Dissecting Configuration Management Policies

Ronald van der Lingen and André van der Hoek

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{vdlingen, andre}@ics.uci.edu

Abstract. A configuration management policy specifies the procedures through which a user evolves artifacts stored in a configuration management system. Different configuration management systems typically use different policies, and new policies continue to be developed. A problem in the development of these new policies is that existing policies (and their implementations) typically cannot be reused. As a basis for a future solution, this paper presents a new configuration management system architecture that focuses on modularly specified policies. In particular, policies consist of a set of constraint modules, which enforce the desired repository structure, and a set of action modules, which govern the desired user interaction. New policies can be developed by combining relevant existing modules from existing policies with new modules that specify the unique aspects of the new policy. We demonstrate how several quite different configuration management policies can be effectively constructed this way.

1 Introduction

Building a new configuration management system is a daunting undertaking. The Subversion project [13], for example, initiated in May 2000, currently comprises over 100,000 lines of code, and has iterated over 14 alpha releases to date. Its first beta release is not scheduled until early next year, and still a number of known issues are yet to be addressed. This is all the more surprising, because the goal of Subversion is merely to provide a better implementation of CVS [2] that addresses a relatively small number of functional "annoyances" in the process [14].

A significant portion of the effort lies in the fact that Subversion is a carefully architected, open, and pluggable implementation with extensive error handling and distributed operation. The inherent non-reusability of the configuration management policy of CVS, however, also played a role. Because the policy of CVS is interwoven throughout its implementation, functionality related to its policy (such as the rules for updating a version tree or selecting artifacts) could not be easily extracted for reuse and had to be reimplemented in Subversion.

A number of projects have attempted to address this issue. EPOS [8] and ICE [21], for instance, provide logic-based infrastructures upon which to implement new configuration management systems by specifying their policies as logical constraints.

Another approach, NUCM [16], provides a programmatic approach that reuses a generic repository and supports implementation of specific policies via a standard programmatic interface. While successful in making the development of new configuration management systems easier, we observe that all of these approaches only focus on reuse of a generic configuration management model. In particular, the infrastructures provide generic data modeling facilities that are reused for expressing policies, but the policies themselves are not reused.

This paper introduces a new configuration management system architecture that focuses on policy reuse. It is loosely based on the generic model of NUCM, but additionally recognizes that policies can be modularly specified and implemented. In particular, a policy is defined by a set of *constraint modules*, which enforce the desired repository structure on the server side, and a set of *action modules*, which govern the desired user interaction on the client side. Four types of constraint modules exist: storage, hierarchy, locking, and distribution. They are complemented by four types of action modules: selection, evolution, concurrency, and placement. A policy is formed by choosing relevant implementations of each type of constraint module and action module. Reuse occurs when new policies are formed by combining one or more existing modules with newly developed modules that capture the unique aspects of the new policy. A single constraint or action module, thus, can be used in the specification of multiple policies.

Below, we first introduce some background information to form the basis for the remainder of the paper. We then introduce our new architecture and discuss the available constraint and action modules in detail. We continue by showing how several quite different configuration management policies are constructed using our approach. We then briefly compare our architecture to existing architectures and conclude with a discussion of the potential impact of our work and an outlook at our future plans.

2 Background

To understand our approach, it is necessary to clearly delineate a configuration management *model* from a configuration management *policy*. The two are often intertwined, which is not surprising given that a typical CM system requires only a single model tuned to a single policy. As a result, terminology has not been quite clear and early papers on configuration management policies, for instance, referred to them as models [6].

Here, we define a configuration management model as the data structures used to track the evolution of one of more artifacts and a configuration management policy as the procedures through which a user evolves the artifacts stored in those data structures. For example, the version tree model is generally used in conjunction with the check-in/check-out policy [12]. As another example, the change sets policy [11] generally uses specialized data structures to capture and relate individual change sets.

While efficiency reasons may prescribe the use of a customized model in support of a particular policy, research has shown how certain policies can be emulated with

other models. For instance, the change sets policy can be supported using the model for the change package policy [20]. Generalizations of this approach provide generic models that support a wide variety of policies [1,16,19,21]. In sacrificing efficiency for generality and expressiveness, these generic models can be used for the exploration and evaluation of new policies [15]. Once a desired policy has been determined, it may need to be reimplemented in an optimized fashion with a specialized model should efficiency be of great concern. Evidence shows, however, that effective systems can be built directly on top of the generic infrastructures [16,21]. Still, reuse of policies themselves remains difficult, a problem we tackle in this paper to make the construction of new configuration management systems even easier.

3 New System Architecture

We base our approach on the observation that configuration management policies do not have to be treated as single, monolithic entities. Rather, they can be dissected into smaller modules, each governing one aspect of one or more policies. As illustrated in Figure 1, we distinguish constraint modules from action modules. Constraint modules are placed on the server side of a configuration management system, and enforce the desired repository structure. The purpose of a storage constraint module, for instance, is to limit the potential relationships among a set of baselines and changes. Among others, those relationships may be restricted to form a linear structure, a version tree, a version graph, or even an arbitrary graph.

Action modules govern the desired user interaction on the client side of a configuration management system. Part of the purpose of a concurrency action module, for example, is to determine which baselines and changes must be locked before a user starts making changes. Example strategies could be to lock all baselines and all changes, only baselines and changes on the current branch, or no baselines or changes at all.

Our architectural decision to place constraint modules on the server side and action modules on the client side is a deliberate choice. While both types of modules could have been placed together on either the server or client side, separating them as in Figure 1 has two significant advantages. First, it places constraint checking at the level of the server, which reduces the need to send state information from the server to a client every time other clients commit changes. Second, it has the advantage of being prepared for a future in which different clients operate with different policies.

Different constraint and action modules can be associated with different artifacts. This allows the architecture to support policies that treat different artifacts differently (e.g., the policy underneath CVS [2] supports file versioning but does not support directory versioning). To avoid having to individually associate modules with every single artifact, default constraint and action modules can be specified.

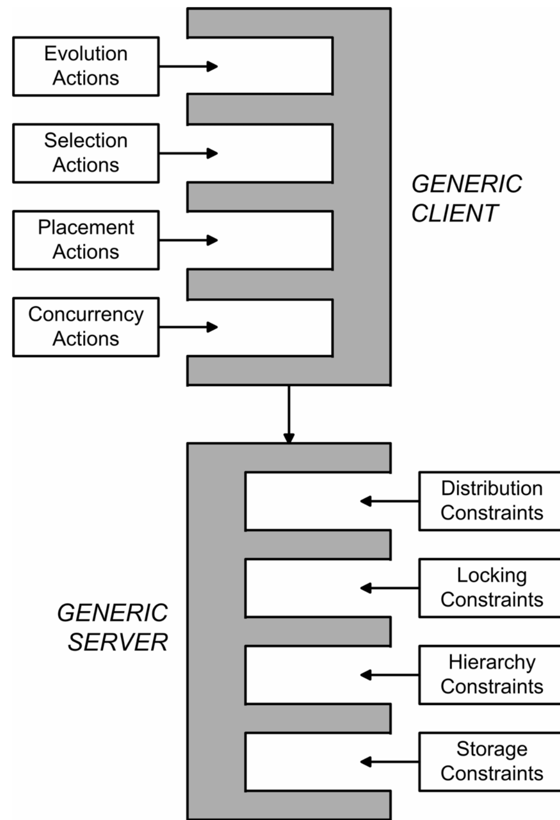

Fig. 1. New configuration management system architecture.

3.1 Constraint modules

The architecture supports four types of constraint modules: storage, hierarchy, locking, and distribution. These types were selected because they represent the common dimensions along which current configuration management policies vary. As our research continues, we will be exploring adding additional dimensions as necessary.

An important aspect of the architecture is that each constraint module is specified independently from the others, making it possible to replace one of the constraint modules without influencing the operation of the others. Of note is also that the sole purpose of a constraint module is to simply enforce the desired repository structure. As such, each constraint module only verifies if a particular operation adheres to the specified constraints. The standard, generic part of the architecture interprets the re-

sults of each constraint module and either commits the operation (if all constraints are satisfied) or aborts the operation (if one or more constraints are violated).

Below, we briefly discuss the role of each type of constraint module.

3.1.1 Storage constraint module

The architecture is built on baselines and changes. At the storage layer, a baseline represents a version of a single artifact (containment is supported by the hierarchy layer, see Section 3.1.2), as stored in its complete form. A change represents an incremental version, and is stored using a delta mechanism.

The purpose of a storage constraint module is to limit the number of baselines and changes that may be created by a user, as well as to limit the potential relationships that may exist among those baselines and changes. For example, directories in CVS cannot be versioned. The storage constraint module for those directories, therefore, would limit the generic server to only be able to store one baseline of a directory at a time. Files, on the other hand, can be versioned in CVS and are stored as baselines and changes that are related in a directed graph. The storage constraint module for files, therefore, would allow a directed graph but still disallow parts of that graph to be disconnected.

To understand the role of storage constraints, Figure 2 presents the structures that result from applying a number of different storage constraint modules. Figure 2a, for instance, shows a baseline-only structure that results from a very conservative storage constraint module. Figure 2b shows the traditional version tree that results from allowing intermittent baselines and changes that each have a single parent. The version graph in Section 2c is the result of slightly different constraints that allow a single

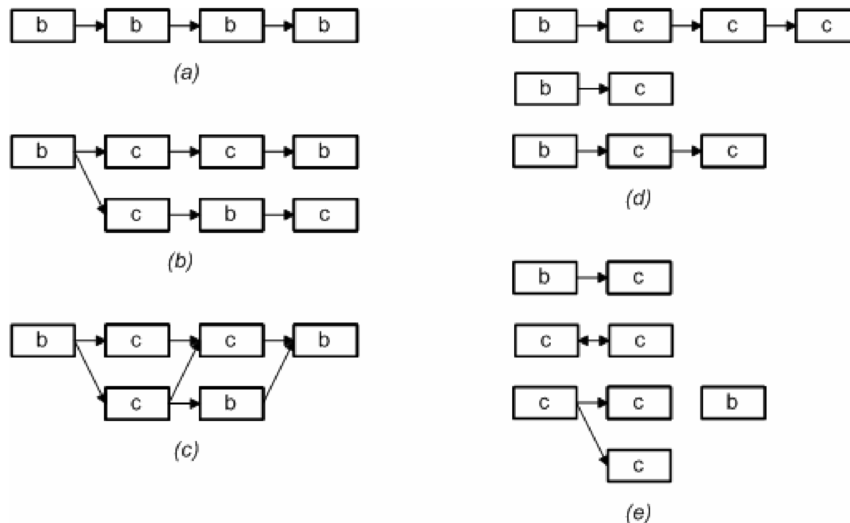

Fig. 2. Structures resulting from different storage constraint modules.

baseline or change to have multiple parents, as long as no cycles result. The examples of 2d and 2e show disconnected graphs, as suited for change package [20] and change set [11] policies, respectively.

3.1.2 Hierarchy constraint module

The purpose of a hierarchy constraint module is to limit a user in forming potentially arbitrary hierarchal structures. The dimensions along which different policies vary are threefold: (1) whether or not the hierarchical composition of artifacts is allowed, (2) whether or not a single artifact can be part of more than one higher level artifact, and (3) whether or not cyclic relationships are allowed. We have defined and implemented a hierarchy constraint module for each of the meaningful combinations of these three dimensions.

Individually, hierarchy constraint modules are not very powerful. Combining them with storage constraint modules, however, makes for a storage facility that is more generic than its predecessor (i.e., the storage model of NUCM [16]). In particular, our new architecture supports containers that: (a) have actual content in addition to containment information and (b) explicitly distinguish baselines from changes.

3.1.3 Locking constraint module

The mechanisms that different configuration management policies use for concurrency control vary. Some are optimistic and rely on automated merging [7] for conflict resolution. Others are pessimistic and rely on locking to avoid conflicts altogether. In terms of persistent state that must be preserved over some timeframe, locks are the only entities that must be stored. Therefore, concurrency-related decisions are made by a concurrency action module (see Section 3.2.1), but locking related checks are performed by a locking constraint module. In particular, a locking constraint module determines whether a request carries the correct set of locks.

Several different locking constraint modules exist that range from not enforcing any locking at all (e.g., in case of a change set policy) to enforcing the simultaneous locking of all baselines and all changes (e.g., in case of a policy like the one in SCCS [10]). Perhaps the most commonly used strategies are to either lock a single node, or to lock a node and its branch.

Locking constraints are specified independently from any hierarchy constraints and therefore cannot cross different levels in the storage hierarchy. Nonetheless, attaching the appropriate locking constraint to each level, combined with the use of an appropriate concurrency action module (see Section 3.2.1), will provide such functionality.

3.1.4 Distribution constraint module

The last type of constraint module pertains to the distribution of artifacts over multiple different servers. Different configuration management systems differ in their policies in a number of different ways. First, some support no distribution at all. Second, those that support distribution may support distribution of an artifact as a whole or of individual baselines or changes. Finally, a policy may support replication of artifacts as a whole or of individual baselines or changes.

The purpose of a distribution constraint module, then, is to ensure that the appropriate distribution policy is followed by a user (e.g., a request to replicate a single baseline cannot be honored if only an artifact as a whole can be replicated). Again, a distribution constraint module operates on a single artifact. If the distribution of hierarchically organized artifacts needs to be coordinated, a placement action module can do so.

3.2 Action modules

In addition to the constraint modules, four types of action modules exist: selection, concurrency, evolution, and placement. Action modules represent the decision points in a configuration management policy. They, for example, make sure to lock the right set of artifacts or to establish appropriate parent relationships when a user merges two or more branches.

Action modules augment the basic user interaction protocol that is supported by the architecture. This interaction protocol consists of two requests. First, a user populates a workspace with artifacts and indicates whether they want read or write access. Second, after the desired changes have been made, they place the changes back in the repository. The actual behavior of the system in response to these requests depends on the particular modules that are "plugged in". For example, if a concurrency action module is plugged in that supports simple locking, populating a workspace results in the artifacts being locked and placing changes back in the repository results in them being unlocked. As described below, each type of action module provides a range of options from which different configuration management policies are composed.

3.2.1 Selection action module

A selection action module interprets a request for populating a workspace and automatically incorporates other artifacts as necessary. Particularly important dimensions in this process are whether or not a request is hierarchical (i.e., whether a request for a collection additionally leads to populating of the workspace with constituent artifacts), whether or not latest versions should be selected (i.e., the policy of DVS [3] is to select by default the latest version of any constituent artifact), and whether or not additional selection criteria should be interpreted (i.e., attribute selection or configuration specifications [4]).

The power of the selection action module lies in it being a counterpart to the hierarchy constraint module. If hierarchical artifacts are permitted, the selection action module makes it easy for a user to meaningfully obtain a set of hierarchically related artifacts with a single request.

Note that the selection action module does not actually populate a workspace, but rather only determines the artifacts with which it should be populated. The generic client takes care of taking the list of artifacts and actually placing them in the workspace. This has two advantages. First, new selection action modules become simpler to implement and require fewer details concerning the internals of the architecture.

Second, it provides another form of reuse as the generic client implements the mechanisms for obtaining artifacts from the server.

3.2.2 Concurrency action module

Similar to the way the selection action module complements the hierarchy constraint module, the concurrency action module serves as a complement to the locking constraint module. When a user requests write access to an artifact that is being placed in their workspace, the concurrency action module determines the appropriate set of baselines and changes that must be locked. As described in Section 3.1.3, the set may be empty in case a change set policy is desired, but it also may involve locking multiple baselines or changes.

A concurrency action module also takes care of determining the set of baselines and changes to be unlocked when a user commits their changes to the server. Normally, of course, this will be the same set that was locked in the first place.

In order to support concurrency policies at the hierarchical level, a concurrency action module specifies two Booleans: the first determines whether locking a container artifact should lead to locking of its constituent artifacts and the second whether locking an artifact should lead to its parent artifacts being locked. Both cases of transitive locking, although rare, are present in certain configuration management policies. Note that the Booleans only indicate a desired behavior. The actual child or parent artifacts are locked according to their own concurrency action modules.

A few configuration management systems incorporate concurrency policies that involve direct workspace-to-workspace communication [5,18]. At the moment, we consider this out of scope since we focus on policies in the setting of client-server based configuration management systems. Our eventual goal, however, is to support such peer-to-peer interaction.

3.2.3 Evolution action module

While a storage constraint module limits the relationships that may exist among baselines and changes, certain choices still must be made. For example, when a user wants to create a new version of an artifact, that version must receive a version number, it must be determined whether it should be stored as a baseline or change, and it must be determined what relationships are desired with the other baselines and changes. An evolution action module, guided by appropriate user input, takes care of these kinds of decisions.

Evolution policies range from simply replacing the current version of the artifact in case the artifact is not versioned to storing the new version of the artifact as an independent change in case of a change set policy. Perhaps the most common policy is to add a new version to the end of the branch from which it originated, unless a newer version already exists, in which case a new branch is created. In effect, this policy creates a version tree.

If a version graph is desired, selection information kept in the workspace will help in determining when merge arcs are needed. In particular, if a user requested a workspace to be populated with a version that represents the merging of two branches

(simply by asking for those two versions, the architecture automatically merges the two), this information is kept in the workspace and fed into the versioning policy by the standard client.

Because the layers in our architecture are orthogonal, the evolution of hierarchical artifacts (containers) is performed in the same manner as the versioning of atomic artifacts. The only exception to this rule is that, again, an evolution action module can specify two Booleans to determine whether committing a new version of a collection should lead to the committing of parent and/or child artifacts.

3.2.4 Placement action module

The last action module in our architecture is the placement action module, which makes decisions regarding the distribution of artifacts, or versions of artifacts. In particular, when a user commits an artifact, the placement action module determines the location where the new baseline or change has to be stored. Normally, that will be the server from which the artifact originated, but it is also possible for a policy to store and or even replicate artifacts on servers that may be closer to the user that is currently making modifications. Again, we note that this module only determines where an artifact will be stored, the actual mechanism of doing so is provided by the generic server implementation.

Just like the concurrency and evolution action modules, the placement action module has two Booleans determining whether parent and/or child artifacts should be colocated with the artifact being manipulated. This way, a policy can restrict a subtree of artifacts to be located at a single server.

4 Examples

In order to demonstrate the use of our new architecture, we have studied which constraint and action models are needed to model some of the more prolific existing configuration management policies. Below, we first demonstrate how to construct a policy similar to the policy of RCS [12]. We then demonstrate how to construct a policy similar to the policy of CVS [2], and show how variants of that policy can be easily constructed. As an example of a rather different kind of policy, we then show how to model the change set policy [11].

In all of these examples, it is important to note that user interaction with the configuration management system remains the same. Users request artifacts for reading and/or writing, and place modified artifacts back into the repository. By plugging in different constraint and action modules, the configuration management system adjusts its behavior to the desired policy. This may result in some user actions to be prohibited (for example, the policy of RCS does not support hierarchical composition), but generally a user will notice little difference.

4.1 RCS

Table 1 illustrates how one would build a configuration management system with a policy similar to RCS. This policy is a relatively simple one and focuses on versioning of individual artifacts in a non-distributed setting. As a result not all features of our architecture are used. In particular, the hierarchy constraint module disallows any hierarchical composition and the distribution constraint module prevents distribution of artifacts over multiple repositories. Consequently, the placement action module is empty, since no decisions regarding placement of artifacts have to be made. Similarly, the value of the Boolean indicators that determine whether the remaining three action modules operate in a hierarchical fashion should be set to false.

Versions are stored in a connected graph of baselines and changes, which in effect forms a version tree in which branching is allowed. A new version is stored on the branch from which it was created, unless an earlier version was already stored on the branch, in which case a new branch is started. When a user selects two versions of an artifact to put in the workspace, the result is a merged artifact and a merge link in the version graph when the result is placed back in the repository. Locking is per baseline or change, and arbitrary nodes can be locked.

Table 1. Policy of RCS as Constructed in Our Architecture.

Module	*Implementation*
Storage constraint	Acyclic, fully connected graph of baselines and changes
Hierarchy constraint	No hierarchical composition
Locking constraint	At least a single node at a time
Distribution constraint	No distribution
Selection action	At most two versions at a time
Concurrency action	Lock/unlock a single node
Evolution action	Store on existing or new branch
Placement action	No placement

4.2 CVS

The policy that CVS uses differs from the policy of RCS in several ways. First, whereas RCS has a pessimistic policy that resolves around the use of locks to prevent conflicts, CVS has an optimistic policy that relies on the use of merging to resolve conflicts as they occur. Second, CVS supports the construction of hierarchal artifacts via the use of directories. Directories themselves, however, cannot be versioned.

Table 2 presents the modules one would use to model the policy of CVS using our architecture. Quite a few modules are different from those of the previous policy in Table 1. In particular, the locking constraint module is now void, as is the concurrency action module. Hierarchical composition is enabled, but restricted to single parents in order to only allow a strict tree of hierarchically structured artifacts. In addition, a new

storage constraint module is introduced to handle directories, which can only be stored as a single baseline.

The storage constraint module for files remains unchanged, but the accompanying evolution action module has been modified slightly. Rather than automatically creating a branch if a newer version of an artifact exists (if the artifact is a file), the evolution action module will fail and inform the user they have to resolve the conflict. This is in concordance with CVS, which allows parallel work but puts the burden of conflict resolution on those who check in last. Directories are not versioned in CVS, therefore the evolution action module for directories simply replaces the one existing baseline.

Although the number of modules that were changed in comparison with the RCS policy may be surprising, we observe that the development of a module is a relatively simple task. In fact, we plan to have all of the modules discussed thus far available as standard modules distributed as part of the architecture.

Table 2. Policy of CVS as Constructed in Our Architecture.

Module	*Implementation*
Storage constraint	Acyclic, fully connected graph of baselines and changes for files; single baseline for directories
Hierarchy constraint	Hierarchical composition, single parent
Locking constraint	No locking
Distribution constraint	No distribution
Selection action	At most two versions at a time
Concurrency action	No locking
Evolution action	Store on existing branch if no conflict for files; replace existing baseline for directories
Placement action	No placement

Changing from a CVS-like policy to a Subversion-like policy is trivial with our architecture. Simply applying the existing storage constraint module for files to directories turns our CVS implementation into a Subversion implementation. Compared to the scenario presented in Section 1, this is a significant difference in effort.
Similarly, the effort involved in making the CVS policy distributed is also small. Since distribution is orthogonal to the other modules, the other modules are simple reused and do not need to be changed. For instance, to replicate artifacts or move artifacts to the repository closest to a user, one would plug in the appropriate distribution constraint and placement action modules.

4.3 Change set

The change set policy does not use a version tree, but instead stores individual baselines and changes independently. Individual versions are constructed in the workspace

by applying a set of changes to a baseline. Because changes are considered completely independent, locking is not necessary.

Table 3 shows the modules used to construct the change set policy with our architecture. Because the change set policy stores baselines independently from each other, the storage constraint module allows disconnected graphs. Because changes are always based on a baseline, however, the storage constraint module enforces dependencies from changes on baselines. The evolution action module is slightly different from its counterpart for the CVS policy, since it only establishes relationships between changes and the baselines upon which they are based.

Our change set policy reuses the hierarchy constraints from the CVS policy, and similarly has no locking and no concurrency action module. Should distribution be desired, the same distribution constraint and placement action modules can be reused from the CVS policy, showing the strength of our architecture in separating concerns and promoting reuse.

Table 3. Change Set policy as Constructed in Our Architecture.

Module	*Implementation*
Storage constraint	Acyclic graph of baselines and changes that depend on baselines
Hierarchy constraint	Hierarchical composition, single parent
Locking constraint	No locking
Distribution constraint	No distribution
Selection action	Arbitrary baselines and changes
Concurrency action	No locking
Evolution action	Store new version as an independent change depending on the baseline
Placement action	No placement

5 Related Work

Several previous projects have explored the issue of easing the development of new configuration management systems. ICE [21], EPOS [8,19], and NUCM [16] were among the first to provide a generic (also called unified) configuration management model upon which to build new configuration management systems. ICE and NUCM, however, have the problem that a configuration management policy is treated as a single monolithic unit. Some reuse is possible, but neither approach explicitly promotes it. The approach of EPOS is better in that regard, since its architecture explicitly separates different parts of a policy in different layers. Due to strong interdependences among the different layers, however, significant amounts of policy reuse remain elusive.

Parisi and Wolf [9] leverage graph transformations as the mechanism for specifying configuration management policies. While certain compositional properties are demonstrated with respect to configuration management policies, the policies they

explore are so closely related (e.g., those underneath RCS [12] and CVS [2]) that it is unclear how the approach may apply to more diverse policies (e.g., change set [11]).
Finally, the unified extensional model [1] is an attempt at creating a generic model that is solely tailored towards extensional policies. While advantageous in providing specific support for building such policies, the approach is limited with respect to policy reuse since it focuses on reuse of the model, not policies.

6 Conclusion

We have introduced a new configuration management system architecture that explicitly supports policy reuse. Key to the approach is our treatment of configuration management policies as non-monolithic entities that are modularized into sets of constraint and action modules. We are currently implementing the architecture. While we still have some significant implementation effort in front of us, early results are encouraging in that the architecture can be faithfully implemented and supports policy modularization.

The initial goal of the project is to reduce the time and effort involved in implementing a new configuration management system. Especially when it is not clear which policy is best suited for a particular situation, our architecture has benefits in speeding up the exploration and evaluation process.

In the long term, our architecture has the potential of making significant impact on two long-standing problems in the field of configuration management. First, we believe the architecture is a step forward towards building a configuration management system that can be incrementally adopted. An organization would start with relatively simple policies, but over time relax those policies and introduce more advanced capabilities as the users become more familiar with the system. For instance, the use of branches (a notoriously difficult issue [17]) could initially be prohibited with a restrictive storage constraint module. Later on, when the users have a good understanding of the other aspects of the desired policy, the original storage constraint module is replaced with a storage constraint module that allows the use of branches.
The second problem that the architecture may help to address is that of policy interaction. Existing configuration management systems typically enforce a policy on the client side. Our architecture, on the other hand, enforces policy constraints on the server side. This represents a step towards addressing the problem of policy interaction, since it allows a server to continue to enforce the original policy as associated with an artifact.

Acknowledgements

This research is supported by the National Science Foundation with grant number CCR-0093489. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement

numbers F30602-00-2-0599 and F30602-00-2-0608. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References

[1] U. Asklund, L. Bendix, H.B. Christensen, and B. Magnusson. *The Unified Extensional Versioning Model*. Proceedings of the Ninth International Symposium on System Configuration Management, 1999: p. 100-122.

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

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

[4] R. Conradi and B. Westfechtel, *Version Models for Software Configuration Management.* ACM Computing Surveys, 1998. 30(2): p. 232-282.

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

[6] P.H. Feiler. *Configuration Management Models in Commercial Environments*. Software Engineering Institute, Carnegie Mellon University, 1991.

[7] T. Mens, *A State-of-the-Art Survey on Software Merging.* IEEE Transactions on Software Engineering, 2002. 28(5): p. 449-462.

[8] B.P. Munch. *Versioning in a Software Engineering Database - the Change-Oriented Way*. Ph.D. Thesis, DCST, NTH, 1993.

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

[10] M.J. Rochkind, *The Source Code Control System.* IEEE Transactions on Software Engineering, 1975. SE-1(4): p. 364-370.

[11] Software Maintenance & Development Systems Inc. *Aide de Camp Product Overview*. 1994.

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

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

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

[15] 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.

[16] A. van der Hoek, A. Carzaniga, D.M. Heimbigner, and A.L. Wolf, *A Testbed for Configuration Management Policy Programming.* IEEE Transactions on Software Engineering, 2002. 28(1): p. 79-99.

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

[18] A.I. Wang, J.-O. Larsen, R. Conradi, and B.P. Munch. *Improving Coordination Support in the EPOS CM System*. Proceedings of the Sixth European Workshop in Software Process Technology, 1998: p. 75-91.

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

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

[21] A. Zeller and G. Snelting, *Unified Versioning through Feature Logic.* ACM Transactions on Software Engineering and Methodology, 1997. 6(4): p. 398-441.