A Testbed for Configuration Management Policy Programming

André van der Hoek, *Member*, *IEEE*, Antonio Carzaniga,
Dennis Heimbigner, *Member*, *IEEE*, and Alexander L. Wolf, *Member*, *IEEE Computer Society*

Abstract—Even though the number and variety of available configuration management systems has grown rapidly in the past few years, the need for new configuration management systems still remains. Driving this need are the emergence of situations requiring highly specialized solutions, the demand for management of artifacts other than traditional source code and the exploration of entirely new research questions in configuration management. Complicating the picture is the trend toward organizational structures that involve personnel working at physically separate sites. We have developed a testbed to support the rapid development of configuration management systems. The testbed separates configuration management repositories (i.e., the stores for versions of artifacts) from configuration management policies (i.e., the procedures, according to which the versions are manipulated) by providing a generic model of a distributed repository and an associated programmatic interface. Specific configuration management policies are programmed as unique extensions to the generic interface, while the underlying distributed repository is reused across different policies. In this paper, we describe the repository model and its interface and present our experience in using a prototype of the testbed, called NUCM, to implement a variety of configuration management systems.

Index Terms—Configuration management, configuration management policies, distributed configuration management, policy programming, peer-to-peer, version control.

✦

1 INTRODUCTION

SINCE its beginnings in the early 1970s, the field of configuration management (CM) has slowly but surely evolved. The marketplace for CM products is now worth well over one billion dollars per year [8]. More than one hundred commercial CM systems, representing a wide range of functionality, are currently available. While some are simple clones of SCCS [41] and RCS [47], others have pushed the state of the art quite considerably by offering a full spectrum of functionality [14]. Most CM systems, however, fall somewhere in between, each providing some distinguishing combination of functionality.

Despite the variety of available systems, several compelling reasons exist to continue the development of new CM systems. First, in the current generation, the basic functionality provided by a given CM system is fixed; if specialized functionality is needed in a particular situation (e.g., required compliance with company-wide standards [40] or e-mail-based synchronization of workspaces [33]), it becomes difficult to provide. A second reason is that existing CM systems tend to focus on the management of source code; if other types of artifacts need to be managed and configured (e.g., Web sites [34],

software architectures [11], or legal databases [29]), only a limited amount of support is available. A third reason is that existing CM systems are based on certain underlying assumptions; if new approaches are developed that are in conflict with some of these assumptions (e.g., the approach based on feature logic [53]), little help is available to implement them.

Already a daunting task in and of itself, the construction of a CM system is further complicated by the fact that many of today's projects are carried out in a distributed fashion. In these projects, multiple collaborating participants are physically dispersed over a number of geographical locations, sometimes even belonging to different companies. 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 probably decentralized collaboration.

We have developed a testbed to support the rapid construction of new, potentially distributed, CM systems. The testbed embodies an architecture that separates 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 architecture is the definition of an *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 components covering the major aspects of a configuration management repository, namely, storage, distribution, naming, access, and attributes. Similarly, the programmatic interface consists of

• A. van der Hoek is with the Institute for Software Research, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425. E-mail: andre@ics.uci.edu.
• A. Carzaniga, D. Heimbigner, and A.L. Wolf are with the Software Engineering Research Laboratory, Department of Computer Science, University of Colorado, Boulder, CO 80309.
  E-mail: {carzanig,dennis,alw}@cs.colorado.edu.

seven orthogonal categories of functions, including access, versioning, querying, and distribution. CM policies are programmed as extensions to the generic interface, while the underlying distributed repository is reused across different policies. Structured this way, the testbed supports direct and flexible experimentation with new CM policies.

Overall, the design of the abstraction layer was guided by the following high-level objectives:

- *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 [35]. Similarly, if the functions in the programmatic interface automatically created a new version of an artifact whenever one of its constituent parts is modified, CM policies in which the evolution 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 considerable amount of research on the issue [2], [10], [13], [25], providing proper support for the distributed operation of a CM system is a complicated task. Therefore, it is 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 permit the 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. Further, 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 arbitrary kinds of artifacts.* As previously mentioned, CM systems are increasingly needed to manage artifacts other than source code. To allow such specialized CM systems to be constructed, 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 [27]. Incorporating such a biased differencing algorithm into the abstraction would violate its ability to properly handle different kinds of artifacts.

- *The abstraction layer should be able to 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.

NUCM (Network-Unified Configuration Management) is our prototype implementation of the testbed. It has been key to the development of several innovative CM systems, including DVS [9] and SRM [49]. As a prototype, NUCM was not designed to exhibit the robustness or completeness that one would expect of a commercial implementation of the abstraction layer. Similarly, the CM systems we built using NUCM were not designed to be particularly robust or complete (although two of them are currently in everyday use). Instead, our focus was on being able to evaluate the utility of the abstraction layer in supporting CM policy programming.

Fig. 1 illustrates the architecture of NUCM in terms of an example repository structure. A CM system that uses a NUCM repository consists of two parts: the generic NUCM client and a particular CM policy. The generic NUCM client implements the programmatic interface and, thus, is the foundation upon which particular CM policies are implemented. This is illustrated in Fig. 1, where two CM policies, namely, policy X and policy Y, both use the generic NUCM client to store and version the artifacts that they manage. In general, a single 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.

The figure also shows that NUCM provides the concept of a *logical* repository that is made up of *physical* repositories. The artifacts in each physical repository are managed by a NUCM *server*. Combined, the NUCM servers for the physical repositories provide access to the complete logical repository. In particular, when artifacts that reside in a different physical repository than the one managed by one of the NUCM servers are requested, that NUCM server will communicate with the other ones to provide access to the artifact.

This paper presents the design of our abstraction layer for CM policy programming and our experiences to date in using NUCM to evaluate the utility of the abstraction layer. We begin in Section 2 by discussing the generic repository model. Section 3 presents the programmatic interface
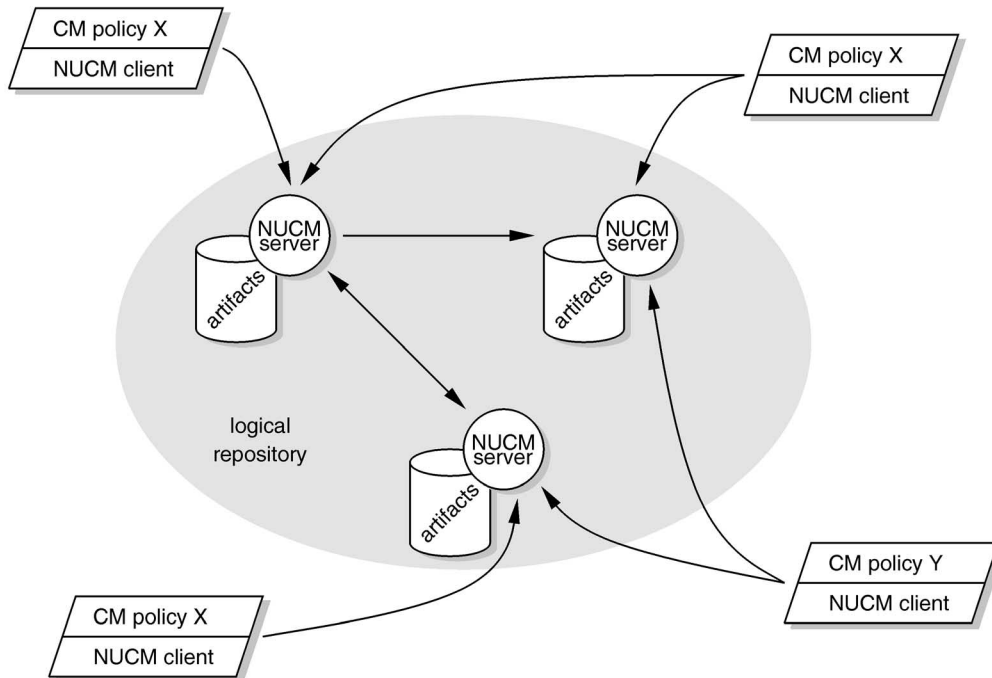
Fig. 1. NUCM architecture.

through which the artifacts are stored and manipulated. Section 4 describes how we have used NUCM to construct a number of rather different CM systems. Related work is discussed in Section 5 and we conclude with a brief look at future work in Section 6.

2 A GENERIC CM REPOSITORY MODEL

The first part of our abstraction layer is the generic repository model. It consists of five components: a storage model, a distribution model, a naming model, an access model, and an attribute model. The storage model defines the mechanisms for versioning and grouping artifacts, the distribution model defines the different ways in which artifacts can be arranged across different sites, the naming model defines the way individual artifacts can be identified in a distributed repository, the access model defines the primary method of access to artifacts stored in a distributed repository, and the attribute model 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 for 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 Section 3 to manipulate the CM repository into the desired behavior.

2.1 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 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.

Fig. 2 illustrates the basic concepts of atoms, collections, and their member relationships. 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 member relationships 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 source files that implement both systems.
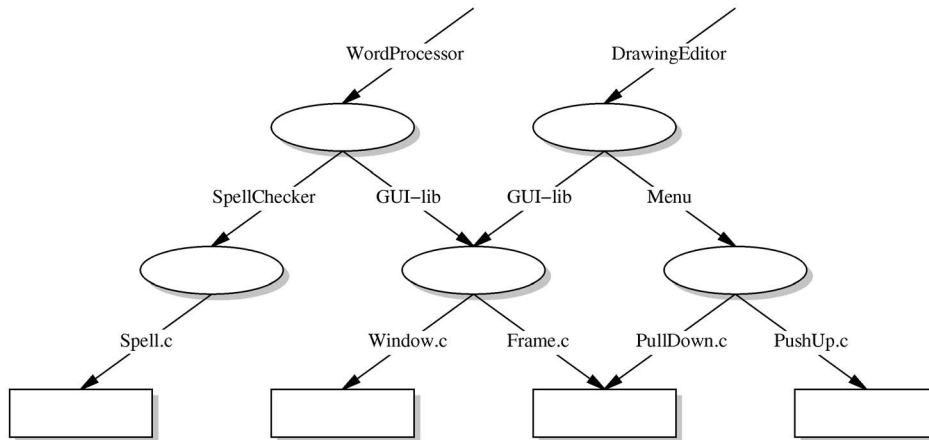
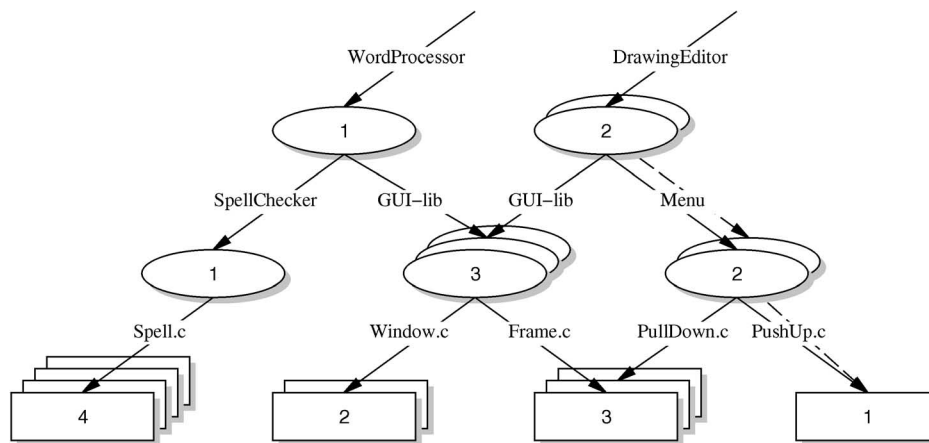Fig. 2. Example repository contents without versions.

Fig. 3. Example repository contents with versions.

In contrast to other approaches, such as CME [25] or ScmEngine [10], 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 are 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.

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. While this 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 Section 4 reuses much of the policy code of an earlier CM system. This earlier CM system also uses the version-tree relationship and was built using NUCM [50].

Fig. 3 shows how the directed graph of artifacts presented in Fig. 2 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.

2.2 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

---

1. As further discussed in Section 3, the "age" of versions merely indicates their relative order of creation. In fact, the contents of the versions may, depending on the policy, change over time.
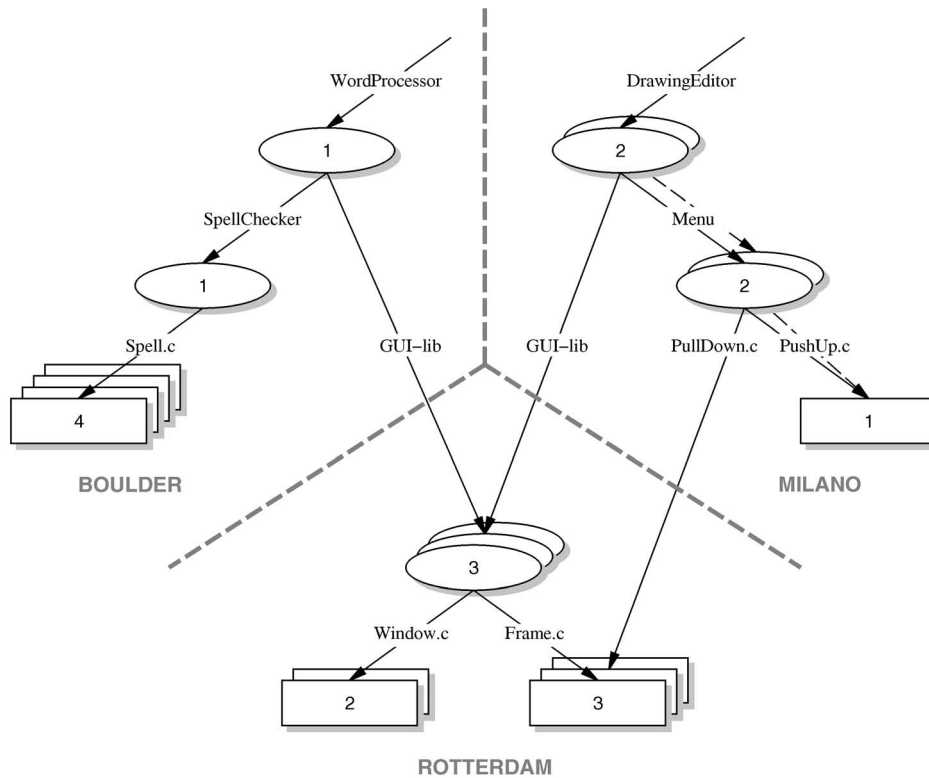
Fig. 4. Example repository contents of Fig. 3 as distributed over three different sites.

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, with no centralized "master" repository controlling 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, member relationships may span the geographical boundaries that exist among physical repositories.

The physical 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.

Fig. 4 presents these concepts with an example distributed repository. Shown is the repository of Fig. 3 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 `DrawingEditor`, 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 member relationships among artifacts in different physical repositories that creates logical repositories. Without the membership of version 2 of the collection `GUI-lib` within version 1 of the collection `WordProcessor`, for example, the logical repository formed by 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 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 flexibility, combined with a

peer-to-peer architecture, allows many different distribution schemes to be mapped onto the distribution model. As further demonstrated elsewhere [48], these schemes include the following:

- a single physical repository that is accessed by many CM clients, thus creating a client-server system like DRCS [36];
- several physical repositories that represent a hierarchy of distributed workspaces in which changes in lower level workspaces are gradually promoted up the hierarchy, thus duplicating the essence of the functionality of such systems as NSE [19] and PCMS [46]; and
- 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 [2].

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 afforded by the repository model allows experimentation with new distribution policies. Once proven to be of use, the implementation of an experimental policy can be optimized for performance.

2.3   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. First, 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. Second, 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 member relationships 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.

The hierarchical name of an artifact adheres to the following template:

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

Thus, names in the abstraction layer can be viewed as URLs that are extended with version qualifiers. Version qualifiers provide a means to specify particular versions of artifacts. Because the storage model allows a single version of an artifact to be a member of multiple (versions of multiple) collections, this kind of naming scheme allows an artifact to have multiple names. For example, assuming the logical repository shown in Fig. 4, the following four names are all equally valid as the fully qualified name of version 1 of the atom `PushUp.c`:

```
//Milano/DrawingEditor:1/Menu:1/PushUp.c:1
//Milano/DrawingEditor:1/Menu:2/PushUp.c:1
//Milano/DrawingEditor:2/Menu:1/PushUp.c:1
//Milano/DrawingEditor:2/Menu:2/PushUp.c:1
```

For convenience, the use of version qualifiers is optional. If a version qualifier is not included, then the interpretation of the name defaults to the version of the artifact that is the actual member of the containing collection. For example, the name

```
//Milano/DrawingEditor:2/Menu/PushUp.c
```

also refers to version 1 of atom `PushUp.c` since version 2 of collection `Menu` is the member of version 2 of collection `DrawingEditor` and version 1 of atom `PushUp.c` is the member of version 2 of collection `Menu`. Thus, it defaults to the following fully qualified name:

```
//Milano/DrawingEditor:2/Menu:2/PushUp.c:1
```

Similarly, because version 2 of collection `DrawingEditor` is the member of the repository in Milano, the following two names also refer to version 1 of atom `PushUp.c`:

```
//Milano/DrawingEditor/Menu:2/PushUp.c
//Milano/DrawingEditor/Menu/PushUp.c
```

2.4   Access Model

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. When designing a CM system, the use of a workspace provides three 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, typically residing much closer in proximity to the originator of changes than the physical repository. Finally, a workspace is unobtrusive in that it provides existing applications with access to versioned artifacts without the need to modify those 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 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 subdirectories, and atoms materialize as files. For example, version 2 of the collection

`DrawingEditor` as presented in Fig. 4 would have the following directory structure when materialized into a workspace on a UNIX file system.

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

ClearCase [3] 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 by the repository. In contrast, workspaces in our access model follow the model that is used by DRCS [36] and DCVS [24], where 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.

In traditional CM systems, the user of a workspace is a human. The user of the workspace in our access model, however, is primarily intended to be a CM system that, in turn, provides tailored styles of access to their ultimate human users. This is illustrated in Fig. 5. 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 might be transformed by a CM policy for presentation to a human user, resulting in the top layer. Note that the bottom two layers are standard and managed internally beneath the abstraction layer. The top layer, however, can be of any shape or form, since it is determined by the CM policy program.

2.5 Attribute Model

To facilitate the storage of metadata in a repository, the repository model incorporates a simple attribute model. An attribute in this model is an untyped name/value pair that can be dynamically associated with a particular version of
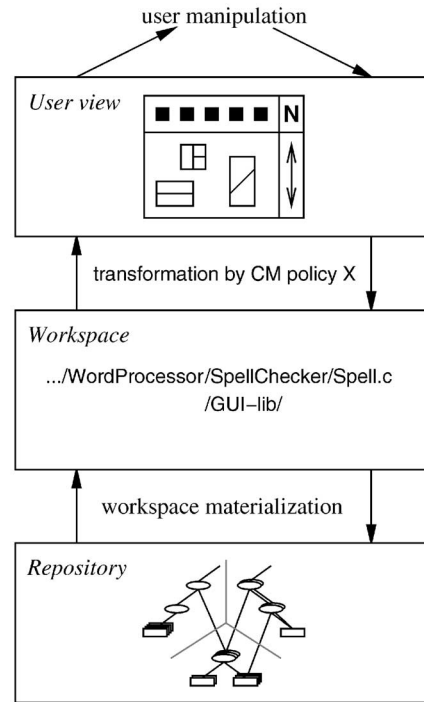

Fig. 5. Access model.

an artifact. Each such version can have its own unique set of attributes, and this set can change over time. The CM policy determines both a naming scheme for the attributes and a set of values that the attributes may assume.

An example is provided in Fig. 6, which shows the attributes that have been associated with the various versions of the atom `Spell.c`. The CM policy managing these versions has labeled them all with the attributes `Author`, `Version`, and `ChangeComment`. Furthermore, if a new version of an artifact fixes 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
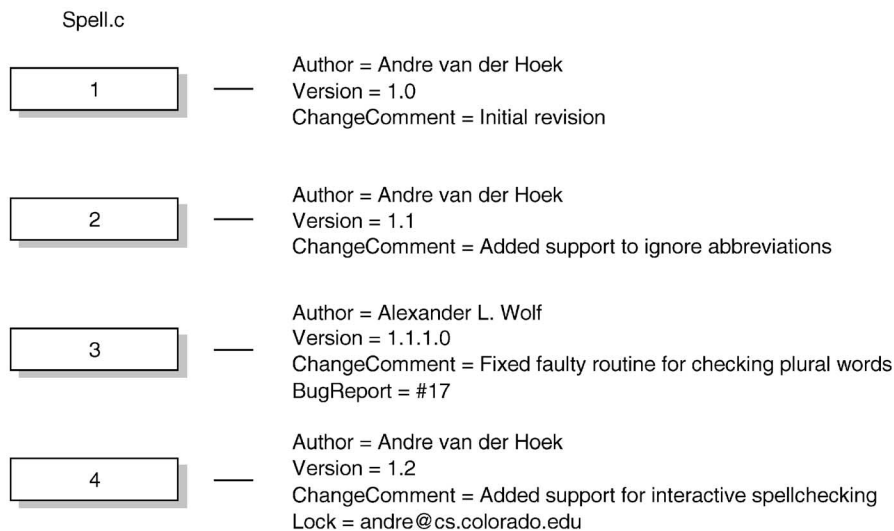

Fig. 6. Example attributes associated with the atom `Spell.c`.

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`).

3   REPOSITORY 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 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 programmer.

An important characteristic of the programmatic interface is the orthogonality among the various functional 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. In Section 4, we present some of the CM policies that we have constructed this way. Below, we introduce, per category, the individual interface functions that constitute the programmatic interface to the generic repository model.

3.1   Access Functions

Access to the artifacts in a repository is, as discussed in Section 2.4, obtained through a workspace in which artifacts are materialized upon request. Once the artifacts are materialized, other interface functions become available to manipulate them. In particular, versioning functions can be used to 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 artifact. A workspace, then, has to be constructed in an incremental fashion. This mechanism allows a CM system to populate a workspace with only the artifacts that it needs. 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.

3.2   Versioning Functions

Once an artifact has been opened in a workspace, the following versioning functions become available to create and store new versions of the artifact:

```
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 3.3).

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 3.7 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. An `nc_abortchange` performed on 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. This forces the CM system either

TABLE 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.

to commit any changes or to abandon them, thereby avoiding unintentional loss of changes.

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
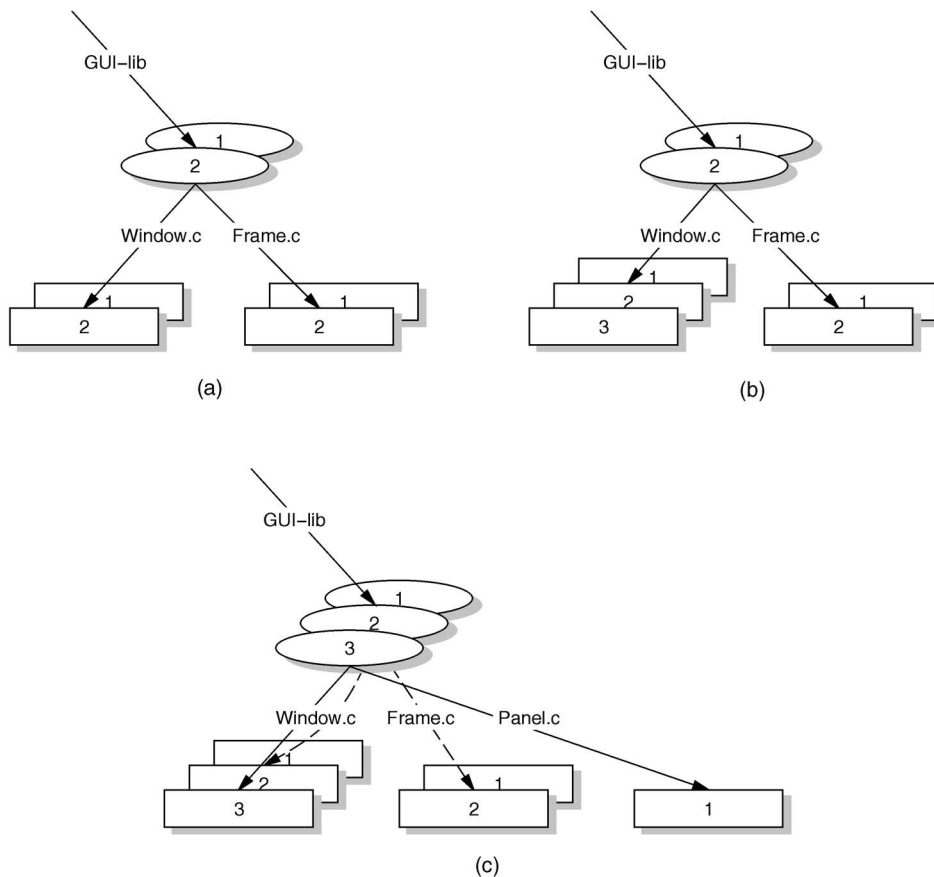
Fig. 7. Progressive states of an example collection.

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. 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 [30] or CoED [6]), other CM policies only version collections when the actual structure of the collection (i.e., its artifact membership) has changed (e.g., ShapeTools [31] or ClearCase [3]). 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, suppose we have a repository containing the artifacts depicted in Fig. 7a. Assume further that, using the function `nc_open`, a workspace has been created that contains version 2 of the artifact `GUI-lib` and its containing artifacts. To be able to modify the atom `Window.c` in the workspace, we invoke the function `nc_initiatechange`. Once the desired changes have been made, we use the function `nc_commitchange` to store a new version of the atom `Window.c` in the repository. The result is shown in Fig. 7b. The repository now contains three versions of the atom `Window.c`, but note that the collection `GUI-lib` has not changed since we did not invoke the function `nc_initiatechange` on that collection. Had we used the function `nc_commitchangeandreplace` instead of the function `nc_commitchange`, no new version would have been created for the artifact `Window.c`. In fact, the structure of the repository would still have looked like the one of Fig. 7a, even though the actual contents of version 2 of the atom `Window.c` would have changed.

3.3   Collection Functions

Similar to the way an editor can be used to change an atom in a workspace, collections need to be changed via some kind of mechanism. 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. This scheme allows many changes to a single collection to be grouped into a single change in the repository.

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 Fig. 4). 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. The new artifact will contain 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.

The function `nc_replaceversion` complements the other collection functions because 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. A situation for which this functionality is especially useful is when a CM policy programmer would like to provide an "undo" facility. For example, the facility can be used to replace a newer version of an artifact in a collection with an older one.

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 system 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 `nc_remove`, `nc_add`, and `nc_rename`.

To illustrate the collection functions, we continue the example of Fig. 7b. Assume that all artifacts are still open in the workspace. To manipulate the collection `GUI-lib`, the function `nc_initiatechange` is first used to gain proper permission. Then, to update the atom `Window.c` to its latest version, the function `nc_replaceversion` is applied. In addition, to provide a panel as opposed to a frame in the collection `GUI-lib`, the function `nc_remove` is used to remove the atom `Frame.c` and the function `nc_add` is used to add the atom `Panel.c`. These changes are transferred to the repository using the function `nc_commitchange`. As a result, the repository looks as shown in Fig. 7c. A new version of the collection `GUI-lib` has been created that reflects the new membership. In addition, because we used the function `nc_commitchange` instead of the function `nc_cimmitchangeandreplace`, 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 `GUI-lib`, then version 2 of the atom `Window.c` and version 2 of the atom `Frame.c` are listed as the collection members, whereas, if the function `nc_list` is used on version 3 of the collection, then version 3 of atom `Window.c` and version 1 of atom `Panel.c` are listed as members.

3.4 Distribution Functions

An important aspect of the distribution model discussed in Section 2.2 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 a need exists for control over the location of artifacts. 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 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.

3.5 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. During deletion, however, a specific rule is enforced: A version of an artifact can only be deleted if that version is not a member of a collection. Consider the example in Fig. 7c. A CM policy is allowed to delete version 1 of atom `Window.c`, but the deletion of version 2 is disallowed because it is a member of version 2 of the collection `GUI-lib`. While it may seem as though we are making a policy decision through this restriction, it is one that is intended simply to preserve the consistency of the repository structure.

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 reached in the versioned directed graph of artifacts (consider applying the function `nc_remove` on version 2 of the atom `Frame.c`). Second, it is possible to create 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 both cases, the storage space occupied by the artifact needs to be automatically reclaimed by an implementation of the abstraction layer.

3.6 Query Functions

The programmatic interface would not be complete without the ability to examine the state of artifacts. For example, when multiple clients share access to an artifact, they should be able to determine whether any new versions of the artifact have been created by another client. 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 system 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.

Although the names of the interface functions speak for themselves, we provide a one-sentence description and typical use of each. The function `nc_gettype` determines whether an artifact is a collection or an atom and is often used when recursively opening a collection and all its containing artifacts in a workspace. To manage version relationships, such as a revision history, the function `nc_version` can be used to determine the version of an artifact before and after the function `nc_commitchange` has been used to store some changes. The function `nc_lastversion` returns the version number of the last version of an artifact, and is used to check for new versions of an artifact that may have been added by another client. If some versions of an artifact have been deleted from a repository, the function `nc_existsversion` can be used to verify whether or not a particular version is still available. Finally, the functions `nc_isopen` and `nc_isinitiated` operate on artifacts in a workspace and are used to verify whether an artifact has been opened and whether it is allowed to change, respectively.

3.7 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 nonexistent. 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 numbers 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 attribute 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, the functions are powerful enough to directly model the locking schemes employed in such existing CM systems as RCS [47], CCC/Harvest [44], and others. If more sophisticated locking schemes are required, then a separate lock manager, such as Pern [23], should be used instead. This approach is consistent with the desire for locking to be orthogonal to the other functionalities of the interface.

4 THREE NUCM-BASED CM SYSTEMS

The abstraction layer, including its repository model and programmatic interface, has been implemented in the NUCM prototype[2] and was used to develop several CM systems, including the three, rather different ones described in this section. At present, two of those systems, namely, DVS [9] and SRM [49], are in everyday use, while the third system, WebDAV, represents an experimental implementation of an emerging standard in Web versioning [22], [52]. We also created proof-of-concept implementations of the widely known check-in/check-out and change-set policies [18], but used an earlier version of the prototype to do so; those implementations are presented elsewhere [50].

Below, we discuss each system and use parts of their implementations to illustrate how NUCM can be used to program particular CM policies. It should be noted that the policies themselves are not the contribution. Instead, the strength of NUCM lies in the ease with which these policies were constructed and the limited amount of work needed to make them suitable for use in a wide-area setting.

4.1 DVS

DVS (Distributed Versioning System) [9] is a versioning system that is focused on providing a distributed environment in which documents can be authored collaboratively.[3] 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 appropriate tools, and commit their changes from the workspace to a storage facility. This policy is similar to the one employed by RCS [47], except that DVS explicitly recognizes and versions collections and, moreover, operates in a wide-area setting.

DVS exhibits several characteristics that illustrate the power of the abstraction layer.

- *No special code needed to be developed for DVS to operate across a network.* DVS relies entirely on the mechanisms included in NUCM to support distribution. In fact, DVS can not only operate in a client-server mode, but it is also possible to federate multiple physical repositories into a single logical repository that is used by DVS.

- *Only approximately 3,000 new lines of source code were needed to create the full functionality of DVS.*[4] The newly written source code mainly 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 basic versioning, is inherited from NUCM.

- *The separation of policy from repository allows certain evolutions in the policies to occur incrementally.* This characteristic of NUCM came upon us unexpectedly. On one occasion, DVS was being used by 10 people at five different sites to jointly author a document. It turned out that the policy provided by DVS did not completely match the desired process. In response, some of the DVS functionality was changed and some new functionality was added. When the second version of DVS was subsequently and incrementally deployed to the various sites, no disruption of work occurred. Because the NUCM repository required no downtime and the artifacts in the repository needed no change, slightly different policies could be used by multiple authors at the same time.

To demonstrate how DVS is built upon the functions in the NUCM interface, Fig. 8 presents a portion of the DVS source code. (Note that all error handling has been removed from the example source code shown in this section.) The use of a NUCM interface function is highlighted by a "*". Illustrated is a procedure that synchronizes a workspace with the latest versions of the artifacts in a NUCM repository. The procedure allows either a single artifact or a recursive set of artifacts to be synchronized, depending on the value of the parameter recursive.

The procedure consists of three parts. In the first part, the version of the artifact in the workspace and its latest version in the repository are determined through the use of the functions nc_version and nc_lastversion. If these versions are the same, the artifact is up to date with respect to the repository. If they are not, the second part of the procedure takes care of synchronizing the two by replacing the version in the workspace with the version in the repository. Additionally, before the latest version of the artifact is opened in the workspace, the current version is closed if it had been opened previously. To avoid the loss of changes that may have been made to an artifact, the routine first verifies whether the current version has been initiated for change. If so, the current version of the artifact is preserved in the workspace and the artifact is not synchronized. The third and final part of the procedure deals with the recursive nature of the synchronization of a workspace. If the artifact to be synchronized is a collection, its contained artifacts are obtained and each of these artifacts is in turn synchronized through a recursive call.

4.2 SRM

SRM (Software Release Manager) [43], [49] is a tool that addresses the problem of software release management.[5] SRM supports the release of "systems of systems" from multiple, geographically distributed sites. In particular,

---

2. http://www.cs.colorado.edu/serl/cm/nucm.html.
3. http://www.cs.colorado.edu/serl/cm/dvs.html.

4. In this paper, all counts of source code lines include empty lines and comments.
5. http://www.cs.colorado.edu/serl/cm/SRM.html.

```
1     int synchronize_workspace(const char* pathname, int recursive)
2     {
3         //
4         // Part 1:  Determine the current and latest version of the artifact.
5         //
6         strip_version_r(pathname, strippedpath);
7 *       nc_version(strippedpath, "", currentversion);
8 *       nc_lastversion(strippedpath, "", lastversion);
9
10        //
11        // Part 2:  If needed, get the latest version of the artifact, unless
12        // it is checked out.
13        //
14        if (strcmp(lastversion, currentversion) != 0) {
15            do_open = 1;
16 *          if (nc_isopen(strippedpath, "."))
17 *              if (!nc_isinitiated(strippedpath, "."))
18 *                  nc_close(strippedpath, ".", 0);
19              else
20                  do_open = 0;
21            if (do_open) {
22                set_version_x(strippedpath, lastversion);
23 *              nc_open(strippedpath, ".", ".", "");
24            }
25        }
26
27        //
28        // Part 3:  If necessary, recursively synchronize the workspace.
29        //
30        if (recursive) {
31 *          atype = nc_gettype(strippedpath, ".");
32            if (atype == COLLECTION) {
33 *              nc_list(strippedpath, "", &members);
34              chdir(strippedpath);
35              start = members;
36              while (members != NULL) {
37                  synchronize_workspace(members->name, recursive);
38                  members = members->next;
39              }
40 *              nc_destroy_memberlist(start);
41              if (strcmp(strippedpath, ".") != 0)
42                  chdir("..");
43            }
44        }
45    }
```

Fig. 8. DVS routine to synchronize a workspace.

SRM tracks dependency information to automate the packaging and retrieval of components. Software vendors are supported by a simple release process that hides the physical location of dependent components. Customers are supported by a simple retrieval process that allows selection and downloading of components whose physical locations are hidden.

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 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.

Of relevance to the discussion in this paper is the flexibility that NUCM provides in the creation of a distributed repository. In particular, we examine the way new participants can join a federated SRM repository. To facilitate this functionality, each participating site maintains a NUCM repository that contains the releases they have created. Additionally, one of the NUCM repositories in the federation maintains a collection that contains all releases from all sites. This is illustrated in Fig. 9 by the repositories in Rotterdam and Boulder. Both repositories contain a collection nucm_root that contains a local collection my_releases and a global collection all_releases. In each repository, the collection my_releases contains the releases made by that site. The collection all_releases, which is shared by both sites, contains all the releases. Note that the repository in Milano is not part of the SRM federation at this point.

The procedure join, presented in Fig. 10, illustrates how a new physical repository can join an existing SRM federation. It operates by creating a new collection for local releases, my_releases, and linking to the existing collection that contains all releases, all_releases. To do so, it first sets up a workspace containing its main collection nucm_root, then allows the collection to change
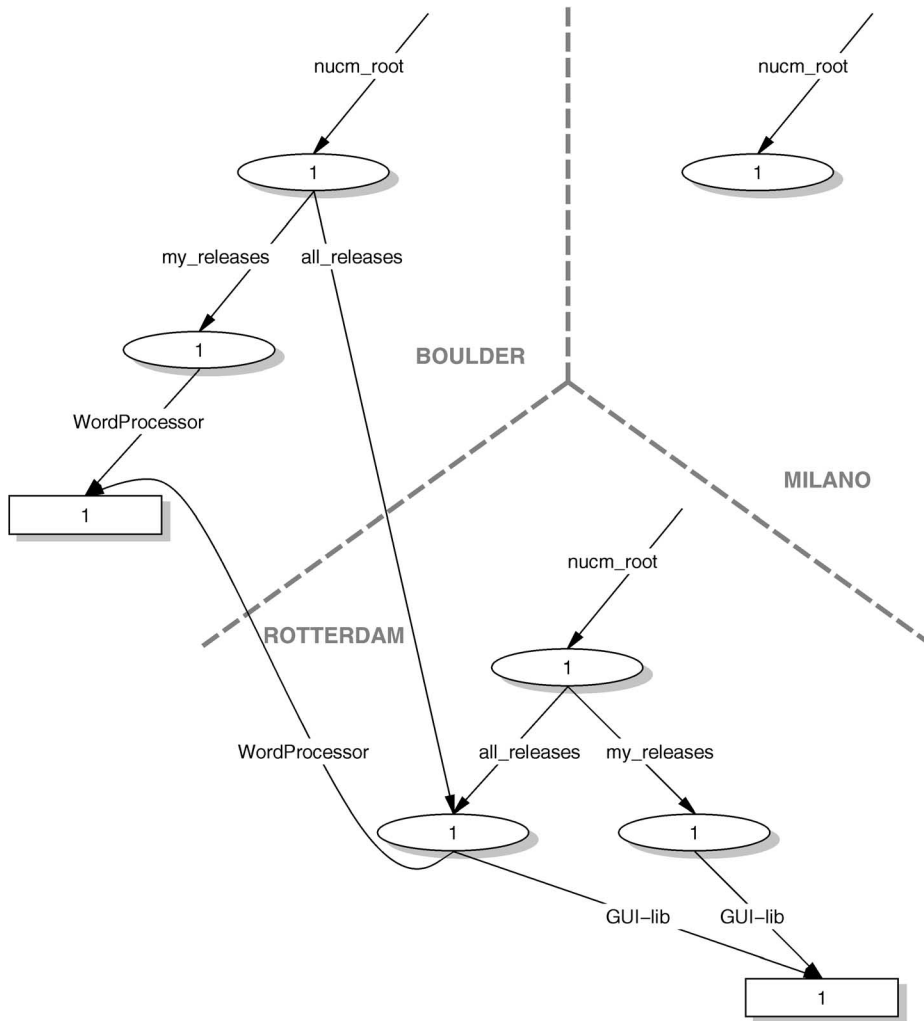
Fig. 9. Federated SRM repository before Milano joins the federation.

```
 1    int join(const char* host, const char* port)
 2    {
 3       //
 4       // Part 1: Set up a workspace.
 5       //
 6       sprintf(all_releases, "//%s:%s/nucm_root/all_releases", host, port);
 7       sprintf(my_nucmroot, "//%s:%s/nucm_root", NUCMHOST, NUCMPORT);
 8       sprintf(my_releases, "WORKSPACE/my_releases");
 9       sprintf(collection, "WORKSPACE/nucm_root");
10 *    nc_open(my_nucmroot, "", WORKSPACE, "");
11 *    nc_initiatechange(collection);
12
13       //
14       // Part 2: Add a new artifact for my personal releases.
15       //
16       mkdir(my_releases);
17 *    nc_add(my_releases, "", collection, "");
18
19       //
20       // Part 3: Import an existing artifact for the list of all releases.
21       //
22 *    nc_add(all_releases, "", collection, "");
23 *    nc_commitchangeandreplace(collection, "");
24 *    nc_close(collection, "", 0);
25    }
```

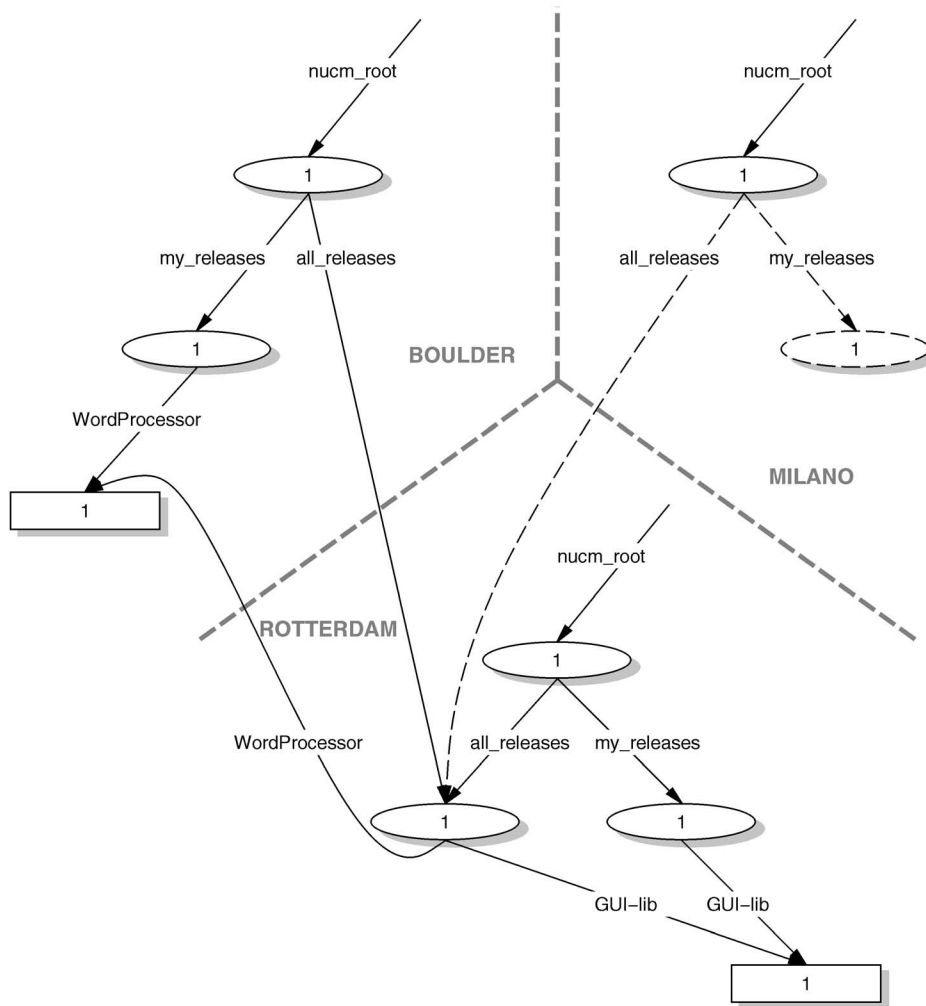Fig. 10. SRM routine to join a federation.

Fig. 11. Federated SRM repository after Milano joins the federation.

by using the function `nc_initiatechange` and, subsequently, uses the function `nc_add` to add the newly created collection `my_releases` and the existing collection `all_releases` to the collection `nucm_root`. The collection `nucm_root` is then stored in the repository using the function `nc_commitchangeandreplace` and the workspace is finally removed by using the function `nc_close`. The result of all these actions is shown in Fig. 11. Assuming that the Milano site is being joined with the SRM repository of Rotterdam and Boulder, the dashed lines indicate the new artifact and the membership relationships that are created by the procedure `join`. Once the artifact and the relationships are created, Milano is a full part of the SRM federation; when it adds new releases to the repository, they can be accessed from all sites.

The main advantage in using NUCM to develop SRM is that distribution could be isolated. Only two lines in the example source code of Fig. 11 explicitly deal with distribution: In line 6 and line 7, the remote repository that contains the collection `all_releases` and the local repository that is going to join the SRM repository are explicitly identified. After that, all policy programming is, in fact, transparent with respect to distribution. This particularly exhibits itself in other portions of the SRM policy code. Adding or removing releases to the SRM repository can

simply be programmed as additions and removals to the collections `my_releases` and `all_releases` since NUCM tracks the physical location of these collections. Similarly, through the collection `all_releases`, any site can retrieve releases from the other sites without having to know where a release is physically located.

This strength of isolating distribution exhibits itself throughout SRM. Only a small part of the complete implementation, less than two percent, 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.

4.3 WebDAV

WebDAV [22], [52] is an emerging standard that proposes to add authoring and versioning primitives to the HTTP protocol [20]. In particular, the standard proposes extensions in the following five areas:

- *Metadata.* To be able to describe Web resources, WebDAV proposes the creation of new HTTP methods that add metadata to Web resources, as well as methods to query and retrieve the metadata.
- *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.

Although the objective of WebDAV (i.e., providing an infrastructure for distributed authoring and versioning) is slightly different from the objective of NUCM (i.e., providing a distributed repository to construct configuration management policies), the interface methods that have been proposed for both are strikingly similar. Only two major differences exist. First, NUCM includes a naming model that explicitly incorporates a mechanism to refer to versions of artifacts, whereas the naming scheme of WebDAV does not define such a mechanism. Second, WebDAV specifies a particular versioning policy, namely, the RCS-like lockable version tree, whereas NUCM is generic with respect to versioning policies.

Because of the similarity between NUCM and WebDAV, it seems advantageous to use NUCM to implement WebDAV, at least in order to quickly determine the utility of its policy. To this end, we created a simple HTTP server that is also a NUCM client. Most of the new HTTP methods translate into direct calls to the NUCM interface, but some require more work. In particular, the versioning routines of WebDAV prescribe a policy that is based on a version tree. To implement this tree, we have to map the versions in the tree to versions of NUCM artifacts. In our implementation, this mapping is created by storing two NUCM artifacts for each WebDAV artifact, namely, the actual artifact and an associated artifact that stores the version tree for that artifact. In addition, the version tree artifact has attributes associated with it that map each version number in the tree to a NUCM version number.

Fig. 12 shows how one of the procedures in our WebDAV implementation, namely, `checkin`, takes advantage of this approach. (Note that, because the WebDAV standard has continued to evolve, the example given here is not completely consistent with the current version of the standard.) The function stores a new version of an artifact and updates the version tree accordingly. Its functionality can be divided into five separate parts.

In the first part, several parameters used in the remainder of the function are determined. The names of the artifact being checked in and its corresponding version tree artifact are constructed first. Subsequently, the type of the artifact being checked in and its NUCM version number under which it was checked out are obtained.

In the second part of the function, the new version of the artifact is read from the WebDAV client and, subsequently, stored in the repository through the use of the function `nc_commitchange`.

The third part of the function serves an important role: It is the part that updates the version tree. We do not show the actual algorithm that determines the new version number since it does not involve any use of NUCM functions. Instead, it is shown how the version tree is obtained from the repository, updated with the new version information and stored back into the repository. Note the use of the function `nc_commitchangeandreplace` to replace the version tree since there is no need to store multiple versions of the version tree itself.

The fourth part of the function sets new attributes for some of the artifacts in the repository. In particular, it preserves the type of the artifact that was checked in and relates the version in the version tree with the NUCM version of the new artifact. Note that the type information is attached to the new version of the artifact, for which a new NUCM name is first constructed.

Finally, in the fifth part of the function, the artifact that was previously checked out and locked for modifications is unlocked so that other users can now modify this version.

Once again, the reusability of NUCM proved to be valuable in the implementation of WebDAV. The total number of lines of source code that were developed to create a prototype WebDAV implementation was only 1,500, of which approximately 40 percent accounts for a graphical user interface that can be used to perform WebDAV operations.

Admittedly, our experimental implementation does not cover all the functionality of WebDAV. However, the limited amount of code that needed to be developed and the minimal time required to produce that code together demonstrate an important aspect of NUCM: It can be used to quickly demonstrate 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 be explored with an early implementation.

5 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 [47], SCCS [41], Sablime [5]), others provide more complete configuration management solutions (e.g., CVS [7], CoED [6], Perforce [38]), and yet others provide integrated environments that incorporate process management and/or problem tracking facilities (e.g., Adele [17], ClearCase [3], Continuus [12]). With respect to distribution, some of the CM systems are only suited for use at a single site (e.g., EPOS [35], ShapeTools [31], SourceSafe [32]), others incorporate a simple, sometimes Web-based, client-server interface (e.g., DCVS [7], Perforce [38], WWCM [25]), and yet others provide more advanced distribution mechanisms such as replication (e.g., ClearCase Multisite [2], Continuus DCM [13], PVCS SiteSync [28]). To understand the position of the abstraction layer and its implementation in NUCM in this large space of CM systems, we examine the evolution of CM system architectures.

Fig. 13a shows the architecture that has traditionally been used: A CM system is constructed as a single, monolithic entity that tightly integrates its storage mechanism with its CM policy. This approach is still the way in which most

```
 1    int checkin(const char* path, const char* oldversion, FILE* client)
 2    {
 3        //
 4        // Part 1: Determine necessary information.
 5        //
 6        sprintf(tree, "//%s:%s/nucm_root/TREE/%s", NUCMHOST, NUCMPORT, path);
 7        sprintf(treefilename, "%s/%s", WORKSPACE, (rindex(tree, '/')+1));
 8        sprintf(artifact, "//%s:%s/nucm_root/%s:%s", NUCMHOST, NUCMPORT, path, nucmversion);
 9        sprintf(filename, "%s/%s", WORKSPACE, (rindex(artifact, '/')+1));
10 *      nc_getattributevalue(tree, "", oldversion, nucmversion);
11 *      nc_getattributevalue(artifact, "", "TYPE", type);
12
13        //
14        // Part 2: Store new version of the artifact.
15        //
16 *      nc_open(artifact, "", WORKSPACE, "");
17 *      nc_initiatechange(filename, "");
18        fd = open(filename, O_TRUNC — O_WRONLY);
19        while ((n = fread(bytes, 4096, 1, client)) ¿ 0)
20            write(fd, bytes, n);
21        close(fd);
22 *      nc_commitchange(filename, "", newnucmversion);
23 *      nc_close(filename, "", 0);
24
25        //
26        // Part 3: Update the version tree.
27        //
28 *      nc_open(tree, "", WORKSPACE, "");
29 *      nc_initiatechange(treefilename, WORKSPACE);
30        fd = fopen(treefilename, "r+");
31        ...
32        ... /* Determine new version number */
33        ...
34        sprintf(line, "%s CHILD OF %s"n", newversion, oldversion);
35        fputs(line, fd);
36        fclose(fd);
37 *      nc_commitchangeandreplace(filename, "");
38 *      nc_close(filename, "", 0);
39
40        //
41        // Part 4: Set new attributes.
42        //
43        sprintf(newartifact, "//%s:%s/nucm_root/%s:%s", NUCMHOST, NUCMPORT, path, newnucmversion);
44 *      nc_testandsetattribute(tree, "", newversion, newnucmversion);
45 *      nc_testandsetattribute(newartifact, "", "TYPE", type);
46
47        //
48        // Part 5: We are done, unlock the artifact.
49        //
50 *      nc_removeattribute(artifact, "", "LOCK");
51    }
```

Fig. 12. WebDAV routine to check in a file.

CM systems are built, as exemplified by CoED [6] and DSCS [33], both of which were only recently developed. The advantage of this architecture is that it allows a CM system to optimize its storage to precisely match the needs of the CM policy. A clear disadvantage, however, is that a resulting CM system tends to be rather inflexible [15]. Moreover, such a CM system typically has to be constructed from the ground up, which is a major effort even today.

As CM systems have become more advanced, some have turned towards using a commercial, generic database management system as the underlying storage mechanism (e.g., ClearCase [3], Continuus [12], TrueCHANGE [45]). Illustrated in Fig. 13b, the advantage of this solution is that the database management system provides a reliable and reusable platform that offers such services as transactions, concurrency control, and rollbacks. These services no longer have to be implemented by the organization that develops the CM system.

Following the pattern of providing an increased level of abstraction with which to build CM systems, Fig. 13c illustrates that NUCM represents the next step in this evolutionary pattern. Although NUCM currently does not provide the same level of robustness and reliability as a database, the abstraction layer that it provides has the advantage of being highly specialized towards configuration management. Thus, as compared to 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 implementation. As stated, NUCM currently lacks such essential
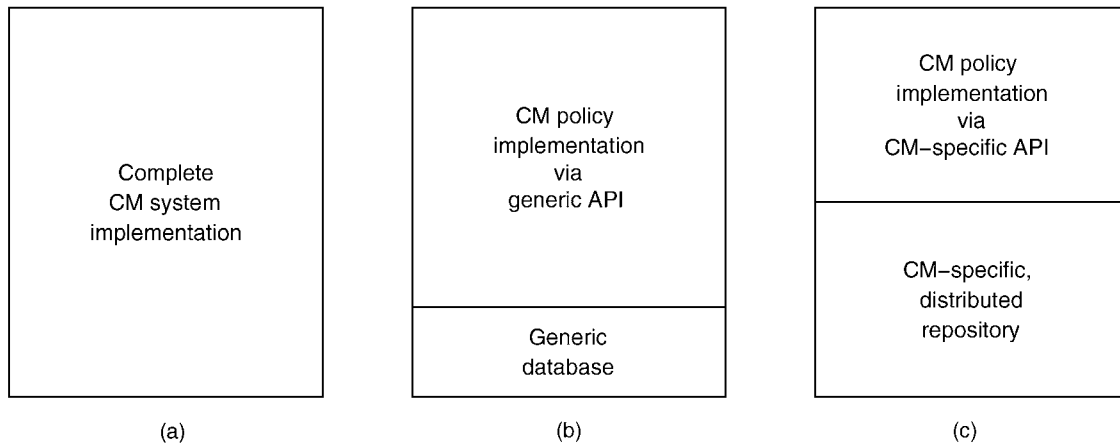
Fig. 13. Evolution of CM system architectures.

services as transactions, rollbacks, and caching, but it is anticipated that these can be incorporated in future implementations. Then, actual CM systems can be implemented based on the abstraction layer defined in this paper, while exhibiting the same qualities as a CM system built on top of a traditional database.

NUCM is only one of several systems that fall into the category of Fig. 13c. The other systems are CME [25], Gradient [4], CoMa [51], and ScmEngine [10].

CME extends RCE [26], itself a programmatic interface to RCS [47], with collection management. CME is similar to NUCM 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, the programmatic interface of CME is not generic with respect to CM policies since it only contains functions that implement the check-in/check-out policy. The second difference is that CME is not distributed since it only interfaces to a single repository at a single site. Thus, whereas NUCM provides support for the construction of a variety of distributed CM policies, CME only provides support for the construction of centralized CM policies that are based on the check-in/check-out model.

Gradient is a CM repository that is based on automatic replication. Each update that is made to an artifact is broadcast 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 NUCM in that 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 check-in/check-out policy).

CoMa is perhaps the one system that is closest in nature to the functionality provided by NUCM. 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 NUCM. 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 NUCM, however, CoMa is limited in that it only supports the construction of variations of the check-in/check-out policy. Moreover, it does not support the distribution of artifacts over multiple physical locations. Thus, even though CoMa is more generically applicable than CME, it is similarly limited in that it only supports a small number of centralized CM policies.

ScmEngine is a distributed CM repository based on the X.500 directory protocol [39]. 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 CM client programs. This distribution mechanism is, in essence, the same as the one defined by the distribution model of the abstraction layer. However, the remainder of the repository model and the programmatic interface provided by ScmEngine are significantly weaker than the ones defined by our abstraction layer. The repository model does not include collections and lacks the concept of version qualifiers to navigate in the version space. Moreover, the programmatic interface is very specific and lacks support for the construction of a wide variety of CM policies, only supporting the traditional check-in/check-out policy.

Outside the domain of configuration management, we can identify groupware and versioned databases as two important lines of work that are closely related to the work presented in this paper. In groupware, the need for distribution, versioning, and workspaces seems to imply that our abstraction layer could be appropriate for use in constructing a groupware system. However, this is not so. Whereas the abstraction layer is based on the principle of workspaces that provide isolation from changes by others, groupware systems tend to focus on collaborative workspaces [16], [21]. The set of issues involved in supporting each is rather different and, consequently, we believe groupware, even though related, falls outside of the domain of NUCM.

Versioned databases (e.g., ODE [1], TVOO [42]) are related to NUCM since NUCM itself can be viewed as a versioned database. In fact, many of the features of NUCM are shared by versioned databases. However, an important

difference exists, which is the presence of a specific repository model and its associated programmatic interface. Whereas these are generic in nature in a versioned database (e.g., an entity relationship model with SQL), both are highly specialized by our abstraction layer. In essence, one could consider the abstraction layer that is incorporated in NUCM 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).

6 CONCLUSION

For the past few years, the field of configuration management has been in a consolidation phase with the research results of the 1980s being transferred to the commercial products of the 1990s. Nonetheless, new CM systems are still being proposed and constructed. Some of these are new entries in an increasingly competitive marketplace. Others implement proprietary solutions that are tailored to the situation at hand. Yet others explore new ground and form the basic research that will lead to the next generation of CM systems. During their design and implementation, though, all face what we consider to be one of the most pertinent problems in the field of configuration management: No suitable platform exists that can serve as a flexible testbed for the rapid construction of potentially distributed CM systems.

Based on the critical observation that, to effectively address this problem it is necessary to separate CM repositories from CM policies, this paper has introduced a novel abstraction layer that represents a first step towards addressing this problem. The abstraction layer precisely defines 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.

The abstraction layer was designed to facilitate the rapid construction of, and experimentation with, CM policies. However, it has proven to facilitate more than that. DVS and SRM, two of the systems that were originally constructed to demonstrate the applicability and flexibility of the abstraction layer, have evolved into complete CM systems. Both are now in use in settings that involve multiple parties in multiple geographical locations and both continue to evolve with respect to the functionality they provide. The abstraction layer, thus, not only supports the construction of new CM policies, but also their gradual evolution into more mature systems.

Our work does not end here. Although we certainly believe that the abstraction layer is a step in the right direction towards providing a generic, reusable, and distributed platform for CM policy programming, much work remains to be done. In particular, it is our belief that the abstraction layer facilitates the construction of standard policy libraries, thereby even further reducing the effort of implementing a CM system. Moreover, we expect to be able to use the abstraction layer as a vehicle for exploring other important problems in configuration management. For example, we believe the abstraction layer provides a suitable platform for investigating the problems of CM policy integration [37].

REFERENCES

[1] R. Agrawal, S. Buroff, N.H. Gehani, and D. Shasha, "Object Versioning in ODE," *Proc. Seventh Int'l Conf. Data Eng.,* pp. 446–455, Apr. 1991.
[2] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner, "ClearCase MultiSite: Supporting Geographically-Distributed Software Development," *Software Configuration Management: Int'l Conf. Software Eng. SCM-4 and SCM-5 Workshops Selected Papers,* pp. 194-214, 1995.
[3] Atria Software, *ClearCase Concepts Manual.* Natick, Mass., 1992.
[4] D. Belanger, D. Korn, and H. Rao, "Infrastructure for Wide-Area Software Development," *Proc. Sixth Int'l Workshop Software Configuration Management,* pp. 154-165 1996.
[5] Bell Labs, Lucent Technologies, *Sablime v5.0 User's Reference Manual.* Murray Hill, New Jersey, 1997.
[6] L. Bendix, P.N. Larsen, A.I. Nielsen, J.L.S. Petersen, "CoED—A Tool for Versioning of Hierarchical Documents," *Proc. Eighth Int'l Symp. System Configuration Management,* 1998.
[7] B. Berliner, "CVS II: Parallelizing Software Development," *Proc. 1990 Winter USENIX Conference,* pp. 174-187, 1990.
[8] C. Burrows and I. Wesley, *Ovum Evaluates Configuration Management.* Burlington, Mass.: Ovum Ltd., 1998.
[9] A. Carzaniga, *DVS 1.2 Manual.* Dept. of Computer Science, Univ. of Colorado, Boulder, June 1998.
[10] J.X. Ci, M. Poonawala, and W.-T. Tsai, "ScmEngine: A Distributed Software Configuration Management Environment on X.500," *Proc. Seventh Int'l Workshop Software Configuration Management,* pp. 108-127, 1997.
[11] 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 Eng. Inst., Pittsburgh, Penn., May 1998.
[12] Continuus Software Corporation, *Continuus Task Reference.* Irvine, Calif., 1994.
[13] Continuus Software Corporation, *Distributed Code Management for Team Engineering.* Irvine, Calif., 1998.
[14] S. Dart, "Concepts in Configuration Management Systems," *Proc. Third Int'l Workshop Software Configuration Management,* pp. 1–18, 1991.
[15] S. Dart, "Not All Tools are Created Equal," *Application Development Trends,* vol. 3, no. 10, pp. 39–54, Oct. 1996.
[16] A. Dix, T. Rodden, and I. Sommerville, "Modeling the Sharing of Versions," *Proc. Sixth Int'l Workshop Software Configuration Management,* pp. 282-290, 1996.
[17] J. Estublier and R. Casallas, "The Adele Configuration Manager," *Configuration Management, Trends in Software,* W. Tichy, ed., no. 2, pp. 99–134, 1994.
[18] P.H. Feiler, "Configuration Management Models in Commercial Environments," Technical Report SEI-91-TR-07, Software Eng. Inst., Pittsburgh, Penn., Apr. 1991.
[19] P.H. Feiler and G. Downey, "Transaction-Oriented Configuration Management: A Case Study," Technical Report CMU/SEI-90-TR-23, Software Eng. Inst., Pittsburgh, Penn., 1990.
[20] R. Fielding, J. Gettys, J.C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol–HTTP/1.1," Internet Proposed Standard RFC 2068, Jan. 1998.
[21] P. Fröhlich and W. Nejdl, "WebRC: Configuration Management for a Cooperation Tool," *Proc. Seventh Int'l Workshop Software Configuration Management,* pp. 175-185, 1997.

[22] Y.Y. Goland, E.J. Whitehead, Jr., A. Faizi, S. Carter, and D. Jensen, "HTTP Extensions for Distributed Authoring—WEBDAV," Internet Proposed Standard RFC 2518, Feb. 1999.

[23] G.T. Heineman, "A Transaction Manager Component for Cooperative Transaction Models," PhD thesis, Columbia Univ., Dept. of Computer Science, New York, June 1996.

[24] T. Hung and P.F. Kunz, "UNIX Code Management and Distribution," Technical Report SLAC-PUB-5923, Stanford Linear Accelerator Center, Stanford, Calif., Sept. 1992.

[25] J.J. Hunt, F. Lamers, J. Reuter, and W.F. Tichy, "Distributed Configuration Management via Java and the World Wide Web," *Proc. Seventh Int'l Workshop Software Configuration Management,* pp. 161-174, 1997.

[26] J.J. Hunt and W.F. Tichy, *RCE API Introduction and Reference Manual.* Germany: Xcc Software, 1997.

[27] J.J. Hunt, K.-P. Vo, and W.F. Tichy, "Delta Algorithms: An Empirical Analysis," *ACM Trans. Software Eng. and Methodology,* vol. 7, no. 2, pp. 192–214, Apr. 1998.

[28] INTERSOLV, *PVCS VM SiteSync and Geographically Distributed Development.* Rockville, Md., 1998.

[29] R. Leung, "Versioning on Legal Applications Using Hypertext," *Proc. Workshop Versioning in Hypertext Systems,* Sept. 1994.

[30] Y.-J. Lin and S.P. Reiss, "Configuration Management with Logical Structures," *Proc. 18th Int'l Conf. Software Eng.,* pp. 298–307, Mar. 1996.

[31] A. Mahler and A. Lampen, "An Integrated Toolset for Engineering Software Configurations," *Proc. ACM SOFSOFT/SIGPLAN Software Eng. Symp. Practical Software Eng. Environments,* pp. 191–200, Nov. 1988.

[32] Microsoft Corporation, *Managing Projects with Visual SourceSafe,* Redmond, Wash., 1997.

[33] B. Milewski, "Distributed Source Control System," *Proc. Seventh Int'l Workshop Software Configuration Management,* pp. 98-107, 1997.

[34] Mortice Kern Systems, Inc., *Untangling the Web: Eliminating Chaos,* Waterloo, Canada, 1996.

[35] B.P. Munch, "Versioning in a Software Engineering Database—the Change-Oriented Way," PhD thesis, DCST, NTH, Trondheim, Norway, Aug. 1993.

[36] B. O'Donovan and J.B. Grimson, "A Distributed Version Control System for Wide Area Networks," *Software Eng. J.,* Sept. 1990.

[37] F. Parisi-Presicce and A.L. Wolf, "Foundations for Software Configuration Management Policies Using Graph Transformations," *Proc. 2000 Conf. Foundational Aspects of Software Eng.,* Mar. 2000.

[38] Perforce Software, *Networked Software Development: SCM over the Internet and Intranets.* Alameda, Calif., Mar. 1998.

[39] Recommendation X.500 (08/97)–Information Technology–Open Systems Interconnection–the Directory: Overview of Concepts, Models and Services, Aug. 1997.

[40] R.J. Ray, "Experiences with a Script-Based Software Configuration Management System," *Software Configuration Management: Int'l Conf. Software Eng. SCM-4 and SCM-5 Workshops Selected Papers,* 1995.

[41] M.J. Rochkind, "The Source Code Control System," *IEEE Trans. Software Eng.,* vol. 1, no. 4, pp. 364–370, Dec. 1975.

[42] L. Rodriguez, H. Ogata, and Y. Yano, "An Access Mechanism for a Temporal Versioned Object-Oriented Database," *IEICE Trans. Information and Systems,* vol. E82–D, no. 1, pp. 128–135, Jan. 1999.

[43] R.A. Smith, "Analysis and Design for a Next Generation Software Release Management System," Master's thesis, Univ. of Colorado, Boulder, Dec. 1999.

[44] Softool Corp., *CCC/Manager, Managing the Software Life Cycle across the Complete Enterprise,* Goleta, Calif., 1994.

[45] Software Maintenance & Development Systems, Inc., *Aide de Camp Product Overview,* Concord, Mass., Sept. 1994.

[46] SQL Software, *The Inside Story: Process Configuration Management with PCMS Dimensions,* Vienna, Va., 1998.

[47] W.F. Tichy, "RCS, A System for Version Control," *Software—Practice and Experience,* vol. 15, no. 7, pp. 637–654, July 1985.

[48] A. van der Hoek, "A Generic, Reusable Repository for Configuration Management Policy Programming," PhD thesis, Dept. of Computer Science, Univ. of Colorado, Boulder, Jan. 2000.

[49] A. van der Hoek,, R.S. Hall, D.M. Heimbigner, and A.L. Wolf, "Software Release Management," *Proc. Sixth European Software Eng. Conf.,* pp. 159-175, Sept. 1997.

[50] A. van der Hoek,, D.M. Heimbigner, and A.L. Wolf, A Generic, Peer-to-Peer Repository for Distributed Configuration Management," *Proc. 18th Int'l Conf. Software Eng.,* pp. 308–317, Mar. 1996.

[51] B. Westfechtel, "A Graph-Based System for Managing Configurations of Engineering Design Documents," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 6, no. 4, pp. 549–583, 1996.

[52] E.J. Whitehead, Jr., "World Wide Web Distributed Authoring and Versioning (WebDAV): An Introduction," *StandardView,* vol. 5, no. 1, pp. 3–8, Mar. 1997.

[53] A. Zeller and G. Snelting, "Unified Versioning Through Feature Logic," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 4, pp. 398–441, Oct. 1997.

André van der Hoek received a joint BS and MS degree in business-oriented computer science from the Erasmus University, Rotterdam, and the PhD degree in computer science from the University of Colorado at Boulder. He is an assistant professor in the Department of Information and Computer Science and a faculty member of the Institute for Software Research, both at the University of California, Irvine. His research interests include configuration management, software architecture, configurable distributed systems, and software education. He has developed several CM systems, was a cochair of the Ninth International Symposium on System Configuration Management, and is chair of the Tenth International Workshop on Software Configuration Management. He is a member of the IEEE and the IEEE Computer Society.

Antonio Carzaniga received the Laurea degree in electronic engineering and the PhD degree in computer science from Politecnico di Milano, Italy. He is currently a research associate with the Department of Computer Science at the University of Colorado at Boulder. His research interests are in the areas of distributed systems engineering, software engineering, computer networks, content-based routing, middleware, programming languages, and software engineering tools.

Dennis Heimbigner received the PhD degree in computer science from the University of Southern California, Los Angeles. He is a research faculty member in the Department of Computer Science, University of Colorado at Boulder. Prior to that, he was at TRW in Redondo Beach, California. His current research interests include distributed computing, peer-to-peer computing, and configuration management. He has published papers in the areas of configuration management, distributed computing, software process, software development environments, concurrent programming, and programming language semantics. He is a member of the IEEE and the IEEE Computer Society.

Alexander L. Wolf received the PhD degree in computer science from the University of Massachusetts at Amherst. He is a faculty member in the Department of Computer Science, University of Colorado at Boulder. Previously, he was at AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests are in the discovery of principles and development of technologies to support the engineering of large, complex software systems. He has published papers in the areas of software engineering environments and tools, software process, software architecture, configuration management, distributed systems, and persistent object systems. Dr. Wolf served as program cochair of the 2000 International Conference on Software Engineering (ICSE 2000), is currently serving as vice chair of the ACM Special Interest Group in Software Engineering (SIGSOFT), and is on the editorial board of *ACM Transactions on Software Engineering and Methodology* (TOSEM). He is a member of the IEEE Computer Society.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.