reliability of NUCM, even in a widely distributed setting.

Although SRM is not a traditional CM system, it still uses the functionality provided by the abstraction layer to the fullest. Collections, versioning, advanced naming, and peer-to-peer distribution are all used in the design and implementation of SRM. As such, SRM is an excellent example of a system that uses many of the features of the abstraction layer.

The main advantage in using the abstraction layer to construct SRM is the fact that distribution could be isolated. Only a small part of the complete implementation explicitly deals with distribution. The remainder of the implementation is concerned with the actual functionality of SRM itself and, in fact, relies on the distribution transparency provided by the internal storage layer of SRM. This particularly exhibits itself in the functionality of adding a release to the SRM repository. This can simply be programmed as an addition to the local collection `my_releases` and the global collection `all_releases`. Since NUCM tracks the physical location of these collections, the addition can be programmed completely in terms of interaction with just the local physical

repository.

7.3 WebDAV

WebDAV [Whi97] is an emerging standard that proposes to add authoring and versioning primitives to the HTTP protocol [FGM+98]. In particular, the standard proposes extensions in the following five areas.

- **Properties.** To be able to describe Web resources, WebDAV proposes the creation of new HTTP methods that add properties (or attributes) to Web resources, as well as methods to query and retrieve the properties.

- **Collections.** To be able to structure Web resources into higher-level constructs, WebDAV proposes the creation of new HTTP methods that allow Web resources to be grouped into collections, as well as methods that change the membership of collections.

- **Name space management.** To be able to efficiently move, copy, and delete Web resources, WebDAV proposes the creation of new HTTP methods that manipulate the Web name space.

- **Locking.** To avoid multiple entities updating a single Web resource in parallel and consequently losing changes, WebDAV proposes the creation of new HTTP methods that allow Web resources to be locked and unlocked for exclusive write access.

- **Version management.** To be able to keep a history of Web resources, WebDAV proposes the creation of new HTTP methods that allow Web resources to be versioned.

Two observations can be made with respect to WebDAV and the abstraction layer. A first observation is that, although the objective of WebDAV (providing an infrastructure

for distributed authoring and versioning) is slightly different than the objective of the abstraction layer (providing a distributed repository to construct configuration management policies), the interface methods that have been proposed by both are strikingly similar. Only two major differences exist. First, the abstraction layer includes a naming model that uses version qualifiers to navigate in the version space, whereas WebDAV leaves the versioning aspects of naming undefined. Second, WebDAV specifies a particular versioning policy, namely the version tree that is modified by checking out and checking in artifacts, whereas the abstraction layer is generic with respect to versioning policies.

The second observation is that, because of the similarity between the abstraction layer and WebDAV, it seems advantageous to implement WebDAV using the abstraction layer. Properties and locking can be mapped onto attributes, collections are identical, name space management maps onto collections and the physical distribution of artifacts, and version management can be implemented as a CM policy.

The remainder of this section describes the details of a prototype of WebDAV as implemented with the testbed. It should be noted, however, that the prototype implements a subset of the functionality prescribed by an earlier version of the standard and is clearly out of date with respect to its current version.

7.3.1 Design

The high-level design of the WebDAV prototype is shown in Figure 7.4. The core of the design is formed by an integration of an HTTP server with the NUCM client. As WebDAV requests come in from WebDAV-enabled browsers, the HTTP server either handles a request itself (in case a regular HTTP request is made) or interprets a request and uses the NUCM client to implement the versioning policy (in case a WebDAV request is made). Note that the NUCM client interacts with only a single NUCM server. Since each HTTP server is responsible for managing its own name space and

does not interact with any other HTTP servers, the underlying storage mechanism (in this case NUCM) does not have to be distributed.

Most of the new HTTP methods translate into direct calls to the programmatic interface, but some require additional work. In particular, the versioning routines of WebDAV prescribe a policy that is based on a version tree. For this part, a basic checkout/checkin policy like the one described in Section 5.1.1 is adopted. The only change regards the use of workspaces. Whereas in the standard checkout/checkin policy these are used as the working area for a user, in the WebDAV prototype the workspace is used to communicate between the NUCM client and the HTTP server. Since artifacts are further forwarded by the HTTP server to the WebDAV-enabled browser in which the user manipulates the artifacts, no need exists for the artifacts to remain in the workspace after a WebDAV operation. Therefore, the workspace management mechanism of the standard checkout/checkin policy is adjusted to remove artifacts after a checkout and open artifacts before a checkin.

7.3.2 Implementation

Because WebDAV is an evolving standard [GWF+99] and the prototype was built almost two years ago, the implementation described here is outdated with respect to the current version of the standard. Nonetheless, the prototype can be used for the purpose of illustrating the strength of the abstraction layer.

The actual prototype implementation only addresses the locking, versioning, and collection aspects of the WebDAV standard. Properties are ignored, since they map easily onto the attributes of the abstraction layer. The name space management functions are not included since they did not exist at the time.

The WebDAV prototype consists of about $1,500$ lines of source code that not only implement the policy, but also a very simple WebDAV-aware HTTP server and a WebDAV client. Specifically, the WebDAV client is implemented as a Java applet

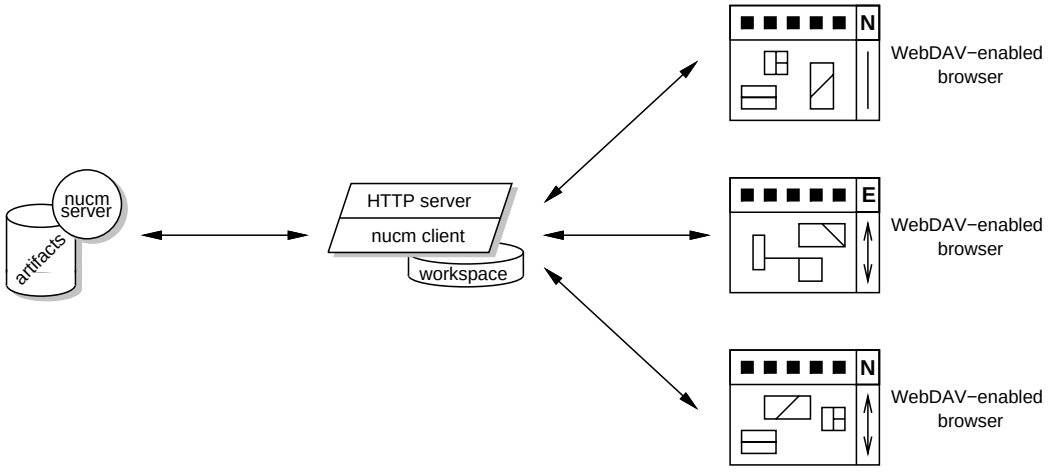


Figure 7.4: Architecture of WebDAV Prototype.

that can be loaded into a standard Web browser. The Java applet makes requests to the WebDAV-aware HTTP server. This server, in turn, implements collection and checkout/checkin functions to provide the desired behavior prescribed by the standard.

The functionality provided by the WebDAV-aware HTTP server is by far not complete. It is only an implementation for the purpose of demonstrating the prototype and leaves out almost all of functionality that is typically provided by an industrial-strength version.

7.3.3 Observations

Unlike DVS and SRM, the WebDAV prototype has not been used other than for simple demonstrations. Nonetheless, the prototype still offers two valuable lessons with respect to the value of the abstraction layer. The first lesson lies in the fact that the checkout/checkin policy incorporated in the prototype actually represents a reuse of an earlier implementation of the checkout/checkin policy [vdHHW96]. Since only a few modifications needed to be made with respect to the handling of workspaces, the policy aspects of the prototype could be completed within a single day.

The remainder of the prototype was implemented within a few weeks. Admittedly, the experimental implementation does not cover all the functionality of the WebDAV standard. However, the limited amount of code that needed to be developed and the rapid development time demonstrate an important aspect of the abstraction layer: it can be used to support the rapid development of prototype CM policies. The development of a standard like WebDAV can particularly benefit from such an approach, since the ramifications of specific policy decisions can almost instantly be tried out with an actual implementation.

7.4 Summary

This chapter has demonstrated the utility and validity of the abstraction layer. The utility is demonstrated by the fact that the NUCM prototype can actually be employed as a useful basis upon which particular CM systems are constructed. Characteristics of these CM systems are that they operate in a distributed nature, store different kinds of artifacts, and all utilize a different kind of CM policy. The presence of these characteristics shows that the abstraction layer indeed provides support in those areas that it is meant to support.

The validity of the abstraction layer is demonstrated by the fact that all three CM systems could easily be implemented. The effort required was small. In particular, the effort required to create those parts that implement the various CM policies was relatively small when compared to existing CM systems. DVS consists of only about $3,000$ lines of source code, the policy of SRM is roughly $1,700$ lines, and the prototype of the WebDAV standard is only $1,500$ lines. This represents a significant reduction when compared to other CM systems.

The most important advantage, however, is the fact that this reduced effort facilitates the exploration of new CM policies that can rapidly be tailored to the situation at hand. The WebDAV prototype represents an excellent example. As a particular standard was being defined, a prototype implementation of the standard could be rapidly constructed. Potential evaluations resulting in adjustments and refinements to the policy could have been performed had the NUCM prototype been available to the group defining the standard. DVS provides another example. Although its first implementation did facilitate a group of distributed collaborators to jointly author a document, its evolution over time has resulted in a complete CM system that fully addresses its requirements.

Chapter 8

Related Work

In its many years of existence the discipline of configuration management has produced numerous industrial and research systems. Some provide only version control facilities, (e.g., RCS [Tic85], SCCS [Roc75], Sablime [Bel97]). Others provide more complete configuration management solutions (e.g., CVS [Ber90], CoED [BLNP98], Perforce [Per98]). Yet others provide integrated environments that incorporate process management and/or problem tracking facilities (e.g., Adele [EC94], ClearCase [Atr92], Continuus [Con94]). A similar categorization can be made with respect to distribution. Some of the CM systems are only suited for use at a single site (e.g., EPOS [Mun93], ShapeTools [ML88], SourceSafe [Mic97]). Others incorporate a simple (sometimes Web-based) client-server interface (e.g., DCVS [Ber90], Perforce [Per98], WWCM [HLRT97]). Yet others provide more advanced distribution mechanisms such as replication or workspace distribution (e.g., ClearCase Multisite [AFK+95], Continuus DCM [Con98], NeumaCM+ MultiSite [Neu98], PVCS SiteSync [INT98a]).

To understand the position of the testbed in the large spectrum formed by all these systems, this chapter presents related work from three different points of view. First, the chapter examines the architectural evolution of CM systems, culminating in the architecture imposed by the testbed introduced in this dissertation. Then, the chapter discusses some other research systems that can be considered alternatives to the testbed. Finally, the chapter briefly contrasts the testbed to several other approaches

not originating in the field of configuration management.

8.1 Architectural Evolution

Figure 8.1 shows the evolution over time of the high-level architecture embedded in CM systems. The earliest architecture, shown in Figure 8.1a, distinguishes itself by the fact that it tightly integrates the storage mechanism employed by the CM system with the CM policy that is presented to the user. First utilized by such CM systems as SCCS [Roc75] and RCS [Tic85], this architecture is still the most popular to date. Such well-known and widely-used systems as PVCS [INT98b], SourceSafe [Mic97], and Perforce [Per98] have been constructed this way. Certainly, an advantage of the architecture is that it allows a CM system adhering to it to optimize the storage for the needs of the CM policy. Perforce, for example, is known and actually marketed for its fast operation, which can be attributed to a storage facility that is highly tuned to its change package policy. A disadvantage of the architecture, however, is that a CM system that is based on it tends to be rather inflexible [Dar96]. For example, PVCS and SourceSafe have been on the market for a long time, but the functionality provided by each is stagnant and has not changed significantly for several years now. In today's market, this can be a serious competitive disadvantage.

Another disadvantage is that the creation of a CM system adhering to the monolithic architecture requires an implementation completely from scratch. Typically, the storage requirements are unique and must be tailored to the CM policy. It is likely, therefore, that no suitable infrastructure can be reused. Consequently, the implementation of all the infrastructure has to be performed by the development organization. The effort involved in this process may range from several months for a relatively simple CM system to a number of years for a full-fledged, feature-rich CM system.

Despite these two disadvantages, most CM systems are still built with a monolithic architecture. This is exemplified by both CoED [BLNP98] and DSCS [Mil97],

each of which was (recently) developed in this manner.

As CM systems have become more mature and advanced, some have turned towards using a commercial, generic database as the underlying storage mechanism (e.g., ClearCase [Atr92], Continuus [Con94], TrueCHANGE [Sof94b]). Illustrated in Figure 8.1b, the advantage of this architecture is that a database provides a reliable and reusable platform that offers such services as schema definition, constraint enforcement, transactions, concurrency control, and rollbacks. These services no longer have to be implemented by the organization that develops the CM system. Generally, a reduction in development effort and an increase in reliability result from the adoption of this type of solution.

Disadvantages of the use of a database as the storage mechanism are not immediately visible. However, circumstantial evidence seems to suggest that they do not necessarily provide a truly reusable platform. First, many of the new CM systems that have recently been developed have chosen not to use a database. Instead, they have opted to develop proprietary storage mechanisms. Second, the databases used by ClearCase and TrueCHANGE are heavily modified versions that provide optimized and tailored access. Some unknown set of problems caused the vendors of these CM systems to actually acquire the rights to the source code of the respective database used in order to solve these problems.

Continuing the evolutionary pattern, the testbed contributed by this dissertation represents the next step. Instead of providing just generic database services, Figure 8.1c illustrates that the testbed provides a repository that is highly specialized towards configuration management. CM policies are constructed by reusing the implementation of a repository model and programmatic interface that are specifically designed to support configuration management policy programming. As compared to the use of a generic database, the model and interface defined by the abstraction layer raise the level of abstraction with which CM policies can be constructed and thereby facilitate their rapid

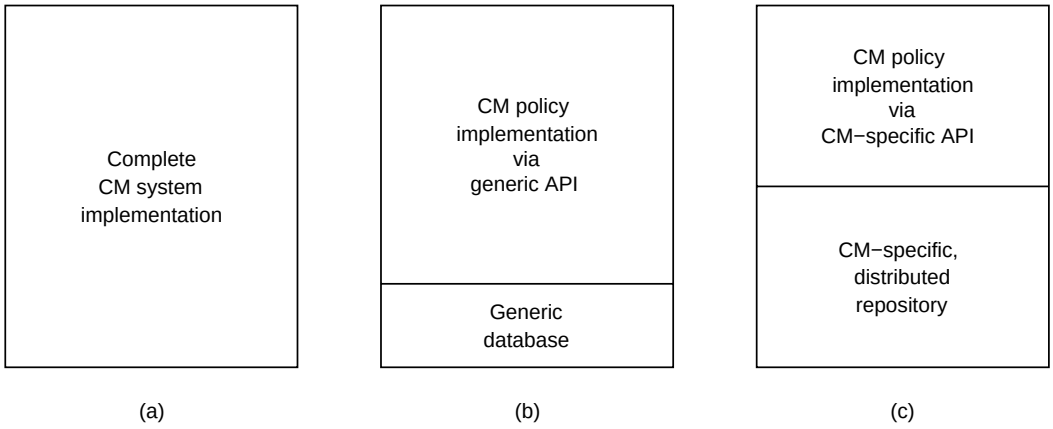


Figure 8.1: Evolution of the Architecture Embedded in CM Systems.

implementation. Still, a disadvantage of the testbed is that a CM system built on top of it may not be as highly tailored and optimized as a solution that is completely designed and build from scratch. Additionally, the testbed lacks exactly the functionality that makes the use of a database as the storage mechanism so attractive. In particular, missing are such services as transactions, rollbacks, and caching. It is hoped that most of this functionality will eventually be incorporated into an industrial-strength implementation of the testbed. Actual CM systems can then be implemented based on the abstraction layer defined in this dissertation. At this point in time, however, the testbed is primarily suited for CM system prototyping and experimentation.

Since the first inception of the testbed [vdHHW96], several other papers have recognized and advocated the use of a CM-specific repository as an architectural layer upon which CM systems are build. Conradi and Westfechtel identify the availability of an instrumentable version engine, supporting both a locking-based and a merging-based versioning style, as an important requirement for further advances in the field of configuration management [CW97]. Similarly, Zeller identifies a three-tiered architecture, consisting of CM primitives, CM protocol, and CM policy, as the pivotal basis upon which further CM research and development should take place [Zel97].

The layered architecture proposed by Conradi and Westfechtel is closest in nature to the testbed. In fact, its six architectural layers (i.e., basic delta storage, version rules and/or version graphs, change-based and state-based version model, product model and data model, transaction support and model, uni-version workspace and virtual file system) are remarkably similar in incremental functionality to the nine layers identified in Figure 6.2. The only difference is that Conradi and Westfechtel incorporate a richer generic versioning model, consisting of the product model and data model, instead of the versioned directed graph upon which the testbed is based. Unfortunately, their architecture has not been implemented and only remains a sketch.

The approach taken by Zeller is rather different from that of the testbed discussed

in this dissertation. Based on feature logic, a single unified formalism is proposed in which a variety of CM policies are implemented [ZS97]. Although the formalism is powerful enough to unify and integrate all four CM policies identified by Feiler [Fei91a], it lacks a high-level programmatic interface and does not provide any support for distribution.

An important observation that should be made is that no single architecture is right for all situations. For example, in certain cases a monolithic architecture is desired over the use of a database or even over the use of the testbed introduced in this dissertation. Perforce [Per98] provides an excellent example of this situation. Perforce is designed and implemented for speed and leverages the close connection between the storage mechanism and CM policy to its fullest advantage in obtaining as fast a performance as possible. Given the generic nature of the testbed, building Perforce on top of it would be an unwise decision.

Nonetheless, for the purposes that the testbed was designed it is by far the best choice of the three architectures. The high-level programmatic interface, combined with the generic repository model, facilitates rapid exploration of new CM policies that manage arbitrary kinds of artifacts and operate in a distributed setting. Additionally, these CM policies may evolve as requirements or desired functionalities change. It should also be noted that, based on the experience with NUCM, it turns out that the performance of a CM policy that uses NUCM oftentimes is satisfactory. Only when the performance becomes a bottleneck, the final CM policy that results after all experimentation has finished should be reimplemented in order to optimize the performance as much as possible.

8.2 Alternative Abstractions

A small number of CM systems are actually built on top of an existing CM system. The two most prominent systems exemplifying this kind of reuse are Asgard [MC96],

which extends ClearCase [Atr92] with an activity-based software change management process, and Ragnarok [Chr98], which uses RCS [Tic85] as the underlying storage mechanism for its architecture-based CM system. However, neither ClearCase nor RCS can be considered a reusable platform upon which to build a wide variety of CM policies. ClearCase is a CM system itself, and its policy and implementation are far too restrictive to be generic. The creation of Asgard concerns a very specific, ClearCase-oriented solution to a problem that is complementary to the functionality already provided by ClearCase.

For different reasons RCS can be dismissed as a generic platform. Although it provides parts of the functionality defined by the testbed , RCS only addresses a small subset of that functionality. Moreover, its original implementation lacks a programmatic interface. RCE [HT97] solves that problem but still only provides an interface that is RCS-specific and does not include such facilities as collections and distribution.

For similar kinds of reasons, most other CM systems are not suitable as a reusable platform for configuration management programming. The testbed defined in this dissertation is unique in providing this kind of functionality. Nonetheless, several approaches deserve to be mentioned because they also fall into the category of Figure 8.1c. These systems are CME, CoMa, Gradient, and ScmEngine. CME [HLRT97] is an extension that adds collection management to RCE [HT97]. CME is similar to the testbed in that it provides an architectural separation of the repository from the actual system that stores and versions the artifacts. However, two significant differences exist. First of all, the programmatic interface of CME is not generic with respect to CM policies. It only contains functions that implement a simple variant of the composition policy. The second difference is that CME is not distributed. It only interfaces to a single repository at a single site. Thus, whereas the testbed provides support for the construction of a large variety of distributed CM policies, CME only provides support for the construction of centralized CM policies that are based on composition.

CoMa [Wes96] is perhaps the one system that is closest in nature to the functionality provided by the testbed. CoMa introduces graph rewriting as a method of constructing specific CM policies. Based on a composition model, it utilizes graph rewriting rules to assert and enforce constraints. These constraints govern the evolution of the artifacts that are managed. The goal of CoMa is to evolve the interrelated sets of heterogeneous artifacts that are created throughout the software life cycle. Naturally, it therefore shares some of its goals with the testbed. Specifically, it needs to manage different kinds of artifacts and it needs to tailor its CM policy to the artifacts that are managed. As compared to the testbed, however, CoMa is limited in that it only supports the construction of variations of the composition policy. Moreover, it does not support the distribution of artifacts over multiple physical locations. Finally, CoMa does not incorporate a workspace model. It is solely a specification of the storage model that is involved. Thus, even though CoMa is more generically applicable than CME, it is similarly limited in that it only supports centralized CM policies that are based on composition.

Gradient [BKR96] is a CM repository that is based on automatic replication. Each update that is made to an artifact is broadcasted instantly as a delta to all replicas. Because Gradient only allows incremental modifications to the artifacts it manages, and furthermore assumes that modifications are independent of each other, it permits simultaneous updates to a single artifact at multiple sites. Gradient is similar in spirit to the testbed. It provides an architectural separation of the storage mechanism from the CM system that uses it. But, as with CME, Gradient only supports a specific policy, both with respect to distribution (where it only supports replicated repositories) as well as with respect to CM policy (where it only supports the checkout/checkin policy).

ScmEngine [CPT97] is a distributed CM repository based on the X.500 directory protocol. X.500 directory entries contain metadata describing the artifacts that are stored in physical repositories. Access servers leverage the standard X.500 directory

protocol to create a logical repository that can be accessed by client CM programs. This distribution mechanism is, in essence, the same as the one defined by the distribution model of the testbed. However, the remainder of the repository model and the programmatic interface provided by ScmEngine are significantly weaker than the ones defined in this dissertation. Specifically, the repository model does not include collections and lacks the concept of version qualifiers to navigate in the version space. Similarly, the programmatic interface is very specific and lacks support for the construction of a wide variety of CM policies, only supporting the traditional checkout/checkin policy.

8.3 Other Domains

Outside the domain of configuration management two important lines of work can be identified that are closely related to the work presented in this dissertation. Specifically, the fields of groupware and versioned databases share many of the concerns of the testbed. In groupware, the need for distribution, versioning, and workspaces seems to imply that the testbed layer could be useful in supporting the construction of a groupware system. However, this is not so. Whereas the testbed is based on the principle that workspaces provide isolation from changes made by other users in other workspaces, groupware systems tend to focus on collaborative workspaces (e.g., WebRC [FN97], the generic model of version management for cooperative applications [DRS96]). Especially in the case of distribution, the set of issues involved in supporting each type of workspace is rather different. Consequently, groupware, even though closely related, falls outside of the domain of the testbed.

Versioned databases (e.g., Ode [ABGS91], TVOO [ROY99]) are related to the testbed since the testbed itself can be viewed as a versioned database. In fact, many of the features of the testbed are shared by versioned databases. Nonetheless, an important difference exists, which is the presence of a specific repository model and an associated programmatic interface in the testbed. Whereas these are generic in nature

in a versioned database (e.g., an entity relationship model with SQL), both are highly specialized by the testbed towards configuration management. In essence, one could consider the testbed to be a layer on top of a versioned database that implements a particular schema (the repository model) and provides a number of standard views and operations (the programmatic interface).

Chapter 9

Conclusions

This dissertation addresses the problem of CM system development. Specifically, it presents a first step towards the creation of a reusable platform that facilitates the rapid construction of, and experimentation with, new—potentially distributed—CM systems. Although only a first step, the testbed provides two important benefits to a CM system developer.

- New, prototype, CM systems can be rapidly developed, even if they need to operate in a distributed setting.
- New CM policies can be flexibly explored, even if they depart from traditional assumptions and limitations.

It is expected that, in the short term, the testbed leads to the creation of design methods for CM systems that use an implementation of the testbed in prototyping the desired functionality. Due to the limited reliability and optimization of the current implementation, this prototyping phase would then be followed by a reconstruction of the eventual CM system from scratch. In the long term, it is expected that the implementation of the testbed will evolve into a fully functional, highly reliable, and highly optimized platform that no longer requires such a costly reimplementation phase.

The critical contribution of the testbed is its architectural separation of CM repositories from CM policies. To do so, a precise abstraction layer is defined that consists of

a generic model of a distributed repository and a programmatic interface for implementing, on top of the repository, specific CM policies. Characteristics of the abstraction layer are its policy independence, its ability to manage a wide variety of different kinds of artifacts, its inherent distributed operation, and its ability to support traditional CM functionality.

A prototype implementation of the testbed, called NUCM, is in actual use. Although primarily used by the members of the Software Engineering Research Laboratory at the University of Colorado, some of the policies that are implemented with this prototype are used by others as well. Additionally, the testbed seems to have already influenced some other ongoing efforts. In particular, evidence seems to suggest that Perforce [Per98] has adopted an old version of the distribution model [vdHHW96], that TrueCHANGE (formerly Aide de Camp [Sof94b]) has adopted software release management as part of its suite of CM products, and that WebDAV [Whi97] has adopted the collection mechanism provided by the abstraction layer.

The remainder of this chapter is structured as follows. First, the strengths and weaknesses of the testbed are summarized. Then, the chapter concludes with a brief discussion of future work.

9.1 Strengths

The strength of the abstraction layer lies in its ability to support the rapid creation of prototype CM systems, as well as in its ability to support the flexible exploration of CM policies. A comprehensive evaluation, consisting of mappings of ten CM policies onto the testbed, as well as implementations of three CM policies onto the implementation of the testbed, demonstrates that the testbed achieves its goals. Specifically, characteristics of the example CM policies are that they operate in a distributed setting, store and manage different kinds of artifacts, and all utilize a different kind of CM policy.

Two additional benefits are identified. First, not only can the testbed itself be reused in the programming of CM policies, but, as it turns out, most of the CM policies that are created with the testbed possess a certain amount of extensibility as well. Therefore, new CM policies can often reuse parts of repository designs and programmatic logic from other, existing CM policies. Closely related to this observation is the second additional benefit. The ease with which variants of example policies can be mapped, as well as the ease with which the CM policies that are actually implemented can be evolved, indicates that the testbed not only provides a reusable abstraction upon which to construct prototype CM systems, but also possesses properties that allow these prototype CM systems to evolve as the required functionality changes. This represents an important benefit that warrants further exploration beyond the scope of this dissertation.

9.2 Weaknesses

Of course, to achieve its strengths, any generic abstraction layer typically sacrifices certain properties, leading to the possibility that the abstraction layer may not be applicable in certain situations. The testbed, naturally, also has some of these weaknesses. In particular, the following three weaknesses can be identified.

- **The testbed is not applicable to the management of fine-grained artifacts.**

  In particular, the current access model prescribes that artifacts be accessed via the file system. Clearly, this is an undesirable situation if a multitude of fine-grained artifacts are managed. The overhead of accessing these artifacts via the file system is simply too high.

- **The testbed leads, at times, to heavy-weight solutions.**

  An example of this undesirable effect is provided by the long transaction pol-

icy discussed in Section 5.1.3. The simplest way to create this policy with the testbed is to store each long transaction in a separate physical repository. Clearly, this is a solution that requires additional storage space and a multitude of NUCM servers running simultaneously if implemented with the NUCM prototype. Compared to a proprietary implementation that optimally stores and manages long transactions, the efficiency of the solution with the testbed is inferior.

- **Certain experiments may not be possible with the testbed.**

  The testbed and its prototype implementation hide certain aspects from the developer of a CM policy. For example, the current implementation of the testbed does not optimize any of its communication among clients and servers. As a result, an experiment in which a communication-intensive CM policy is placed on a large number of widely distributed clients may simply not be achievable. These and other limitations, caused by the fact that the testbed and its implementation simply do not allow a developer to tune communication parameters, may prohibit the applicability of the testbed in certain situations.

9.3 Future Work

The work discussed in this dissertation only represents the beginnings of a new and promising approach to the creation of CM systems. Much work remains to be done. In particular, immediate research questions can be identified in investigating whether the abstraction level provided by the testbed can be raised even further, whether the functionality provided by the testbed can be broadened, whether the testbed can be applied to different domains, and whether extensions can be made that improve the functionality of the testbed without changing its model or interface. Each one of those questions is briefly discussed below.

- **Can the level of abstraction provided by the testbed be raised?**

  Even though the testbed provides significant benefits in creating new CM policies, an examination of the policies that have been constructed seems to indicate that it might be possible to further raise the level of abstraction with which CM policies are programmed. In particular, as evidenced by the descriptions, repository models, and core policy designs discussed in Chapter 5, the example policies share some parts of their repository designs as well as pieces of their functionality. Therefore, a question that is worth considering is whether it is possible to create a high-level CM policy programming language in which constructs encapsulate parts of the example CM policies introduced in Chapters 5 and 7. Such a language might be declarative (e.g., a CM policy is created by specifying desired sets of properties, such as "locking" or "composition") or programmatic in nature (e.g., a CM policy is created by programming in terms of high-level constructs, such as "lock" or "compose"), but it is unclear at this moment what the exact nature of such a language will be or even whether such a language can actually be constructed.

- **Can the functionality provided by the testbed be broadened?**

  The testbed does not cover the full spectrum of functionality that was introduced by Dart [Dar91]. Instead, the testbed concentrates on providing only storage, distribution, versioning, and access facilities. An interesting question to explore is whether the facilities provided by the testbed can be broadened to include some of the functionality that was intentionally left out. If successful, this would further enhance the usability of the testbed. For example, consider the construction, with the testbed, of a CM policy that also incorporates a distributed Make-like facility. Currently, the latter requirement would involve incorporating a solution that is created outside of the testbed. This results in

the complication that a CM policy has to use both the programmatic interface provided by the testbed and a different interface that is provided by the implementation of the distributed derivation process. A more desirable solution uses a single interface encompassing both types of functionality. Important questions to be raised in the creation of such a unified interface regard the kinds of functionality that can be generically added to the testbed (e.g., derivation processes, process engines, merge interfaces) and whether the orthogonality that currently underlies the testbed can be maintained.

- **Can the testbed be applied to other domains?**

  Currently, the testbed is exclusively focused on the creation of prototype CM systems. Obviously, an interesting avenue to explore is the application of the principles and techniques developed within the testbed to other domains. For instance, consider the domain of groupware. Although quite different from the domain of configuration management, the two domains share such characteristics as a multitude of artifacts, distribution, evolution, and versioning. Research questions arise as to whether the same principles and techniques—or even the current incarnation of the testbed itself—can be applied to the construction of groupware systems. Additional questions arise as to whether the orthogonality of the functional categories can be preserved while being extended with other categories, such as access rights or artifact sharing facilities.

- **Can the functionality provided by the testbed be improved without changing its model or interface?**

  The research questions discussed above are primarily focused on the use and applicability of the testbed itself. Another research question arises if we consider the way the testbed is actually implemented. For example, as sets of physical repositories interact to provide CM clients access to the artifacts they maintain,

it might be possible to improve the overall access time of these clients by employing an advanced caching mechanism. Of course, given the fact that the testbed itself is policy independent, such a caching mechanism must be policy independent as well. A potential avenue to explore is the application of access pattern detection techniques (e.g., each time the function `nc_getattributevalue` is used on some artifact, it is followed by a use of the function `nc_open` on the same artifact, or each time the function `nc_open` is used on a collection, it is followed by the repeated use of the function `nc_open` on all of the member artifacts of the collection). These patterns could be used to predict, prefetch, and cache those artifacts that are expected to be used in the near future. Potential techniques to be adapted may be found in the field of process discovery [CW98c].

Bibliography

[AB89] V. Ambriola and L. Bendix. Object-oriented configuration control. In Proceedings of the Second International Workshop on Software Configuration Management, pages 133–136. ACM SIGSOFT, 1989.

[ABGS91] R. Agrawal, S. Buroff, N.H. Gehani, and D. Shasha. Object versioning in Ode. In Proceedings of the 7th International Conference on Data Engineering, pages 446–455, Kobe, Japan, April 1991.

[AFK+95] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting geographically-distributed software development. In Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, pages 194–214, New York, New York, 1995. Springer-Verlag.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology, 6(3):213–249, July 1997.

[Arc99] Archimedes Software, Kirkland, Washington. BugBase Slide Show, 1999.

[Atr92] Atria Software, Natick, Massachusetts. ClearCase Concepts Manual, 1992.

[Bel97] Bell Labs, Lucent Technologies, Murray Hill, New Jersey. Sablime v5.0 User's Reference Manual, 1997.

[Ber90] B. Berliner. CVS II: Parallelizing software development. In Proceedings of 1990 Winter USENIX Conference, Washington, D.C., 1990.

[BFGL94] S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE: An environment for software process analysis, design, and enactment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, Software Process Modeling and Technology, pages 223–248. Wiley, 1994.

[BKR96] D. Belanger, D. Korn, and H. Rao. Infrastructure for wide-area software development. In Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, pages 154–165, New York, New York, 1996. Springer-Verlag.

[BLNP98] L. Bendix, P.N. Larsen, A.I. Nielsen, and J.L.S. Petersen. CoED – a tool for versioning of hierarchical documents. In Proceedings of the Eighth International Symposium on System Configuration Management, number 1439 in Lecture Notes in Computer Science, pages 174–187, New York, New York, 1998. Springer-Verlag.

[BSK94] I.S. Ben-Shaul and G.E. Kaiser. A paradigm for decentralized process modeling and its realization in the Oz environment. In Proceedings of the 16th International Conference on Software Engineering, pages 179–188. IEEE Computer Society, May 1994.

[BT96] G.A. Bolcer and R.N. Taylor. Endeavors: A process system integration infrastructure. In Proceedings of the Fourth International Conference on the Software Process, pages 76–85. IEEE Computer Society, December 1996.

[Buf95] J. Buffenbarger. Syntactic software merging. In Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, pages 153–172, New York, New York, 1995. Springer-Verlag.

[BW98] C. Burrows and I. Wesley. Ovum Evaluates Configuration Management. Ovum Ltd., Burlington, Massachusetts, 1998.

[Car98] A. Carzaniga. DVS 1.2 Manual. Department of Computer Science, University of Colorado, Boulder, Colorado, June 1998.

[Chr98] H.B. Christensen. Experiences with architectural software configuration management in Ragnarok. In Proceedings of the Eighth International Symposium on System Configuration Management, number 1439 in Lecture Notes in Computer Science, pages 67–74, New York, New York, 1998. Springer-Verlag.

[CL93] S.-Y. Chiu and R. Levin. The Vesta repository: A file system extension for software development. Technical Report 106, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, June 1993.

[Cle95] G.M. Clemm. The Odin system. In Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, pages 241–262, New York, New York, 1995. Springer-Verlag.

[Con94] Continuus Software Corporation, Irvine, California. Continuus Task Reference, 1994.

[Con97] R. Conradi, editor. Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, New York, New York, May 1997. Springer-Verlag.

[Con98] Continuus Software Corporation, Irvine, California. Distributed Code Management for Team Engineering, 1998.

[CPT97] J.X. Ci, M. Poonawala, and W.-T. Tsai. ScmEngine: A distributed software configuration management environment on X.500. In Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 108–127, New York, New York, 1997. Springer-Verlag.

[CW97] R. Conradi and B. Westfechtel. Towards a uniform version model for software configuration management. In Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 1–17, New York, New York, 1997. Springer-Verlag.

[CW98a] P.C. Clements and N. Weiderman. Report on the Second International Workshop on Development and Evolution of Software Architectures for Product Families. Technical Report SEI–98–SR–003, Software Engineering Institute, Pittsburgh, Pennsylvania, May 1998.

[CW98b] R. Conradi and B. Westfechtel. Version models for software configuration management. ACM Computing Surveys, 30(2):232–282, June 1998.

[CW98c] J.E. Cook and A.L. Wolf. Discovering Models of Software Processes from Event-Based Data. ACM Transactions on Software Engineering and Methodology, 7(3):215–249, July 1998.

[Dar90] S. Dart. Spectrum of functionality in configuration management systems. Technical Report SEI–90–TR–11, Software Engineering Institute, Pittsburgh, Pennsylvania, December 1990.

[Dar91] S. Dart. Concepts in configuration management systems. In Proceedings of the Third International Workshop on Software Configuration Management, pages 1–18. ACM SIGSOFT, 1991.

[Dar96] S. Dart. Not all tools are created equal. Application Development Trends, 3(10):39–54, October 1996.

[DRS96] A. Dix, T. Rodden, and I. Sommerville. Modeling the sharing of versions. In Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, pages 282–290, New York, New York, 1996. Springer-Verlag.

[EC94] J. Estublier and R. Casallas. The Adele configuration manager. In W. Tichy, editor, Configuration Management, number 2 in Trends in Software, pages 99–134. Wiley, London, Great Britain, 1994.

[EC95] J. Estublier and R. Casallas. Three dimensional versioning. In Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, pages 118– 135, New York, New York, 1995. Springer-Verlag.

[EP84] V.B. Erickson and J.F. Pellegrini. Build — a software construction tool. AT&T Bell Laboratories Technical Journal, 63(6):1049–1059, July–August 1984.

[Est95] J. Estublier, editor. Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, New York, New York, 1995. Springer-Verlag.

[Est99] J. Estublier, editor. Proceedings of the Ninth International Symposium on System Configuration Management, number 1675 in Lecture Notes in Computer Science, New York, New York, 1999. Springer-Verlag.

[FD90] P.H. Feiler and G. Downey. Transaction-oriented configuration management: A case study. Technical Report CMU/SEI–90–TR–23, Software Engineering Institute, Pittsburgh, Pennsylvania, 1990.

[Fei91a] P.H. Feiler. Configuration management models in commercial environments. Technical Report SEI–91–TR–07, Software Engineering Institute, Pittsburgh, Pennsylvania, April 1991.

[Fei91b] P.H. Feiler, editor. Proceedings of the Third International Workshop on Software Configuration Management, ACM Press, June 1991. Association for Computer Machinery.

[Fel79] S.I. Feldman. MAKE — a program for maintaining computer programs. Software—Practice and Experience, (9):252–265, April 1979.

[FGM+98] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1, January 1998. Internet Proposed Standard RFC 2068.

[FKR+95] G. Fowler, D. Korn, H. Rao, J. Snyder, and K.-P. Vo. Configuration management. In B. Krishnamurthy, editor, Practical Reusable UNIX Software, chapter 3. Wiley, New York, New York, 1995.

[FN97] P. Fröhlich and W. Nejdl. WebRC: Configuration management for a cooperation tool. In Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 175–185, New York, New York, 1997. Springer-Verlag.

[Gad95] C. Gadonna. MISTRAL User Manual V1. LGI, May 1995. ESPRIT Project 5327, REBOOT.

[GKY91] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version description in EPOS. Software Engineering Journal, 6:378–386, 1991.

[GS93] D. Garlan and M. Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, Advances in Software Engineering and Knowledge Engineering, volume 1. World Scientific, New Jersey, 1993.

[GWF+99] Y.Y. Goland, E.J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen. HTTP extensions for distributed authoring – WEBDAV, February 1999. Internet Proposed Standard RFC 2518.

[Hei96] G.T. Heineman. A Transaction Manager Component for Cooperative Transaction Models. PhD thesis, Columbia University, Department of Computer Science, New York, New York, June 1996.

[HHW99] R.S. Hall, D.M. Heimbigner, and A.L. Wolf. A cooperative approach to support software deployment using the Software Dock. In Proceedings of the 1999 International Conference on Software Engineering, pages 174–183, New York, New York, May 1999. ACM Press.

[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.

[HLRT97] J.J. Hunt, F. Lamers, J. Reuter, and W.F. Tichy. Distributed configuration management via Java and the World Wide Web. In Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 161–174, New York, New York, 1997. Springer-Verlag.

[HT97] J.J. Hunt and W.F. Tichy. RCE API Introduction and Reference Manual. Xcc Software, Germany, 1997.

[HVT98] J.J. Hunt, K.-P. Vo, and W.F. Tichy. Delta algorithms: An empirical analysis. ACM Transactions on Software Engineering and Methodology, 7(2):192–214, April 1998.

[INT98a] INTERSOLV, Rockville, Maryland. PVCS VM SiteSync and Geographically Distributed Development, 1998.

[INT98b] INTERSOLV, Rockville, Maryland. Using PVCS for Enterprise Distributed Development, 1998.

[Leu94] R. Leung. Versioning on legal applications using hypertext. In Proceedings of the Workshop on Versioning in Hypertext Systems, September 1994.

[LM88] A. Lampen and A. Mahler. An object base for attributed software objects. In Proceedings of the EUUG Autumn '88 Conference, pages 95–105, Cascais, Portugal, October 1988.

[LR96] Y.-J. Lin and S.P. Reiss. Configuration management with logical structures. In Proceedings of the 18th International Conference on Software Engineering, pages 298–307. Association for Computer Machinery, March 1996.

[LT98] P. Lindsay and O. Traynor. Supporting fine-grained traceability in software development environments. In Proceedings of the Eighth International Symposium on System Configuration Management, number 1439 in Lecture Notes in Computer Science, pages 133–139, New York, New York, 1998. Springer-Verlag.

[Mag98] B. Magnusson, editor. Proceedings of the Eighth International Symposium on System Configuration Management, number 1439 in Lecture Notes in Computer Science, New York, New York, 1998. Springer-Verlag.

[MC96] J. Micallef and G.M. Clemm. The Asgard system: Activity-based configuration management. In Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, pages 175–186, New York, New York, 1996. Springer-Verlag.

[Mic97] Microsoft Corporation, Redmond, Washington. Managing Projects with Visual SourceSafe, 1997.

[Mil97] B. Milewski. Distributed source control system. In Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 98–107, New York, New York, 1997. Springer-Verlag.

[ML88] A. Mahler and A. Lampen. An integrated toolset for engineering software configurations. In Proceedings of the ACM SOFSOFT/SIGPLAN Software Engineering Symposium on Practical Software Engineering Environments, pages 191–200, Boston, Massachusetts, November 1988.

[Mor96] Mortice Kern Systems, Inc., Waterloo, Canada. Untangling the Web: Eliminating Chaos, 1996.

[Mun93] B.P. Munch. Versioning in a Software Engineering Database — the Change-Oriented Way. PhD thesis, DCST, NTH, Trondheim, Norway, August 1993.

[Neu98] Neuma Technology Corporation, Ottawa, Canada. NeumaCM+ FAQ's, September 1998.

[OG90] B. O'Donovan and J.B. Grimson. A distributed version control system for wide area networks. Software Engineering Journal, September 1990.

[Ous94] J.K. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley Publishing Company, 1994.

[Per98] Perforce Software, Alameda, California. Networked Software Development: SCM over the Internet and Intranets, March 1998.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. SIGSOFT Software Engineering Notes, 17(4):40–52, October 1992.

[Ray95] R.J. Ray. Experiences with a script-based software configuration management system. In Software Configuration Management: ICSE SCM-4 and SCM-5 Workshops Selected Papers, number 1005 in Lecture Notes in Computer Science, pages 282–287, New York, New York, 1995. Springer-Verlag.

[Roc75] M.J. Rochkind. The source code control system. IEEE Transactions on Software Engineering, SE–1(4):364–370, December 1975.

[ROY99] L. Rodriguez, H. Ogata, and Y. Yano. An access mechanism for a temporal versioned object-oriented database. IEICE Transactions on Information and Systems, E82–D(1):128–135, January 1999.

[SDK+95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. IEEE Transactions on Software Engineering, 21(4):314–335, April 1995.

[Sei96] C. Seiwald. Inter-file branching — a practical method for representing variants. In Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, pages 67–75, New York, New York, 1996. Springer-Verlag.

[Smi99] R.A. Smith. Analysis and design for a next generation software release management system. Master's thesis, University of Colorado, Boulder, Colorado, December 1999.

[Sof94a] Softool Corp., Goleta, California. CCC/Manager, Managing the Software Life Cycle across the Complete Enterprise, 1994.

[Sof94b] Software Maintenance & Development Systems, Inc., Concord, Massachusetts. Aide de Camp Product Overview, September 1994.

[Som96] I. Sommerville, editor. Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, New York, New York, March 1996. Springer-Verlag.

[SQL98] SQL Software, Vienna, Virginia. The Inside Story: Process Configuration Management with PCMS Dimensions, 1998.

[Sta96] Starbase Corporation, Irvine, California. StarTeam Web Connect Users's Guide, 1996.

[Tib96] D. Tibrook. An architecture for a construction system. In Proceedings of the Sixth International Workshop on Software Configuration Management, number 1167 in Lecture Notes in Computer Science, pages 76–87, New York, New York, 1996. Springer-Verlag.

[Tic84] W.F. Tichy. The string-to-string correction problem with block moves. ACM Transactions on Computer Systems, 2(4):309–321, November 1984.

[Tic85] W.F. Tichy. RCS, a system for version control. Software—Practice and Experience, 15(7):637–654, July 1985.

[Tic88] W.F. Tichy. Tools for software configuration management. In Proceedings of the International Workshop on Software Versioning and Configuration Control, pages 1–20, 1988.

[Tic89] W.F. Tichy, editor. Proceedings of the Second International Workshop on Software Configuration Management, ACM Press, November 1989. Association for Computer Machinery.

[TMA+96] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead, Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. ACM Transactions on Software Engineering and Methodology, 22(6):390–406, June 1996.

[Tow99] Tower Concepts, Oneida, New York. Razor Manual, 1999.

[vdHHHW97] A. van der Hoek, R.S. Hall, D.M. Heimbigner, and A.L. Wolf. Software release management. In Proceedings of the Sixth European Software Engineering Conference, number 1301 in Lecture Notes in Computer Science, pages 159–175, New York, New York, September 1997. Springer-Verlag.

[vdHHW96] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. A generic, peer-to-peer repository for distributed configuration management. In Proceedings of the 18th International Conference on Software Engineering, pages 308–317. Association for Computer Machinery, March 1996.

[vdHHW98] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Software architecture, configuration management, and configurable distributed systems: A ménage a trois. Technical Report CU–CS–849–98, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[vdHHW99] A. van der Hoek, D.M. Heimbigner, and A.L. Wolf. Capturing architectural configurability: Variants, options, and evolution. Technical Report CU–CS–895–99, Department of Computer Science, University of Colorado, Boulder, Colorado, December 1999.

[Wes96] B. Westfechtel. A graph-based system for managing configurations of engineering design documents. International Journal of Software Engineering and Knowledge Engineering, 6(4):549–583, 1996.

[Whi97] E.J. Whitehead, Jr. World Wide Web distributed authoring and versioning (WebDAV): an introduction. StandardView, 5(1):3–8, March 1997.

[Win88] J.F.H. Winkler, editor. Proceedings of the International Workshop on Software Versioning and Configuration Control. B.G. Teubner, 1988.

[WS97] L. Wingerd and C. Seiwald. Constructing a large product with Jam. In Proceedings of the Seventh International Workshop on Software Configuration Management, number 1235 in Lecture Notes in Computer Science, pages 36–48, New York, New York, 1997. Springer-Verlag.

[Zel97] A. Zeller. Configuration Management with Version Sets: A Unified Software Versioning Model and its Applications. PhD thesis, Technischen Universität Braunschweig, Braunschweig, Germany, April 1997.

[ZS97] A. Zeller and G. Snelting. Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology, 6(4):398–441, October 1997.

Appendix A

Details of the Programmatic Interface

A.1 Access Functions

A.1.1 nc_open

A.1.1.1 Signature

```
int nc_open(const char* path,
            const char* p_cwd,
            const char* prefix_target,
            const char* t_cwd)
```

A.1.1.2 Functionality

Gains access to the artifact determined by the parameters `path` and `p_cwd` and places its contents in the workspace determined by the parameters `prefix_target` and `t_cwd`. The contents of the artifact is always retrieved from a NUCM logical repository, even if the parameters `path` and `p_cwd` point to an artifact already existing in another workspace.

A.1.1.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.1.1.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbOpen	:	the artifact is already open in the workspace
NdbExist	:	a different artifact with the same name already exists in the workspace
NdbUnequal	:	the path and target represent two different artifacts that have the same name
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.1.2 nc_close

A.1.2.1 Signature

```
int nc_close(const char* target,
             const char* t_cwd,
             int forced)
```

A.1.2.2 Functionality

Relinquishes access to an artifact. The artifact is removed from the workspace. If the artifact is a collection, all members of the collection are removed as well. If the artifact is a collection and itself or one or more of its members is initiated, the close does not succeed unless the close is forced. A close is forced if the value of the parameter `forced` is 1, a close is not forced if the value of the parameter `forced` is 0.

A.1.2.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.1.2.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbInitiated	:	the artifact is a collection and itself or one or more of its members is initiated and the close is not forced
NdbNoClose	:	the artifact cannot be closed because the current working directory is (part of) this artifact

A.2 Versioning Functions

A.2.1 nc_initiatechange

A.2.1.1 Signature

```
int nc_initiatechange(const char* target,
                      const char* t_cwd)
```

A.2.1.2 Functionality

Initiates change on the artifact determined by the interpretation of the parameters `target` and `t_cwd`. Only artifacts in a workspace can be initiated. Once the artifact is initiated, it can be modified. However, it cannot be initiated again until either `nc_commitchange`, `nc_commitchange`, or `nc_commitchangeandreplace` has been called for the artifact. It is possible for a single artifact to be initiated multiple times as long as each initiation takes place in a different workspace.

A.2.1.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.2.1.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbInitiated	:	the artifact is already initiated in the workspace

A.2.2 nc_abortchange

A.2.2.1 Signature

```
int nc_abortchange(const char* target,
                   const char* t_cwd,
                   int forced)
```

A.2.2.2 Functionality

Aborts change on the artifact determined by the interpretation of the parameters `target` and `t_cwd`. The contents of the artifact is restored to the contents of the version that was initiated, and the artifact can no longer be modified. If the artifact is a collection, all members of the collection are closed. If the artifact is a collection and one or more of its members is initiated, the abort does not succeed unless the abort is forced. An abort is forced if the value of the parameter `forced` is 1, an abort is not forced if the value of the parameter `forced` is 0.

A.2.2.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.2.2.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbInitiated	:	the artifact is a collection and one or more of its members is initiated and the abort is not forced
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available
NdbNoClose	:	the change cannot be aborted because the current working directory is (part of) this artifact

A.2.3 nc_commitchange

A.2.3.1 Signature

```
int nc_commitchange(const char* target,
                    const char* t_cwd,
                    char* version)
```

A.2.3.2 Functionality

Commits change on the artifact determined by the interpretation of the parameters `target` and `t_cwd`. A new version of the artifact is created in a logical NUCM repository and the contents of the new version is set to the current contents of the artifact. The previous version of the artifact in the repository remains unchanged. After committing change, the artifact can no longer be modified. The parameter `version` is set to the version number of the newly created artifact. It is assumed that the parameter `version` holds enough space to place the version number in. The resulting value is NULL-terminated.

A.2.3.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.2.3.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbNotInit	:	the artifact is not initiated
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.2.4 nc_commitchangeandreplace

A.2.4.1 Signature

```
int nc_commitchangeandreplace(const char* target,
                              const char* t_cwd)
```

A.2.4.2 Functionality

Commits change on the artifact determined by the interpretation of the parameters `target` and `t_cwd`. No new version of the artifact is created in a NUCM logical repository. Instead, the contents of the initiated version of the artifact is set to the current contents of the artifact. After committing change, the artifact can no longer be modified.

A.2.4.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.2.4.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbNotInit	:	the artifact is not initiated
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.3 Collection Functions

A.3.1 nc_add

A.3.1.1 Signature

```
int nc_add(const char* path,
           const char* p_cwd,
           const char* prefix_target,
           const char* t_cwd)
```

A.3.1.2 Functionality

Adds the artifact determined by the parameters `path` and `p_cwd` to the collection determined by the path `prefix_target` and `t_cwd`. The collection to which the artifact is added has to be initiated. Three types of additions are possible:

- **addition of a new artifact**: the artifact is a file or directory in the file system that is currently not maintained by NUCM. If the artifact is not present in the current workspace, it is copied there. A new artifact is created in the NUCM logical repository, and its contents is set to the current contents of the artifact in the workspace. A new collection is by definition empty when added.
- **import of an existing artifact in another workspace**: the artifact is an artifact that is already maintained by NUCM and has been opened in another workspace. The artifact is copied into the current workspace. No new artifact is created in the NUCM logical repository, instead both the collection where the artifact came from and the collection to which it has been added reference the same artifact.
- **mount of an existing artifact in a NUCM logical repository**: the artifact is an artifact that is maintained by a NUCM logical repository. The artifact is copied from the NUCM logical repository into the current workspace. No new artifact is created in the NUCM logical repository, instead the collection references the existing artifact. This variant of the addition interface function is the function that is capable of combining artifacts from various NUCM physical repositories into a NUCM logical repository.

The initial version of the artifact that is created is 1.

A.3.1.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.3.1.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbLonely	:	no collection present in the workspace to which the artifact can be added
NdbNotInit	:	the collection to which to add the artifact is not initiated
NdbExist	:	a different artifact with the same name already exists in the workspace
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.3.2 nc_remove

A.3.2.1 Signature

```
int nc_remove(const char* target,
              const char* t_cwd,
              int forced)
```

A.3.2.2 Functionality

Removes the artifact determined by the parameters `target` and `t_cwd` from the collection in the workspace. If the artifact has been opened, the artifact is removed from the workspace. If the artifact is a collection, its members are removed from the workspace as well. If the artifact is a collection and itself or one or more of its members is initiated, the remove does not succeed unless the remove is forced. A remove is forced if the value of the parameter `forced` is 1, a remove is not forced if the value of the parameter `forced` is 0. The collection from which the artifact is removed has to be initiated.

A.3.2.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.3.2.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbLonely	:	no collection present in the workspace from which the artifact can be removed
NdbNotInit	:	the collection from which to remove the artifact is not initiated
NdbInitiated	:	the artifact is a collection and itself or one or more of its members is initiated and the remove is not forced
NdbNoClose	:	the artifact cannot be removed because the current working directory is (part of) this artifact

A.3.3 nc_rename

A.3.3.1 Signature

```
int nc_rename(const char* target,
              const char* t_cwd,
              const char* new_name)
```

A.3.3.2 Functionality

Renames the artifact determined by the parameters `target` and `t_cwd` in the collection in the workspace to the name given in the parameter `new_name`. If the artifact has been opened, it is also renamed in the workspace. The collection in which the artifact is renamed has to be initiated.

A.3.3.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.3.3.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbLonely	:	no collection present in the workspace in which the artifact can be renamed
NdbNotInit	:	the collection in which to rename the artifact is not initiated
NdbExist	:	a different artifact with the same name already exists in the workspace

A.3.4 nc_replaceversion

A.3.4.1 Signature

```
int nc_replaceversion(const char* target,
                      const char* t_cwd,
                      const char* new_version,
                      int forced)
```

A.3.4.2 Functionality

Replaces the current version of the artifact determined by the parameters `target` and `t_cwd` in the collection in the workspace with the version determined by the parameter `version`. If the artifact has been opened, it is also replaced in the workspace. If the artifact is a collection, all members of the collection are closed. If the artifact is a collection and one or more of its members are initiated, the replacement does not succeed unless the replacement is forced. A replacement is forced if the value of the parameter `forced` is 1, a replacement is not forced if the value of the parameter `forced` is 0. The collection in which the version of the artifact is replaced has to be initiated.

A.3.4.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.3.4.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbLonely	:	no collection present in the workspace in which the artifact can be replaced
NdbNotInit	:	the collection in which to replace the artifact is not initiated
NdbInitiated	:	the artifact is a collection and one or more of its members is initiated and the replace is not forced
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available
NdbNoClose	:	the version of the artifact cannot be replaced because the current working directory is (part of) this artifact

A.3.5 nc_copy

A.3.5.1 Signature

```
int nc_copy(const char* path,
            const char* p_cwd,
            const char* prefix_target,
            const char* t_cwd,
            const t_server* server)
```

A.3.5.2 Functionality

Copies the history of the artifact determined by the parameters `path` and `p_cwd` to the NUCM server determined by the parameter `server`, and then adds this newly created artifact to the collection determined by the path `prefix_target` and `t_cwd`. The new artifact is opened in the workspace. The collection has to be initiated.

A.3.5.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.3.5.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbLonely	:	no collection present in the workspace to which the artifact can be added
NdbNotInit	:	the collection to which to add the artifact is not initiated
NdbExist	:	a different artifact with the same name already exists in the workspace
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.3.6 nc_list

A.3.6.1 Signature

```
int nc_list(const char* path,
            const char* p_cwd,
            const t_memberlist* memberlist)
```

A.3.6.2 Functionality

Sets the value of the parameter `memberlist` to the list of artifacts contained in the NUCM collection determined by the interpretation of the parameters `path` and `p_cwd`. Each member in the list contains the name and version of an artifact contained in the NUCM collection. The type `t_memberlist` is defined as:

```
typedef struct memberlist {
    char name[MAXPATHLEN + 1];
    char version[MAX_VERSION_LENGTH];
    struct memberlist* next;
} t_memberlist;
```

The resulting list of members is allocated for the caller, but needs to be de-allocated by the caller by using the function `nc_destroy_memberlist`.

A.3.6.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.3.6.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbAtomList	:	the artifact is an atom and cannot be listed
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.4 Distribution Functions

A.4.1 nc_setmyserver

A.4.1.1 Signature

void nc_setmyserver(const t_server* server)

A.4.1.2 Functionality

Sets the default NUCM server that serves this client to the server addressed by the parameter `server`. This default NUCM server is the server where newly created artifacts are stored, and needs to be set before `nc_add` can be used. The type `t_server` is defined as:

```
typedef struct {
    char name[MAXHOSTNAMELEN + 1];
    char version[MAX_PORT_LENGTH + 1];
} t_server;
```

A.4.1.3 Return Values

None

A.4.1.4 Error Codes

None

A.4.2 nc_getlocation

A.4.2.1 Signature

```
int nc_getlocation(const char* path,
                   const char* p_cwd,
                   t_server* server)
```

A.4.2.2 Functionality

Determines the address of the NUCM server where the history of the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd` is physically located, and places the resulting server address in the parameter `server`.

A.4.2.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.4.2.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.4.3 nc_move

A.4.3.1 Signature

```
int nc_move(const char* path,
            const char* p_cwd,
            const t_server* server)
```

A.4.3.2 Functionality

Physically moves the history of the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd` to the NUCM server addressed by the parameter `server`. The move has no effect other than that the history of the artifact is moved to a different physical location, all operations on the artifact continue to operate as normal.

A.4.3.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.4.3.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.5 Deletion Function

A.5.1 nc_destroyversion

A.5.1.1 Signature

```
int nc_destroyversion(const char* path,
                      const char* p_cwd,
                      const char* version)
```

A.5.1.2 Functionality

Physically removes a single version of the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd` from a NUCM logical repository. The version that is removed is determined by the parameter `version`, and is only removed if it is not referenced by any collections.

A.5.1.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.5.1.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbRefcount	:	the version of the artifact cannot be removed because it is still referenced by one or more collections
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.6 Query Functions

A.6.1 nc_gettype

A.6.1.1 Signature

```
int nc_gettype(const char* path,
               const char* p_cwd)
```

A.6.1.2 Functionality

Determines the type of the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd`.

A.6.1.3 Return Values

COLLECTION	:	everything went ok and the artifact is a collection
ATOM	:	everything went ok and the artifact is an atom
-1	:	an error occurred

A.6.1.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.6.2 nc_version

A.6.2.1 Signature

```
int nc_version(const char* path,
               const char* p_cwd,
               char* version)
```

A.6.2.2 Functionality

Sets the value of the parameter `version` to the version number of the artifact determined by the interpretation of the parameters `path` and `p_cwd`. It is assumed that the parameter `version` holds enough space to place the version number in, i.e., is at least `MAX_VERSION_LENGTH` in size. The resulting value is NULL-terminated.

A.6.2.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.6.2.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.6.3 nc_lastversion

A.6.3.1 Signature

```
int nc_lastversion(const char* path,
                   const char* p_cwd,
                   char* version)
```

A.6.3.2 Functionality

Sets the value of the parameter `version` to the version number of the last created version of the artifact determined by the interpretation of the parameters `path` and `p_cwd`. It is assumed that the parameter `version` holds enough space to place the version number in, i.e., is at least `MAX_VERSION_LENGTH` in size. The resulting value is NULL-terminated.

A.6.3.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.6.3.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.6.4 nc_existsversion

A.6.4.1 Signature

```
int nc_existsversion(const char* path,
                     const char* p_cwd,
                     const char* version)
```

A.6.4.2 Functionality

Determines whether the version determined by the parameter `version` of the artifact determined by the interpretation of the parameters `path` and `p_cwd` exists.

A.6.4.3 Return Values

1	:	everything went ok and the version exists
0	:	everything went ok and the version does not exist
-1	:	an error occurred

A.6.4.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.6.5 nc_isinitiated

A.6.5.1 Signature

```
int nc_isinitiated(const char* target,
                   const char* t_cwd)
```

A.6.5.2 Functionality

Determines whether the artifact determined by the interpretation of the parameters `target` and `t_cwd` is initiated.

A.6.5.3 Return Values

1	:	everything went ok and the artifact is initiated
0	:	everything went ok and the artifact is not initiated
-1	:	an error occurred

A.6.5.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM

A.6.6 nc_isopen

A.6.6.1 Signature

```
int nc_isopen(const char* target,
              const char* t_cwd)
```

A.6.6.2 Functionality

Determines whether the artifact determined by the interpretation of the parameters `target` and `t_cwd` has been opened in the workspace.

A.6.6.3 Return Values

1	:	everything went ok and the artifact is open
0	:	everything went ok and the artifact is not open
-1	:	an error occurred

A.6.6.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM

A.7 Attribute Functions

A.7.1 nc_testandsetattribute

A.7.1.1 Signature

```
int nc_testandsetattribute(const char* path,
                           const char* p_cwd,
                           const char* attrname,
                           const char* value)
```

A.7.1.2 Functionality

If and only if the attribute determined by the parameter `attrname` does not exist for the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd`, creates this attribute in a NUCM logical repository and sets its value to the value of the parameter `value`.

A.7.1.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.7.1.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbExist	:	the attribute exists already and cannot be overwritten
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.7.2 nc_setattribute

A.7.2.1 Signature

```
int nc_setattribute(const char* path,
                    const char* p_cwd,
                    const char* attrname,
                    const char* value)
```

A.7.2.2 Functionality

Always sets the value of the attribute determined by the value of the parameter `attrname` for the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd` to the value of the parameter `value`. If the attribute does not exist, it is created. If the attribute does exist, its value is overwritten.

A.7.2.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.7.2.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.7.3 nc_getattributevalue

A.7.3.1 Signature

```
int nc_getattributevalue(const char* path,
                         const char* p_cwd,
                         const char* attrname,
                         char* value)
```

A.7.3.2 Functionality

Returns the value of the attribute `attrname` for the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd`. It is assumed that the parameter `value` holds enough space to place the value of the attribute in. The resulting value is NULL-terminated.

A.7.3.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.7.3.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbNonExist	:	the attribute does not exist
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available

A.7.4 nc_removeattribute

A.7.4.1 Signature

```
int nc_removeattribute(const char* path,
                       const char* p_cwd,
                       const char* attrname)
```

A.7.4.2 Functionality

Physically removes the attribute determined by the parameter `attrname` for the NUCM artifact determined by the interpretation of the parameters `path` and `p_cwd` from a NUCM logical repository.

A.7.4.3 Return Values

0	:	everything went ok
-1	:	an error occurred

A.7.4.4 Error Codes

NdbOk	:	everything went ok
NdbUnknown	:	the path does not represent a NUCM artifact
NdbNotOpen	:	the artifact exists in the workspace but is not an artifact that is managed by NUCM and was therefore not opened by NUCM
NdbNonExist	:	the attribute does not exist
NdbConnect	:	the server is down or not reachable
NdbComm	:	a communication error occurred
NdbProtocol	:	a failure occurred in the NUCM protocol
NdbNotAvail	:	service is not available