

# Separation of Concerns in Software Configuration Management

Mark C. Chu-Carroll

March 23, 2001

## 1 Introduction

Separation of concerns is one of the foundational rules of software engineering. Separating the aspects of systems that perform different roles simplifies the code of software systems, making those systems easier to implement, easier to understand, and less prone to bugs.

Similarly, software configuration management (SCM) systems are one of the foundational tools of software engineering. SCM tools have proved to be so valuable to software engineering that no significant system is implemented without the use of SCM tools. However, in the SCM systems themselves, many important concerns remain unseparated, particularly the key concerns of program storage and program organization. This lack of separation limits the functionality that can be provided by the SCM system.

I have been building an SCM called Coven system based on separating the concerns of program storage, program organization, and inter-programmer coordination. I believe that by separating these aspects of the SCM system, that the resulting system becomes more powerful, and enhances both the basic functionality of the system, in particular its ability to be used to handle separation of the overlapping concerns that make up the real systems developed using the SCM system. A more detailed description of the Coven SCM system can be found in [3].

## 2 Multidimensional Software Configuration Management

A key to our approach is observing that in a conventional SCM system, source files are used as a basis for addressing two distinct concerns: storage, and organization. The storage concern is an obvious notion: code must be placed in a source file to be stored by a file-based system. But the placement of code into a *particular* source file, and into a particular location in a particular source file is not just for the purpose of storage: it can communicate information about the system to other programmers. The layout of code into source files creates an organizational decomposition of the system into discrete units which demonstrates a viewpoint on the structure or meaning of the system. This is what we call the *organizational* concern of software development.

The storage concern is actually a key basis for much of the functionality of the SCM system. The atomic units in the storage concern are also the basic units used by the system for providing versioning, coordination, consistency, and communication.

The fundamental problem with the current situation is that by the lack of separation between storage and organization, SCM systems treat source files as atomic units (because they are atomic for storage purposes), when in fact they are semantically composites made up of collections of smaller independent code fragments, and other concerns can only be addressed properly by treating source files as non-atomic, and allowing features to be provided in terms of the smaller entities that make up a source file.

By providing the key functionalities of versioning and coordination in terms of large, complex compounds, the SCM system is thus crippled, and cannot provide a variety of facilities which are potentially extremely useful. In particular, it limits the abilities of the SCM system to address issues of communication and coordination among teams of programmers collaboratively developing large systems – which is exactly the domain for which SCM functionality is most critical.

The solution that I propose to this is *multidimensional software configuration management*, which separates the key concerns of program storage, program organization, and inter-programmer coordination. To achieve the goals of multidimensional SCM, an SCM system must provide the following facilities.

1. Programs (and related information) are stored as fine-grained artifacts, where a stored artifact is the smallest independent semantic unit of the language used to write the artifact.
2. Organizational views are provided through dynamically generated composite objects called *virtual source files* (VSFs). A VSF looks like a normal source file, except that its elements are actually independent entities.
3. Coordination facilities are provided in terms of collections of stored artifacts. In a locking SCM system, locks would apply not to individual program artifacts, but to dynamically specified collections of artifacts that are related to a particular change.
4. Versioning is provided in terms of consistent views of a system. That is, when a change is performed, it is not recorded as a series of updates to individual artifacts: it is recorded as an atomic change to the system. Thus, the system records a history of reproducible *consistent project versions*, where a consistent project version represents the precise state of a project at some point in time.

In the rest of this paper, I will describe Coven, the multidimensional software configuration management system being developed by my group at IBM Research, and how it meets these requirements.

## 2.1 Program Organization and Virtual Source Files

In Coven, we perform storage and versioning of code in small, fine-grained artifacts. These artifacts correspond to the independent semantic units within a source file in a conventional SCM system. (For example, for storing Java programs, an artifact is a field or method declaration; for Z, it is a schema, a definition, or a block of descriptive text.)

Program organization in Coven is provided through a dynamic facility to assemble these fine grained artifacts into compound units called *virtual source files* (VSFs). VSFs play no role in program storage, but exist solely for the purpose of communicating organizational meaning. VSFs themselves can be named and stored by the system, and can in turn be members of higher-order VSFs which correspond to virtual source directories, subprojects, etc. A given artifact is not restricted to being a member of one VSF, and a given VSF is not restricted to being a member of one virtual directory.

This separation of storage from organization allows programmers to create multiple, orthogonal code organizations in terms of VSFs, each of which describes a distinct viewpoint of the structure of the system. Through this mechanism, programmers familiar with particular parts of the system can store views that serve as an illustration or explanation of that part of the system.

In addition to this service for code understanding, the VSF system allows programmers to store code in structures well suited to different tasks: to perform a particular task, the programmers can choose the organization best suited to that task.

- All artifacts that contain a call to the method "registerGenerator":  

```
all x | x calls "registerGenerator"
```
- All artifacts that implement the method "getProperty", call the method "submitQuery", and contain an assignment to the field "\_xmlRepr":  

```
all x |
  x implements "getProperty" AND
  x calls "submitQuery" AND
  x assigns "_xmlRepr"
```
- All artifacts that are a part of the dynamic menu contribution subsystem:  

```
all x |
  x implements "contributeMenu" OR
  x creates "MenuContribution" OR
  (exists m : MethodCall |
    m in x AND
    (m.name="generateMessage" AND m.param[1]="menu.submit") OR
    (m.name="subscribeMessage" AND m.param[1]="menu.contribute"))
```

Figure 1: Example queries for VSF generation

Coven provides support for dynamically generating VSFs through the use of a query language. Programmers can specify queries to describe what program artifacts should be included in a particular VSF, or what VSFs should be

included in a virtual directory, etc. The use of a query language allows programmers to compactly and understandably describe the contents of a program artifact. (We present our argument for the necessity of a dynamic query system, along with the details of the query language and its efficient implementation in [4].) When a VSF is named and stored by the system, the query that generated the VSF is stored under the name, and can be incrementally (and automatically) updated as the programmer modifies the set of artifacts that make up the VSF. A few sample queries taken from our study of a highly decoupled programming environment are illustrated in figure 1.

## 2.2 Locks and Coordination

Like program organization, in most SCM systems, coordination between programmers overlaps with storage: coordination is provided through features like locks or optimistic conflict detection, and that functionality is provided entirely in terms of atomic source files. This leads to problems with strong coordination facilities, because they are provided in terms of the atomic unit of the storage concern, when in fact, they should be provided in terms of the semantic units of the underlying programming language.

The purpose of coordination facilities is to allow programmers to communicate information about impending changes. The coordination facilities allow the system to maintain a consistent codebase, allowing multiple programmers making concurrent changes, but preventing conflicting changes from being entered into the shared code repository. There are two main methods used to perform coordination: locking (or pessimistic) coordination, and optimistic coordination.

In locking coordination, the SCM system allows programmers to lock the stored program artifacts that they intend to change. Only one programmer is allowed to have a lock on a given artifact at a given moment. If this mechanism is used correctly, programmers are prevented from making conflicting changes.

In optimistic coordination, the SCM system makes no attempt to actively coordinate the programmers. It allows programmers to make concurrent changes without bothering to prevent conflicts. However, when code is integrated into the shared repository, it detects any conflicts between the code being integrated, and the current version in the shared repository. If any conflicts are detected, the code integration is halted, and the conflicts must be resolved, possibly with the assistance of the SCM system.

I claim that locking coordination is the correct approach: the SCM should make its best effort to prevent conflicts from occurring. Optimistic coordination is only desirable because locks are provided on the wrong objects, and this, in turn, is because of the fundamental error of treating source files as atomic units.

Coordination is properly provided in terms of collections of artifacts: locks should be applied to the exact collection of artifacts affected by an impending change. In a conventional SCM system, where coordination occurs in terms of files, this is impossible. But in a system like Coven, where the concern of coordination is properly separated from both storage and organization, it is natural to provide locking coordination of the correct structures.

In Coven, the same query language used for specifying VSFs is used for specifying locks. A lock can be applied to any collection of fine-grained program artifacts specified by a query. The locked code collection need not correspond to any existing VSF, but can cross all of the stored program organizations in any way that the programmer desires.

There are two significant advantages to this approach. First, the programmer can lock *exactly* the set of artifacts associated with a change. Instead of locking an entire source file (as in most file-based SCM systems), or locking individual fine-grained artifacts (as in most fine-grained SCM systems), the programmer can precisely specify the scope of the change, and lock exactly the relevant artifacts.

Second, this mechanism enhances the understanding of other programmers. While the change is in process, a programmer who tries to create a conflicting lock will be informed of the conflict, and will see the description of the conflicting change. If they want to break the lock, they are free to do so, but the programmer holding the lock will be notified of the break, and the description of the change that required the break.

## 2.3 Consistent Project Versioning

Consistent project versioning (CPV) is an essential feature for any modern SCM. By CPV, we mean that the SCM system provides some mechanism of managing change by recording changes that cross organizational units as atomic units. This can be done by a variety of mechanisms, most prominently including change packages (as used by many systems such as ClearCase[9], Adele[5, 2] or Infuse[8]), or project oriented versioning (as used by systems like PRCS[6]).

Like the other requirements that we have discussed, consistent versioning requires crossing standard organizational dimension, and is absolutely crucial for much of the advanced functionality of current SCM systems. Unlike the other features proposed here, this is already widely implemented: some form of it is present in *every* modern SCM system in use today. However useful it is in its current form, this feature is limited by the fact that it crosses the dominant decomposition of the system, without acknowledging the fact that the organizational dimension of the system is made up of non-atomic units that can be decomposed into small semantically meaningful pieces.

A consistent change across the system is recorded in terms of deltas applied to *source files*. The fact that a particular change affects only a small portion of given source file is recorded by the system, but the fact that it affects precisely one semantic entity in the source file is lost, because the system cannot recognize the semantic entities that make up the source file.

In Coven, we perform consistent project versioning using a hierarchical project model based on the project-versioning model of systems like PRCS. Each time a programmer makes a change that spans a collection of artifacts, the system creates new versions of the changed artifacts, and then records a new version of the system containing the updated artifacts. With this mechanism, the system has a record of exactly what artifact versions coexisted with one another at any given time. This addresses one of the key problems with the change package model: under change packages, it can be difficult to identify whether a particular version extracted from the system corresponds to a prior state of the system.

### 3 Related Work

The idea of multidimensionality used by Coven originated in the separation of concerns community[12], and the ideas were developed in discussions with IBM's HyperSpaces[7] team. This notion is closely related to the dynamic view mechanisms of the Gwydion Sheets environment[11], and Desert[10], and the dynamic hierarchical configurations in ICE[13], Adele, and Infuse[8].

The versioning model used by Coven is based on a variant of the project-based versioning model proposed by PRCS[6], with a consistency model based on the feature logic of ICE[13]. Using fine-grained versioning as a mechanism for inter-programmer coordination was explored by the COOP/Orm project[1]. COOP/Orm was a collaborative programming environment, based on extremely tight collaboration between programmers, and used an even finer grained form of storage and versioning than Coven. However, it still enforced a single organizational view of the system.

### 4 Conclusions

In this paper, I have described how multidimensional software configuration management provides more powerful support for software development than conventional SCM systems. It does this by separating the distinct concerns of program storage, program organization, and inter-programmer coordination within the SCM system. This separation enables a variety of features, including dynamic multidimensional program organization through virtual source files, lock based coordination of appropriate program artifacts, and enhanced communication between programming teams.

### References

- [1] B. Magnusson and U. Asklund. Fine grained version control of configurations in COOP/Orm. In *ICSE '96 SCM-6 Workshop*, pages 31–48, 1996.
- [2] 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.
- [3] M. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Proceedings of FSE 2000*, 2000.
- [4] M. C. Chu-Carroll. Efficiently retrieving artifacts in a fine-grained scm repository. Submitted to FSE 2001.

- [5] J. Estublier and R. Casallas. *Configuration Management*, chapter The Adele Configuration Manager. Wiley and Sons, Ltd., 1994.
- [6] J. MacDonald, P. Hilfinger, and L. Semanzato. Prcs: the project revision control system. In *Proceedings of SCM 8*, pages 33–45. Springer Verlag, 1998.
- [7] H. Ossher and P. Tarr. Multi-dimensional Separation of Concerns and the Hyperspace Approach. In *Proceedings of the Symposium on Software Architectures and Component Technology*. Kluwer, 2000.
- [8] 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.
- [9] Rational ClearCase. Pamphlet at "www.rational.com", 2000.
- [10] S. Reiss. Simplifying data integration: the design of the Desert software development environment. In *Proceedings of ICSE 18*, pages 398–407, 1996.
- [11] R. Stockton and N. Kramer. The Sheets hypercode editor. Technical Report 0820, CMU Department of Computer Science, 1997.
- [12] 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.
- [13] A. Zeller. Smooth operations with square operators: the version set model in ICE. In *ICSE '96 SCM-6 Workshop*, pages 8–30, 1996.