An Experimental, Pluggable Infrastructure for Modular Configuration Management Policy Composition

Ronald van der Lingen and André van der Hoek
Department of Informatics
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{vdlingen,andre}@ics.uci.edu

Abstract

Building a configuration management (CM) system is a difficult endeavor that regularly requires tens of thousands of lines of code to be written. To reduce this effort, several experimental infrastructures have been developed that provide reusable repositories upon which to build a CM system. In this paper, we push the idea of reusability even further. Whereas existing infrastructures only reuse a generic CM model (i.e., the data structures used to capture the evolution of artifacts), we have developed a novel experimental infrastructure, called MCCM, that additionally allows reuse of CM policies (i.e., the rules by which a user evolves artifacts stored in a CM system). The key contribution underlying MCCM is that a CM policy is not a monolithic entity; instead, it can be composed from small modules that each address a unique dimension of concern. Using the pluggable architecture and base set of modules of MCCM, then, the core of a desired new CM system can be rapidly composed by choosing appropriate existing modules and implementing any remaining modules only as needed. We demonstrate our approach by showing how the use of MCCM significantly reduces the effort involved in creating several representative CM systems.

1. Introduction

Despite the availability of many different CM systems that range in functionality from relatively simple version archives to advanced process-based environments, new CM systems continue to be developed on a regular basis. Some of these systems enter the marketplace, either commercially [11,15,22] or as free alternatives [6,8,26]. Others are developed primarily for in-house use [1,18]. Yet others are academic in nature and explore the boundaries of CM [4,5,8,9].

Unfortunately, designing and implementing a new CM system is difficult [7]. Anecdotal evidence indicates that it regularly takes a minimum of several years and tens of thousands of lines of code to create a fully-functional new CM system. Consider, for instance, the Subversion project [26]. Initiated in May 2000, Subversion now consists of over 100,000 lines of code and has iterated through more than 25 (alpha) releases. It still has a number of issues to be addressed and completion of the project is not expected soon. This is surprising given the modest goal of Subversion: to provide a better implementation of CVS that resolves certain functional deficiencies [27].

Various factors contribute to needing this kind of effort when building a new CM system. Some factors are inherent to the CM system being built and cannot be avoided, for instance when a CM system is to contain novel functionality. That functionality has to be created from scratch (e.g., one goal of Subversion is to operate securely in distributed settings; goals of other new CM systems include a unique graphical user interface; etc.). Other factors, however, are fundamental to the field at large. In this paper, we focus on one of those fundamental factors: *the lack of an effective platform with which CM policies can be compactly expressed and (at least partially) reused in the implementation of new CM systems*. CM policies constitute the rules by which a user evolves the artifacts stored in a CM system. Most CM policies share similar concerns and their implementations are often similar in nature. In Subversion, for instance, a non-trivial part of its code base reimplements significant parts of the CM policy of CVS. If such common code could be factored out and reused across multiple CM systems, significant savings in terms of time and effort could be achieved.

In this paper, we describe MCCM, a new experimental infrastructure that we specifically designed to address this problem. Compared to existing infrastructures [29,32,34], which only support reuse of a generic CM model (i.e., the data structures used to capture the evolution of artifacts), MCCM pushes the idea of reusability further—from just reusing a generic CM model to additionally allowing reuse of critical parts of CM policies. MCCM is based on the key observation that a CM policy is not monolithic; rather, it can be composed from small, reusable modules that each address a unique dimension of concern. Using these modules, the core of a desired new CM system can be rapidly assembled by first choosing an appropriate set of existing modules, then implementing any additional modules as needed, and finally plugging the resulting set of modules into the generic, pluggable, and distributed architecture provided by MCCM.

MCCM explicitly distinguishes constraint modules and action modules. Constraint modules plug into the MCCM server (repository) side to maintain, as prescribed by a desired CM policy, the consistency of the CM model. For example, constraint modules may be used to prevent branching or disallow replication of artifacts over multiple repositories. Action modules plug into the MCCM client (workspace) side to, also per the desired policy, enact the rules of that policy. For example, action modules may lock necessary artifacts, or determine whether particular actions should operate hierarchically (e.g., a commit of a collection leads to a commit of its constituent artifacts).

To demonstrate the use of MCCM, we performed two experiments. In our first experiment, we built approximations of RCS [25], CVS [3], and Subversion [26], as well as a prototype CM system based on change sets. Although none of these systems are as fully functional as their real-world counterparts, they illustrate reuse of modules across a variety of CM policies. Our second experiment compares an existing CM system, DVS [4], as it was originally implemented using NUCM [29] and now re-implemented using MCCM. This experiment shows the key benefit of our approach: a significant reduction in effort.

The contributions of this work are two-fold. Theoretically, we show that CM policies can be broken down into individual modules that each address a unique dimension of concern. Practically, we contribute the pluggable architecture and base set of modules that constitute the MCCM infrastructure. Combined, these lay the basis for an effective new approach to CM system building—one based on extensive reuse.

The remainder of this paper is structured as follows. In Section 2, we discuss relevant background material within the field of CM. We introduce our high-level approach in Section 3, and discuss the implementation of MCCM in Section 4. We present our experience in using MCCM in Section 5, discuss related work in Section 6, and conclude in Section 7 with an outlook at our future work.

2. Background

While a few CM systems have the exact same CM policy (e.g., RCS and its clones), most CM systems differ in their policies in one way or another. Some differ in relatively minor (but still important) ways, such as whether a version number is reserved immediately when an artifact is checked out [12] or assigned only after a new version of the artifact is checked in [25]. Others differ in more dramatic ways, such as whether or not they support branching [31] or whether or not they the support change sets [33].

Of interest to this paper is the evolving relationship between CM models, or the way artifacts are structured and stored in CM systems, and the CM policies used to govern changes to these artifacts. Early CM systems, such as RCS [25], SCCS [21], and Sablime [2], were created from the ground up. It was therefore possible to design and implement the CM models of these systems to precisely support their associated policies. Consequently, the data structures used by these CM systems to capture the evolution of artifacts are finely tuned and highly specialized to their respective CM policies.

Recognizing that it is inefficient to build each new CM system from scratch, architects of CM systems in the late 80's and early 90's searched for ways to reduce this effort. The reuse of existing CM systems and database systems in their entirety as storage facilities proved to be a good initial solution. For instance, CVS [3] and several other CM systems were built by wrapping RCS with additional functionality, and ClearCase [13] (then DSSE) was built upon a commercial database system (Raima).

Unfortunately, while advantageous from a reuse point of view, this approach also has its drawbacks. In particular, it creates a serious disconnect between the CM model as provided by the underlying technology and the needs of the CM policy as implemented by the new CM system. As an example, the CM model of RCS does not support the storage and management of collections of artifacts, something that is needed by the CM policy of CVS. As another example, generic database systems rarely provide versioning facilities, something that is needed by all CM systems. This disconnect must be addressed, and generally requires the implementation of mappings from an ideal CM model (as supported internally by the code implementing the new CM system) to an actual CM model (as supported by the reused database or CM system). Unfortunately, in the case of advanced CM policies these mappings become so complex that the approach of reuse is often deemed not pragmatic. Many new CM systems therefore are still implemented from the ground up.

Several experimental infrastructures have been created in response to this problem (e.g., EPOS [32], NUCM [29], ICE [34]). These infrastructures blend the aforementioned approaches of reusing an entire CM system and reusing a generic database. In particular, they each provide a reusable repository that is specifically designed to support the building of new CM systems. The repository intrinsically supports the storage of versioned artifacts, exposes a programmatic interface specifically designed for manipulat-

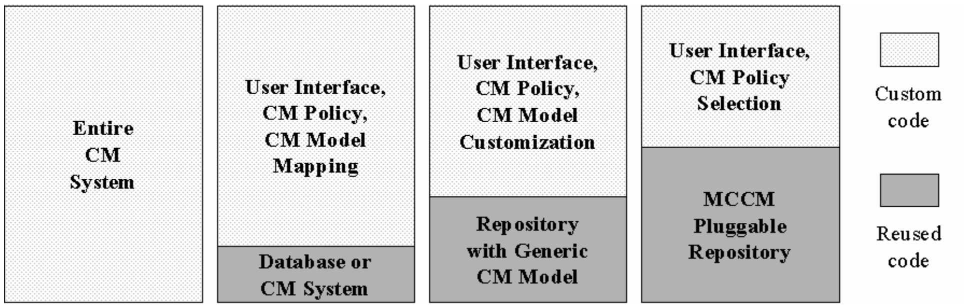
Entire CM System
User Interface, CM Policy, CM Model Mapping
Database or CM System
User Interface, CM Policy, CM Model Customization
Repository with Generic CM Model
User Interface, CM Policy Selection
MCCM Pluggable Repository
Custom code
Reused code

Figure 1. Evolution of CM System Architecture.

ing those artifacts, and internally leverages a generic CM model that can be used to support a broad variety of CM policies. A new CM system, then, is implemented by utilizing the functions in the programmatic interface to customize the generic repository to the needs of the desired CM policy, and building the rest of the CM system (including CM policy) on top of the customized repository.

These infrastructures have undergone some preliminary evaluations by their designers in terms of building prototype CM systems. Although these prototypes are largely experimental in nature, they demonstrate that it is possible to implement a broad variety of CM policies, and show a varying, but sizeable reduction in the effort it takes to implement new CM systems (sometimes reaching a ten-fold reduction).

Summarizing the discussion, Figure 1 depicts the architecture of CM systems as it evolved over time. Dark shading represents reuse of existing facilities; light grey highlights the part that is necessarily unique to the CM system being built. The balance between these two has changed from building an entire CM system from the ground up, to reusing an existing database or CM system, to leveraging a reusable repository with a generic CM model that can be customized and accessed through an associated programmatic interface. Our work represents the next step in this evolutionary path: by reusing CM policies in association with a generic, pluggable repository, the amount of unique code that needs to be designed and developed to create a new CM system is further reduced as compared to previous approaches.

3. Approach

The design and implementation of MCCM was sparked by an experiment in which we modeled ten different CM policies using NUCM [28]. By design, these policies covered a diverse set of approaches and philosophies, ranging in complexity from a basic check out/check in policy to an advanced change set policy. Moreover, they varied in the distribution of artifacts over multiple repositories as well as in the kinds of artifacts managed. While we expected a set of widely differing approaches to the implementations of the policies, we observed an interesting pattern quite to cerns and often addressed these concerns in similar ways. This was all the more surprising given our use of NUCM, which provides a generic CM model and already factors out much code common to manipulating this model. This means that the details of the policies *themselves* are much more alike than one would initially suspect.

MCCM capitalizes upon this observation by providing an architecture designed to exploit the similarities of CM policies. Shown in Figure 2 (and updated from our initial architecture introduced in [30]), the architecture separates the different concerns of CM policies by providing a generic client and a generic server that can be customized through the use of small, reusable modules. By plugging in desired modules (which form the CM policy) into the slots of the MCCM client and server (which provide standard facilities for workspace access and a customizable CM model, respectively), the core of a new CM system can be rapidly composed.

Modules are either constraint or action modules. Constraint modules plug into the server side of MCCM and maintain the consistency of its generic CM model per the needs of a desired CM policy. Action modules plug into the client side of MCCM to, also per the desired policy, enact the rules of that policy. A good way to characterize the functionality of constraint and action modules is to view a constraint module as setting allowable boundaries and an action module as deciding, within those boundaries, the exact operations that are to take place.

MCCM supports five dimensions along which individual CM policies may differ. Each dimension is addressed by a pair of modules, namely a constraint module and an action module. For example, how multiple versions of an artifact are stored as baselines and changes is determined by the combination of a storage constraint module and an evolution action module. Analogously, how different artifacts are distributed and replicated over multiple repositories is determined by a distribution constraint module and a placement action module. The remaining three dimensions are concerned with the hierarchical operation of a CM policy (composition constraint & hierarchy action modules), the rules by which the CM policy supports con-

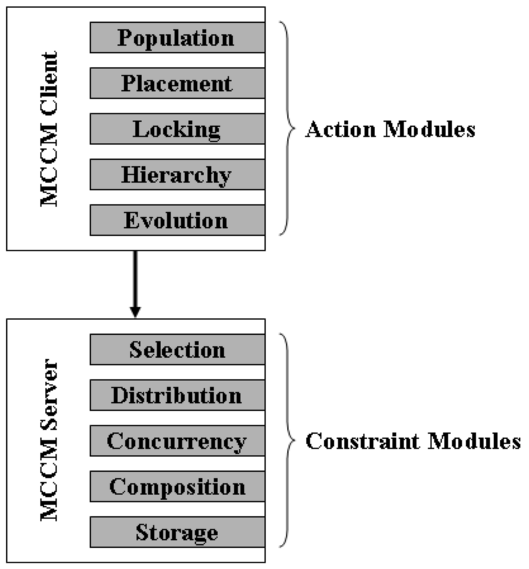

Figure 2. MCCM Architecture.

current modification of artifacts (concurrency constraint & locking action modules), and the way in which the CM policy selects and places artifacts in workspaces (selection constraint & population action modules).

MCCM separates constraint and action modules from each other because it increases the level of reuse: a single constraint module can be combined with multiple action modules to form different CM policies. As a trivial example, a constraint module disallowing the use of branching may be combined with, among others, a storage action module that stores each version of an artifact as a baseline or a storage action module that stores the first version as a baseline and any subsequent versions as changes from the previous version. A slightly different policy is formed, but the constraints on the generic CM model only need to be implemented once.

In building a new CM system, a designer can use modules in three ways: (1) associated with an individual artifact stored in MCCM, (2) associated with a group of artifacts as defined by their artifact type, or (3) through use of default modules. In searching for modules to apply when an artifact is manipulated, MCCM first checks whether the artifact has individual modules that should be used, then checks whether its artifact type has any applicable modules, and only then resorts to the default modules. MCCM therefore supports the creation of CM policies in which applicable rules depend on the nature of the artifacts being stored. It is common, for instance, that the rules of a CM policy vary depending on whether an artifact is a file or a directory. By setting the default modules to the rules for files, and as necessary associating different modules with the DIRECTORY artifact type (only as necessary, since typically not all modules need to be different), these kinds of policies can be easily implemented using MCCM.

It is important to realize that the result of plugging in a desired set of modules into MCCM does not constitute an entire CM system. As illustrated in the last architecture of Figure 1, the role of MCCM is to serve as a reusable, core infrastructure. Upon this core, the user interface of a CM system, the dynamic association of modules with artifacts (if desired), and any further details of the CM policy (such as attaching descriptive attributes or enforcing high-level processes) must still be programmed. However, those remaining parts are typically intrinsically unique to different CM systems, and they have to be programmed regardless of how much reuse of other parts takes place.

4. Implementation

MCCM consists of three parts: a server, a client, and a set of base modules that can be plugged into the client and server. The MCCM server implements a distributed, peer-to-peer repository with facilities for storing multiple versions of artifacts as sets of baselines and changes, hierarchically relating artifacts, locking artifacts, distributing and replicating artifacts over multiple servers, and selecting (versions of) artifacts from the repository. The MCCM client builds upon the server to provide workspace access to artifacts. Furthermore, it exposes a programmatic interface (implemented and accessible using Java RMI) upon which the remainder of a desired CM system can be built. This interface revolves around three functions: OPENARTIFACT, INITIALIZECHANGE, and COMMITCHANGE. These are policy neutral: they represent the stages that any artifact goes through when undergoing modifications, irrespective of which CM policy is used. Any CM system built using MCCM, thus, invokes these three functions in exactly the same order (albeit sometimes at different times). They can therefore be considered a main loop, to which the modules are attached to change its behavior.

The programmatic interface contains a large number of additional functions. Space prohibits us to discuss them all in detail. In brief, though, they make it possible to add and remove artifacts to (from) collections, to rename artifacts, to cancel changes and close workspaces, to move and replicate artifacts across multiple servers, to associate textual attributes with artifacts, to set default modules and attach or detach modules to (from) particular (types of) artifacts, and to query the state of artifacts, workspaces, and repositories. It is important to note that, compared to other infrastructures, these functions exclusively reflect user actions; policies are programmed using modules and not using the MCCM client programmatic interface.

In designing MCCM, a primary consideration was that individual modules should be small in their implementations. Key to achieving this goal are two factors: (1) internally, MCCM makes use of simple, dedicated graph structures that record the current state of the repository, and (2) each module addresses one single concern, leveraging the functions in the interfaces of the relevant graph structures to either check certain conditions (a constraint module) or decide upon a particular course of action (an action module). For instance, a storage constraint module only needs to implement a single function that checks whether or not the graph of baselines and changes for an artifact violates any of its storage constraints. As another example, an evolution action module needs to implement only four functions: one that returns a version identifier for the first version of an artifact; two that return a version identifier for a next version of an artifact (with or without user input), and one that decides whether new versions are only created for artifacts that have changed or also for artifacts that did not change (some policies will store a duplicate). Because of these small and focused responsibilities, most of our base modules are implemented in 100 lines of code or less.

4.1. Constraint Modules

The MCCM server provides a generic CM model, but places no restrictions on how this model is used. To tailor the generic CM model to the needs of a new CM policy, constraint modules must be used to restrict the server's behavior to only permit the kinds of uses that do not violate the CM policy. Five kinds of constraint modules exist, the role of each of which we describe below.

4.1.1. Storage constraint modules. At its core, the server can store baselines, changes, and dependency relations, all on a per-artifact basis. That is, each artifact is stored separately and each incremental change is stored as a *baseline* containing the entire version of the artifact or as a *change* containing a delta with the changes made in a workspace. Dependency relations among baselines and changes represent inclusion hierarchies: whenever a change is selected, its dependency relations are recursively traced until each of the preceding baselines are found. Intermittent changes as well as any baselines that reside on different branches are then merged to create a single version of the artifact. Dependency relations are not mandatory; that is, the storage model is *not* a version tree in which deltas are used for saving space. Instead, our model reflects the change set approach in its use of baselines and changes, and uses dependency relations as the basis for enforcing the constraints of storage constraint modules.

Together, baselines, changes, and dependency relations form a (potentially disconnected) graph. Different storage constraint modules can be used to enforce particular structures on this graph—structures that reflect the needs of the desired CM policy. As an example, Figure 3 illustrates the structures that result from use of several of our base storage constraint modules. In Figure 3a, each new version of an artifact can only be stored as a baseline. This policy is relaxed in Figure 3b, where baselines and changes can be used intermittently and where branching is allowed. Subsequent modules further allow merging (Figure 3c), independent lines of development (Figure 3d), and independent storage of changes with the possibility of mutual dependencies (Figure 3e).

Of note is that this mechanism does not recreate existing CM models, but instead emulates them. For instance, together with the standard mechanism of always merging in dependency relations, the use of the storage constraint module shown in Figure 3c creates a storage facility that is equivalent to the well-known version tree model. Similarly, Figure 3d and Figure 3e support the needs of the change package and change set CM policies, respectively.

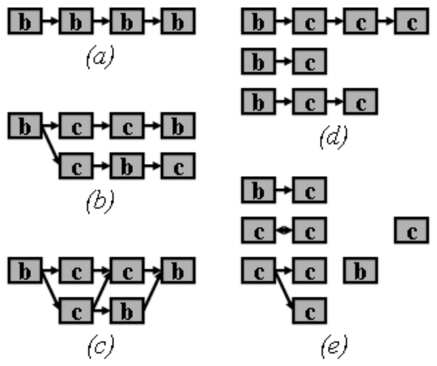

Figure 3. Results of Applying Different Storage Constraint Modules ('b' = baseline, 'c' = change).

4.1.2. Composition constraint modules. MCCM distinguishes atoms and collections. Atoms are not further broken down for CM purposes; collections may contain other artifacts (both collections and atoms). As with the storage model, MCCM maintains a graph for capturing the composition of artifacts. For efficiency purposes, the graph is stored as a set of small graphs (one per artifact). Conceptually, though, the graph is treated as one large graph.

Figure 4 illustrates an example of composition. Dashed lines indicate that a baseline or change of one artifact contains other baselines and changes of other artifacts. Note that composition is per baseline or change, and is not necessarily constant throughout an artifact. For instance, Figure 4 shows a collection for which each baseline is composed of different artifacts. Not all CM policies need this flexibility, but advanced CM policies typically do need it. In the change set policy, for example, composition identifies the constituents of a logical change, which are small in number since they typically only change a few artifacts.

By default, the composition graph is extremely permissive: it can be disconnected, it may contain cycles, etc. By using a composition constraint module, this behavior can be restricted to precisely support the needs of the desired CM policy. Our base modules consider the following dimensions: (1) whether or not hierarchical composition is allowed, (2) whether or not an artifact can be part of multiple, higher-level artifacts, and (3) whether or not cyclic relationships are allowed. Other dimensions, such as that a baseline always contain baselines or a change always contain changes can be added as necessary.

The combination of the generic server with just storage constraint and composition constraint modules already is very expressive. A simple version of the storage needs of CVS, for instance, can be defined by associating two constraint modules each to the standard DIRECTORY and FILE artifact types. Since CVS does not version directories, the storage constraint module for directories should limit their storage to be as a single baseline only. The composition constraint module for directories should allow hierarchical, non-cyclic composition. Files, on the other hand, must have the storage constraint module illustrated in Figure 3c and a composition constraint module that disallows containment of other artifacts.

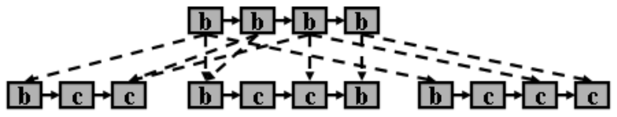

Figure 4. Example of Hierarchical Composition.

4.1.3. Concurrency constraint modules. The graph with which MCCM records locks on artifacts extends the storage graph discussed in Section 4.1.1. As such, locks in the CM model are per-artifact, and can be attached to individual baselines or changes. Each invocation to the MCCM server to establish a lock on an artifact, however, may request more than one baseline or change to be locked. To ensure that such a request follows the proper CM policy, MCCM provides a base set of concurrency constraint modules that range in functionality from not enforcing any locking at all (a number of CM policies, after all, rely on merging instead of locking), to requiring locking of at least a single baseline or change (a typical strategy), to requiring locking of at least an entire branch (assuming that a complementary storage constraint module is used, another typical strategy), to requiring locking of an entire artifact (as often used for maintenance purposes).

We note that MCCM currently just supports write-only locks. While sufficient for modeling all of our experimental CM policies, we recognize that a more advanced read-write locking scheme may be desired and in the near future will enhance MCCM to support such a scheme.

4.1.4. Distribution constraint modules. MCCM supports distribution and replication of artifacts at a fine-grained level: different baselines and changes for a single artifact can be stored across different servers, and a single baseline or change may be replicated across multiple servers for redundancy and performance reasons. Naturally, not every CM policy supports these options. Therefore, distribution constraint modules can be used to limit how artifacts are distributed and replicated over multiple servers. In particular, a concurrency constraint module is provided with a distribution graph that contains a map of which baselines and changes are stored where. Based on this graph, our base distribution constraint modules can restrict a CM policy to have no distribution (replication), to only support distribution (replication) of an artifact as a whole, or to not restrict distribution (replication) and simply adopt the fine-grained model provide by MCCM. Other strategies can be implemented as desired.

4.1.5. Selection constraint modules. Selection constraint modules restrict, per artifact, the combination of baselines and changes that may be placed in a workspace. They, for instance, ensure that any selection includes at least one baseline to which to apply the changes or that all selected baselines and changes are from the same branch. At first sight, this may seem unnecessary, since storage constraint modules prevent any illegal combinations of baselines and changes from being committed. However, selection constraint modules still serve two important roles. First, it is useful to know a-priori whether a selected combination is invalid, such that any unnecessary work is avoided. Second, and more importantly, the set of situations that can be avoided with selection constraint modules is greater than with just storage constraint modules. For instance, the first situation described above (a selection must include at least one baseline) cannot be enforced “after the fact”.

4.2. Action Modules

Within the bounds established by constraint modules, there still is considerable freedom as to which particular actions are taken. Action modules reduce this freedom to one particular choice. As with constraint modules, action modules base their decisions on the graphs that are made available to them, are small and focused in nature, and only *direct* MCCM (they do not perform the operations, but simply decide which operations must happen). Often, user input is a factor in these decisions, for instance when a user wants to (in a CVS-like policy) manually initiate a branch or explicitly create a new major revision. Obtaining this kind of user input is the responsibility of the CM system implementer, and it must be passed to the MCCM client using the programmatic interface. MCCM then passes the information to the relevant action modules.

4.2.1. Evolution action modules. When an artifact has undergone modifications and must be stored back into the repository, an evolution action module decides whether it is stored as a baseline or change, whether any dependency relations must be created, what the identifier is with which the new baseline or change can be identified, and whether a new version is stored even if the artifact has not changed from the previous version. To do so, MCCM provides the evolution action module with the current graph of the artifact at hand and with the original selection that was made to put the artifact in the workspace (see Section 4.2.5). In addition, any user preferences are passed on for the module to take into consideration.

Thus far, we have implemented a small number of base evolution action modules. In particular, available for reuse are a module that stores each version as an entire baseline, a module that stores a first version as a baseline and any subsequent version as a change (creating a branch when needed), and a module that simply maintains one baseline that is continually replaced when a new version is stored (as used for directories in the CVS policy, for instance).

4.2.2. Hierarchy action modules. Hierarchy action modules serve a very important role within the MCCM architecture: they decide if certain operations occur recursively over the parents and children of an artifact under consideration. For instance, hierarchy constraint modules decide whether or not a lock on a collection results in attempts to lock its constituent artifacts. Although recursive propagation typically is downwards, CM policies such as those of COOP/Orm [17] demand recursive propagation upwards. Hence the desirability of both must be specified by all hierarchy constraint modules.

Note that hierarchical operation does not mean that the same modules are applied throughout; instead, an operation is performed recursively, but is parameterized by the constraint and action modules of each artifact under consideration. This allows CM policies such as those of CVS (in which the rules for directories and files differ) to still benefit from hierarchical operation.

We note that the presence of hierarchical operation introduces some difficult issues with respect to the completion of certain actions. For instance, it may happen that deep in the recursion a lock fails or a commit cannot complete because it violates the rules of the CM policy. In such cases, the MCCM infrastructure rolls back the entire operation. This creates a transactional behavior and guarantees that an operation either completes successfully or does not complete at all.

4.2.3. Locking action modules. A locking action module complements a concurrency constraint module in choosing which baselines and changes must be locked when an artifact is to undergo change in a workspace. In support of optimistic CM policies, the simplest locking action module does not lock any baselines or changes at all. More pessimistic locking action modules, however, determine the precise set of locks by inspecting the current graph of baselines and changes, accounting for the selection placed in a workspace, and incorporating any user preferences. Several commonly-used base modules are available, each enforcing different semantics in terms of amount of locking (e.g., single baseline or change, branch, entire artifact). If another kind of locking scheme is needed, it can be easily derived from any of these existing modules.

4.2.4. Placement action modules. A placement action module decides the location of artifacts when multiple servers manage a federation of distributed repositories. In particular, it assigns a specific repository when a new artifact is created or when a new baseline or change is committed. Additionally, it determines which set of baselines and changes to move as well as which set of baselines and changes to replicate at any given time.

Analogously to other action modules, a placement action module operates on a per-artifact basis and uses the graph of baselines, changes, and their current locations for making its decisions. Currently, we have implemented two base placement action modules. One prohibits distribution and replication, the other supports full replication of all artifacts. Intermediate placement action modules are under development.

4.2.5. Population action modules. The final type of action modules are population action modules. Based on the current storage graph (as discussed in Section 4.2.1), these modules are responsible for selecting a particular version of an artifact to be placed in a workspace. This version is dynamically constructed by merging the set of baselines and changes that are selected by the module. Selection is especially important in the context of hierarchical composition. While most CM policies by default use a contained version of an artifact, other policies (such as the one of DVS [4]) always choose the latest version. Currently, we have implemented these two as the set of base modules.

Clearly, this module is the most influenced by any user preferences that are expressed. Often, a user wants a specific version of an artifact to be placed in their workspace. This choice must be respected, although it may potentially be enhanced by a population action module to incorporate any dependencies that should have been included for consistency purposes in the initial user selection as well.

4.3. Discussion

Pluggable architectures have successfully been used in a wide variety of domains, including, among others, development environments [19], operating systems [23], and web servers [24]. Their use has been ignored, however, in the area of generic infrastructures for CM system building. It is our belief that this is largely due to the misperception that CM policies are monolithic entities that—in their entirety—are built on top of whichever generic facilities are available in the infrastructure that is (re)used. As the preceding discussion illustrates, MCCM refutes this misperception, and demonstrates that it is possible to provide a pluggable infrastructure that supports the composition of CM policies out of reusable modules.

The strength of our approach lies in the following two, closely related factors:

1. The generic client and server implement the mechanisms by which the repository is maintained and the workspace is manipulated. This allows the implementation of modules to concentrate on making decisions and not on actually carrying out those decisions.
2. Each of the modules addresses a different concern. This is most evident within the set of constraint modules and within the set of action modules. But even in the case of paired constraint and action modules, each module addresses a different concern (maintaining the consistency of the generic CM model versus choosing a particular course of action).

Clearly, successful operation of MCCM cannot be guaranteed and depends on the way it is used. It is critical, for example, that compatible constraint and action modules be selected, since MCCM itself cannot detect when the selected modules have conflicting semantics. It is the responsibility of a designer, therefore, to ensure a coherent overall CM policy (something they must do regardless of which infrastructure is used).

An interesting question about the use of MCCM is how to address crosscutting concerns. For example, currently it is not possible to create a locking action module that only places locks on artifacts residing in a specific repository. Our experience in building CM policies indicates, though, that such crosscutting concerns are rarely needed. Should they nevertheless be needed, they can be programmed on top of the programmatic interface of the MCCM client. In the example above, for instance, a designer should use the programmatic interface to first verify the location of an artifact and then explicitly request a lock (instead of relying on the action module to place the locks). Should we encounter particular dimensions along which crosscutting concerns are frequent, we will change the architecture of MCCM to make available relevant graphs to the constraint and action modules that should address those crosscutting concerns (in the example above, we would make available the distribution graph to locking action modules).

5. Experience

To show how MCCM promotes reuse in the composition of CM policies, we built approximations of RCS [25], CVS [3], and Subversion [26], as well as a prototype CM system based on change sets. While none of our example implementations is as fully functional as its original counterpart (all core functionality is implemented, but we omitted parts of the user interface and administrative options), all of the systems are functional and allow a user to version artifacts in accordance with the CM policies.

Table 1 summarizes our experience. For each example system, the table lists the modules that were used in composing it. Several observations are in place. First, a high number of base modules were reused, confirming that the availability of base modules significantly eases development. Second, nine custom modules were developed, but these modules were reused ten times—amortizing their development cost over multiple CM systems. We furthermore will add the custom modules to our library of base modules, which in future will reduce the need for developing new modules for other new CM systems.

We also observe that it is trivial to change the policies of these systems. For instance, should we desire a distributed version of CVS that supports multiple servers and replication of artifacts, we can simply take the placement action module used in the Subversion system and plug it into the CVS system. No further changes are needed.

Table 1. Example Compositions of CM Policies (Modules: 'S' = Standard, 'R' = Reused, 'C' = Custom).

	Modules	RCS	CVS	SubVersion	Change Sets
Constraint modules	*Storage*	(S) Version tree	(S) Version tree (files) (S) Single baseline (directories)	(S) Version tree	(C) Individual changes with dependent baselines
	Composition	(S) No composition	(S) Single parent	(S) Single parent	(S) Single parent
	Concurrency	(S) Full artifact locking	(S) No locking	(S) No locking	(S) No locking
	Distribution	(S) No distribution	(S) No distribution	(S) Full artifact distribution & replication	(S) No distribution
	Selection	(C) At most two baselines or changes	(R) RCS module	(R) RCS module	(C) Baseline plus set of dependent changes
Action modules	*Evolution*	(C) Store on existing or new branch, or use user input	(R) RCS module for files (S) Single baseline (directories)	(R) RCS module	(C) Independent change depending on original baseline
	Hierarchy	(S) No hierarchical operations	(C) Evolve parents (files) (C) Evolve children and parents & populate children (directories)	(R) CVS module for files (R) CVS module for directories	(R) CVS module for files (R) CVS module for directories
	Locking	(S) Full artifact locking	(S) No locking	(S) No locking	(S) No locking
	Placement	(S) No distribution	(S) No distribution	(S) Full artifact distribution & replication	(S) No distribution
	Population	(C) Select newest version or use user input	(R) RCS module (files) (S) Select baseline (directories)	(R) RCS module	(C) Use user input and select missing baselines & changes

The time and effort involved in developing each system was small: each was completed within a few days and comprised a modest amount of new source code (RCS: 564 LOC, CVS: 446 LOC, Subversion: 341 LOC, change sets: 615 LOC). This new code includes all of the code for the textual user interfaces we implemented for each of these systems (which we built on top of the MCCM client programmatic interface). The amount of code dedicated to the development of new modules, thus, is even smaller.

To confirm that use of MCCM also saves effort when it comes to real-world systems, we performed a second experiment. In this experiment we compared an existing CM system in active use, DVS [4], as it was originally implemented using NUCM [29] and now re-implemented using MCCM. Implementing the new version of DVS only took a few days and required less than 1500 new lines of code. Compared to the original version of DVS, which required 3000 lines of code, this represents a significant amount of savings. While some of the savings can be attributed to a change in programming language (from C in the original DVS to Java in the new DVS), a close analysis of the code reveals that the reuse of modules as promoted by MCCM accounts for as much as 75% of the savings.

In sum, our experiments show that MCCM is a viable and effective approach to CM policy composition and can support a developer in rapidly assembling the core of different CM systems with highly variable CM policies.

6. Related Work

EPOS [32] and ICE [34] are logic-based approaches to creating an infrastructure similar to MCCM. As discussed in Section 2, however, their infrastructures are focused on supporting the reuse of CM models only. Nonetheless, a few of the policies implemented with ICE are able to reuse existing logic rules from other policies, which, while seemingly incidental, indicates that the use of ICE as a vehicle for policy reuse should be further explored.

NUCM [29] is a predecessor to our work. Unlike ICE and EPOS, it is not based on logic; rather, it provides a programmatic interface to its generic repository. MCCM inherits this API-based approach, but compared to NUCM provides an enhanced architecture that supports pluggable modules. Furthermore, whereas NUCM only supports the reuse of a generic CM model, MCCM inherently supports the reuse of CM policies.

WebDAV [10] and DeltaV [14] are two web standards that together add extensive distributed authoring and versioning techniques to the HTTP protocol. While squarely aimed at the web, the two standards combined provide a set of features that is similar to those provided by NUCM: they provide a generic CM model and, through new HTTP methods, they provide a programmatic interface for building web clients. While exact details and relative strengths and weaknesses differ between the WebDAV/DeltaV protocols and NUCM, the overall result is largely similar, and the combination of the two standards suffers the same kinds of drawbacks when compared to MCCM.

Parisi-Presicci and Wolf introduce a formal approach to policy programming [20]. Using graph transformations, different CM policies can be specified. Their approach is promising in demonstrating reuse of parts of CM policies, but unfortunately falls short of demonstrating reuse across a broad variety of CM policies. Thus far, they have only included existential, state-based policies in their experiments. MCCM, on the other hand, supports reuse across existential and intentional CM policies as well as across change-based and state-based CM policies.

Finally, BAMBOO is a new infrastructure leveraging a generic CM domain model to generate new CM repositories [16]. While in the early stages of design and thus far focusing on the generation of CM models, BAMBOO is promising in being first in taking a generative approach to reuse in CM systems. We believe it could be extended to include the generation of different CM policies, but as of now it has not demonstrated so. We do note that the CM models generated by BAMBOO have more dimensions of freedom than the set of modules allowed by MCCM. As such, a combination of both approaches represents an attractive avenue of future research.

7. Conclusions

This paper has successfully demonstrated the feasibility and practical reality of the idea of CM policy composition. Through a novel infrastructure that consists of a generic, pluggable architecture and an associated set of constraint and action modules, MCCM can achieve high levels of reuse in the implementation of the core of new CM systems. As demonstrated by the application of MCCM to the implementation of several representative CM systems, this reuse translates into a significant reduction in effort as compared to existing implementation strategies (i.e., from scratch or leveraging other existing infrastructures).

Our infrastructure is still experimental in nature: additional dimensions of variability exist among CM policies that our infrastructure does not yet address; the infrastructure can be made more robust by hosting it on a reliable, transactional database system; and we wish to extend the approach to other domains such as content and document management. Even so, MCCM makes two important contributions. The first is the theoretical result that it is possible to modularly compose CM policies; the second is the practical demonstration of our approach through our pluggable architecture and base set of modules.

While MCCM addresses an important problem, we believe its true value is still ahead. Two long-standing problems in the CM domain are those of incremental adoption and CM policy interaction. The first problem stems from a desire to gradually introduce users to all of the features of a CM system. The second problem pertains to the need of different groups to work together—even when each group uses a different CM policy. MCCM provides an excellent basis upon which to explore these two issues. We believe, for instance, that it is possible to address the first problem by modifying MCCM to support modules on an individual user basis. Only after a user has demonstrated mastery of the basic principles will new, more permissive modules be made available to them. With respect to the second problem, MCCM's strong separation of its generic CM model from individual CM policies, combined with its separation of constraint and action modules, provides a starting point for exploring how different CM policies may leverage the common model and constraint modules for policy interaction. Clearly, both research problems are strongly related, and—in addition to further refining MCCM—our future work will address the two in concert.

Acknowledgments

We thank our research group, and in particular Anita Sarma and Eric Dashofy. Effort funded by the National Science Foundation under grant numbers CCR-0093489 and IIS-0205724.

References

[1] D. Belanger, D. Korn, and H. Rao, *Infrastructure for Wide-Area Software Development*, Proc. of the Sixth International Workshop on Software Configuration Management. 1996, Springer-Verlag: p. 154-165.

[2] Bell Labs Lucent Technologies. *Sablime v5.0 User's Reference Manual*. 1997.

[3] B. Berliner. *CVS II: Parallelizing Software Development*. Proceedings of the USENIX Winter 1990 Technical Conference, 1990: p. 341-352.

[4] A. Carzaniga. *DVS 1.2 Manual*. Department of Computer Science, University of Colorado at Boulder, 1998.

[5] M.C. Chu-Carroll and S. Sprenkle. *Coven: Brewing Better Collaboration through Software Configuration Management*. Proc. of the Eighth International Symposium on Foundations of Software Engineering, 2000: p. 88-97.

[6] CollabNet, *SourceCast*, http://www.collabnet.org/products/sourcecast/, 2002.

[7] S. Dart, *Not All Tools Are Created Equal*, in *Application Development Trends*. 1996. p. 39-54.

[8] M. de Jonge. *Source Tree Composition*. Proc. of the Seventh Interantional Conference on Software Reuse, 2002.

[9] J. Estublier. *Defining and Supporting Concurrent Engineering Policies in SCM*. Proc. of the Tenth International Workshop on Software Configuration Management, 2001.

[10] Y. Goland, et al. *HTTP extensions for distributed authoring -- WEBDAV (RFC 2518, Standards Track)*. 1999.

[11] Hansky, *Firefly*, http://www.hansky.com/products/ff/firefly.html, 2003.

[12] J.J. Hunt, et al. *Distributed Configuration Management via Java and the World Wide Web*. Proc. of the Seventh International Workshop on Software Configuration Management, 1997: p. 161-174.

[13] IBM Rational, *ClearCase*, http://www.rational.com/products/clearcase/, 2003.

[14] IETF Delta-V Working Group, *Versioning Extensions to WebDAV, Internet Draft*, http://www.webdav.org/deltav/protocol/, 2000.

[15] Inobyte, Ltd., *Global Source*, http://www.inobyte.com/, 2003.

[16] S. Kim, et al. *SCM Domain Modeling and Repository Kit Design*. University of California, Santa Cruz, 2003.

[17] B. Magnusson and U. Asklund. *Fine Grained Version Control of Configurations in COOP/Orm*. Proc. of the Sixth International Workshop on Software Configuration Management, 1996: p. 31-48.

[18] J. Micallef and G.M. Clemm, *The Asgard System: Activity-Based Configuration Management*, Proc. of the Sixth International Conference on Software Configuration Management. 1996, Springer-Verlag: p. 175-186.

[19] Object Technology International, Inc., *Eclipse Platform*, http://www.eclipse.org/, 2003.

[20] F. Parisi-Presicce and A.L. Wolf. *Foundations for Software Configuration Management Policies using Graph Transformations*. Proc. of the Third International Conference on Fundamental Approaches to Software Engineering, 2000: p. 304-318.

[21] M.J. Rochkind, *The Source Code Control System*. IEEE TSE , 1975. SE-1(4): p. 364-370.

[22] SCMLabs, Inc., *Quartet*, http://www.scmlabs.com/, 2003.

[23] A.S. Tanenbaum, *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, N.J., 1992.

[24] The Apache Software Foundation, *Apache HTTP Server Project*, http://www.apache.org/, 2003.

[25] W.F. Tichy, *RCS, A System for Version Control*. Software - Practice and Experience, 1985. 15(7): p. 637-654.

[26] Tigris.org, *Subversion*, http://subversion.tigris.org/, 2002.

[27] Tigris.org, *Subversion Frequently Asked Questions*, http://subversion.tigris.org/project_faq.html, 2002.

[28] A. van der Hoek. *A Generic, Reusable Repository for Configuration Management Policy Programming*. Ph.D. Thesis, University of Colorado at Boulder, Department of Computer Science, 2000.

[29] A. van der Hoek, et al., *A Testbed for Configuration Management Policy Programming*. IEEE TSE, 2002. 28(1): p. 79-99.

[30] R. van der Lingen and A. van der Hoek. *Dissecting Configuration Management Policies, Software Configuration Management*. Proc. of the ICSE Workshops SCM 2001 and SCM 2003 (Selected Papers), 2003: p. 177-190.

[31] C. Walrad and D. Strom, *The Importance of Branching Models in SCM*. IEEE Computer, 2002. 35(9): p. 31-38.

[32] B. Westfechtel, B.P. Munch, and R. Conradi, *A Layered Architecture for Uniform Version Management*. IEEE TSE, 2001. 27(12): p. 1111-1133.

[33] D. Wiborg Weber. *Change Sets versus Change Packages: Comparing Implementations of Change-Based SCM*. Proc. of the Seventh International Workshop on Software Configuration Management, 1997: p. 25-35.

[34] A. Zeller and G. Snelting, *Unified Versioning through Feature Logic*. ACM TOSEM, 1997. 6(4): p. 398-441.