Design, Implementation, and Evaluation of a Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

ABSTRACT

The Revision Control System (RCS) is a software tool that helps in managing multiple revisions of text. RCS automates the storing, retrieval, logging, identification, and merging of revisions, and provides access control. It is useful for text that is revised frequently, for example programs and documentation.

This paper presents the design and implementation of RCS. Both design and implementation are evaluated by contrasting RCS with SCCS, a similar system. SCCS is implemented with forward, merged deltas, while RCS uses reverse, separate deltas. (Deltas are the differences between successive revisions.) It is shown that the latter technique improves run-time efficiency, while requiring almost no extra space.

1. Introduction

Software changes constantly. Any useful, nontrivial software system requires correction, adaptation, extension, and improvement throughout its entire life. The constant modifications result in multiple versions of the system, collectively called a *family of systems*. Belady and Lehman[1] have analyzed the phenomenon of constant change and argued that evolution into a family is practically unavoidable for a software system of sufficient size.

Maintaining and managing a software family can be a nightmare. As the size of the family grows, it becomes more and more difficult to avert chaos and to limit the combinatorial explosion of the changes. A suitable technique is to plan for a family during design and make sure that every family member can be configured from as many common parts as possible. Skillful management, design, and implementation must be combined to maintain order and to keep the family together.

This paper presents the Revision Control System (RCS), a software tool that helps in controlling the evolution of software families. The basic function of RCS is that it manages *revisions* of text. A revision is created by manually changing a document, usually by means of a text editor.* RCS can be used for any type of text, in particular programs and documentation. RCS stores and retrieves multiple revisions of text, logs changes, identifies revisions, merges revisions, and controls access to them. The space overhead of storing multiple revisions is minimized by saving only the differences between successive pairs.

The basic idea of RCS is not new. There are several other systems that have a similar purpose, for example SCCS[2] and SDC. [3] Most of the early revision control systems are limited in that they treat each system part in isolation and do not consider configurations of parts. RCS avoids this limitation, and corrects some other design flaws. RCS is also implemented in a novel way, namely with reverse, separate deltas, which improve its performance considerably.

In Sections 2 and 3 we present design and implementation of RCS. Section 4 contains the evaluation. We compare RCS with SCCS, and describe an experiment demonstrating that reverse, separate deltas (as used in RCS) can lead to better performance than forward, merged deltas (as used in SCCS), while costing almost no extra space. The evaluation should be of value to designers of similar systems.

2. Design of RCS

The user interface of RCS has been tuned for the UNIX programming environment. [4] However, readers not familiar with UNIX should be able to follow the discus-

* We do not use the more general term of ***version*** here, because that would include text that is generated by transformation. For example, object code generated by a compiler from a source program is a type of version that is not handled by RCS.

0270-5257/82/0000/0058$00.75 © 1982 IEEE

sion without problems, since the basic ideas are independent of a particular operating system.

Suppose a programmer, having created a file named *mod* containing a program, wishes to put it under control of RCS. He may plan a series of modifications, from which it would be difficult to recover without a back-up copy. He may also anticipate numerous revisions of the program and want to store them in a space efficient way. He issues the following command:

```
ci mod
```

Ci, short for checkin, deposits revisions into RCS files. RCS files contain multiple revisions of text and are managed by RCS commands. In this example, suppose no RCS file for *mod* exists. *Ci* therefore creates and initializes the file *mod.v* and deposits into it the contents of *mod* as revision 1.1. By convention, all RCS files end in *.v*. *Ci* records the date and time of the deposit as well as the programmer's identification. Since this is the initialization, *ci* also prompts for a short description of the program. The description can be used as part of the program documentation.

When it becomes necessary to change *mod*, the programmer executes the checkout command

```
co mod
```

This command retrieves a copy of the latest revision stored in *mod.v* and places it into the file *mod*. The programmer can now edit, compile, test, and debug *mod*. At all times, he is assured that his old program is still available. When he thinks that his modifications have led to a new revision that is worth saving, he can check it in by executing *ci* again. The command

```
ci mod
```

deposits the contents of *mod* as a new revision into *mod.v*, increments the revision number by one, and records date, time, and programmer id. *Ci* also prompts for a log message summarizing the change. At the time of deposit, the information about the change is still fresh in the programmer's mind, and the prompting is a gentle reminder to supply it. One can later read the complete log of all revisions and figure out what happened to a program without having to compare source code listings.

It is also possible to assign a revision number explicitly, provided it is higher than the previous ones. For example, if all existing revisions are numbered at level 1 (i.e., if they have numbers of the form 1.1, 1.2, etc.), then the command

```
ci -r2 mod
```

starts numbering at level 2 and assigns 2.1 to the new revision.** Correspondingly, *co* can be instructed to retrieve revisions by number. The command

```
co -r2.4 mod
```

retrieves the latest revision with a number between 2.1 and 2.4. Thus, revision numbers in the *co* command are actually cutoff numbers. Similarly, revisions can be retrieved by cutoff date. The command

```
co -d2/19 mod
```

retrieves the latest revision that was checked in on or before 00:00:00 o'clock on Feb. 19 of the current year. *Co* accepts dates in free format and defaults omitted portions intelligently.

It is also possible to retrieve revisions by author and state. The state indicates the status of a revision. By default, the state is set to ***experimental*** at checkin time. A revision may be promoted to the status ***stable*** or *released* by changing its state attribute. *Co* retrieves revisions according to any combination of revision number, date, author, and state.

RCS employs locking to prevent two or more persons from depositing competing changes to the same revision. Suppose two programmers check out revision 2.4 and modify it. Programmer A deposits his revision first, and programmer B somewhat later. Unfortunately, programmer B knows nothing about A's changes, so the effect is that A's changes are "undone" by B's deposit. A's changes are not lost since all revisions are saved, but they are confined to a single revision.

RCS prevents this conflict by locking. In order to check in a new revision, a programmer must have locked the previous one. At most one programmer at a time may lock a particular revision, and only this programmer may append the next revision to it. Locking can be done with both *co* and *ci*. Whenever someone intends to edit a revision (as opposed to reading or compiling it), he should check it out and lock it by using the *-l* parameter on *co*. On subsequent checkin, *ci* checks for the existence of the lock and then removes it. If the programmer wants to check in a revision but wishes to continue modifying it, he can use the *-l* parameter on *ci*, which moves the existing lock to the newly checked-in revision, and suppresses the deletion of his working file. This shortcut saves an extra *co* operation.

** A UNIX command convention is that all parameters that are not filenames are preceded by a '—'.

There is one exception to this rule: The owner of an RCS file does not need to lock. This exception simplifies the commands for RCS files that are the responsibility of a single programmer. In case an RCS file is updated by several people, this exception can be disabled.

The locks described above are long-term locks. They indicate that someone intends to deposit a new revision at a later time. In the meantime, other updates of the RCS file may go ahead. However, every update of an RCS file must occur in a critical section. For instance, a *ci* command must not be executed simultaneously with an operation deleting an old revision (see Section 2.2). Thus, all operations on an RCS file must enter a critical section ensuring mutual exclusion of updates. A *ci* operation that succeeds in entering the critical section may have to abort if it cannot find the required long-term lock for a specific revision.

2.1. The Revision Tree

The above situation of two programmers modifying the same revision may actually signify a branch in the development. If both programmers want their modifications to remain separate, then RCS can be instructed to maintain two revisions with a common ancestor. These two revisions may again be modified several times, giving rise to a tree with two branches. RCS allows the construction of a tree of revisions and provides facilities for joining branches.

An RCS revision tree has a main branch, called the *trunk*, along which the revisions are numbered 1.1, 1.2, ..., 2.1, 2.2, etc. A revision may sprout one or more side branches. Branches are numbered *fork*.1, *fork*.2, ..., etc, where *fork* is the number of the fork revision. Revisions on a branch again have increasing numbers, with the branch number as a prefix. Branch revisions may sprout additional branches. Figure 1 illustrates an example tree with 4 branches (not counting the trunk).

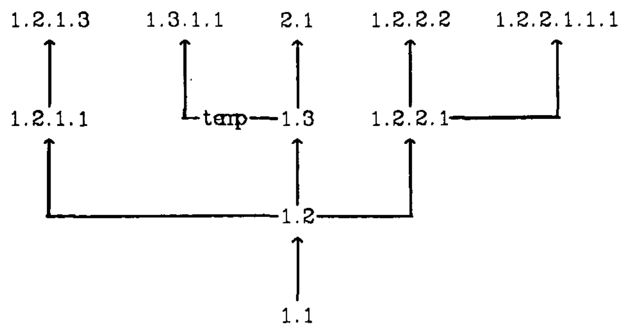

Fig. 1: A revision tree with 4 side branches.

Revisions and branches may be labelled symbolically. For instance, branch 1.3.1 could be labelled *temp*. Revisions on a labelled branch can then be identified using the branch label as a prefix. In our example, revision 1.3.1.1 is the same as *temp.1*. It is also possible to give a symbolic name to an individual revision. This label can then serve as a prefix for branches starting with that revision.

Symbolic labels are mapped to revision numbers and have a variety of uses. For instance, branches can be labelled with the identification of the programmer working on them such that a programmer need not remember "his" branch numbers in several RCS files. Special configuration labels can be assigned to branches or revisions in several RCS files in such a way that a single checkout command can collect the proper revisions for a whole configuration. For example, assume that a system family consists of RCS files *f1.v*, ..., *fn.v*. Assume furthermore that we labelled a specific branch in every file with *configx* if the revisions on this branch belong to configuration *configx*. Then the command

```
co -rconfigx *.v
```

retrieves the latest revisions of all parts that make up *configx*, although the actual revision numbers may be non-uniform. Since several labels may map to the same revision number, sharing of parts among several configurations is possible.

Every revision in the tree consists of the following attributes: a revision number, a checkin date and time, the author's identification, a state, a log message, and the actual text. All these items are determined at the time the revision is checked in. The revision number is either given explicitly in the *ci* command, or it is determined by incrementing the number of the revision that the programmer locked previously. The programmer must hold a lock for the latest revision on a branch if he wants to append to that branch, or he must be the owner of the RCS file and the latest revision on that branch must be unlocked. A new revision is either appended to an existing branch or starts a new branch. Insertion in the middle of branches is not allowed. Starting a new branch does not require a lock.

We discussed the state attribute and the log message already. There is no fixed set of states, but *co* has an option to check out revisions according to their state attributes. The important aspect of the log message is that RCS reminds the programmer to supply the information. Of course, an uncooperative person may answer with an empty message, but his name is recorded anyway.

The bulk of a revision is contained in the text attribute. RCS stores only *deltas*, i.e., differences between revisions. From the user's point of view, the differences are completely transparent; RCS encourages him to think in terms of complete revisions.

There is also some administrative data stored in an RCS file. This data consists of a table mapping symbolic labels to revision numbers, a list of locks (pairs of programmer identifications and revision numbers), and an access list. The access list specifies who may alter the RCS file. If the access list is empty, everybody with normal write permission for the file may change it.

2.2. Auxiliary RCS Commands

There are several auxiliary RCS commands. The command *rlog* displays the log entries and other information about revisions in a variety of formats. The command *rcs* changes RCS file attributes. *Rcs* can be used to shrink and expand the access list, to change the symbolic labels, to reset the state attribute of revisions, and to delete revisions. It also has a facility to lock and unlock revisions, as well as to "force" locks. A programmer forces a lock if he removes a lock held by somebody else. Forcing of locks is sometimes necessary if a programmer forgets to release his locks. *Rcs* allows the forcing, but also sends a mail message to the programmer whose lock was broken.

A special option on the *co* command permits the joining of revisions. Revisions *r1* and *r3* are joined with respect to revision *r2* by applying to a copy of *r3* all changes that transform *r2* into *r1*. For example, joining revision 2.1 and 1.3.1.1 in Fig. 1 with respect to their common ancestor, 1.3, has the effect of retrieving 2.1 and incorporating into it all changes that lead from 1.3 to 1.3.1.1. The resulting revision can be edited or checked back in as a new revision. (Revision 1.3 may actually be omitted; *co* finds the youngest common ancestor automatically.)

Co informs the user whether there is an overlap between the changes from *r2* to *r1* and from *r2* to *r3*. In that case, the resulting revision contains the overlapping sections properly flagged, and the user must edit the file and select the desired sections by hand.

The join operation is completely general in that it may be applied to any triple of revisions. A perhaps less obvious application is if $r1 < r2 < r3$ on the same branch. In this case, joining *r1* and *r3* with respect to *r2* has the effect of undoing in a copy of *r3* all changes that led from *r1* to *r2*. There are also multiple joins, in which the result of one join becomes revision *r1* of the next join.

2.3. Identification

In a system family of moderate size, it is desirable to "stamp" every revision with its number, creation date, author, etc. The stamping provides a means of identification and is done fully automatically by RCS. To obtain a standard identification, the source text should contain the marker *$Header$* in a convenient place, for example in a comment at the beginning of the program. If this revision is later checked out, the marker will be replaced with a character string of the format

```
$Header: RCSfile revision date time author state$
```

where the six fields contain the actual values. Assume a revision checked out with such a header has number 1.2. The programmer may edit this revision and check it back in with number 1.3. He need not update the above stamp, because RCS does this automatically. Whenever revision 1.3 is checked out, *co* searches for markers of the form *$Header: ...$* as well as *$Header$* and replaces them with the proper stamp. Note that the update of the stamp must be done at checkout and not checkin time, because the state of a revision may change over time.

Additional markers like *$Author$*, *$Date$*, *$RCSfile$*, etc., generate portions of the *$Header$* stamp. The marker *Log* has a special function. It accumulates the complete log of a given branch in the revision itself. Whenever *co* finds the markers *Log* or *$Log: ...$*, it inserts the current log message right after it, preceded by the header discussed above. Thus, when a programmer checks out a revision for modification, the whole history is readily available in the source file. For example, revision 1.3.1.1 in Figure 1 could contain the following history.

```
$Log:    checkout.v $
1.3.1.1  82/01/20  20:32:11  wft
started a new branch for PDP-11.

1.3      82/01/05  10:03:23  wft
added option -p to co for printing to stdout

1.2      81/12/20  21:44:23  pjd
added check for multiply defined names

1.1      81/12/01  03:20:12  wft
initial revision
```

Note that if a revision is checked out, the log contains all entries up to (and including) that revision. Since the revisions are actually stored as deltas, each log entry occurs in only one delta. Thus, the space required for accumulating the log is negligible.

The identification technique can also be used to stamp object files. This is done by placing some of the markers discussed above into character strings that are compiled into the object modules. For example in the language C, the declaration

```
char RCSid[] = "$Header$";
```

initializes the array *RCSid* with the standard identification string. This string will appear in the object module after compilation. Another auxiliary RCS command, *ident*, extracts all such strings from a compiled or linked program, as well as from source. With this facility, it becomes simple to determine which revisions and which modules went into a certain software system. This facility is invaluable for program maintenance.

3. Implementation of RCS

To conserve space, RCS stores deltas rather than complete revisions. The grain of change is the line, i.e., if any single character is changed on a line, RCS considers the whole line changed. We chose this approach because UNIX provides the program *diff*, which computes deltas on a line-by-line basis. *Diff* uses hashing and is quite fast, but may occasionally fail to find the minimum difference. In practice, this deficiency causes no problem, since the changes from one revision to the next normally affect only a small fraction of the lines.

An important implementation decision concerns how to store the deltas. One can either merge the deltas or keep them separate. SCCS uses merged deltas, RCS separate deltas. Merged deltas work as follows. Suppose we store the initial revision unchanged and compute the delta for the second revision with *diff*. Assume the delta indicates that a single block of lines was changed. Merging the delta into the initial revision involves marking the original block of lines as excluded from revision 2 and higher, inserting the block of replacement lines (which may be longer, shorter, or empty) right after the first block, and marking the second block as included in revision 2 and higher. Merging additional deltas works analogously, except that excluded and included blocks may overlap. To regenerate a revision, a special program scans through the revision file and extracts all those lines that are marked for inclusion in the desired revision. For a detailed discussion of this technique see Rochkind.[2]

Merged deltas have the property that the time for regeneration is the same for all revisions. The whole revision file must be scanned to collect the desired lines. If all revisions are of approximately the same length, the time for copying the desired lines into the output file is also the same for all revisions. Thus, regeneration time is a function of the number of revisions stored and the average length of each revision. However, there is a high cost involved in merging a new delta. First, the old revision must be regenerated to let *diff* compute the delta. Next, the delta is edited into the revision file. This operation is complicated, because it must consider overlapping changes and branches.

Separate deltas are conceptually simpler and have a significant performance advantage if arranged properly. They work as follows. Suppose we store the initial revision unchanged. For the second revision, *diff* produces an edit script that will generate the second revision from the first. This script is simply appended to the revision file.

An edit script consists of a sequence of editor commands to delete and insert lines. A deletion command specifies the range of lines to delete, whereas an insertion command specifies after which line to start adding lines, followed by the actual lines to insert. On regeneration, the initial revision is extracted into a temporary file, a simple stream editor is invoked, and the edit script is fed into the editor. This operation regenerates the second revision. Later revisions are stored and regenerated analogously.

The above method applies deltas in a forward direction. The initial revision is stored intact and can be extracted quickly, but all other revisions require the editing overhead. Since the initial revision is accessed much less frequently than the newest one, the deltas should actually be applied in the reverse direction. In such an arrangement, the newest revision is stored intact, and deltas are used to regenerate older revisions. RCS uses this idea. Reverse deltas are not harder to implement than forward deltas, since *diff* generates a reverse delta if the order of its arguments is reversed.

The advantage of separate, reverse deltas is that the revision accessed most often can be extracted quickly -- all that is needed is a copy of a portion of the revision file. Regeneration time for the newest revision is merely a function of its length and not of the number of revisions present. Adding a new revision is also faster than with merged deltas. First, generate the latest revision (which is fast) and execute *diff* to produce the reverse delta. Next, concatenate the new revision, the reverse delta for the previous revision, and the remaining deltas. The concatenation is faster than the merging.

The disadvantage of reverse, separate deltas is that the regeneration of old revisions may take longer than with merged deltas. The problem is that the application of n deltas requires n passes over the text. An analysis of the editing cost is given in Section 4.

Branches need special treatment if we use reverse deltas. The naive solution would be to keep complete copies for the ends of all branches, including the trunk. Clearly, this is unacceptable because it requires too much space. The following arrangement solves the problem. The latest revision on the trunk is a complete copy, the deltas on the trunk are reverse deltas, but deltas on side branches are forward deltas. Regenerating a revision on a side branch proceeds as follows. First, copy the latest revision on the trunk; second, apply reverse deltas until the fork revision for the branch is obtained; third, apply forward deltas until the desired revision is reached.

RCS uses this scheme. Figure 2 shows the tree of Figure 1, with each node represented as a triangle whose tip points in the direction of the delta. Note that regenerating a branch revision always incurs some editing overhead. However, if active branches form towards the end of the trunk, only a few deltas need to be applied. Section 4 shows that the performance of applying 2 deltas to the most recent revision is not worse than using merged deltas.

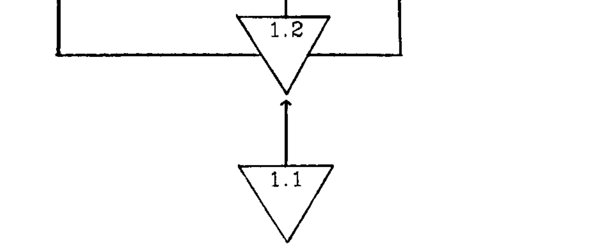

Fig. 2: A revision tree with forward and reverse deltas.

4. Evaluation

In this section, we compare design and performance of RCS and SCCS. We use a version of SCCS that was originally provided by Bell Labs as part of the programmer's workbench[5] and was adapted at UC Berkeley to the VAX-11. Our purpose is not to criticize the developers of SCCS. SCCS has proven to be an enormously useful tool, and the basic idea of keeping a set of differences has withstood the test of time. We merely wish to discuss some annoying shortcomings of SCCS and how future revision control systems should be improved to become even more useful. We also present performance measurements that make the implementation tradeoffs clear.

4.1. Design

A frequent source of errors in SCCS is that all commands require the revision file as a parameter, although the user would rather specify the working file. The revision file contains the revisions and is managed by SCCS; the working file contains a single checked-out revision and is edited by the user. Since the user is focusing his attention on the working file, SCCS should permit him to supply the working file name.

To avoid this problem, the user of RCS can actually specify the working file, or the revision file, or both. The last form is useful if neither the revision file nor the working file are in the current directory. For example, the RCS command

```
co path1/mod.v path2/mod
```

extracts a revision from *mod.v* in directory *path1* and places it into file *mod* in directory *path2*. If the revision file is omitted, the RCS commands first look for the revision file in the subdirectory *RCS* and then in the current directory. Thus, the user need not clutter his working directories with revision files. The file naming conventions of RCS have been designed such that it can be combined with the tool MAKE.[6] MAKE performs automatic system regeneration after changes and depends on file name suffixes. SCCS was built before MAKE and the two are somewhat incompatible.

The access control in SCCS is sometimes too strict. If a revision is locked, it is impossible to force the lock unless one has extra privileges. Since forcing of locks is occasionally necessary, all users normally acquire that privilege. However, forcing a lock by privileged users leaves no trace. We chose a more flexible approach for RCS. Forcing a lock is possible with a special command, but it always leaves a highly visible trace, namely a message in the mailbox of the user whose lock was broken. Thus, RCS allows work to proceed while delaying the resolution of the update conflict, instead of vice versa.

Automatic identification of revisions based on special markers in the source file is another idea that originated with SCCS. However, the identification mechanism in SCCS is awkward to use. First, markers are not mnemonic and therefore difficult to remember. Second, the SCCS checkout command overwrites the marker with the actual value. Thus, the location of the original marker is lost, and the value cannot be updated automatically on later checkouts. SCCS therefore offers a special case: If the checked-out revision is locked for editing, the expansion of the markers is suppressed. This option keeps the markers in place, but has two other disadvantages. First, revisions that are checked out for editing are not stamped. Thus, the revision being modified contains no identification at all. Second, sometimes one checks out a revision unlocked, but edits it anyway. This happens in a number of circumstances. In some cases, one intends to make only a small modification, expecting to throw it away when done. Unfortunately, these little projects tend to grow such that it becomes worthwhile to save the modifications. In other cases, one checks out a revision unlocked because one lacks the locking privilege, or because the revision has been locked for too long without progress. Rather than wait until the responsible person returns and resolves the conflicts, one checks out a revision unlocked and proceeds with modifications to meet one's schedule. Now one is forced to remove the old stamps and reinsert the markers by hand. Often, these annoying corrections are simply not done, leaving outdated stamps around. Because of these complications, the identification mechanism in SCCS is often not used in practice.

RCS avoids all these problems. Markers are always expanded correctly, and they are easy to remember. RCS also provides a facility for accumulating the log in the source file.

SCCS provides no symbolic revision names, making it awkward to specify which revisions constitute a specific configuration if the revisions do not share the same numbers. One can usually manage to keep revision numbers and dates in synchrony for the initial release. However, as soon as maintenance becomes necessary while the next release is already in development, branches are introduced and the numbering becomes non-homogeneous. Symbolic names are a clean way of restoring order in such situations.

SCCS requires the user to know that revisions are stored as deltas. The user can specify explicitly which deltas to exclude or include during a SCCS checkout operation. This low-level facility is needed because SCCS provides no commands for merging revisions. In RCS one need not consider such implementation details. One can specify the merging of two branches directly, without having to figure out which deltas to exclude or include.

In all fairness, we need to point out that SCCS offers many features that are missing from RCS. For example, SCCS performs complete checksumming, and provides flags that control the creation of branches and the range of revision numbers. We feel that many of these features are unnecessary and contribute to the bulkiness of SCCS. We realize, however, that some of these features may creep into RCS eventually. In any case, the relative performance of RCS and SCCS, to be discussed in the next section, should not be affected by the presence or absence of these features, since they require negligible time and space for processing.†

4.2. Performance

In this section, we analyze the relative performance of reverse, separate deltas (as used in RCS) and forward, merged deltas (as used in SCCS). The measurements were collected on a VAX/11-780 with 4 Mbytes of main memory, running version 4.1 of the Berkeley UNIX. The measurements are load, machine, and operating system dependent. One should therefore consider performance ratios between RCS and SCCS rather than the absolute numbers.

4.2.1. Design of the Experiment

To obtain useful data, we synthesized a set of benchmark files with the average number of revisions, the average number of changes per revision, and the average number of lines per revision. Since there is only little data available on the use of RCS at this time, we based our benchmark files on statistics reported for SCCS. Rochkind[2] observed that the average length of a single revision is about 250 lines. Kernighan[4] found independently that the average UNIX file length to be slightly over 240 lines. Rochkind furthermore reports

† An exception is perhaps checksumming. RCS co derives some of its speed advantage from processing only part of the RCS file. However, a full checksum would require processing the complete file. An incremental checksum, one for each delta, is probably more appropriate for separate deltas, and would preserve the speed advantage of RCS co.

that the average number of revisions is 5, with a space overhead of 35 percent. Assuming that all revisions are of the same length (this assumption will be justified below), then each of the 4 changes (excluding the initial one) accounts for 35/4 percent of the initial revision, or 22 changed lines.

The missing statistics are the average line length and the average length of a block of changed lines. This data was derived from our environment. One of the most popular editors on our VAXes is one that keeps a backup copy for every file touched. We wrote a program that finds pairs of backup copies and edited versions and compares them. A sample of about 900 files revealed the following. The average length of a changed block of lines was approximately 6 lines, and the average line length was 33 characters. To our surprise, we also found that the average file length was 243 lines and the average number of lines changed was 19. Such a close match justifies that we "mix" observations from two different environments to synthesize the test data.

Our data also showed that backup copy and edited version were of almost the same length. This means that modifications do not change the file size significantly, and our assumption of equal length of all revisions is justified.

Based on this data, we created 2 sets of 10 benchmark files containing 1 to 10 revisions each. One set was for RCS, the other for SCCS. The initial revision consisted of 250 lines of 33 characters. In all other revisions, we changed a total of 22 lines in 2 blocks of 5 lines and 2 blocks of 6 lines. These blocks were equally spread through the file, and did not overlap until the 7th revision. The effect of overlapping changes is probably insignificant, because no serious degradation in performance was observed for the 7th and higher revisions. An exception is SCCS ci, which seems to be sensitive to overlaps (see below).

The timings shown were measured with a single user on the system. In all comparisons, every pair of points was obtained by executing the corresponding RCS and SCCS operations alternately 10 times and taking the average. Thus, changes of the system load affected both SCCS and RCS commands equally. The same measurements were taking on a heavily loaded system. Due to the inaccuracy of the UNIX clock, the timings varied widely in this case. To obtain more accurate readings, we increased the size of the revisions 20 times. Thus, the initial revision was 5000 lines, and a single change involved 440 lines in 4 blocks. Measurements with these enormous files on a heavily loaded system showed the same trends as the small files on a lightly loaded system.

4.2.2. Results

Figure 3 shows the time required to check out the latest revision as a function of the number of revisions present. Recall that the latest revision is stored unchanged by RCS. Consequently, the time required by the RCS co operation stays approximately constant, no matter how many revisions are stored. SCCS, on the other hand, scans all revisions. Accordingly, the time increases with every additional revision, until SCCS takes about 2.3 times as long as RCS.

The graph also shows that RCS starts out at a considerable advantage. The initialization cost of SCCS is apparently much higher than for RCS. However, even for files 20 times the average size where initialization is no longer noticeable, SCCS still takes about twice as long as RCS to retrieve the latest revision out of 10.

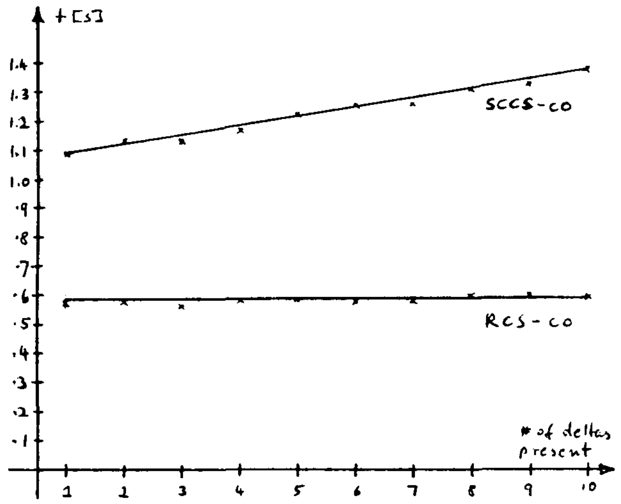

Fig. 3: Time for checkout of latest revision.

Figure 4 shows the time required to check out a revision as a function of the number of deltas applied. This was done on the benchmark file with 5 revisions. The time required for SCCS co remains constant, because SCCS reads the complete file, independent of the revision retrieved. RCS co exhibits quite a different behavior. RCS co is faster for the first three revisions, but slower for the others. The two curves cross over between the third and fourth revisions. The slope of the curve for RCS co reflects the time for the editing passes over the file.[‡] The slope of the curve is sensitive to the size of the revisions. For files 20 times the average (i.e., 5000 lines!) the cross over point is near the second revision.

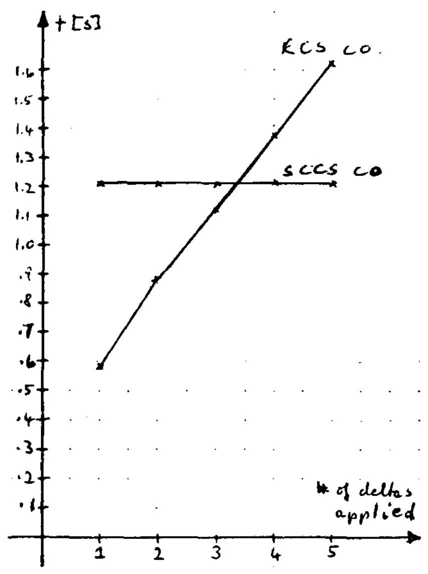

Fig. 4: Time for checkout as a function of the number of deltas applied, 5 deltas present.

Figure 5 shows the time required to add a new revision to the trunk, as a function of the number of revisions present. (Because of the longer executions times, 5 rather than 10 runs per data point provided satisfactory accuracy.) RCS ci is slightly faster for the trunk, but it should approach SCCS ci for side branches because of the editing required to generate the branch tip. Computing the delta accounts for about 60% of RCS ci. The deterioration in performance of SCCS ci between revision 6 and 7 could be due to the overlapping changes in revisions 7 and higher. RCS files are 1% to 4% larger than corresponding SCCS files. Apparently, merged deltas are a slightly more efficient encoding of multiple revisions.

Our data demonstrates that reverse, separate deltas outperform merged, forward deltas. Considering the checkout operation for the "average" file with 5 revisions (Fig. 4), we note that the average of the RCS checkout times for the 4 older revisions is the same as for SCCS. Assuming that these revisions are accessed with decreasing frequency, RCS should be somewhat faster than SCCS for those. In addition, RCS is a factor of 2 faster for the latest revision. In practice, older revisions are accessed rarely, which should give RCS an advantage of close to a factor of 2 overall. Clearly, more data is needed, especially on the use of branches, before more accurate comparisons can be made.

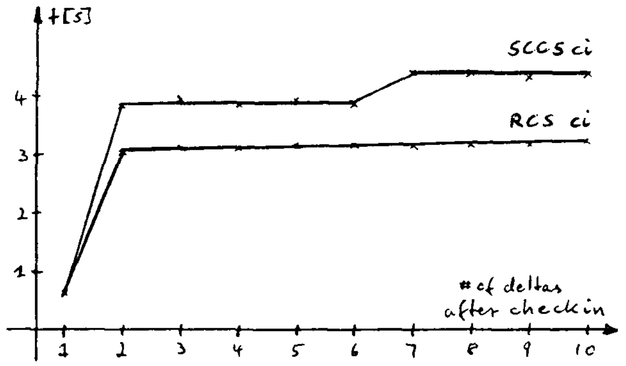

Fig. 5: Time for checkin on the trunk.

4.3. Future Work

RCS has been instrumented to collect statistics about its use. In particular, it records the number of deltas that need to be applied to generate a desired revision. This data will show how much more often the most recent revision is accessed. We are also collecting data on the average number of revisions per RCS file. We believe that an average of 5 is too low. For example, Glasser[7] reports an average of 6.6 revisions per SCCS file. We hypothesize that the number of revisions present is actually a function of the age of the file.

The ideal behavior of RCS would be if the checkout time for older revisions remained constant, just as in SCCS. We are investigating whether this can be achieved by merging the edit scripts. This technique would give fast performance for the latest revision, and require a single editing pass for all others.

5. Conclusions

We presented design and implementation of a revision control system, and evaluated it against a similar system. We showed experimentally that an implementation with reverse, separate deltas outperforms one with forward, merged deltas. The experiment consisted of timing various operations on a set of benchmark files.

‡ An earlier implementation invoked a general purpose text editor, *ed*, as a separate process to perform the regeneration of old revisions. This resulted in an enormous performance penalty: 3 to 5 times the cost of SCCS co!

Because of the lack of adequate metrics, the user interface design could only be evaluated subjectively, although the design improvements may turn out to be more valuable than the performance improvements.

Acknowledgments: Many people contributed to this project, and I am grateful to all of them. Special thanks go to Bill Joy and Eric Allman from Berkeley, who thoroughly criticized my design and made sure I did not make an undesirable system. David Arnovitz implemented 2 (!) prototypes, and Tim Korb and Stephan Bechtolsheim patiently used RCS despite some problems at first.

References

1. Belady, L.A. and Lehman, M.M., "The Characteristics of Large Systems," pp. 106-138 in *Research Directions in Software Technology*, ed. Peter Wegner,M.I.T. Press (1979).
2. Rochkind, Marc J., "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1**(4) pp. 364-370 (Dec. 1975).
3. Habermann, A. Nico, *A Software Development Control System*, Technical Report, Carnegie-Mellon University, Department of Computer Science (Jan. 1979).
4. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Software – Practice and Experience* **9**(1) pp. 1-15 (Jan. 1979).
5. Ivie, Evan L., "The Programmer's Workbench - A Machine for Software Development," *Communications of the ACM* **20**(10) pp. 746-753 (Oct. 1977).
6. Feldman, Stuart I., "Make - A Program for Maintaining Computer Programs," *Software – Practice and Experience* **9**(3) pp. 255-265 (March 1979).
7. Glasser, Alan L., "The Evolution of a Source Code Control System," *Software Engineering Notes* 3(5) pp. 122-125 (Nov. 1978). Proceedings of the Software Quality and Assurance Workshop.