1982

Design, Implementation and Evaluation of a Revision Control System

Walter F. Tichy

Report Number:
81-397

Design, Implementation, and Evaluation of a Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

CSD-TR-397

ABSTRACT

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

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 runtime efficiency, while requiring no extra space.

Keywords: Experimental computer science, programming environments, software maintenance, software tools, version control.

March 25, 1982

Design, Implementation, and Evaluation of a Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

CSD-TR-397

1. Introduction

An important characteristic of software is that it changes constantly. The plasticity of software fosters a mode of development in which modification of a released software product is the norm rather than the exception. Some of the changes are necessary to correct errors, i.e., to make the program consistent with its specifications. Other changes move a software system away from its original specifications. "Improved" versions of heavily used software products seem to arise almost spontaneously. This latter phenomenon, dreaded by every system builder, arises because a successful product, depended upon by a large user community, will always be applied in unexpected ways or unforeseen situations, generating the desire and even necessity to add all kinds of extensions, bells, and whistles.

The constant modifications produce a family of related software systems. As the size of the family grows, the management of the family becomes more and more difficult. Management is necessary for keeping the cost of the evolving family down and for averting chaos. The cost can be limited by avoiding duplication of effort and by configuring every family member from as many standard parts as possible. Skillful management, design, and implementation must be combined to prevent chaos 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 system families. RCS stores and retrieves multiple revisions of program and other text. It logs changes, identifies revisions, merges revisions, and controls access to them. The space overhead of

storing multiple versions is minimized by saving only the differences between successive revisions.

The basic idea of RCS is not new. There are several other systems that have a similar purpose, for example SCCS[Roc75a], and SDC[Hab79a]. 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 perform an experiment to demonstrate 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[Ker79a]. However, readers not familiar with UNIX should be able to follow the description without problems, since the basic ideas are independent of a particular operating system.

Suppose a programmer wishes to put a file *mod* containing program text under control of RCS. He may plan a series of modifications, from which it would be difficult to recover without a back-up copy in case the modifications go wrong. It could also be that the programmer anticipates numerous revisions of the program, and that he wants to save them in a space efficient way. Whatever his motivation, he issues the following command:

```
ci -i mod
```

Ci, short for checkin, deposits revisions into RCS files. RCS files contain multiple revisions of text that are managed by RCS. In this example, the option *-i* indicates that an RCS file for *mod* must be initialized. *Ci* therefore creates a 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. If the -i option is present, it 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 the old version of his program is still available. When he thinks that his modifications have led to a new version 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 assign 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 -d19/2 mod
```

retrieves the latest revision that was checked in on or before Feb. 19, 23:59:59 o'clock of the current year.

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.

So far, we have been inaccurate about one detail, namely about the locking of revisions. RCS must 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.

This conflict is prevented in RCS by locking. In order to check in a new revision, a programmer must lock 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* option 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* option 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.

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 if they are the responsibility of a single programmer. In case an RCS file is updated by several people, the owner of the file should always lock although locking is not enforced, or he should be someone who is not permitted to deposit new revisions. Otherwise, a conflict situation as outlined above could arise.

2.1. The Revision Tree

The above situation of two programmers modifying the same revision should actually be handled with 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 are again numbered sequentially, using 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). Revisions and branches may actually be numbered in arbitrary increments. For instance, revision 3.2 may directly precede revision 3.8.

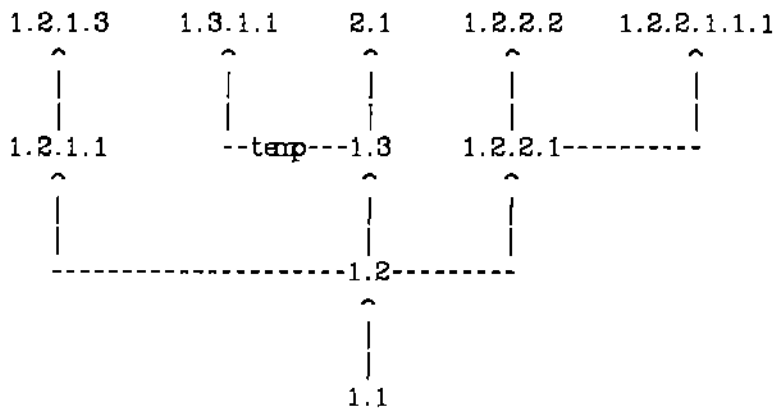

Fig. 1: A revision tree with 4 side branches.

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 must be appended to an existing branch or start 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, which are 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 two auxiliary RCS commands. *Rlog* displays the log entries and other information about revisions in a variety of formats. *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*. If *r1* and *r3* are on two separate branches that have *r2* as a common ancestor, joining has the effect of incorporating into a copy of *r3* all changes that lead from *r2* to *r1*. The resulting revision can be edited or checked back in as a new revision. *Co* will inform the user if there is an overlap between the changes from *r2* to *r1* and from *r2* to *r3*. In that case, the user has to examine and edit the resulting revision. (Revision *r2* may actually be omitted; *co* finds the youngest common ancestor automatically.)

The join operation is completely general in that it may be applied to any triple of revisions. A 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

$Header: RCSfile revisionnumber date time author state$

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 version with smaller table.

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 symbolic 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. A third auxiliary RCS command, *ident*, extracts all such strings from a compiled and linked program. Thus, it is extremely simple to determine which revisions and which modules went into a certain software system. Such a facility is invaluable for program maintenance.

3. Implementation of RCS

RCS stores deltas for conserving space. 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.

Another 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 [Roc75a].

Merged deltas have the property that the time for regeneration is the same for all revisions. The whole revision file must be scanned for collecting 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 some performance advantages 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. On regeneration, the initial revision is extracted into a temporary file, a simple stream editor is invoked, and the edit-script is piped 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 the 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 much quicker than the merging.

The disadvantage of reverse, separate deltas is that the regeneration of old revisions takes longer than with merged deltas. The problem is that the application of n deltas requires n passes over the latest revision. Also, the editing cost is incurred every time an old revision is regenerated, whereas merged deltas require editing only once per delta during the merge. Section 4 presents data to determine how much more often the latest revision should be accessed to obtain a net saving in processing time.

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 the editing overhead. However, if active branches appear towards the end of the trunk, only a few deltas need to be applied.

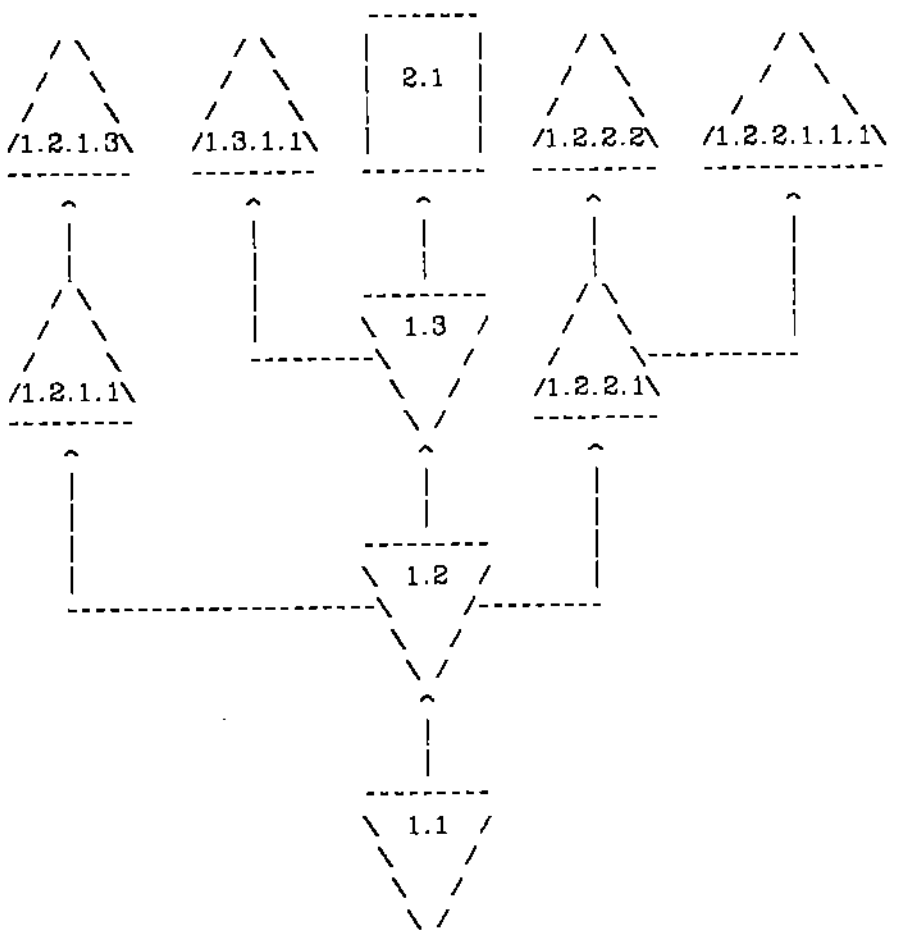

Fig. 2: A revision tree with forward and reverse deltas.

4. Evaluation

In this section, we compare design and performance of RCS and SCCS. 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[Fel79a]. MAKE performs automatic system regeneration after changes and depends on file name suffixes. SCCS was built before MAKE and the two were never integrated properly.

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 the 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, the 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 small a 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 case, 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 problems and complications, the identification mechanism in SCCS is often not used in practice.

RCS avoids all these problems. The 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.[1]

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.

[1] An exception is perhaps checksumming. RCS co derives 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.

4.2.1. Design of the Experiment

To obtain useful data, we had to construct a benchmark file 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 measurements on statistics reported for SCCS[Roc75a]. Rochkind observed that the average length of a single revision is about 250 lines. This was confirmed independently in[Ker79a], where the average UNIX file length was found to be slightly over 240 lines. Rochkind furthermore reports 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).

We performed initial timings of SCCS and RCS operations. These showed that the SCCS checkout operation was on the average 50% slower than the equivalent RCS operation for the latest revision. Due to the inaccuracy of the UNIX clock, these measurements were consistent only for a lightly loaded system. If the system was heavily loaded, the timings varied widely. To obtain more accurate measurements, 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. All timings given below were measured with those enormous files. Consequently, the measurements are greatly exaggerated, and one should only consider performance ratios rather than the absolute numbers.

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 curves shown were measured with a single user logged on. The maximum variation in the measurements was less than 5% of the average and considered insignificant. We took similar sets of measurements on a lightly loaded (about 10 users) and on a heavily loaded system (over 30 users). On the lightly loaded system, the times required were slightly higher, and the curves were no longer smooth. However, because of the alternate execution of RCS and SCCS operations, the ratios between corresponding operations were the same as in the single user case. On the heavily loaded system, the measurements varied considerably with changing load conditions, and were up to 30% higher than on the single user system. Still, the ratios stayed about the same.

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, has to scan all revisions. The graph shows that the ratio between SCCS co and RCS co increases steadily, until SCCS co takes about twice

as long as RCS co. For the average case with 5 revisions, SCCS co is about 60% slower than RCS co.

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 latest revision, but slower for all others. The two curves cross over for the predecessor of the latest revision. The slope of the curve for RCS co reflects the time for the editing passes over the file.[2]

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 long executions times, 5 rather than 10 runs per data point provide enough accuracy. The maximum variation is within 1% of the average for all but the first pair of points.) SCCS ci requires 20% to 30% more time than RCS ci. Computing the delta accounts for about 60% of RCS ci. Appending to side branches should be more expensive for RCS ci, because of the editing required to generate the branch tip. The deterioration in performance of SCCS ci between revision 6 and 7 could be due to the overlapping changes in revisions 7 and higher.

Our data demonstrates that reverse, separate deltas outperform merged, forward deltas if the latest revision is accessed more often than all others. Considering only the checkout operation, RCS and SCCS require about the same total time if the latest revision is checked out slightly over twice as often as the others (assuming equal frequency for all others). If this ratio is lower, SCCS-style deltas are preferable, otherwise RCS-style deltas. We believe that the ratio of 2/1 is easily exceeded in practice, because one needs to recover old revisions only rarely.

[2] 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!

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 whether the initial revision is accessed frequently enough to warrant the use of reverse deltas. We are also collecting data on the average number of revisions per revision file. We believe that an average of 5 is too low. For example, Glasser[Gla78a] 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. One way to achieve this would be to keep the latest revision intact, but to merge 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 may outperform one with forward, merged deltas. The experiment consisted of timing various operations on a set of benchmark files.

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

Fel79a. Feldman, Stuart I., "Make - A Program for Maintaining Computer Programs," *Software – Practice and Experience* **9**(3) pp. 255-265 (March 1979).

Gla78a. 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.

Hab79a. Habermann, A. Nico, *A Software Development Control System*, Technical Report, Carnegie-Mellon University, Department of Computer Science (Jan. 1979).

Ker79a. Kernighan, Brian W. and Mashey, John R., "The UNIX Programming Environment," *Software – Practice and Experience* **9**(1) pp. 1-15 (Jan. 1979).

Roc75a. Rochkind, Marc J., "The Source Code Control System," *IEEE Transactions on Software Engineering* **SE-1**(4) pp. 364-370 (Dec. 1975).

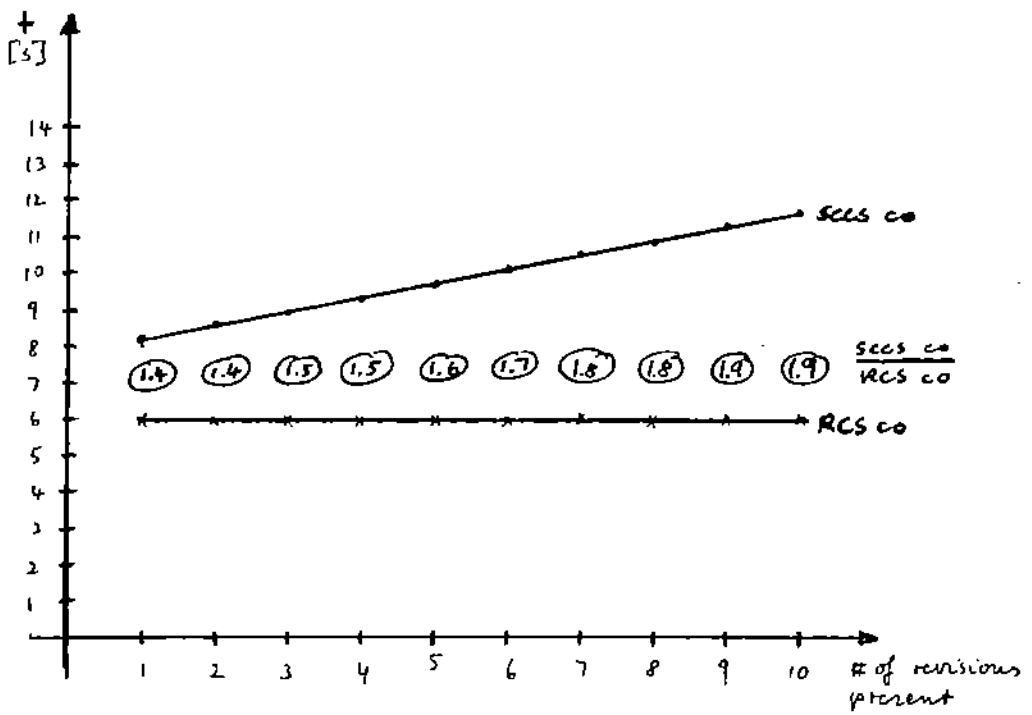

Fig.3 : Time for checkout of latest revision

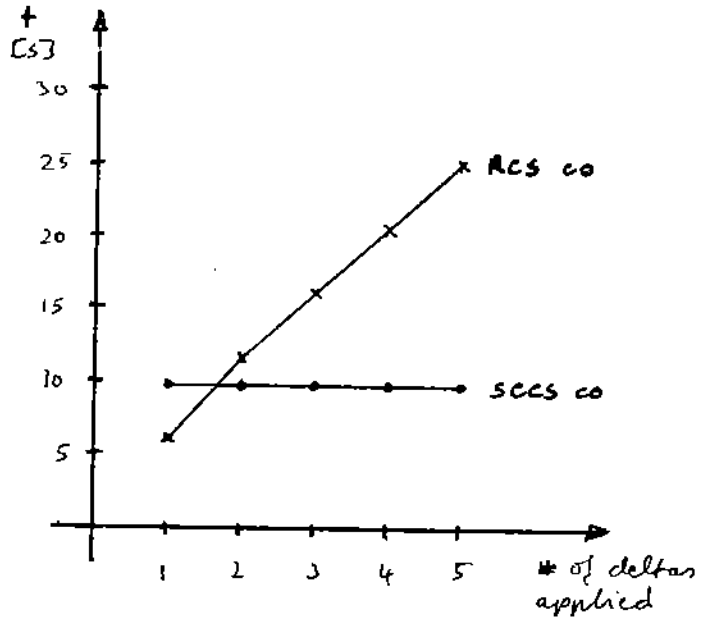

Fig. 4 : Time for checkout as a function of the number of deltas applied.

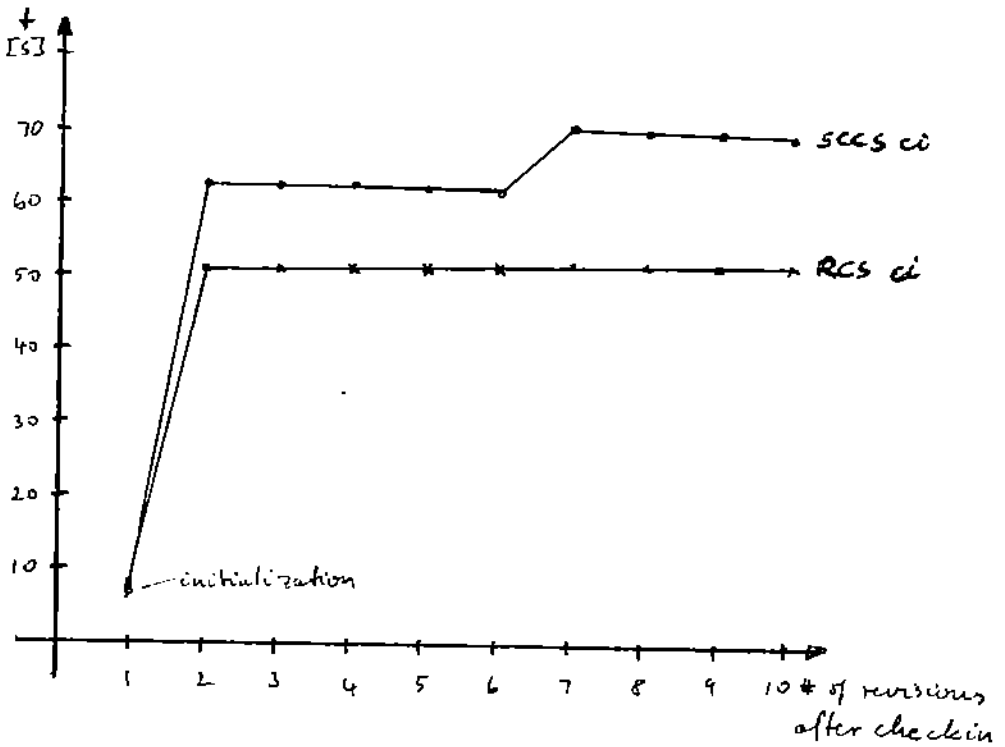

Fig. 5: Time for checkin as a function of the number of revisions present.

Appendix A: How to get started with RCS

Suppose you have a file f.c that you wish to put under control of RCS. Invoke

```
ci -i f.c
```

This command creates f.v and stores f.c into it as revision 1.1. It will also delete f.c. To get f.c back, type

```
co f.c
```

You can now edit f.c and check it in as revision 1.2 by invoking

```
ci f.c
```

To avoid the deletion during ci (in case you want to continue editing), invoke

```
ci -l f.c
```

The -l option on ci leaves f.c alone and saves you one co operation. This option can also be used during initialization.

Suppose you have files with names f.c and f.h which would result in two RCS files f.v. There are two ways to to get around this problem. One solution is to change f.h to f_h, fh.c, or fheader.c. The other solution is to tell RCS not to strip off the suffix. This can be done by specifying the RCS filenames explicitly during initialization, like in

```
ci -i f.c,v f.h,v
```

In this case, ci creates f.c,v and f.h,v and looks for f.c and f.h in your current directory. Once f.c,v and f.h,v have been created, you can use

```
ci f.c f.h
co f.c f.h
```

and ci will figure out what RCS files to take. You can also use a mixture, like

```
ci -i f.c f.h,v
```

This command puts f.c into f.v and f.h into f.h,v. This alternative is convenient for default rules in MAKE.

It is a good idea to make an extra directory called RCS in your working directory. Co and ci will first look there to find RCS files (ending in ,v). Of course, if you specify RCS files explicitly, you must precede them with RCS/. The above command becomes

```
ci -i f.c RCS/f.h,v
```

Ci creates RCS/f.v for f.c, if RCS exists, otherwise it creates f.v in your working directory. RCS/f.h,v is given explicitly, and ci looks for a file f.h in your working directory. After initialization, everything goes back to normal and

```
co f.c f.h
ci f.c f.h
```

will work as expected.

Combining MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form .v.c, because this will always keep a copy of your working files around, even those you are not working on. Instead, do this:

```
.SUFFIXES: .v

.v.o:
        co -q $<
        cc $(CFLAGS) -c $*.c
        rm -f $*.c

prog:   f1.o f2.o .....
        cc f1.o f2.o ..... -o prog
```

This rule has the following effect. If a file f.c does not exist, and f.o is older than f.v, MAKE checks out f.c, compiles f.c into f.o, and then deletes f.c From then on, MAKE will use f.o until you change f.v again.

If f.c exists (presumably because you are working on it), the default rule .c.o takes precedence, and f.c is compiled into f.o, but not deleted.

To avoid confusion of f.c and f.h, store f.c in f.v and f.h in f.h.v. The default rule will apply only for f.c; you need to write an explicit checkout rule for f.h.

If you keep your RCS file in the directory ./RCS, all this won't work and you have to write explicit checkout rules for every file, like

```
f1.c:   RCS/f1.v
        co -q f1.c
```

Unfortunately, these rules do not have the property of removing unneeded .c-files.

Appendix B: RCS Manual Pages

NAME

ci – check in RCS revisions

SYNOPSIS

ci [options] file ...

DESCRIPTION

Ci stores new revisions into RCS files. Each filename ending in ',v' is taken to be an RCS file, all others are assumed to be working files containing new revisions. *Ci* deposits the contents of each working file into the corresponding RCS file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section of *co* (1)).

1) Both the RCS file and the working file are given. The RCS file is of the form *path1/file*,v or *path1/file.sfx*,v and the working file is of the form *path2/file.sfx*, where *path1/* and *path2/* are (possibly different or empty) paths, *file* is the the filename stem common to both files, and *.sfx* is a (possibly empty) suffix.

2) Only the RCS file is given. Then the corresponding working file is *file.sfx* in the current directory.

3) Only the working file is given. Then the corresponding RCS file is of the form *file.sfx*,v or *file*,v, and *co* tries to find it first in the directory *./RCS*, and then in the current directory.

If *file.sfx* is stored in *file.v*, then *.sfx* of the working file may be omitted; *ci* will add it if nonempty.

For *ci* to work, the caller's login must be on the access list, except if the access list is empty or the caller is the superuser or the owner of the file. To append a a new revision to an existing branch, the tip revision on that branch must be locked by the caller. Otherwise, only a new branch can be created. This requirement is not enforced for the owner of the file. (An existing lock by somebody else may be broken with the *rcs* command.)

For each revision checked in, *ci* prompts for a log message. The log message should summarize the change and is terminated with a line containing a single '.' or a control-D. If several files are checked in, *ci* asks whether to reuse the previous log message.

–i[*rev*] creates a new RCS file, deposits the contents of the working file as the initial revision, and assigns revision number *rev* to it (default: 1.1). The suffix attribute is derived from the name of the working file and the access list is initialized to empty. Instead of the log message, descriptive text is requested (see -t below). If the RCS file already exists, an error message is printed.

–r[*rev*] assigns the revision number *rev* to the checked-in revision, releases the corresponding lock, and deletes the working file. This is also the default.

If *rev* is omitted, *ci* derives the new revision number from the caller's last lock. If the caller has locked the tip revision of a branch, the new revision is appended to that branch. The new revision number is obtained by incrementing the tip revision number. If the caller locked a non-tip revision, a new branch is started at that revision by incrementing the highest branch number at that revision. The default initial branch and level numbers are 1. If the caller has no lock and is the owner of the file, the new revision is appended to the trunk.

If *rev* indicates a revision number, it must be higher than the latest one on the branch to which *rev* belongs, or must start a new branch.

If *rev* indicates a branch instead of a revision, the new revision is appended to that branch. The level number is obtained by incrementing the tip revision number of that branch. If *rev* indicates a non-existing branch, that branch is created with the initial revision numbered *rev.1*.

Exception: On the trunk, revisions can only be appended to the end, but not inserted inside the trunk.

-l[*rev*] works like **-r**, except it locks the new revision after checkin and does not delete the working file. This is useful for saving a revision without having to do another checkout.

-q[*rev*] quite mode; no diagnostic output is printed. *Rev* is assigned to the checked-in revision. The log message must be provided with the **-m** option.

-m*msg* uses the string *msg* as the log message for all revisions checked in.

-n*name* assigns the symbolic name *name* to the number of the checked-in revision. *Ci* prints an error message if *name* is already assigned to another number.

-N*name* same as **-n**, except that it overrides a previous assignment of *name*.

-s*state* sets the state of the checked-in revision to the string specified by *state*. The default is *Exp*.

-t[*txtfile*]
Txtfile is the name of a text file containing descriptive text. If the **-i** option is present and the text file is not given, *ci* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. If the RCS file already exists and the text file is supplied, *ci* replaces the existing text with the new one. If the RCS file exists and the **-t** option is given without the text file, *ci* erases the existing text and replaces it with text supplied from the std. input.

DIAGNOSTICS

For each checked in revision, *ci* prints the RCS file, the working file, and the revision number.

AUTHOR

Walter F. Tichy

FILES

The caller of the command must have read/write permission to the directories containing the RCS file and the working file, and to the RCS file itself. A number of temporary files are created. A semaphore file is created in the directory containing the RCS file. *Ci* always creates a new RCS file and unlinks the old one. This makes links to RCS files useless.

SEE ALSO

co (1), ident(1), rcs (1), rlog (1)

BUGS

ci -l does not update the keywords in the working file.

NAME

co – check out RCS revisions

SYNOPSIS

co [options] file ...

DESCRIPTION

Co retrieves revisions from RCS files. Each filename ending in '.v' is taken to be an RCS file. All other files are assumed to be working files. *Co* retrieves a revision from each RCS file and stores it into the corresponding working file.

Pairs of RCS files and working files may be specified in 3 ways (see also the example section).

1) Both the RCS file and the working file are given. The RCS file is of the form *path1/file*.v or *path1/file.sfx*.v and the working file is of the form *path2/file.sfx*, where *path1/* and *path2/* are (possibly different or empty) paths, *file* is the the filename stem common to both files, and *.sfx* is a (possibly empty) suffix.

2) Only the RCS file is given. Then the corresponding working file is *file.sfx* in the current directory.

3) Only the working file is given. Then the corresponding RCS file is of the form *file.sfx*.v or *file*.v, and *co* tries to find it first in the directory *./RCS*, and then in the current directory.

If *file.sfx* is stored in *file.v*, then *.sfx* of the working file may be omitted; *co* will add it if nonempty.

Revisions of an RCS file may be checked out locked or unlocked. Locking a revision avoids overlapping updates. A revision checked out for reading or processing (e.g. compiling) should not be locked. A revision checked out for editing and later checkin must normally be locked. Locking a revision currently locked by another user is illegal. (A lock may be broken with the *ci* or the *rcs* command.) *Co* with locking requires the caller to be on the access list of the RCS file, unless he is the owner of the file or the superuser, or the access list is empty. There are no accesslist restrictions for *co* without locking.

A revision is selected by number, creation date, author, or state. If none of these options is specified, the latest revision on the trunk is retrieved. When the options are applied in combination, the latest revision that satisfies all of them is retrieved. An error message results if the options cannot be satisfied. A revision number may be attached to either of the options **-l**, **-u**, **-p**, **-q**, or **-r**. A *co* command applied to an RCS file with no revisions creates a zero-length file. *Co* always performs keyword substitution (see below).

-l[*rev*] locks the checked out revision for the caller. See option **-r** for handling of the revision number *rev*.

-u[*rev*] does not lock the checked out revision. This option is the default. Besides for reading, it is needed if the caller intends to edit the revision and place it on a different branch. See option **-r** for revision number *rev*.

-p[*rev*] prints the retrieved revision on the std. output rather than storing it in a file. This option is useful when the *co* command is part of a pipe.

-q[*rev*] quiet mode; diagnostics are not printed.

-d*date* retrieves the latest revision whose date is less than or equal to *date*. The standard form of a date has six digit fields separated by periods (July 12, 1982, 6:11:05 pm is written as 82.7.12.18.11.05). Trailing fields and the year may be omitted. (Ex.: "co -d7.12" retrieves the

newest release created on or before 82.7.12.23.59.59, if the year is 1982.) Most popular forms of dates and times are recognized, but must be quoted if they contain spaces. (Ex.: "6/12 00:01 am", "18 Aug.")

-r*rev* retrieves the latest revision whose number is less than or equal to *rev*. If *rev* indicates a branch rather than a revision, the latest revision on that branch is retrieved. *Rev* is composed of one or more numeric or symbolic fields separated by '.'. The numeric equivalent of a symbolic field is specified with the -n option of the commands *ci* and *rcs*.

-s*state* retrieves the latest revision whose state is set to *state*.

-w[*login*] retrieves the latest revision written by the user with login name *login*. If the argument *login* is omitted, the callers login is assumed.

-j*joinlist* generates a new revision which is the join of the revisions on *joinlist*. *Joinlist* is a comma-separated list of pairs of the form *rev2:rev3*, where *rev2* and *rev3* are (symbolic or numeric) revision numbers. For the initial such pair, *rev1* denotes the revision selected by the options -l, ..., -w. For all other pairs, *rev1* denotes the revision generated by the previous pair. (Thus, the output of one join becomes the input to the next.)

For each pair, *co* joins revisions *rev1* and *rev2* with respect to *rev3*. This means that all changes that transform *rev3* into *rev2* are applied to *rev1*. This is particularly useful if *rev1* and *rev2* are the ends of two branches that have *rev3* as a common ancestor. If *rev2* > *rev3* > *rev1* on the same branch, joining has the effect of undoing the changes that lead from *rev3* to *rev2* in *rev1*.

If *rev3* is omitted, the youngest common ancestor is assumed. If any of the arguments indicate branches, the latest revisions on those branches are assumed. If the option -l is present, the initial *rev1* is locked.

KEYWORD SUBSTITUTION

Strings of the form *$keyword$* and *$keyword:...$* embedded in the text are replaced with strings of the form *$keyword: value$*, where *keyword* and *value* are pairs listed below. Keywords may be embedded in literal strings or comments to identify a revision.

Initially, the user enters strings of the form *$keyword$* and checks in the file. After the first checkout, these strings are replaced with *$keyword: value$*. If this revision is modified and checked back in, the value field is no longer correct. However, on a subsequent checkout, *co* again replaces strings of the form *$keyword:...$* with the correct *$keyword: value$*. Thus, the keyword values are automatically updated. Warning: Do not tamper with expanded keywords except for deleting them.

Keywords and their corresponding values:

$Author$ The login name of the user who checked in the revision.

$Date$ The date and time the revision was checked in, in the format YY.MM.DD.hh.mm.ss.

$Header$ A standard header containing the RCS file name, the revision number, the date, the author, and the state.

Log The log message supplied during checkin, preceded by a header containing the RCS file name, the revision number, the author, and the

date. Existing log messages are NOT replaced. Instead, the new log message is inserted after *$Log:...$*. This is useful for accumulating a complete change log in a source file.

$Revision$
The revision number assigned to the revision.

$Source$ The RCS file name.

$State$ The state assigned to the revision with *rcs -s* or *ci -s*.

$Suffix$ The suffix recorded with *rcs -x* or *ci -i*.

DIAGNOSTICS

The RCS file name, the working file name, and the revision number retrieved are written to the diagnostic output.

EXAMPLES

Suppose the current directory contains a subdirectory 'RCS' with a RCS file 'io.v'. Then all of the following commands retrieve the latest revision from 'RCS/io.v' and store it into 'io.c', provided 'RCS/io.v' has its suffix attribute set to 'c'.

```
co io;  co io.c;  co RCS/io.v
co io RCS/io.v;  co RCS/io.v io;
co RCS/io.v io.c;  co io.c RCS/io.v
```

AUTHOR

Walter F. Tichy

FILES

The caller of the command must have write permission in the working directory and either read permission (for reading) or read/write permission (for locking) in the directory which contains the RCS file.

A number of temporary files are created. A semaphore file is created in the directory of the RCS file to prevent simultaneous update.

SEE ALSO

ci (1), ident (1), rcs (1), rlog (1)

BUGS

The option **-j** does not work for revisions larger than 64K bytes.

NAME

ident – identify files

SYNOPSIS

ident file ...

DESCRIPTION

Ident searches the named files for all occurrences of the pattern *$keyword:...$*, where *keyword* is one of

Author
Date
Header
Log
Revision
Source
State
Suffix

These patterns are normally inserted automatically by the RCS command *co (1)*, but can also be inserted manually.

Ident works on text files as well as object files. For example, if the C program in file f.c contains

```
char rcsid[] = "$Header: Header information$";
```

and f.c is compiled into f.o, then the command

```
ident f.c f.o
```

will print

```
f.c:
     $Header: Header information$
f.o:
     $Header: Header information$
```

AUTHOR

Walter F. Tichy

SEE ALSO

ci (1), co (1), rcs (1), rlog (1).

BUGS

NAME

rcs – create RCS files or change RCS file attributes

SYNOPSIS

rcs [options] file ...

DESCRIPTION

Rcs creates new RCS files or changes attributes of existing ones. An RCS file contains multiple revisions of text, an access list, a change log, descriptive text, and some control attributes. For *rcs* to work, the caller's login name must be on the access list, except if the access list is empty, the caller is the owner of the file or the superuser, or the **-i** option is present.

Files ending in '.v' are RCS files, all others are working files. If a working file is given, *rcs* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

–i creates and initializes a new RCS file. If the file already exists, an error message is printed.

–a*logins* adds the login names appearing in the comma-separated list *logins* to the access list of the RCS file.

–A*oldfile* replaces the access list of the RCS file with a copy of the access list of *oldfile*.

–e*login* erases the login names appearing in the comma-separated list *logins* from the access list of the RCS file.

–l[*rev*] locks the revision with number *rev*. If a branch is given, the latest revision on that branch is locked. If *rev* is omitted, the latest revision on the trunk is locked. Locking prevents overlapping changes. A lock is removed with *ci* or *rcs -u* (see below). The default is to leave the locks of an existing file unchanged, and to leave a new file unlocked.

–u[*rev*] unlocks the revision with number *rev*. Normally, only the locker of a revision may unlock it. Somebody else unlocking a revision breaks the lock. This causes a mail message to be sent to the original locker. The message contains a commentary solicited from the breaker. The commentary is terminated with a line containing a single '.' or control-D.

–n*name*[:*rev*]
associates the symbolic name *name* with the branch or revision *rev*. If *rev* is omitted, the most recent revision on the main trunk is assumed. *Rcs* prints an error message if *name* is already assigned to another number.

–N*name*[:*rev*]
same as **-n**, except that it overrides a previous assignment of *name*.

–o*list* deletes ("outdates") the revisions given in the comma-separated *list* of revisions and ranges. A range consisting of a branch means all revisions on that branch. A range *rev1–rev2* means revisions *rev1* to *rev2* on the same branch, *–rev* means from the beginning of the branch containing *rev* up to and including *rev*, and *rev–* means from revision *rev* to the end of the branch containing *rev*. None of the outdated revisions may have branches.

–s*state*[:*rev*]
sets the state attribute of the revision *rev* to *state*. If *rev* is omitted, the latest revision on the trunk is assumed. Any character string is

acceptable for *state*. A useful set of states is *Exp* (for experimental), *Stab* (for stable), and *Rel* (for released). By default, *ci* sets the state of a revision to *Exp*.

-t[*txtfile*]
Txtfile is the name of a text file containing descriptive text. If the **-i** option is present and the text file is not given, *rcs* prompts the user for text supplied from the std. input, terminated with a line containing a single '.' or control-D. If the RCS file already exists and the text file is supplied, *rcs* replaces the existing text with the new one. If the RCS file exists and the **-t** option is given without a text file, *rcs* erases the existing text and replaces it with text supplied from the std. input.

-x[*sfx*] Sets the suffix attribute of the RCS file to *sfx*. *Sfx* may be any character string allowable in a filename, but without '.', ',', ';', and ':' The suffix is used in the filenames generated by *co(1)*. The default is the empty string.

AUTHOR

Walter F. Tichy

FILES

Rcs creates a semaphore file in the same directory as the RCS file to prevent simultaneous update. For changes, *rcs* always creates a new file. On successful completion, *rcs* deletes the old one and renames the new one. This strategy makes links to RCS files useless.

SEE ALSO

ci (1), co (1), ident (1), rlog (1).

BUGS

NAME

rlog – print log messages and statistics of RCS files

SYNOPSIS

rlog [–**d***dates*] [–**l**[*lockers*]] [–**r***revisions*] [–**s***states*] [–**w***logins*] [–**a**] [–**n**] [–**t**] file ...

DESCRIPTION

Rlog prints information about RCS files. Files ending in '.v' are RCS files, all others are working files. If a working file is given, *rlog* tries to find the corresponding RCS file first in directory ./RCS and then in the current directory, as explained in *co* (1).

–**d***dates* prints information about revisions with creation dates in the ranges given by the comma-separated list of *dates*. A single date means the range between the floor and ceiling values of the omitted trailing fields. For example, *81.9* means the range *81.9.1.0.0.0–81.9.30.23.59.59*. A range of the form *d1–d2* means the range *floor(d1)–ceil(d2)*. A range of the form *–d* means *0–ceil(d)* (i.e., all revisions deposited on or before *d*). A range of the form *d–* means *floor(d)–now*, where *now* is the current date/time (i.e., all revisions dated *d* or later). The current year may be omitted in all dates.

–**l**[*lockers*]
prints information about locked revisions. If the comma-separated list *lockers* of login names is given, only the revisions locked by the given login names are printed. If the list is omitted, all locked revisions are printed.

–**r***revisions*
prints information about revisions given in the comma-separated list *revisions* of revisions and ranges. A range *rev1–rev2* means revisions *rev1* to *rev2* on the same branch, *–rev* means revisions from the beginning of the branch up to and including *rev*, and *rev–* means revisions starting with *rev* to the end of the branch containing *rev*. An argument that is a branch means all revisions on that branch. A range of branches means all revisions on the branches in that range.

–**s***states* prints information about revisions whose state attributes match one of the states given in the comma-separated list *states*.

–**w***logins* prints information about revisions written (checked-in) by users with login names appearing in the comma-separated list *logins*.

For the options **-d**, **-l**, **-r**, **-s**, and **-u**, *rlog* prints the file name, extension, access list, symbolic names, and total number of revisions, followed by entries for the revisions in reverse chronological order for each branch. For each revision, *rlog* prints revision number, author, date, time, state, log message, and number of lines added and deleted. If no option is given, information about all revisions is printed. Combinations of options print the intersection of the revisions selected by each option. The options below print information that is not associated with revisions.

–**a** prints the access list.

–**n** prints the list of symbolic names.

–**t** prints the descriptive text.

AUTHOR

Walter F. Tichy

SEE ALSO

ci (1), co (1), ident(1), rcs (1).

BUGS

NAME

rcsfile – format of RCS file

DESCRIPTION

An RCS file is an ASCII file. Its contents is described by the grammar below. The text is free format, i.e., spaces, tabs and new lines have no significance except in strings. Strings are enclosed by '@' (doublequote). If a string contains a '@', it must be doubled.

The meta syntax uses the following conventions: '|' (bar) separates alternatives; '{' and '}' enclose optinal phrases; '{' and '}*' enclose phrases that may be repeated zero or more times; '{' and '}+' enclose phrases that must appear at least once and may be repeated; '<' and '>' enclose nonterminals.

```
<rcstext>     ::=  <admin> {<delta>}* <desc> {<deltatext>}*

<admin>       ::=  head        {<num>};
                   suffix      {<id>};
                   access      {<id>}*;
                   symbols     {<id> :<num>}*;
                   locks       {<id> :<num>}*;

<delta>       ::=  <num>
                   date        <num>;
                   author      <id>;
                   state       {<id>};
                   branches    {<num>}*;
                   next        {<num>};

<desc>        ::=  desc        <string>

<deltatext>   ::=  <num>
                   log         <string>
                   text        <string>


<num>         ::=  {<digit>{.}}+

<digit>       ::=  0 | 1 |... | 9

<id>          ::=  <letter>{<idchar>}*

<letter>      ::=  A | B | ... | Z | a | b | ... | z

<idchar>      ::=  Any printing ASCII character except
                   space, tab, carriage return, new line,
                   and <special>.

<special>     ::=  ; | : | , | @

<string>      ::=  @{any ASCII character, with '@'doubled}*@
```

Identifiers are case sensitive. Keywords are in lower case only. The sets of keywords and identifiers may overlap.

SEE ALSO

ci (1), co (1), rcs (1), rlog (1).