

Container-Based SCM and Inter-File Branching

A Promising Model Meets A Capable Technology

Laura Wingerd
Perforce Software, Inc.
2320 Blanding Avenue,
Alameda, CA 94501 USA
laura@perforce.com

April 2003

Abstract

Container-based software configuration management is a process model designed to handle complex component packaging variants. Inter-file Branching is a file variant representation method designed to handle complex relationships between branched files. Certain characteristics and effects of Inter-file Branching are not only analogous to but ideal for container-based software configuration management. This paper explores the requirements of container-based software configuration management and the features of Inter-file Branching that complement them.

Overview

This paper has three parts. The first introduces container-based software configuration management at a very high level. It makes the point that for the development of large systems, commercial software systems in particular, the value of a software configuration management (SCM) methodology lies in its ability to reduce complexity. To illustrate that, this paper describes a hypothetical implementation of a container-based SCM process -- necessarily simplified, of course.

The second part of this paper describes Inter-file Branching and compares it to traditional *intra*-file branching mechanisms. If you've had hands-on experience using file-based SCM tools -- like CVS, for example -- you will find this comparison enlightening.

But what do the underlying branching mechanics of an SCM tool have to do with implementing an SCM methodology? Wouldn't any sophisticated SCM tool be a suitable candidate to support a container-based process? The third part of this paper answers those questions by showing how the characteristics of Inter-file Branching inherently support the requirements of container-based SCM.

What Is Container-Based SCM?

In very large, very complex software engineering endeavors, the single-product paradigm of software configuration management falls short. What if the product consists of hundreds of components? What if it is distributed in a dozen -- or a hundred -- different configurations? To manage such complexity, the *container-based* software configuration management approach can be useful. A bit like object oriented programming, or component-based development, container-based SCM views any end product as an assemblage of components, each of which evolves independently of the other.

For example, let's say we're taking our children on a trip. We pack their toothbrushes, their clothes, their toys, and their snacks. It's up to us to anticipate our children's needs and to make sure we have all their necessities in the car before taking off. Compare this to organizing a trip for adults. We know where we're going, we know who's signed up to participate, and we've hired a bus. We give each participant a description of the trip and tell them how much luggage each of them is allowed. When the participants show up on the day of the trip, we don't count toothbrushes and we don't check to see if they remembered to bring spare batteries for their cameras. Instead, we simply make sure all our participants are on board with their luggage.

Container-based SCM is like the bus trip. In fact, it is like a holiday tour company that runs hundreds of bus trips for thousands of participants. On a well-run holiday tour, participants pack their own things and board the right buses; buses leave with the right participants and luggage on board. By passing the responsibilities of packing and boarding on to each of the participants, we have made it easier to embark on a tour with 45 adults than with five children.

With container-based SCM, increasing the size of a system need not complicate the task of managing its evolution. As elements that comprise the system pass from one stage to another they are "packed" so that only the containers, not their contents, need tracking. This is not to say that individual elements are not available to downstream delivery stages. Just as holiday tour participants can get to the things in their luggage at any stop during the tour, components in a container-based system can refer to elemental objects at any stage of their development. But in the same way that the concierge of a hotel along the route of a tour need not be concerned with exactly which toothbrushes are arriving that night, the processes managing a stage of software development need not be concerned with the individual elements involved at that stage.

Components and Streams

In the SCM world, files are the smallest elements that can be identified, accessed and modified individually. Files evolve; each change to a file creates a new version that supercedes an old one, and each file has a known state at any point in its evolution. Individual file versions can be inspected, compared, and labeled. A file can be branched at a particular version and a variant of it can evolve from that version. In the evolution of a large software system the number of files and revisions involved can

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content
Management

climb into the millions. It's easy to see that, unless files can be packed into containers, the relationships between them will be too complex to manage.

Container-based SCM builds on the familiar notion of grouping files by their in-use relationship to each other and their relationship to a production phase. Every vendor, consultant, and SCM pundit has favorite names for these groupings, but in this paper they'll be referred to as *components* and *streams*.

Components are files grouped together because they contribute to a common function. For example, the files compiled and linked together to make an executable program can be grouped into a single component. The files that constitute a program's user manual can be grouped into another. Components can contain other components. An executable program component and a user manual component, for example, can be combined into a component of files that belong together as a product. Because components can form hierarchies, and because programs that operate on files understand directory hierarchies, directory hierarchies make very good containers for components.

To be useful, a component has to be more than a collection of files in a directory hierarchy. It has to be an SCM object in its own right, with a history and a known state at any point in that history. Like a file, a component may be inspected, compared, labeled, and branched into variants that evolve independently. Furthermore, grouping files into components must be a reversible process. In other words, inspection of a component should yield individual files with their identities, histories, and relationships intact.

While a component identifies files grouped by their relationship to each other during use, a *stream* identifies components passing through the same stage of evolution together. Streams can be designated for development, testing, system integration, porting, and release packaging, among other things. Components are either created in or delivered to streams. The composition, configuration, and packaging of components can change as they progress from stream to stream. For example, a group of components might start out in a development stream. Some, but not all, of those components might be delivered to a system integration stream where they are combined with components from another development stream. The combined component might then be delivered to a testing stream.

Streams are not hierarchical -- they don't contain other streams. Streams *are* like components, however, in that they have a history, they can be labeled, they can be compared, and they can be inspected to yield individual components with their identities, histories, and relationships intact.

Streams and components are not the subject of developer activities, however. Developers check in changes to files. Therefore, as files are versioned, so are components and streams. A component's version is a reference to the file versions it contains; a stream's version is a reference to the component versions it contains. Creating a new version of a file creates a new version of the component that contains it, of that component's parent components, and of the stream that contains them all.

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content
Management

A Container-Based SCM Scenario

How does container-based SCM reduce complexity? Consider the case of LHC (Large Hypothetical Corporation), provider of a vast array of large software systems. They develop sector-specific products for their Fortune 500 customers around the world. One of those products is a financial planning application called "LFin." LHC's LFin customers are financial institutions in the US and Europe. Each LFin package sold is customized not only for the language and currency of the locale in which it is used, but also with the corporate graphics of the customer using it. The current schedule requires that LFin be packaged for two customers, MonoBanque (language: French; currency: euro) and BigBank (language: English; currency: US dollar).

LFin consists of, among other things, three major components: a web application server (AS), a user interface (UI), and an online help system (OH). Each of those components is developed by a separate LHC division; each is a significant software system in its own right, built from source files and delivered in the form of executable files, configuration files, installer programs, test suites, and internal documentation. The AS component is the same for all LFin configurations, while the OH and UI components are customized per specification. The online help text, for example, must be written in the appropriate language and make reference to the appropriate currency. The UI must do those things, and it must display the customer's corporate colors and logo as well.

All three of these components, AS, OH, and UI, are built from subcomponents and precursor components designed to support many LHC products in addition to LFin. The LFin UI, for example, is based on a general-purpose windowing application framework (WAF). The UI developers are not involved with the development of WAF; they receive completed, released WAF components from another LHC division. They "LFin-ize" it by packaging it with modules that implement the LFin logic, then localize it (configure it with language- and currency-specific modules) and customize it (package it with customer-specific graphics and color schemes) to produce the final LFin UI.

Thus, by the time LFin is ready for release packaging, the components -- and the files -- that make up the LFin UI have already been through several stages of development, each of which takes place in a stream. The UI division, for example, pick up a WAF version from the WAF release stream and put it into their development stream. In the development stream they work on files, generate new components, reconfigure components, and do whatever it takes to produce the LFin UI. When they have a version of the LFin UI ready to deliver, they put it in *their* release stream.

At LHC it is the release manager who is responsible for collecting, packaging, and testing the software that will be delivered to customers. He collects components from other divisions' release streams and places them into a stream under his own control for packaging and testing. The problems he finds are usually in the realm of mismatched configurations and unmet specifications; these are reported back to the supplying divisions. Instead of attempting to fix problems by modifying files in the release manager's stream, the supplying divisions deliver new versions of their components into their own release streams, replacing previously delivered versions.

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content
Management

The release manager takes delivery of the updated components and reassembles the product packages. Thus, supplying divisions can fix problems in their components without having to know anything about where the release manager puts component files that are assembled into products. Likewise, the release manager can reassemble product packages after defects are fixed without having to know anything about where or how files were changed in the contributing components.

As a process model, container-based SCM is more than just versioning, branching, and releasing product. It reaches into distribution, assembly, and reuse to accommodate two realities of very large-scale software development. One is that a software product may be built and distributed in a myriad of configurations. The other is that the components that make up an end product evolve independently, as if they were products in their own right -- each emerges from its own development organization, each has functionality it provides and expects, and each can be built in many configurations.

What Is Inter-file Branching?

Inter-file Branching is a method used to track modifications to files. In particular, it keeps track of what happens when files are branched into variants and modified in parallel with their originals. Its name comes from the fact that it creates branched files as peers instead of versions of their progenitors, and tracks branching and merging as events that occur between files instead of between versions of a file.

Here's a very simple example. A file called `/www/index.html` is created, modified twice, then branched to a file called `/dev/index.html`. That file is modified once, then branched into a file called `/new/index.html`. Later, the first version of the original file is branched into a file called `/www/test.html`. The resulting inventory of file revisions, in the order in which they were created, is:

```
/www/index.html 1 (new file)
/www/index.html 2 (modification of /www/index.html 1)
/www/index.html 3 (modification of /www/index.html 2)
/dev/index.html 1 (same as /www/index.html 3)
/dev/index.html 2 (modification of /dev/index.html 1)
/new/index.html 1 (same as /dev/index.html 2)
/www/test.html 1 (same as /www/index.html 1)
```

Notice that the original file is now four files, each with its own revision history, and each with the potential to evolve independently. What distinguishes one file from another is its full name (its path name plus its file name). The names and paths of branched files can be as arbitrarily chosen as those of new files.

To date, the only SCM system that uses Inter-file Branching is Perforce. Most SCM systems use the traditional *intra*-file branching method. In the traditional method, branching a file results in a version of the file, just as modifying a file results in a version of a file. The exact sequence of steps in the simple Inter-file Branching example above can't be applied to the traditional branching method because it

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content Management

involves creating new files. However, an analogous set of file revisions can be created with the traditional branching method using branch labels or tags in lieu of new file names. The result will be something like:

```
/www/index.html 1.1      (new file)
/www/index.html 1.2      (modification of 1.1)
/www/index.html 1.3      (modification of 1.2)
/www/index.html 1.3.1.1  (tagged 'dev', same content as 1.3)
/www/index.html 1.3.1.2  (tagged 'dev', modification of 1.3.1.1)
/www/index.html 1.3.1.2.1.1 (tagged 'new', same content as 1.3.1.2)
/www/index.html 1.1.1.1  (tagged 'test', same content as 1.1)
```

As this illustrates, the traditional method of branching can only create new versions of existing files. With Inter-file Branching, by contrast, files can be branched from one name to another and from one directory to another.

Lineage, Revision Identifiers, and Integration History

With the traditional branching method, each file's revision identifier shows its lineage. (Every SCM system has its own scheme of identifying revisions. The example above uses a hybrid of the schemes used by RCS, CVS, and ClearCase.) With Inter-file Branching, on the other hand, the lineage of any given file revision is not evident from its path, name, or revision number.

Also, the traditional branching method allows content to be merged only between revisions of the same file. Inter-file Branching allows merging and copying between files. In the same way that content can be branched from one file to another, it can be merged or copied from one file to another. Furthermore, every time a file is branched, merged, or copied to another file, that event is recorded in the *integration history* of the originating and target files. Thus, while a file's lineage is not evident from its revision identifier, it *is* evident from the integration history associated with that revision.

Aggregate Path History

A side effect of Inter-file Branching is that the aggregate history of a path -- that is, the combined history of the files in a path -- expresses the version history of the path itself. For example, consider the history of the files in the `/main/xyz/` path, shown here:

	A	B	C	D

/main/xyz/foo.c	1	2	3	4
/main/xyz/bar.c	1		2	
/main/xyz/ola.c				1

This example shows three files and four events that affected them. (The events are lettered A, B, C, and D here but they could just as well be dates or other identifiers.)

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content Management

They represent a sequence of events over time.) In Event A, files /main/xyz/foo.c and /main/xyz/bar.c were added, creating the first revision of each. In Event B, /main/xyz/foo.c was modified, creating its second revision. And so forth. Note that at each of those events, the state of the files in the /main/xyz/ path can be identified, and that, outside of those events, there is no state change. Thus the /main/xyz/ path itself has four discrete states, or versions: A, B, C and D.

The version history of a path at any level in the repository can be determined in the same way. Consider this example:

	A	B	C	D	E

/main/abc/f1.c		1		2	
/main/abc/f2.c		1		2	
/main/abc/f3.c		1		2	
/main/xyz/foo.c	1		2	3	4
/main/xyz/bar.c	1		2	3	
/main/xyz/ola.c					1

Here, there are two versions of /main/abc/ (B and D), four versions of /main/xyz/ (A, C, D, and E), and five versions of the /main/ path (A, B, C, D, and E).

Applying Inter-file Branching

As the preceding information suggests, many of the requirements of container-based SCM are complemented by the characteristics of Inter-file Branching. To begin with, components and streams can be modeled simply by using the hierarchical directory structure of the file repository. Paths at a particular level can be designated as containers for streams, and paths below that can be designated as containers for component hierarchies. Delivery of components from stream to stream is simply a matter of branching and merging from one path to another; the underlying system keeps track of the origins and integration histories of individual files.

The simple LHC scenario illustrates this. The UI division at LHC have designated /UI/LFin/ as their LFin release stream container. They'll deliver the customized LFin UI components they have built into that stream, one component container for each customer, resulting in:

```
/UI/LFin/MonoBanque/...  
/UI/LFin/BigBank/...
```

(The '...' notation means 'files in this path.')

The LHC release manager takes delivery of the UI components into his stream. He does that by branching files thus:

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content
Management

```
/UI/LFin/MonoBanque/... -> /RM/LFin/MonoBanque/UI/...  
/UI/LFin/BigBank/... -> /RM/LFin/BigBank/UI/...
```

(Note that, by doing this, he is taking delivery of the latest versions of files in the UI release stream. He could branch labeled or dated versions of files in that stream, if, for some reason, the tip revisions were to be avoided. But because the UI/LFin/ stream is dedicated to delivery of UI components, he has no reason not to take the latest versions.)

The release manager also takes delivery of components from the OH and AS divisions, filling out the complement of LFin components in his stream:

```
/RM/LFin/MonoBanque/UI/...  
/RM/LFin/MonoBanque/AS/...  
/RM/LFin/MonoBanque/OH/...  
/RM/LFin/BigBank/UI/...  
/RM/LFin/BigBank/AS/...  
/RM/LFin/BigBank/OH/...
```

Now, his /RM/LFin/ stream is populated with files he can use to assemble the final product packages. Every one of those files has a history that can be traced back to its supplying division.

Problems detected during product packaging can be tracked to their likely origins by auditing component histories and comparing components between streams. For example, suppose a UI problem in the BigBank LFin product package has been detected. The first step in tracking it down is to see where the LFin UI for BigBank came from. The aggregate history of the /RM/LFin/BigBank/UI/ path shows that it contains only one version, the version that was branched from /UI/LFin/BigBank/. However, inspection shows that the latter component has been changed twice, indicating that two new deliveries of the LFin UI component for BigBank were made subsequently.

(A deeper investigation of revision histories would show that the culprit was actually a component in the WAF stream that was used to build the LFin UI. However, the LHC release manager is only interested in completing his product packaging on time and has neither the need nor the inclination to know which files were modified to correct the problem.)

The release manager can take delivery of the new UI component by merging it into his stream. This will update the /RM/LFin/BigBank/UI/ component with newer files. Note that the release manager does not have to specify which files need updating; individual file merges are controlled by their integration histories. In the release manager's view, the UI component as a whole is being refreshed. Once that is done, a new LFin product package can be assembled for BigBank.

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content
Management

Real-World Complexities

The LHC scenario is a simplification that overlooks many complexities of the real world. It's almost impossible to provide readable examples that take such complexities into account. Were it possible, the examples above would have demonstrated:

- Release package numbering. LHC is surely going to distribute more than one release of LFin to its customers. As long as configuration and assembly requirements remain constant, the LHC release manager could use the same streams for a series of releases. Components delivered to those streams would be labeled according to release number (RM-LFin-Pkg-1.0, RM-LFin-Pkg-2.0, and so forth). But if the releases diverged significantly, or if LHC has a policy of supporting more than one release at a time, each generation of releases would require its own stream.
- Changes to component composition. Software development in the real world involves adding, deleting, renaming, and moving files. In other words, the composition of components is likely to change as components progress through a development phase. The version history of streams in the real world is likely to show significant modifications to component structures.
- Intermediate assembly and testing; ancillary files. In the real world there are many phases between component delivery and packaging. It's a very simple software configuration process that doesn't involve build and test steps for every component delivery. Those steps involve layers of tools, scripts, "glue code," test harnesses, build logs, test logs, and so forth, all of which have their own sources and development cycles.
- Target platform variants. For example, what if LHC's customers do not all use the same computer system hardware? The LFin components delivered to the release manager would have to include subcomponents for each target platform.

Implementation Guidelines

A container-based SCM system implemented with Inter-file Branching is likely to be successful if these guidelines are followed:

- A uniform naming convention for streams, components, labels, and other configuration objects should be established at the outset. Inter-file Branching does not impose any naming conventions of its own.
- SCM operations -- building, testing, labeling, delivery, and so on -- should target streams and components. They should not operate on individual files or file collections not defined as streams or components.
- Configuration and assembly streams should be used as conveyor belts, not as workbenches for upstream developers. Component versions that don't pass muster should be replaced, not fixed right in the stream. Software components

1st BCS CMSG Conference 2003
Implementing CM Everywhere, Change, Configuration & Content Management

should be fixed by their suppliers and handed off in their entirety as newer versions.

- If labels are used, they should be used to identify component and stream *versions*; they should not be used as containers themselves. In other words, if components constitute a release, they should be delivered into the same release stream instead of left in separate streams to be marked by the same release label.

Conclusion

Container-based SCM can simplify the task of managing file variants in large-scale software development. Inter-file Branching has features that complement the requirements of container-based SCM:

- Container-based SCM organizes files into a hierarchy of components and streams. Inter-file Branching organizes files into a hierarchy of repository paths that can be designated for components and streams.
- Container-based SCM must be able to branch components from stream to stream and track their lineage. Inter-file Branching allows hierarchies of files to be branched from one path to another and can determine the lineage of any collection of files.
- Subcomponents, components, and streams must have a history of discrete versions. References to those versions must identify versions of the files within them. Inter-file Branching can aggregate the history of files in a path to yield the version history of that path. A version of a path is also a reference to the versions of the files in the path.

Because of these correlations, Inter-file Branching is an effective technology for implementation of a container-based SCM process.

Bibliography

Karl Fogel and Moshe Bar. *Open Source Development with CVS, 2nd Edition*. Coriolis; 2001.

Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000.

E. J. Reek and H. Thelosen. "Introduction to Container Based Software Configuration Management." [QA Systems](http://www.qa-systems.com/welcome.html), <http://www.qa-systems.com/welcome.html> 31 July 2002.

Christopher Seiwald. "Inter-File Branching: A Practical Method for Representing Variants." Sixth International Workshop on Software Configuration Management (I-SCM6), Berlin, Germany, March 1996; in *Software Configuration Management: Selected Papers of the ICSE SCM-6 Workshop*, edited by Ian Somerville. Springer-Verlag, 1996.

1st BCS CMSG Conference 2003
**Implementing CM Everywhere, Change, Configuration & Content
Management**

Darcy Wiborg Weber. "Requirements for an SCM Architecture to Enable Component-Based Development." Telelogic Technologies North America (formerly Continuous Software Corporation), 24 February 2001.

Brian A. White. *Software Configuration Management Strategies and Rational ClearCase*. Addison-Wesley, 2000.