Multi-Grain Version Control in the Historian System

Makram Abu-Shakra and Gene L. Fisher
Department of Computer Science
California Polytechnic State University
San Luis Obispo, CA 93407
{mabushak, gfisher}@csc.calpoly.edu
http://www.csc.calpoly.edu

Abstract. This paper describes Historian, a version control system that supports comprehensive versioning and features to aid history navigation. Comprehensive versioning is supported through frequent and automated creation of versions which typically results in a large number of versions. To reduce user overhead in history navigation, the hierarchical structure present in most documents is utilized to support fine-grained version control. The series of document editing operations is also organized hierarchically and can be used for navigation as well.

1 Introduction

This paper describes the Historian editor-based version control system. Historian supports fine-grain version control of documents, with an end-user interface that provides efficient and convenient version retrieval. A document is a collection of structured text, where the structuring is defined on a document-specific basis. For example, a prose document is structured as paragraphs within sections within chapters; a program is structured as declarations within functions within files.

A number of existing systems provide features similar to those available in Historian, notably COOP/Orm [10] and Orwell [19]. The contributions of Historian compared to these systems are:

a. support for comprehensive version control at multiple levels of granularity along both structural and temporal lines
b. a graphical user interface for convenient historical navigation and low-overhead version retrieval

Structural granularity is based on the natural hierarchical organization of a document. For example, versioned items in an object-oriented program range from functions, to classes, to modules, as described well in [7]. Temporal granularity is based on the frequency of version creation during document editing. For example, the frequency of version creation can range from saving at each individual edit operation, at the document save operation, or at each document check-in operation.
The support of different levels of granularity for both types provide unique advantages to the software configuration process. While our approach focuses on the effects of granularity on viewing the evolution of documents, existing systems have explored its effects on other areas such as merging alternatives and group awareness.

The focus of Historian is notably more narrow than many full-feature configuration management systems. From the end user's perspective, Historian appears as a largely standard text editor, with specialized features for version navigation and retrieval. In this sense, the end-user look and feel of Historian are much like groupware editors such as PREP [13,14] and SEPIA [5] Historian can be aptly viewed as a specialized component that can be integrated into a larger CM environment.

The term fine-grain version control appears first to have been used by the authors of Orwell [19] to refer to the version control of individual program methods (i.e., functions) as opposed to files or modules in an object-oriented program. This is fine-grain on the structural level. In COOP/Orm [11] the term is extended to mean frequent creation of versions as well. This is fine-grain on the temporal level. While a number of other good systems have addressed fine-grain version control on one or both of these levels, none has focused on convenient and low-overhead retrieval, which we view as an important issue. Historian provides version control that is comprehensive, fully automatic, efficient, and supports focused retrieval.

The motivation for Historian has come from our own frustration as authors when changes to a document eliminate versions that we might later wish to retrieve. Such frustration arises when working alone, and is amplified when working collaboratively with other authors, as reported in [3] for example. While version control systems have always addressed this problem to some extent, systems typically keep track of changes only when files are checked in, or at other explicitly invoked user operations. Comprehensive and fully automatic version control introduces the security that any version of a document is retrievable, without the user having to request version saving explicitly.

There are two major problems with comprehensive version control: excessive use of storage space and the high overhead of navigating through a large number of versions. Solutions to storage problems such as file compression and differencing algorithms are well known and are utilized in Historian to keep memory usage at a minimum. User interface solutions to the latter problem are largely unaddressed in CM systems, and the primary focus of Historian. In order to keep comprehensive version navigation practical, features are introduced to aid the user in searching for desired versions. In our experience, version navigation in most CM systems is already awkward and time-consuming; saving a comprehensive set of versions makes matters even worse.

We began our research with the assumption that reducing user overhead in navigation will make version retrieval a more common practice in software development and will thus increase productivity. In our experience using RCS [20] or RCS-based systems, we have found that saved versions were very rarely
retrieved, even when they might be quite useful in undoing a design or implementation change that did not succeed as expected. That version retrieval becomes more common is fundamental to realizing the importance of reducing version retrieval overhead. In our experience using the current Historian prototype, we have observed that different versions have been retrieved as often several times per hour. This high frequency of retrieval typically occurs during the later development stages of programs. It is at these stages when history is most valuable because changes are often made to "code that already works" and is therefore more likely to be retrieved.

In addition to the quantitative increase in version retrieval, we have observed a number of qualitative improvements in program development with the use of Historian. The user gains a sense of security because no data are ever lost and a certain sense of bravery when modifying code. In addition, the multiple undo feature found in text editors is effectively replaced by Historian's features for very fine-grain version control. We will further elaborate on these effects later in the paper.

In the discussion of Historian, we will distinguish between two types of overhead: system-time and user-time. User overhead is what we consider being the amount of time spent in trying to find a specific version in mind. Version retrieval can be categorized into two fundamental types. The first type is the retrieval of a version whose id is already known, i.e., the version of a system release. This type typically involves little user overhead. The second type involves retrieving a version whose id is not known, when the user must search through history to find the version in mind. Historian aims to reduce the overhead of the latter type. Henceforth in this paper, we assume that the identifier is not known.

2 Related Work

Of relevance to Historian, there are four broad categories of tools that have been used to support document version control:

1. First generation tools developed primarily for software engineering, such as SCCS [16] and RCS [20]
2. Second generation configuration management tools that focus on fine-grain version control, such as COOP/Orm [10,11] Orwell [19] POEM [7] and CAME-ERA [8]
3. Conventional text editors with certain features that support version control, notably Emacs [18] and Microsoft Word [12]

Discussion of the relationship between Historian and these tools follows.

2.1 First and Second Generation CM Tools

RCS and SCCS are very well known in the CM community, having been the basis of use and/or comparison for very many later CM systems. The pros and cons of
first-generation systems have been discussed extensively elsewhere, notably by Conradi and Westfechtel in [1] and [2] RCS is of direct relevance to Historian since we use the RCS engine in the Historian implementation, although the end user is completely unaware of this.

The second generation CM system with features most comparable to Historian is COOP/Orm. Multiple granularity is supported for both structural and temporal levels. While their approach does not focus on version navigation, granularity levels are utilized for the purposes of group awareness and unifying the approaches to version control of documents and configurations. Group awareness is supported at a temporally fine level by making visible incremental document changes to all users concurrently editing the document. Awareness is also supported at a coarse level.

Multiple structural granularity is supported in both COOP/Orm and Historian because both fine and coarse granularity have disadvantages when supported without the other. For example, if only fine granularity were supported, configurations would result in a large number of version combinations, though finer grained versioned items make more sense from the version control point of view because they limit the effects of changes.

The representations of fine-grain versioning used in the CAMERA and POEM systems are comparable to Historian, however the focus of both systems is different. In the case of CAMERA, the system is designed to support powerful techniques for merging different document alternatives. The major focus of POEM is the unification of the logical representations used in normal program development with the representations used for configuration management. Neither focuses on version navigation as in Historian.

2.2 Standard and Collaborative Text Editing Systems

Features to support version control are found in some conventional text editors. These features can be grouped into the following general categories:

1. automatic version saving
2. version comparison
3. undo/redo commands

As representatives of "conventional" editors, consider Emacs and Microsoft Word. Emacs provides all three of these features. Word supports the second and third, with partial support for the first.

What is common to Historian and conventional editors is the concept of a basic action. While the precise definitions of basic action differ between the editors, the concept is fundamental. The editor defines what constitutes a primitive editing action, representing the smallest level of granularity at which changes are recorded. Historian automatically archives changes at this level, whereas conventional editors only support non-archived undo/redo at this level.

Among systems designed specifically for collaborative work, MILO and DIVA provide limited forms of version control. MILO provides time-ordered views of a document, based on time-stamped notes attached to document elements. The
MILO user can traverse the note history, and filter the history based on the contents of notes. The DIVA system provides a catch-up facility that allows users to view document changes from an historical record. The user interface includes animated replay of document changes.

Historian provides a considerably larger degree of support for version control than in MILO or DIVA. In particular, Historian versions are not limited to time-stamped notes or other document attachments.

The Quilt and SEPIA systems focus on version control specifically. Version control in Quilt is less comprehensive, and requires users to define cooperative roles to enable versioning. The most recent SEPIA system [5] does provide comprehensive version control. In comparison to SEPIA, Historian is significantly more focused on the domain of textual documents. SEPIA is a full-scale hypertext system, in which version control is supported at the level of hypertext objects and tasks. While focused retrieval is supported at this level, it is less specifically focused than in Historian, where users concentrate on the textual content of a document, not on more general hypertext objects.

The existing system most closely related to Historian is the PREP editor [15,13,14] PREP allows users to focus on version changes at conventional levels of document organization, such as sentence and paragraph. It also supports flexible comparison between arbitrary version pairs. The contributions of Historian vis a vis PREP are fully automatic version archiving, combined with flexible version search and retrieval. PREP's side-by-side comparison interface is more elaborate than in Historian, but both PREP and Historian convey fundamentally the same comparative information. The advantage in Historian is that versions can be compared incrementally by traversing the archived history. In the broader area of cooperative work, the overall scope of the Historian system is considerably smaller than that of PREP. In particular, support for inter-author communication based on tunable parameters is beyond the current focus of the Historian system.

3 Overview of Historian

This section of the paper presents an overview of the Historian model. The interface and functionality described are somewhat beyond the capabilities of the current working prototype. Precise details of prototype limitations are presented in Section 5.

Historian supports multiple structural granularity by recognizing and utilizing the hierarchical structure of documents. In terminology presented in [10] the smallest versioned item of a document is called an information unit. For example, in a C file information units are functions and in a prose document information units are paragraphs. Larger versioned items, called composite units, are composed of zero or more information units. In a prose document chapters and sections are typical composite units; chapters consist of sections and sections consist of paragraphs. The largest composite unit versioned by Historian is the entire file.
From a version navigation standpoint, structural fine-grain support enables focusing a search on a particular section of a document. If the user is only interested in the evolution of a function, retrieving versions at the file level will typically involve retrieving versions that do not show changes in the function of interest. The user must also search for the section of interest in every version retrieved. Consider the following scenario: Functions f1, f2 and f3 exist in a file and the user modifies f1 and checks in the file, modifies f1 again and checks in, then modifies f2 only and checks in, and finally modifies f3 only and checks in. If the user wants to retrieve f1’s earlier version, then traversing back in time at the file level retrieves three versions that do not show a change in f1. By focusing a search on f1 via fine-grain support, these three uninteresting versions are not retrieved.

Historian implements change propagation as defined in COOP/Orm [11]. A change to an information unit also constitutes a change to the composite unit(s) of which it is a part. Thus, all changes are reflected in all the different levels of structural granularity and the user can switch granularity views at any point including in the middle of a search through history. An early version of a function, for example, can be used to bring up the corresponding version of the entire file.

In order to keep track of the hierarchical structure of a document and its modifications, Historian has been implemented as an extension of the XEmacs text editor. Figure 1 shows an example of the Historian user interface.
The window on the left is the regular text editing window used to modify files. The window on the right is the version browser. The text editing window is used to specify the unit whose versions will be retrieved in the version browser. In this example, a C file is being edited and the versions of one of its functions (an information unit) are being browsed.

Versions are generated automatically by Historian as the user modifies the file. In order to support temporally fine-grained version control and to keep a comprehensive set of versions, Historian generates a version at every edit operation. This granularity can be compared to the change granularity typically supported by the multiple undo features in text editors.

Instead of generating only one series of versions for each unit, several series of versions are generated, each series differing significantly in size. Only one series contains a comprehensive set of versions and versions for the other series are generated less frequently. In this manner, Historian supports multiple temporal granularity. By choosing the series through which to traverse, the user can choose between executing a thorough inspection of a unit's history versus executing a quick and skimming search.

The size of a series is controlled by the type of user action that triggers a version save for that series. In our example, four series are generated where each of the following user actions triggers a version save for its corresponding series:

- The Per Edit series: An edit operation.
- The Per Save series: The file is saved.
- The Per Session series: The file is closed.
- The Per Check-In series: The file is checked in.

Note that a version is only saved if the unit has been modified.

The Per Edit series typically contains the largest number of versions since edit operations occur more frequently than file saves, file closes, and file check-ins. Conversely, the Per Check-In series typically contains the least number of versions.

Our example shows the four series in the version browser. For each series the number of the currently displayed version and the total number of versions in the series are shown. Forward and backward arrow buttons are used to sequentially step through the series. One can also retrieve versions by clicking on the version boxes inside the version graph. Versions that are vertically aligned in the version graph represent a single version shared between more than one series.

In our example, the versions of a single function (an information unit) are being browsed. If the versions of the entire file were being browsed instead, the four types of series will still be shown but they will typically be much larger in size since they will contain all of the changes made to the file. This represents a combination of the histories of all the information units of which it is composed.

The number and types of series used for a particular file are configurable by the user. For example, if a temporally fine-grained version control is not desired, only the Per Check-In series need be used.

In addition to the series in our example, other options are available such as the Per Compile (for compilable documents such as C files) and a Marker series.
Versions are generated for the Per Compile series whenever the file is successfully compiled. Traversing this series ensures that every version retrieved is correct to the degree that the compiler ensures correctness.

Versions are generated for the Marker series explicitly by the user to mark specific versions that are deemed important. Adding a version to the Marker series grants quicker access to that version in the future, a form of user-explicit sifting of uninteresting versions. For example, if a programmer is about to make major changes throughout a file, changes which may need to be undone in the future, marking the current file version allows the programmer to make these changes without worry since they can all be undone easily. The Marker series is especially useful for rapid development and exploratory programming.

As with structural granularity, the user can switch temporal granularity at any point during a search. A typical case is one in which the desired version of a unit exists somewhere early in its history and the user wishes to traverse back in time incrementally. This is done using the arrow buttons, starting from the most current version instead of starting at a random point in history. Instead of traversing a series with a comprehensive set of versions, the user would begin with a coarse-grained series such as the Per Check-In series for a skimming traversal. Once the general proximity of the desired version is found, the user would switch to a fine-grained series for a more thorough evolutionary inspection.

4 Technical Details

Historian’s implementation is tightly integrated with the XEmacs text editor [9] in order to support fine-grained version control of both types. The use of XEmacs as the base editor has facilitated the continuous monitoring of changes made to a document. Structurally fine-grain version control requires maintaining the boundary locations of each information unit within a file and the awareness of which units the user modifies. Temporally fine-grain version control requires monitoring the user’s actions and triggering version saves of modified units when appropriate.

Only the versions of information units are physically saved; composite unit versions are created from those of their information units. Due to change propagation, each change in an information unit reflects a change in the composite unit of which it is a part. In order to minimize overhead and memory, new versions are generated for changed information units only, and the current versions of unchanged information units are shared. Specifically, composite units are represented as collections of pointers to information units. Our approach is similar to the revision tree model defined in [11].

Versions are also shared along the different levels of temporal granularity since the same version can exist on more than one series. A version in one series is equal to a version in another series when a version is saved for the second series before any further modification is made to the unit since the version was saved for the first series. In our C example, the three coarser grained series always share all their versions with the finest grained series, the Per Edit series,
since modifications always trigger a new version for that series. However, not all versions of a coarse-grained series share their versions with the finer grained series. In that sense, multiple temporal granularity differs from multiple structural granularity in that the structural type has the property of inclusion between the different levels whereas the temporal type does not.

5 Current Status and Future Work

A working prototype of Historian has recently been developed and only a small number of users have worked extensively with the system. Through this limited experience we have made observations on the effects of using Historian. We have observed a significant increase in the frequency of version retrieval with the use of Historian when compared to other version control systems. Also, the user gains a sense of security from the fact that no data are ever lost and that whatever changes are made, they can be undone quickly. This, in turn, enables the user to modify code that already works yet needs to be changed with less hesitation or worry, supporting rapid development and exploratory programming. Code is seldom commented out for later reference, but rather deleted instead to increase code readability. Finally, the multiple undo feature of XEmacs is effectively replaced by Historian, as discussed earlier.

The current implementation supports a limited number of features of the model discussed in this paper and we plan to continue development. Most importantly, structural coarse-grain support has been designed but not yet fully implemented. Structural fine-grain support and the different levels of temporal granularity are provided. Also, the user interface shown in the example is slightly more sophisticated than the one implemented. Specifically, the version graph is not present in the prototype and the series information is presented in a textual form. Otherwise, the interface of the prototype is as presented in Figure 1. In terms of supported document types, the prototype has been used on C, Lisp, HTML, and UNIX Troff.

A problem with structurally fine-grain version control which needs to be addressed deals with the relationships between units that arise when two units are merged together, one unit is split apart, or a significant portion of one unit is pasted into another. During traversal, the user might want to follow the history unit section that becomes part of the history of another unit as a result of one of these operations. For example, if a large block of code were cut from one function and pasted into another, traversing forward in time through the history of the original function would suddenly indicate the disappearance of that block of code, even though its evolution continues elsewhere. Without special support from the version browser, the user would have to bring up the file version which corresponds to the function version displayed, and search for the new function that contains the desired block of code. Only then could the user resume inspecting the evolution of the block of code. There is still no guarantee the section can be found in the file version because the cut and paste operation could have been executed over a sequence of actions spanning version saves.
While splitting and merging units are relatively rare operations in programming compared to cutting and pasting, splitting and merging paragraphs in prose documents is common. Following the evolution of a paragraph can become quite confusing if proper navigation support is not provided. To the best of our knowledge, this issue has not been addressed by existing systems that implement structurally fine-grain version control, which is understandable because the overhead of version navigation has never been of primary focus. We have a number of user interface ideas each of which solves this problem but we have yet to decide upon which one to implement.

6 Conclusion

This paper has presented the Historian system for multi-grain version control and navigation. Our experiences to date with Historian have been quite positive. We have observed the hoped-for benefits of Historian— that overhead in version retrieval is reduced given multi-grain access to document histories at both structural and temporal levels.

It should be noted that the benefits for experimental programming provided by Historian, in particular the security to make rapid changes to an evolving program, are done in a context that also supports sound software engineering. That is, rapid development is done by an individual during the course of program evolution. When versions are to be shared by colleagues, they are checked in, and become stabilized by the normal mechanisms that lock-out changes to version that have been declared stable.

Our experience with the current Historian prototype is limited to use by its development team and a small number of other software engineers. We plan to conduct a controlled usage study that will provide additional data to help quantify the benefits we have observed in our use of the research prototype.

References