Coven: Brewing Better Collaboration through Software Configuration Management

Mark C. Chu-Carroll
IBM T. J. Watson Research Center 30 Saw Mill River Road Hawthorne, NY 10532, USA
mcc@watson.ibm.com

Sara Sprenkle
Department of Computer Science Levine Science Research Center Duke University
Durham, NC USA
sprenkle@cs.duke.edu

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Evolution – Version Control; D.2.9 [**Software Engineering**]: Management – Software Configuration Management, Programming Teams; D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Human Factors, Management

Keywords

Collaborative programming

ABSTRACT

Our work focuses on building tools to support collaborative software development. We are building a new programming environment with integrated software configuration management which provides a variety of features to help programming teams coordinate their work.

In this paper, we detail a hierarchy-based software configuration management system called *Coven*, which acts as a collaborative medium for allowing teams of programmers to cooperate. By providing a family of inter-related mechanisms, our system provides powerful support for cooperation and coordination in a manner which matches the structure of development teams.

1. INTRODUCTION

Modern software has become increasingly complex as users demand more sophisticated features. This has led to a dramatic increase in the size and complexity of the source code for software systems and a concomitant increase in the size and complexity of the programming teams that build those systems. In such an environment, programmers and their managers must deal with the complex process of coordination and cooperation among the members of the programming team.

We are building a programming environment called *Coven* (COllaborative Versioning ENvironment), that helps programming teams manage coordination in large software projects by providing better support for modern large-scale software development practices. The core of this system is a new software configuration management system which acts as a coordination space, and which is tightly integrated into an experimental collaborative programming environment.

Currently, programmers on large projects generally use two separate sets of tools: a programming environment, and a software configuration management (SCM) system. Most programming environments are typically written as if the programmer were working in isolation, without providing any features to manage cooperation or communication with other programmers. The SCM manages the program source code, and provides a limited set of features for managing interaction between different programmers. The SCM features for allowing interaction between programmers are typically fairly low-level and primitive, and the programming environments provide little to no support for integrating these cooperative SCM features into the programming experience.

Coven provides better support for collaborative software development than existing SCM systems, by providing mechanisms that allow programmers to communicate with one another and coordinate their work. It provides capabilities that reflect to hierarchical structure of real programming projects, where the project consists of a large group of people, subdivided into groups, each of which is responsible for some part of the system. Coven's flexible code repository reflects this structure, allowing large programming teams to coordinate their work in different ways, and at different granularities depending on the needs of, and relationships between, the different project sub-teams. All of its capabilities are integrated with a programming environment specifically designed for use in a team-based environment.

Coven provides these facilities by building a hierarchical code repository system, and integrating a family of features into the hierarchical structure. It stores code at a fine grain, both temporally and spatially, allowing programmers to manage and share code in flexible ways. It provides locking and notification facilities in concert with the hierarchical repository structure in a manner which maximizes the utility of these facilities in a collaborative environment. Finally, it provides flexible browsing capabilities that allow different project sub-teams to reconcile their different structural views of the project.

For the most part, these individual features are not novel. Various forms of replication have been used by TeamWare[22], BitKeeper[3], Infuse[18], and Adele[8, 1]. Our change tracking mechanisms are similar to those used by PRCS[13]. Locks as a coordination mechanism date back at least to RCS[25] and SCCS. Dynamic hierarchical configurations have been handled by systems including Gwydion Sheets[21], ICE[29], Adele, and Infuse. Fragment based versioning and dynamic source file generation were also explored by Lin and Reiss in Desert[20].

However, the way that Coven combines these features in a collaborative setting provides a unique set of capabilities for coordinating work among many programmers that cannot be provided by any of these systems in isolation.

In the rest of this paper, we will discuss how the Coven SCM system provides better support for collaborative team-based software development than current tools. In each section, we will start by presenting an example of a typical team-based development task, and contrast how current systems handle that task with how it would be handled in Coven.

These scenarios will be taken from a common example project. We will use a fictional development project whose goal is to create a new web browser. After analysis and design, the team is divided into two key sub-projects: one whose task is to build the user interface, and one whose task is to handle network communication and protocol support.

The project consists of six programmers, divided into two teams to work on the two key sub-projects. The UI team consists of Alice (responsible for the document model for HTML), Bob (HTML layout and rendering), and Cindy (graphics and Javascript). The communications team consists of Dave (HTTP protocol support), Evelyn (network interface and buffering), and Frank (caching, cookies, and secure communication). The UI team is managed by George, and the network team by Helen.

2. THE REPOSITORY

The Coven repository is the heart of our system. The repository is flexible, based around features that support hierarchical development. The Coven repository is designed to support this structure by providing coordination mechanisms that match the hierarchical structure of the development groups that use it.

2.1 Hierarchical Repository Support

The Coven repository is actually implemented as a family of related repository replicas, which are coordinated through a subscription based messaging system. The use of a repository hierarchy allows project sub-teams to isolate their changes in progress, but still maintain contact and coordination with the rest of the project. We will present the Coven hierarchical repository in two parts: first, we describe how repository replication and the hierarchy work to provide safe change isolation, and then we will describe how changes made in isolation are propagated outward through the system by programmer-initiated releases.

2.1.1 Replication and Change Isolation

The UI team is changing from an earlier HTML parser to one based on the new W3C standard document object model. This is a major internal change to Alice's code, with very little outwardly visible effect.

While the change is in progress, a copy of the system containing the modified code will not be compilable because parts of the document model are in the new form, and parts still in the old. If the changes become visible to the rest of the project group, the entire system will be unworkable: the comm team won't be able to test their code, and even the rest of the UI team won't be able to make progress until the changes are complete and stable.

In simple SCM tools such as RCS or CVS, once code is checked into the repository, it immediately becomes available to all programmers. These systems make some change isolation possible through *branching*, which forms a fork in the path of versioning. However, in all but the most simple cases, managing forks and merges in software systems can be extremely complex, and most projects try to minimize the number of branches, using them only when it cannot be avoided. In such a system, Alice would not check her changes into the repository until she reached a stable point where it would not likely interfere with the work of other team members. In the meantime, she would not be able to check in partial steps without interfering with other team members.

Other systems, such as ClearCase[19] and Vesta[10], make it much easier to create project branches. In these systems, programmers can create an isolated space in which to make changes by producing a project branch which will later be merged into the development stream. Continuus/CM[6] provides a variation of this mechanism, providing branching mechanisms that are specifically tailored towards group structure and project work-flow.

Finally, another family of systems, including Adele[8, 1], CCC[7], and Infuse[18], use an approach based on sub-repositories. In these systems, hierarchies of repository replicas or sub-databases are used to provide a structured method of handling change isolation. Like other members of this family of systems, Coven is built around a hierarchical repository replication system.

The Coven repository is implemented as a hierarchy of linked repository replicas. The system provides a master, central program repository, which manages the master copy of the system. Anything which has been checked in to the master repository is available to all programmers working on the project. This master repository is *replicated* by each project sub-team, to provide a private sub-repository. The sub-repository may be further replicated to produce sub-repositories for smaller sub-teams until each programmer (or each small team of programmers) has a private sub-repository where they can place their own changes. Changes can be checked into these sub-repositories without those changes becoming visible to the rest of the project team. (We call this ability to create frequent isolated project versions *temporal fine granularity*.)

The sub-repositories are linked to their parent using a subscription based notification facility. When relevant events occur, notifications are transmitted between the levels of the tree. For instance, when a unit of code is changed, information about the change is transmitted from the repository where the change occurred to its parent, and if appropriate the parent transmits information about that change to its parent and to its other children.

Using this hierarchical repository support in Coven, Alice could check intermediate versions of her changes into a private repository replica, and not worry about interfering with other team members' work.

2.1.2 Release Management

Alice finishes her changes to the document model and renderer. Now, she is ready to make the code available to other team members. She checks the code in to the repository.

In a simple SCM system, when a user checks their code in to the repository, it instantly becomes the new default version for the entire project. Everyone, both workers on the same subproject and workers on distantly related parts of the project, will simultaneously receive the new version.

In more advanced systems, the mechanisms for change isolation make it easier to gradually merge changes back into the main stream of development. For example, in ClearCase, programmers access the repository using a mechanism called *view expressions* to map filenames in a conventional filesystem to the ClearCase version space. By using the appropriate view expressions, programmers can easily create project branches to get a degree of isolation, and then merge the branch back into the main development path when it reaches a stable point. The appropriate use of branches and view expressions can allow different project sub-teams to maintain different views of the project, and gradually release changes to other programmers. However, especially in the presence of large numbers of branches and hierarchical group structure, the view expressions can become extremely complicated, and can lead to great difficulty in managing a ClearCase repository[28].

The advantage to Coven's approach is that it provides a very simple style of isolation and re-integration which matches natural development group structure and development styles. Changes are automatically released to wider groups of users in a manner which suits the structure of the development group. This natural propagation of current versions happens automatically within the replica tree, without the complexity of either writing configuration expressions (as in ClearCase) or manually managing the branching structure of the repository (as in Vesta).

As we discussed in the previous section, in Coven, changes are initially isolated to a private repository replica. When the programmers responsible for that replica reach a stable point, they need to be able to make their changes accessible to other members of the project. In Coven, this process is called *releasing* a set of changes.

Change releases are integrated into the repository hierarchy. When a programmer releases a set of changes from her replica, the entire set of changes is copied to her replica's parent. After a release, the parent repository reflects a *different* head version of artifacts than any other sub-repositories not involved in the release. The parent then sends out a notification to any other child replicas, informing them that they need to re-synchronize their sub-repository. The replicas, upon receiving this notification, will alert users of the repository of the availability of changes. When the users of the replica are ready, they can re-synchronize with the parent replica, integrating the newly released changes into their replica. This resynchronization is initiated by the user of the replica, rather than automatically, in order to prevent abrupt, unexpected changes from interfering with programmers in the middle of some task.

Because of the locking mechanisms, users rarely make concurrent changes to a single fragment. But under some circumstances, concurrent changes are unavoidable. In such a situation, Coven will perform an optimistic merge of the concurrent changes, and if any errors are detected, it will require a user to intervene and resolve the conflicts.

Using this facility, Alice has the ability to release the code to a particular sub-team of the project. She can start by releasing her code to the other members of the UI team. When they've tested it, and are satisfied that it's ready, it can be released to a wider group. The hierarchical repository support allows team members who work closely together to share code more readily, and to integrate changes more frequently with each other than with team members working on more distant parts of the system.

2.2 Project-Based Versioning Model

The repository stores code in a versioned program store. The store performs versioning at a finer grain, both temporally and spatially, than typical program repositories. [1] Support for change packages and consistent hierarchical projects are provided through a flexible composite artifact mechanism. The use of such a fine-grained repository enhances Coven's coordination features, and enables the use of multi-dimensional program organizations, which we will discuss in section 3.

2.2.1 Fragment-Based Versioning

Alice made changes to the rendering code to support a new document model. When she finished, she checked the code into the sub-repository shared by the UI team. This caused a wide-ranging set of changes, replacing some classes entirely, and altering parts of others. The changes ranged through many files, but tended to be localized to relatively small parts of those files.

In most configuration management tools, the repository stores versions of source files. All of the facilities of the repository from version histories to locks are managed in terms of entire source files. We refer to this as *coarse spatial granularity*. This coarse granularity limits the usefulness of all of those facilities in a collaborative setting:

- A programmer in the process of making a set of changes when a new changeset is released by someone else has difficulty assessing the cost of integrating that changeset immediately versus waiting until his changes are completed.
- Coarse grained locks easily become a bottleneck, even when programmers are trying to change non-conflicting sections of a given source file.
- Change histories of a particular method or function are difficult to extract from the version history of the source file containing the method/function.

The solution to this is to perform versioning of smaller units. Several systems, including COOP/Orm[14] and Desert/Poem[11 20] have created systems with finer artifact granularity.

Coven performs versioning on the level of the smallest independent program fragment. The exact size of a versioned

[1]Temporal fine granularity allows programmers to record change history of small parts of the system frequently, without creating a complex version history for the entire project. Spatial fine granularity allows smaller code elements to be versioned, allowing more precise change histories.

fragment in Coven is dependent on the language(s) being used. For a programming language such as Java or C++, the program fragment size is generally an entire method/function/field declaration. In a text-processing language like LaTeX, a fragment would be a paragraph of text. (A figure illustrating how a Java source file is partitioned into versioned fragments is shown in figure 1.)

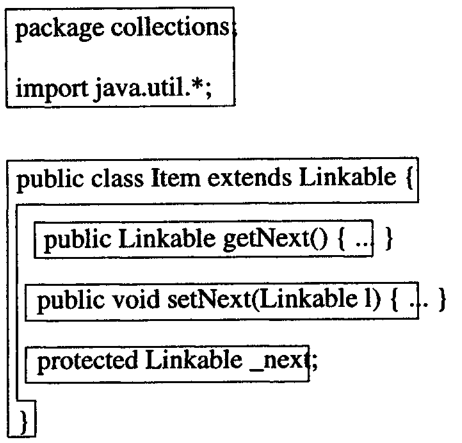

package collections;
import java.util.*;
public class Item extends Linkable {
public Linkable getNext() { .. }
public void setNext(Linkable l) { .. }
protected Linkable _next;
}

Figure 1: Division of Java source into versioned fragments

Each program fragment is versioned individually. This allows programmers to view the change history of an individual item, and to easily understand how program elements will be affected by the integration of a particular changeset. It also provides a vast improvement in terms of communication and coordination: any coordination facility provided by the system can operate in terms of relatively small program units, rather than source files.

Of course, versioning in terms of small fragments means that the system must provide tools for composing these fragments into larger units. Beyond simply maintaining the versions of fragments, the system must also maintain versions of linguistic structures (like classes) and structural entities (like conceptually separated concerns[24]) that are composed of collections of related program fragments.

For instance, to represent the change history of a class, the change history of its fragments is not sufficient. Over time, the class may add or remove methods. These changes should be reflected by a change in an artifact representing the class. This facility is provided in our system through the use of *compound artifacts*, described in the next section.

2.2.2 Compound Artifacts and the Project Model

After Alice checks in her code, the rest of her team starts to make changes to their own code to integrate into the new document model. After several changes are applied, they discover a crucial bug in Alice's code. While she fixes it, the rest of the team needs to revert to the old document model code.

In the simplest SCM systems, reverting to an older version of an entire project can be cumbersome. File based systems like RCS and CVS provide a mechanism called *tagging*, which identifies a version of the project. A tag is a textual label which can be applied to all artifacts in a particular project which marks the version of each artifact that was in place when the tag was applied. The tagging mechanism is completely manual: the programmer chooses to create a tagged version and manually chooses a name for the tag. A tag is only placed when a programmer explicitly does so, and the relationship between different tagged versions is only as clear as the labels the programmer chose to use.

Change package systems like ClearCase[19] make the job significantly easier; since changes are always recorded in *packages* that collect a set of related deltas, one can simply choose the sets of packages to roll back. ClearCase also provides features for recording exactly which artifact versions were used to produce a particular version, allowing rollbacks both to named versions (corresponding roughly to tags in simpler versions), or particular points in time. Systems with a strong notion of configurations as versioned entities, including [8, 1, 10, 20] among many others, retain complete information about the history of all complex artifacts, from single files to subsystems, to entire projects. These systems allow programmers to rollback the system to any past state of any versioned configuration.

Coven's project model is based on the subdivision of a programming project into sub-projects, called *compound artifacts (CAs)*. A CA is a specification of a set of versioned artifacts which are its members. These artifacts may, in turn, be further CAs, nested arbitrarily. When a project contains nested CAs, the contents of those CAs need not be mutually exclusive: two CAs may contain overlapping project subsets. A CA is essentially the same thing as a configuration as discussed earlier; we choose to use a different term because Coven CAs can be very fine-grained entities such as dynamically generated source files, while the common notion of a configuration is a higher-granularity entity. Coven compound artifacts combine both the common configuration notion of most SCM systems and the virtual source file notion of systems like Desert[20] and Sheets[21].

The CAs themselves are versioned artifacts, which change over time. A version of a CA specifies the set of artifact versions which are its members. Between versions of the CA, not only can the particular artifact versions contained in a CA change, but the set of artifacts itself may change. For instance, in the example we discussed in the previous section, a class would be implemented by a CA containing all of the artifacts that contain elements of the class. When a method is deleted and the class CA is checked in to the repository, the artifact containing that method will be removed from the new version of the class CA.

CAs are used for a wide range of purposes in Coven. On a fine grain, a member declaration and its documentation may be contained in a single CA; on a coarser grain, all of the artifacts that make up a particular class declaration are contained in a CA. This hierarchy of ever larger CAs continues until we reach the level where there is a single CA representing the entire project.

A CA version represents a snapshot of the versions of a particular collection of artifacts, as they existed at some point in time. Through the use of the CA system, Coven allows programmers to roll back any set of changes, returning any part of the system to its exact state at some point in the past.

CAs provide the basis for our model of *project consistency*. As we described in the previous section, the history of a CA represents a snapshot of the state of a collection of member artifacts at some point in time. We say that such a set of versions which coexisted together can be referred to as a *consistent project version (CPV).*

Our model extends this notion of consistency with temporal fine granularity of CPVs. A given CA is subdivided into sub-CAs. The CA has a history of CPVs that represent the state of consistent versions of the entire set of artifacts contained within the CA. The sub-CAs also have their history of consistent versions. The sub-projects may have CPVs that are not components of any CPV for the entire project — they represent intermediate states of the subproject between consistent versions of the enclosing project.

In terms of our example, Alice has a consistent version history of the subproject containing her code in the history of her particular subproject CA. While she is working on the new document model, she continues to create new consistent versions of her subproject CA. Until she's ready to release the code to her co-workers, she does not create a consistent version of the UI subproject CA — her consistent versions are not ready to integrate with the rest of the project. But within her subproject, she has a full history of the changes she made, and she can easily revert to any past state of that subproject.

This use of hierarchical (sub)project CAs means that the history mechanisms can operate at multiple temporal granularities. The history of the CA representing the entire project is fairly coarse, representing a history of stable states of the primary components of the system. Within any given version of the entire project, there exists a history of how each component of that system evolved to the new version. Programmers can take advantage of this by having a precise history of their own work, by being able to maintain flexible isolation and integration of their changes, and by being able to view the change histories of other parts of the system with the necessary degree of detail.

2.2.3 Consistency Details

Our use of potentially overlapping subproject CAs introduces a consistency requirement for CA versions. For any artifact contained within a CA, there must be exactly one version of that artifact.

For an example of why this consistency requirement is needed, consider the following. Given a system represented by a CA P, which contains some set of sub-CAs including Q and R. Version P_i of P contains versions Q_j and R_k of Q and R. Q_j and R_k both contain a version of a simple artifact a_l. Someone changes a, producing version a_{l+1}. She checks in this change using sub-project Q, producing version Q_{j+1}. Now she tries to check in project P, with Q_{j+1} and R_k as members. If a version P_{i+1} was created, it would contain both versions a_l and a_{l+1} of a.

The solution to this problem uses a feature logic much like that described in [29]. (Another solution to this problem based on a different approach is described in [8].) Each CA version is represented using a specification containing a list of features describing its members. For each artifact, there is a version feature, specifying the version of that artifact contained within that CA version. When a new CA version is created, the artifact specifications of all artifacts transitively included in the CA are merged. The merge operation for the artifact version features is defined to produce an error for non-matching version values.

When such an error occurs, the system rejects the check-in of code due to the inconsistency. It generates a notification which is sent to the programmer who performed the check-in and to the programmers who checked in the CA versions involved in the conflict, so that the involved programmers can work together to resolve the problem. The system can assist in the resolution of this conflict by providing the programmers with a diff3-like view of their conflicting code, allowing them to identify and resolve the key differences.

2.3 Locking and Coordination

The key to coordination between members of a project team is communication. Coven provides a set of facilities that allow the SCM system to support and enhance communication between team members. The primary communication mechanism provided by Coven is called a *soft lock*. A soft lock allows programmers to attach notifications to parts of the project in a way that allows them to keep in touch with other programmers. We will first present the notion of soft locks, and describe how they allow programmers to communicate. Then we will describe how soft locks are integrated with the repository hierarchy to provide a family of different coordination mechanisms operating at different granularities.

2.3.1 Soft Locks

Since Alice is busy replacing the old document model code, it's important that other programmers not make changes to the code that she's replacing, or their changes will cause hard to resolve conflicts later.

Many SCM systems provide some kind of mechanism for *locking* code. This allows programmers to mark a section of code that they intend to change as "off limits", to prevent other programmers from making conflicting changes. These locks provide a powerful coordination mechanism, allowing a programmer to notify others about their activities to prevent a difficult and expensive problem. But in current SCM systems, locks have a major weakness: since SCM systems typically operate on entire source files, a lock limits access not to a single program construct, but to an entire source file. When a programmer locks a source file to make a change to one program fragment located in that source file, the entire file is locked, and no one can make changes to any other part of the system contained in that file. This improper lock granularity causes locking to become a major bottleneck, where programmers interfere with one another not because of genuine conflict but because they need to work with different program elements that happen to be located in the same source file.

In response to this problem, many SCM systems have adopted an optimistic approach, in which they assume that programmers are well-coordinated from outside of the SCM system and that conflicts will occur very rarely. They therefore discard locking and allow programmers to edit files as needed. When changes are checked in to the repository, the SCM system checks that there are no conflicts, and requires the programmer to resolve the conflicts immediately if they occur.

Coven takes a middle-ground approach. Locking can be a very serious bottleneck to a development team. But that does not mean that tools should deprive programmers of a useful coordination facility. Coven therefore provides a facility called a *soft lock*. A soft lock is placed on code by a programmer to notify others that she is making changes. Associated with the soft lock is a log message placed by the programmer. When another programmer tries to change soft locked code, he is notified that the code is locked, and presented with the lock message. If he decides to go ahead and make changes, the programmer who placed the soft lock is notified of those changes, so that she can be aware of any impending conflicts. When such a conflict is detected, the system allows the involved programmers to know in advance of the potential for a problem, so that they can discuss the appropriate action. The system can provide some communication mechanisms to assist in this process.

As with all of the other mechanisms of Coven, soft locks are integrated into the repository hierarchy, allowing locks to be placed at many different granularities, from locking small program units to locking entire projects. At the finest grain, soft locks can be attached to individual program fragments. At coarser grains, soft locks can be attached to any CA, which implicitly attaches the lock transitively to all artifacts contained by the CA. Thus, a single lock can cover anything from a single fragment to an entire project. The system allows users to choose the lock granularity which is best suited to the task that they are performing, choosing the tradeoffs between managing large numbers of small locks, versus small groups of large locks according to their particular requirements.

2.3.2 Hierarchical Lock Management

Once Alice has checked in her code, other members of her sub-team can look at and alter the code that she's checked in to the repository. She doesn't need to be notified each time they make a change; however, she still needs to know if anyone from the communications sub-team alters anything in the old code that she has replaced but has not released to them yet.

In most SCM systems, a system artifact on a given project branch is either locked or unlocked. To allow code to be locked for some users, but not for others requires building project branches, with the difficulties discussed earlier. Coven uses the repository hierarchy to allows a more flexible locking system without having to explicitly manage multiple project branches, and while retaining a single linear change history for each artifact. It does this by allowing locks to be placed over portions of the repository hierarchy. A lock may be placed at any level in the hierarchy, and may cover either the entire sub-hierarchy, or may exclude sections. In this example, Alice could place a soft lock on the components that she is changing in a manner that locks the components over the entire repository hierarchy, excluding only the sub-repository used by her sub-team.

Internally, each repository replica maintains information about locked artifacts in the repository. Lock information is transmitted between replica levels as necessary.

When a programmer synchronizes her replica with its parent, she may choose to lock some part of the system (that is, some artifact or collection of artifacts) that she (or her sub-team) plan to change. This lock may be placed in a manner which locks the code throughout the entire replica hierarchy, excluding the particular sub-hierarchy where they will be performing the changes. After such a lock, programmers who use the unlocked replica or children of the unlocked replica can make changes to the locked code without triggering the lock notifications, but programmers outside of this sub-hierarchy cannot.

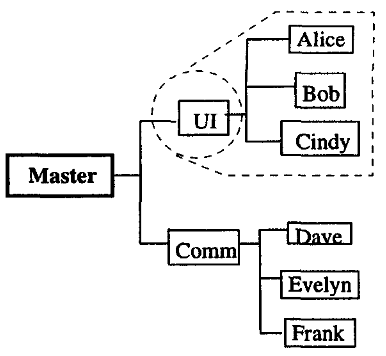


Figure 2: An example of hierarchical locking

An example of the repository replication hierarchy used by our project team is illustrated in figure 2. If Alice locks the code for the document model to make changes in the UI team's repository replica (illustrated by the circle around the UI replica), then the lock is in place through the entire hierarchy excluding the section surrounded by dotted lines: that is, in the master repository replica and in all of the comm team's sub-replicas. Similar locking/coordination features are discussed in [8] and [18].

Using this mechanism, Alice could release the code to her project sub-team, allowing them to use and alter the newly released version of her code, but she will still receive notifications when anyone else tries to change any members of the CA that she was working on.

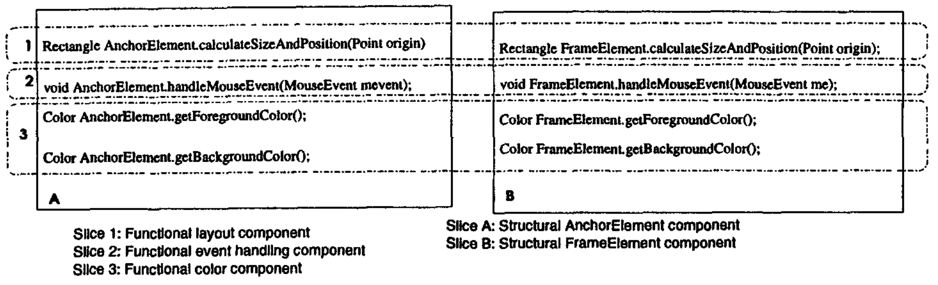
3. MULTIDIMENSIONAL PROGRAM OR GANIZATION

Alice and Bob work together on one of the key pieces of the user interface: the document rendering frame. But they have very different views of it.

Alice views it as a canvas, upon which document elements paint themselves. Her view is based on the document object model itself, on the classes that make up the structure of a document, and how that structure will paint itself onto the screen.

Bob views it as a container for user interface elements. Some of the elements that get painted onto the screen are actually links, and clicking on them should cause the user interface to react. Bob does not particularly care about the document model: he just knows that certain kinds of elements will be involved in UI actions that he needs to program. He needs to know where links are located on the screen and how layout is done. But he does not particular care about the details of the document structure that underly it.

For Alice, the natural way of looking at the program is through an object-oriented AST pattern. A document is represented as a parse tree, where each node in the tree represents a document element. Each document element type is implemented as a class, which has a family of methods that implement the operations of the class. Rendering is just one method among the many. Alice wants to view the code organized according to those classes. For her purposes, the ideal organization of the code into files puts all of the functions for a given class into its own source file.


```
Rectangle AnchorElement.calculateSizeAndPosition(Point origin)
void AnchorElement.handleMouseEvent(MouseEvent mevent);
Color AnchorElement.getForegroundColor();
Color AnchorElement.getBackgroundColor();

Rectangle FrameElement.calculateSizeAndPosition(Point origin);
void FrameElement.handleMouseEvent(MouseEvent me);
Color FrameElement.getForegroundColor();
Color FrameElement.getBackgroundColor();
```
1
2
3
A
B
Slice 1: Functional layout component
Slice 2: Functional event handling component
Slice 3: Functional color component
Slice A: Structural AnchorElement component
Slice B: Structural FrameElement component

Figure 3: Orthogonal Program Organizations

For Bob, the object-oriented pattern is distracting. What he cares about is a limited subset of the code: how layout is performed. He wants to be able to follow the data and control flow of layout on the screen. For him, the best code organization would be one where all of the placement methods are together, so that he could follow the control flow through the layout methods.

In conventional programming environments and SCM systems, programmers must accept one organization of the code. A given program fragment is located in *one, and only one* source file. The programmers must agree on a canonical organization. In our example, that means that either Bob or Alice must accept a code organization which is not well-suited towards their task.

While there is a whole new field of software engineering work, called Multidimensional Separation of Concerns[24, 23], dedicated to this problem, very few SCM systems have addressed it. The CMU Gwydion project[21] started work on this, but never reached the point of building a complete SCM repository. Lin and Reiss's Desert[20] system provides an approach similar to that of Coven, but based on fragments having a primary home in one source file.

Coven takes advantage of fragment based versioning to allow programmers to generate multiple orthogonal program organizations. Each fragment is handled separately by the repository. Programmers access the repository by checking out dynamically generated *collections* of program fragments called *virtual source files (VSFs)*. Each program fragment can be located in many different VSFs, each of which represents a piece of a different decomposition of the program. Alice and Bob can each have their own optimal program organization, without interfering with each other. Internally, a VSF is simply a subproject represented in a manner which is useful for viewing a collection of program fragments in a convenient way.

Figure 3 illustrates a view of a small part of the program, and two orthogonal program organizations. Slices 1, 2, and 3 form a decomposition of the program into components based on functional elements; slices A and B form a decomposition of the program into components based on class structure. The decomposition into functional components is the program view that Bob prefers, while the decomposition into classes is Alice's preference. By using VSFs, Bob and Alice can each use the program organization best suited to the task that they are performing, without any conflict.

3.1 Query Based Repository Access

Bob wants to change some event handling methods in the browser. He needs to look at the layout subsection of the document model code to change how events are signalled by some elements of the rendered document. The code is structured according to the DOM specification, and is therefore organized by classes in the DOM hierarchy. This organization scatters the relevant code across many different source files.

As we discussed in the previous section, in a conventional SCM system, Bob would need to look at code scattered through many different source files. With Coven, he could use a virtual source organization that allows him to virtually restructure the code into the functional elements that he wants. However, generating this new organization is difficult. Specifying a program organization by identifying exactly what fragments should be included in each VSF is prohibitively difficult.

Coven simplifies the identification process by providing a query language that programmers use to specify how to generate the different program organizations. Each time a virtual source file is checked out of the SCM system, the set of fragments contained in that VSF is dynamically generated by executing a query.

Coven's query language consists of a simple core to which query extensions can be added. The initial system provides a set of primitive query types, which can be assembled to form complex queries. Users of the system are free to add new query types, which can be either compositions of existing queries, or entire new components that are dynamically linked into the system.

Each virtual source is specified by a query line with a predicate that specifies a condition that fragments must meet to be included in the source. The system scans the repository, generating the collection of fragments which should be included in the VSF. Once the fragments are collected, the VSF is generated in a form determined by its format clause. By default, the VSF format is an XML document which is used internally by the Coven environment.

VSFs can be exported in other formats by specifying a value for the format clause. The format clause specifies the name of a component called an *exporter*, which is used to generate the VSF in a particular form. Programmers are free to add new exporters to the system, allowing them to generate VSFs in any format convenient for their use. Each exporter is generally paired with an importer, which allows the system to read code into the repository from other formats.

A typical example of the use of exporters is shown in figure 4, where a format type "java" is specified. This exporter would generate the VSF in the format of a typical Java source file, which is usable by a standard Java compiler, or any other tool that reads Java source files. Through the use of a set of simple filter programs, even timestamp dependent tools like Unix "make" can easily be made to interoperate with Coven.

```
// Example 1
source PerformLayout =
  select fragment from browser.ui where
    (fragment impl calcSizeAndPosition)

// Example 2
source MouseEvents =
  select fragment from browser.ui where
    (fragment impl handleMouseEvent) or
      (exist other from browser.ui where
        (other impl handleMouseEvent)
          and fragment dependsOn other)

// Example 3
source AnchorElement.java =
  select fragment from browser.ui where
    (fragments memberOf 'AnchorElement')
    format 'Java'

view Functional = { PerformLayout,
      MouseEvents }
```

Figure 4: Example of queries for the functional organization

In figure 4, we illustrate example queries for a Coven repository containing Java code. The first query selects the set of fragments that contain implementations of a method named "calcSizeAndPosition". The second is similar in that it also selects all methods that implement a particular method (in this case, "handleMouseEvent", but it also includes fragments on which implementations of "handleMouseEvent" depend. The third example illustrates how code can be withdrawn from the repository to be used by other tools. It selects the set of fragments that are members of the class "AnchorElement", and exports them as a standard Java source file, with fragment identity information encoded into comments. The generated VSF will be usable by conventional, non-Coven based tools, including standard compilers.

Example 4 illustrates how VSFs are used to generate program organizations. A program organization (called a *view* in the query language) is nothing more than a collection of virtual source files that represent one organization of the program source.

The use of query-based repository access can lead to added complexity, because programmers may need to write complex query expressions to get exactly the set of fragments in which they are interested. But we believe that this tradeoff is acceptable: our query language makes routine queries extremely simple, but provides the power to do more complex things. The simple queries for simple cases prevent Coven from being significantly more complex than more conventional systems. For the advanced cases where complex queries are required, we are providing a significant capability which is not possible using conventional systems.

Because all of Coven's coordination mechanisms are provided in terms of individual fragments (and collections of fragments), the coordination tools work equally well even when programmers are viewing the code through different program organizations.

4. USER INTERFACE SUPPORT

The use of an SCM system like Coven introduces added complexity to the task of the programmer. The programmer gains expressiveness and organizational flexibility at the cost of dealing with queries, consistency, and the loss of intrinsic context. To hide the added complexity, a programming environment can make the composition and decomposition of programming fragments and the manipulation of multiple program organizations more tractable.

We have built a programming environment which allows the easy integration of external tools into a flexible linked-pane style programming environment. This environment integrates all of the coordination and versioning facilities of Coven into a cohesive programming environment.

The use of such an integrated environment is crucial. Without it, notification mechanisms such as soft locks lose their immediacy, which is a key to strong coordination.

Beyond simply integrating Coven's features, the programming environment enhances them by making them easier to use. For instance, the environment simplifies the process of generating new queries, by providing tools to guide programmers. It eases the use of different views, by providing tools to rapidly and easily switch between them without loss of context.

For instance, consider Dave, working on the HTTP protocol support. He needs to integrate his code with Alice's document model code. But his view of the system is quite different from hers. When he has a question about Alice's code, he can ask her for an explanation. She can use her environment to send a description of her screen to Dave; Dave's environment will then show what Alice is seeing. Once Dave's questions are answered, he can switch his environment back to his preferred view, highlighting the code that he saw from Alice's viewpoint.

5. RELATED WORK

5.1 Cooperative Environments

The COOP/Orm[14] project has developed a collaborative SCM system integrated with a programming environment with goals very similar to Coven. COOP/Orm, however, uses an optimistic approach with synchronous updates. In such a system, programmers can change any code at any time, with the changes immediately reflected in the workspaces of all programmers using the system, and with the system detecting conflicts. Coven takes a different approach, based on our belief that in an environment with large numbers of programmers working on a project, immediate updates are too disruptive to individual programmers. Instead of the immediate update approach of COOP/Orm, we have instead opted to use the hierarchical replication/release strategy with soft locking. COOP/Orm also uses a fine-grained versioning model with hierarchical documents, similar in some ways to our model of fine-grained versioning with hierarchical composition. They provide fine-grained versioning down to a finer level than what Coven provides, performing versioning on the level of individual programming language expressions. We believe that the costs of the complexity imposed on the programmer and the environment to deal with such fine granularity is greater than the benefit that it provides.

Systems including TeamWare[22] and Infuse[18] provide replication support similar to ours. We believe that our system takes better advantage of this replication hierarchy because of its integration with other features, including query based repository access and soft locks.

Adele[8, 1] provides extremely powerful support for software building, along with a form of hierarchical support based on sub-databases. It does this through the use of rather complex object declarations which express dependencies between objects in the repository. This mechanism allows to store and version not just the original source artifacts, but all intermediate products as well. Similarly, Vesta[10], while not providing hierarchical support, does provide excellent support for change isolation, while also managing builds and versioning of intermediate results.

5.2 Dynamic Views and Multidimensional Organization

The idea of multiple program views based on rearranging source organizations was explored by the Gwydion project from CMU[21]. The Gwydion group built a hyper-code programming environment called *Sheets*. Like Coven, the Sheets system subdivides code into program fragments and allows fragments to be dynamically assembled into new virtual source files, which they call sheets. The Sheets environment used a query language to generate sheets viewed by their UI. However, the Sheets system did not integrate SCM with this dynamic view support. Further, this view system was based on their use of a repository which could only be accessed from within their system. Lin and Reiss[11] provide very similar functionality in their desert system, complete with fragment-level versioning, but using a different model of repository access.

The Hyper-J project at IBM, both through their hyperspaces work[24], and their earlier subject-oriented programming[15, 16] have explored the notion of multidimensional program organizations. Their focus has been on implementing software using multidimensional separation of concerns, and then integrating those separated concerns through a powerful composition system.

5.3 Repositories

IBM's VisualAge Smalltalk includes a fine-grained program repository with some collaborative facilities, called ENVY[17]. ENVY includes some locking facilities and a replication model that resembles ours in many ways. ENVY presents a model of the project that strictly divides the system into subparts in static ways. It provides locking support strictly in terms of this statically imposed program organization. This system is not as flexible as Coven, where new organizations may be dynamically generated as needed, and locks can be used to coordinate work on any collection of code desired by the programmer. Further, ENVY provides no query engine nor any other tools for generating custom views of source code. Because of the tight integration with the repository and the lack of any mechanism like programmable our format mechanism for exporting code, third party tools are difficult to use with their system.

ClearCase[19] and related SCM systems provide powerful configuration management with change package support. ClearCase provides this support transparently by presenting itself as a network file system, with the change package support normally hidden from the user. Change packages do provide a form of project consistency, but do not have the expressive power of our consistency model. In particular, ClearCase has no notion of a consistent subproject, which prevents or restricts the use of the flexible coordination mechanisms similar to those provided by Coven. Beyond acting as a code repository, ClearCase also provides significant development support in integrating bug-tracking, associating change packages with bug fixes, and baseline and build management, none of which are currently supported by Coven.

Conventional version control systems, including RCS[25], CVS[4], and PRCS[13], all allow programmers to interact through a common code repository. All provide some tools for programmer interaction. CVS and PRCS both use optimistic methods with conflict detection and resolution, whereas RCS uses exclusive locking. CVS and RCS lack project consistency models. PRCS has a strong project consistency model quite similar to ours except that it relies on files as versioned artifacts.

BitKeeper[3] and NUCM[26, 27] are both distributed repositories, and both have disconnected/replicated modes. But neither has strong coordination features. Both are modeled on distantly connected, non-coordinated development in the open-source style.

WebDAV[9][5] is a protocol for distributed authoring and versioning support on the web based on an extended version of HTTP. WebDAV includes both locking, and some aggregate structure that could be used to represent configurations. WebDAV does not specify the granularity of its artifacts — it merely identifies its versioned artifacts as entities with a URL. WebDAV is designed to be flexible about exactly what kind of repository is accessed by the protocol. It may be possible to build a WebDAV based front-end for Coven.

5.4 Other Forms of Collaborative Support

Many systems, including Adele[8], Vesta[10], Infuse[18] and DSEE[12], have focused attention on software building as a different aspect of the collaborative programming problem. In large systems, managing builds in the presence of multiple users making frequent changes is an extremely complex problem. All of these systems use a variety of caching, process, and policy management systems for easing the build problem. Coven does not yet make any attempt to address this problem; we hope to make this a focus for the future.

Adele[8], Infuse[18], and Marvel/Oz[2] have focused on building SCM systems that support software processes. In these systems, software processes such as testing requirements, code reviews, and code approvals are all enforced through codeable process specifications. This allows the system to enforce certain cooperation/coordination styles, and assist programmers in the appropriate kinds of cooperation and coordination.

6. CONCLUSION AND FUTURE WORK

In this paper, we presented the Coven integrated programming environment and software configuration management system. This system allows programmers to cooperate and coordinate their work through the SCM system that manages their code. By structuring the repository as a hierarchy and integrating a collection of coordination and communication features into the hierarchical structure, it allows the coordination facilities of the repository to be used in a manner which matches the hierarchical style of software development used by large software development teams.

Our future directions point in two main directions: providing better support for collaboration and providing more complete software configuration management support.

To provide better collaborative support, we plan to initially build better user interface support for collaboration. Once we have better UI support, we plan to examine how real programmers work with Coven, and see how we can adapt the collaborative features of the system to better support the real practices of collaborative software development. In particular, we plan to explore more communication facilities that can do more to allow programmers to resolve conflicts detected by the locks and inconsistent CA check-in.

In the software configuration management area, we plan to address the build problem in a collaborative setting, including integrating support for process-oriented repository controls similar to those provided by Adele.

7. REFERENCES

[1] N. Belkhatir, J. Estublier, and W. Melo. Adele 2: A support to large software development process. In *Proceedings of the 1st International Conference on the Software Process*, 1991.

[2] I. Ben-Shaul and G. Kaiser. Federating process-centered environments: the oz experience. *Automated Software Engineering*, 5(1):97–132, January 1998.

[3] Bitkeeper, Inc. BitKeeper source management: Details of operation. Webpage; "http://www.bitkeeper.com/bk05.html".

[4] P. Cederqvist. *CVS Reference Manual*, 1998. Available online at "http://www.loria.fr/ molli/cvs/doc/cvs_toc.html.

[5] G. Clemm and C. Kaler. Versioning extensions to WebDAV. Technical report, IETF, 1999.

[6] Managing your eassets with continuus cm synergy: 2nd generation task-based change management. web-pamphlet at "www.continuus.com", 2000.

[7] S. Dart. Spectrum of functionality in configuration management systems. Technical Report CMU/SEI-90-TR-11, CMU Software Engineering Institute, 1990.

[8] J. Estublier and R. Casallas. *Configuration Management*, chapter The Adele Configuration Manager. Wiley and Sons, Ltd., 1994.

[9] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP extensions for distributed authoring – WebDAV. Technical Report RFC2518, The Internet Society, February 1999.

[10] A. Heydon, R. Levin, T. Mann, and Y. Yu. The vesta approach to software configuration management. Technical Report 1999-01, Compaq SRC, 1999.

[11] Y. Lin and S. Reiss. Configuration management with logical structures. In *Proceedings of ICSE 18*, pages 298–307, 1996.

[12] D. Lubkin. Heterogeneous configuration management with dsee. In *Proceedings of the 3rd Workshop on Software Configuration Management*, pages 153–160, 1991.

[13] J. MacDonald, P. Hilfinger, and L. Semanzato. PRCS: the project revision control system. In *Proceedings of SCM 8*, pages 33–45. Springer Verlag, 1998.

[14] B. Magnusson and U. Asklund. Fine grained version control of configurations in COOP/Orm. In *ICSE '96 SCM-6 Workshop*, pages 31–48, 1996.

[15] H. Ossher and W. Harrison. Combination of inheritance heirarchies. In *Proceedings of the 1992 Conference on Object Oriented Programs, Software, Languages and Applications*, pages 25–40, 1992.

[16] H. Ossher, M. Kaplan, W. Harrison, A. e. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings of the 1992 Conference on Object Oriented Programs, Software, Languages and Applications*, pages 235–250, 1995.

[17] OTI. ENVY/Developer: The collaborative component development environment for IBM visualage and objectshare, inc. visualworks. Webpage: available online at: "http://www.oti.com/briefs/ed/edbrief5i.htm".

[18] D. Perry and G. Kaiser. Infuse: a tool for automatically managing and coordinating source changes in large systems. In *Proceedings of the ACM Computer Science Conference*, 1987.

[19] Rational ClearCase. Pamphlet at "www.rational.com", 2000.

[20] S. Reiss. Simplifying data integration: the design of the Desert software development environment. In *Proceedings of ICSE 18*, pages 398–407, 1996.

[21] R. Stockton and N. Kramer. The Sheets hypercode editor. Technical Report 0820, CMU Department of Computer Science, 1997.

[22] Sun Microsystems, Inc. TeamWare user's guides, 1994.

[23] P. Tarr, W. Harrison, H. Ossher, A. Finkelstein, B. Nuseibeh, and D. Perry, editors. *Proceedings of the ICSE2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, 2000.

[24] P. Tarr, H. Ossher, W. Harrison, and J. S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.

[25] W. Tichy. RCS - a system for version control. *Software: Practice and Experience*, 7(15), 1985.

[26] A. van der Hoek, A. Carzaniga, D. Heimbigner, and A. Wolf. A reusable, distributed repository for configuration management policy programming. Technical Report CU-CS-864-98, University of Colorado Department of Computer Science, 1998.

[27] A. van der Hoek, D. Heimbigner, and A. Wolf. A generic, peer-to-peer repository for distributed configuration management. In *Proceedings of ICSE 18*, March 1996.

[28] D. Weintraub. The Not-So-Official ClearCase page. webpage, 1998. URL="http://www.eclipse.net/ davidw".

[29] A. Zeller. Smooth operations with square operators: the version set model in ICE. In *ICSE '96 SCM-6 Workshop*, pages 8–30, 1996.