1984

RCS: A System for Version Control

Walter F. Tichy

Report Number:
84-474

RCS – A System for Version Control

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

CSD-TR-474

ABSTRACT

An important problem in program development and maintenance is version control, i.e., the task of keeping a software system consisting of many versions and configurations well organized. The Revision Control System (RCS) is a software tool that assists with that task. This paper presents RCS from the user's point of view, describing how to employ RCS for managing multiple versions of individual components as well as configurations.

RCS was originally developed for programming environments, and was intended for storing specifications, source programs, documentation, and test data. Because of its general utility, it is also being used for managing computer graphics, VLSI layouts, form letters, papers, and book chapters.

Besides an introduction to RCS, the paper presents usage data. The statistics show that RCS' method of storing differences is a space and time efficient way of storing multiple revisions. The paper concludes with a survey of the state of the art in version control tools.

Keywords: versions control, revisions, deltas, software tools.

March 13, 1984

RCS – A System for Version Control

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

1. Introduction

The Revision Control System (RCS) is a set of Unix commands that help with version control. Version control is the task of keeping software systems consisting of many versions and configurations well organized. RCS manages revisions of individual components as well as configurations. It keeps a complete development history, and allows backup to any point in the development.

RCS operates on *revision groups*. A revision group is a set of text documents, called *revisions*, which evolved from each other under manual changes. A new revision is usually created by modifying an existing one with a text editor. RCS organizes the revisions in an ancestral tree. The initial revision is the root of the tree, and the tree edges indicate from which revision a given one evolved. RCS may be combined with MAKE [Fel79a], resulting in a powerful package for version control. In addition, RCS offers facilities for "stamping" each revision with a unique marker, for merging updates with customer modifications, and for distributed software development.

Although RCS was originally intended for software, it is useful for any text that is revised frequently and whose previous revisions must be preserved. RCS has been applied successfully to store the source text for drawings, VLSI layouts, documentation, specifications, test data, form letters, and articles.

RCS is designed for both production and experimental environments. In production environments, sophisticated update controls prevent potentially disastrous conflicts and mistakes. In an experimental environment where strong controls are counterproductive, it is possible to disable them.

This paper describes RCS from the user's point of view. As a comprehensive discussion of revision control functions, it is also useful for designers of similar systems. Section 2 provides an introduction to RCS. The succeeding sections discuss the revision tree, locking primitives, and the provisions for configurations. Section 6 presents usage statistics, and Section 7 gives a historical perspective.

2. How to get started with RCS

Suppose we have a text file *f.c* that we wish to put under control of RCS. Invoking the checkin command

ci f.c

creates a new revision group with the contents of *f.c* as the initial revision (numbered 1.1) and stores the group into the file *f.c,v*. Unless told otherwise, the command deletes *f.c*. It also asks for a description of the group. The description should state the common purpose of all revisions in the group, and becomes part of the group's documentation. All later checkin commands will ask for a log entry, which should summarize the changes made. (The first revision is assigned a default log message, which just records the fact that it is the initial revision.)

Files ending in *,v* are called *RCS files* (*v* stands for *v*ersions); the others are called working files. To get back the working file *f.c* in the previous example, we use the checkout command:

co f.c

This command extracts the latest revision from the revision group *f.c,v* and writes it into *f.c*. We can now edit *f.c* and, when finished, check it in back in by invoking

ci f.c

Ci assigns number 1.2 to the new revision. If *ci* complains with the message

ci error: no lock set by <login>

then the system administrator has decided to configure RCS for a production environment by enabling the "strict locking feature." If this feature is enabled, all RCS files are initialized such that checkin operations require a lock on the previous revision (the one from which the current one evolved). Locking prevents overlapping modifications if several people work on the same file. If locking is required, we should have locked the revision during the previous checkout by using the option -l:

co -l f.c

Of course, it is too late now to do the checkout with locking, because we already modified *f.c*. If we would execute a checkout with locking now, it would overwrite our modifications. (To prevent accidental overwrites, *co* senses the presence of a working file and asks whether the user really intended to overwrite it. The overwriting checkout is sometimes useful, namely when one wants to back up to the previous revision.) To be able to proceed with the checkin in the present case, we first execute

rcs -l f.c

This command retroactively locks the latest revision, unless someone else locked it in the meantime. In this case, we have to negotiate with that person and decide whose modifications should take precedence.

If an RCS file is private, i.e., if there is only one person who is going to deposit revisions into it, the strict locking feature is unnecessary and can be disabled. If strict locking is disabled, the owner of the RCS file need not have a lock for checkin; all others still do. Turning strict locking off and on is done with the commands:

rcs -U f.c and *rcs -L f.c*

These commands enable or disable the strict locking feature for each RCS file individually. The system administrator only decides whether strict locking is enabled initially.

If we don't want to clutter our working directory with RCS files, we should create a subdirectory called *RCS* in our working directory, and move all our RCS files there. RCS commands look first into that directory for RCS files. All the commands discussed above will still work, without change*.

Suppose we would like to prevent a working file from being deleted by the checkin command. Reasons for keeping it checked out may be that we would like to compile it or continue editing. We invoke

ci -l f.c

This command checks in *f.c* as usual, but performs an additional checkout with locking afterwards. Thus, the working file does not disappear after the checkin. There is also an option -u for *ci* which does a checkin followed by a checkout without locking. This is useful if we want to compile the file after the checkin. Both options update the identification markers in the working file (see below).

* Pairs of RCS and working files can actually be specified in 3 ways: a) both are given, b) only the working file is given, c) only the RCS file is given. If a pair is given, both files may have arbitrary path prefixes; RCS commands pair them up intelligently.

Besides the operations *ci* and *co*, RCS provides the following commands: *ident* (extract identification markers), *rcs* (change RCS file attributes), *rcsdiff* (compare revisions), *rcsmerge* (merge revisions), and *rlog* (print information about RCS files). A synopsis of these commands appears in the Appendix.

2.1. Automatic Identification

RCS can stamp source and object code with special identification strings, similar to product and serial numbers. To obtain such identification, we place the marker

$Header$

into the text of a revision, for instance inside a comment. The checkout operation will replace this marker with a string of the form

$Header: filename revisionnumber date time author state $

We never need to touch this string, because *co* keeps it up to date automatically. To propagate the marker into object code, we simply put it into a literal character string. In C, this is done as follows:

```
static char rcsid[] = "$Header$";
```

The command *ident* extracts such markers from any file, in particular from object code. *Ident* helps to find out which revisions of which modules were used in a given program. It returns an entire and unambiguous parts list, form which a copy of the program can be reconstructed. This facility is invaluable for program maintenance.

There are several additional identification markers, one for each component of $Header$. The marker

Log

has a similar function. It accumulates the log messages that are requested during checkin. Thus, one can maintain the complete history of a revision directly inside it, by enclosing it in a comment. Below is a partial reproduction of a log contained in revision 4.1 of the file *cl.c*. The log appears at the beginning of the file, and makes it easy to determine what the recent modifications were.

```
/* $Log:       ci.c,v $
 * Revision 4.1  83/05/10  17:03:06  wft
 * Added option -d and -w, and updated assignment of date, etc. to new delta.
 * Added handling of default branches.
 *
 * Revision 3.9  83/02/15  15:25:44  wft
 * Added call to fastcopy() to copy remainder of RCS file.
 *
 * Revision 3.8  83/01/14  15:34:05  wft
 * Added ignoring of interrupts while new RCS file is renamed;
 * avoids deletion of RCS files by interrupts.
 *
 * Revision 3.7  82/12/10  16:09:20  wft
 * Corrected checking of return code from diff.
 * An RCS file now inherits its mode during the first ci from the working file,
 * except that write permission is removed.
 */
```

FIGURE 1: Log entries produced by the marker Log.

Note: Since revisions are stored in the form of differences, each log message is physically stored once, independent of the number of revisions present. Thus, the Log marker incurs negligible space overhead.

3. The RCS Revision Tree

RCS arranges revisions in an ancestral tree. The *ci* command builds this tree; an auxiliary command, *rcs* prunes it. The tree has a root revision, normally numbered 1.1, and successive revisions are numbered 1.2, 1.3, etc. The first field of a revision number is called the *release number* and the second one the *level number*. Unless given explicitly, the *ci* command assigns a new revision number by incrementing the level number of the previous revision. The release number must be incremented explicitly, using the -r option of *ci*. Assuming there are revisions 1.1, 1.2, and 1.3 in the RCS file f.c,v, the command

ci -r2.1 f.c or *ci -r2 f.c*

assigns the number 2.1 to the new revision. Later checkins without the -r option will assign the numbers 2.2, 2.3, and so on. The release number should be incremented only at major transition points in the development, for instance when a new release of a software product has been completed.

3.1. When are Branches Needed?

A young revision tree is slender: It consists of only one branch, called the trunk. As the tree ages, side branches may form. Branches are needed in the following 4 situations.

Temporary Fixes

Suppose we have a tree with 5 revisions grouped in 2 releases, as illustrated below. Revision 1.3 is in operation at customer sites while release 2 is in active development.

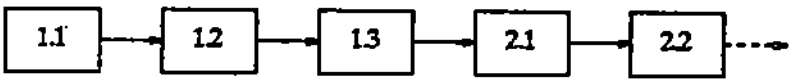

FIGURE 2: A slender revision tree.

Now imagine a customer requesting a fix of a problem in revision 1.3, although actual development has moved on to release 2. RCS does not permit us to splice in an extra revision between 1.3 and 2.1, since that would not reflect the actual development history. Instead, we must create a branch at revision 1.3, and check in the fix on that branch. The first branch starting at 1.3 has number 1.3.1, and the revisions on that branch are numbered 1.3.1.1, 1.3.1.2, etc. The double numbering is needed because we may later create another branch at 1.3, say 1.3.2. Revisions on this branch would be numbered 1.3.2.1, 1.3.2.2, and so on. We create branch 1.3.1 and add revision 1.3.1.1 by executing the following steps:

```
co -r1.3 f.c        -- check out revision 1.3
edit f.c            -- change it
ci -r1.3.1 f.c      -- check it in on branch 1.3.1
```

This sequence of commands transforms the above tree into the one given below. Note that it may be necessary to incorporate the differences between 1.3 and 1.3.1.1 into a revision at level 2. The operation *rcsmerge* automates this process (see the Appendix).

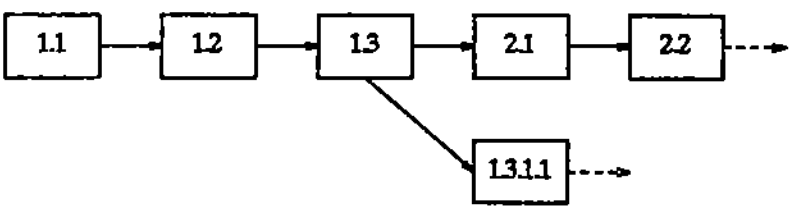

FIGURE 3: A revision tree with one side branch

Distributed Development and Customer Modifications

Suppose we have a situation as above, with revision 1.3 in operation at several customer sites, while release 2 is in development. Local modifications at a customer's site should be placed on a side branch. When the next release is distributed, it should be appended to the trunk of the customer's RCS file, and the customer can then merge his local modifications back into the new release. In the above example, a customer's RCS file would contain the following tree, assuming that he has received revision 1.3, added his

local modifications as revision 1.3.1.1, then received revision 2.4, and merged 2.4 and 1.3.1.1, resulting in 2.4.1.1.

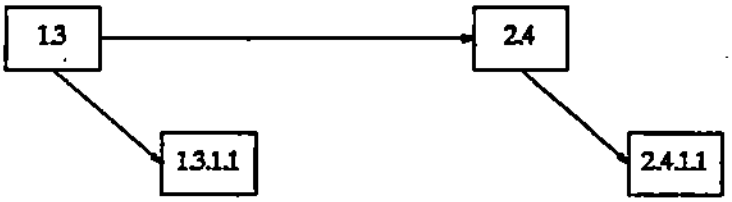

FIGURE 4: A revision tree with local modifications.

This approach is actually practiced in the CSNET project, where several universities and a company cooperate in developing a national computer network.

Parallel Development

Sometimes it is desirable to explore an alternate design or a different implementation technique in parallel with the main line development. Such development should be carried out on a side branch. The experimental changes may later be moved into the main line, or abandoned.

Conflicting Updates

A common occurrence is that one programmer has checked out a revision, but cannot complete his assignment for some reason. In the meantime, another person must perform another modification immediately. In that case, the second person should checkout the same revision, modify it, and check it in on a side branch, for later merging.

Every node in a revision tree consists of the following attributes: a revision number, a checkin date and time, the author's identification, a log entry, a state, and the actual text. All these attributes are determined at the time the revision is checked in. The state attribute indicates the status of a revision. It is set automatically to "experimental" during checkin. A revision can later be promoted to a higher status, for example "stable" or "released". The set of states is user-defined.

3.2. Revisions are Represented as Deltas

For conserving space, RCS stores revisions in the form of deltas, i.e., differences between revisions. This section discusses how RCS arranges deltas; the reader not interested in these details should skip to the next section. RCS completely hides the fact that it is implemented with deltas.

A delta is a sequence of edit commands that transforms one string into another. The deltas employed by RCS are line-based, which means that the only editing commands allowed are insertion and deletion of lines. If a single character in a line is changed, the corresponding edit script deletes that line and inserts the changed one. The program *diff* [Hun76a] produces a minimal line-based delta between 2 text files. A character-based edit script would take much longer to compute, and would not be significantly shorter.

RCS arranges deltas as follows. The most recent revision on the trunk is stored intact. All other revisions on the trunk are stored as reverse deltas. A reverse delta produces a given revision if applied to the successor revision. This implementation has the advantage that extraction of the latest revision is a simple and fast copying operation. Adding a new revision to the trunk is also fast: *Ci* simply adds the new revision intact, replaces the previous revision with a reverse delta, and keeps the rest of the old deltas. Thus, *ci* requires the computation of only one new delta.

The disadvantage with reverse deltas is that regeneration of older revisions takes time proportional to the number of deltas applied. SCCS [Roc75a], a precursor of RCS, arranges deltas such that the cost of regeneration is equally distributed over all revisions. Since usage statistics show that the most recent revision is the one that is retrieved in 95% of all cases (see Section 6), biasing checkout time in favor of the most recent revision results in significant savings. A detailed analysis of the two methods can be found in [Tic82a].

Branches need special treatment. The naive solution would be to store complete copies for the tips of all branches. Clearly, this approach would cost too much space. Instead, RCS uses *forward* deltas for branches. Regenerating a revision on a side branch proceeds as follows. First, extract 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. Figure 5 illustrates a tree with 1 side branch, where the direction of the deltas is indicated by the point of the triangle.

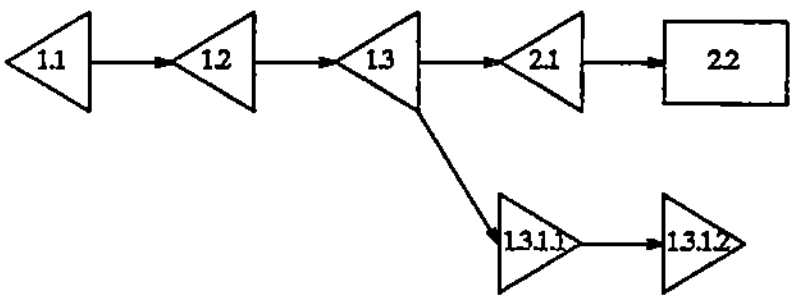

FIGURE 5: A revision tree with reverse and forward deltas.

4. Locking: A Controversial Issue

The locking mechanism for RCS was difficult to design. We first present the problem and its solution in its "pure" form, and then discuss the complications caused by "real-world" considerations.

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 checks in his revision before programmer B. Unfortunately, programmer B has not seen 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. 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* tests the lock and then removes it. At most one programmer at a time may lock a particular revision, and only this programmer may check in the succeeding revision. Thus, while a revision is locked, it is the exclusive responsibility of the locker.

An important maxim for software tools like RCS is that they must not stand in the way. This consideration leads to several weakenings of the locking mechanism. First of all, even if a revision is locked, it can still be checked out. This is necessary if other people wish to compile or process the locked revision while the next one is in preparation. The only operation they cannot do is to lock the revision, or check in the succeeding one. Second, checkin operations on other branches in the RCS file must still be possible; the locking of one revision must not affect any other revision. Third, occasionally, a revision is locked for a long period of time because the programmer is absent or otherwise unable to complete his assignment. If another programmer has to make a pressing change, he has the following 3 alternatives for making progress: a) he can find out who is holding the lock and ask that person to release it; b) he can check out the locked revision, modify it, check it in on a branch, and worry about merging the changes later; c) he can break the lock. Breaking a lock leaves a highly visible trace, namely an electronic mail message that is sent automatically to the holder of the lock, recording the breaker and a commentary requested from him. Thus, breaking locks is permitted, but will not go unnoticed.

---

† Note that this problem is entirely different from the atomicity problem. Atomicity means that concurrent update operations on the same RCS file cannot be permitted, because that may result in inconsistent data. Atomic updates are essential (and implemented in RCS), but do not solve the conflict discussed here. Even if user A's update is deposited atomically before user B's update, the latter one will still cover up the former.

If an RCS file is private, i.e., when a programmer owns an RCS file and does not expect anyone else to do checkin operations, locking is a nuisance. In this case he may disable the "strict locking feature" discussed in Section 2. This means that checkin operations executed by the owner succeed even if he holds no lock, but other programmers must still lock before checkin. In principle, this approach is unsafe, because it may lead to a race condition in which someone else succeeds in slipping in a new revision before the owner has finished the previous one. In practice, however, most private files have protection set such that other programmers are not permitted to write these files anyway, and hence this situation cannot arise.

As added protection, each RCS file contains an access list, which specifies the users who may execute update operations. If an access list is empty, normal Unix file protection applies. Thus, the access list is useful for restricting the set of people who would otherwise have update permission. Just as with locking, the access list has no effect on read-only operations like *co*. This approach is consistent with the Unix philosophy of openness, which contributes to a productive software development environment.

5. Dealing with Configurations

So far, we have seen how RCS deals with revisions of individual components. This section discusses how to handle configurations. A configuration is a set of revisions, where each revision comes from a different revision group, and the revisions are selected according to a certain criterion. For example, in order to build a functioning compiler, one has to combine the "right" revisions from the scanner, the parser, and the code generator. RCS provides a number of facilities to effect a smooth selection.

During development, the usual selection criterion is to choose the latest revision of all components. The *co* command makes this selection by default. For example, the command

```
co *,v
```

retrieves the latest revision of all RCS files in the current directory. If instead one wishes to have the latest revision of a certain release, one can specify just the release number. For instance,

```
co -r2 *,v
```

retrieves the latest revision with release number 2 from each RCS file. This situation typically arises when release 2 has been completed and development has moved on to newer releases. If the highest level number within a given release number is not the one that was distributed, one can employ the state attribute. For example,

```
co -r2 -sReleased *,v
```

retrieves the latest revision with release number 2 whose state attribute is "Released". Of course, the state attribute has to be set accordingly, using the *ci* or *rcs* commands. Another way of specifying a configuration is to use the date as the selection criterion. Suppose a release of an entire system was established on March 4, at 1:00 PM. Then the command

```
co -d"March 4, 1:00 PM" *,v
```

would check out all the components of that release, independent of the numbering. The -d option specifies a "cutoff date", i.e., the retrieved revision is the one with a checkin date that is closest to, but not younger than the date given. The options -d, -r, and -s can be combined, retrieving the latest revision that satisfies all parameters.

In large systems, a single release number or date is not sufficient for selection. For example, suppose one wishes to combine release 2 of one subsystem and release 15 of an older subsystem. Most likely, the creation dates of those releases differ also. Thus, a single revision number or date passed to the *co* command will not suffice to select the right revisions. Symbolic revision numbers help in such circumstances. Each RCS file may contain any number of symbolic names which are mapped to numeric revision numbers. For example, the commands

```
rcs -nVX:2 s,v;    rcs -nVX:15.9 t,v;
```

bind the symbol *VX* to revision number 2 in file *s,v*, and to 15.9 in *t,v*. Now the command

```
co -rVX s,v t,v
```

retrieves the latest revision in release 2 from *s,v*, and revision 15.9 from *t,v*. Judicious use of symbolic revision numbers helps organizing large configurations. Symbolic numbers may be intermixed with numeric ones. Thus, the number VX.5 would select revision 2.5 in *s,v* and branch 15.9.5 in *t,v*.

MAKE [Fel79a] is a program that processes configurations. It is driven by configuration specifications in a special file, called a "Makefile". A common application of MAKE is compiling and linking configurations. MAKE avoids redundant processing steps by comparing creation dates of source and processed objects. For example, when instructed to compile all modules of a given system, it only recompiles those source modules that were changed since they were processed last.

MAKE has been integrated with RCS. When a certain file to be processed is not present, MAKE attempts a checkout operation, performs the required processing, and then deletes the checked out file to conserve space. It inspects the modification date of the RCS file to decide whether reprocessing (and hence checkout) is necessary. Although this approach is slightly

inaccurate in that it may cause redundant compilations (for instance, deleting an old revision changes the modification date of the RCS file and triggers a recompilation), it is conservative in that it does not miss any changes.

MAKE totally ignores an RCS file if the corresponding working file is present. In this situation, a user is most likely in the process of editing the file, and it is therefore the most recent revision to be processed. Accordingly, MAKE does not overwrite it with an implicit checkout, nor delete it after processing.

The selection criteria for configurations discussed earlier can be passed to MAKE either as parameters, or can be embedded directly into the Makefile. Experience with MAKE and RCS has shown that it is convenient to apply RCS to Makefiles as well, resulting in multiple revisions of configuration specifications. Whenever a configuration is baselined or distributed, the best approach is to unambiguously fix the selection criteria with a symbolic revision number, to embed that selection into the Makefile, and then to check in the Makefile. With this approach, old configurations can be regenerated easily and reliably.

6. Usage Statistics

The following usage statistics were collected on 2 DEC VAX-11/780 computers of the Purdue Computer Science Department. Both machines are used for research purposes only. Thus, the data reflects an environment in which the majority of projects involve prototyping and advanced software development.

For the first experiment, we instrumented the *ci* and *co* operations to log the number of backward and forward deltas applied. The data were collected during a 13 month period from Dec. 1982 to Dec. 1983. The following table summarizes the results.

operation	total op's	total deltas applied	mean deltas applied	op's w. > 1 delta		branch op's	
co	7867	9320	1.18	509	(6%)	203	(3%)
ci	3468	2207	0.64	85	(2%)	75	(2%)
ci & co	11335	11527	1.02	594	(5%)	278	(2%)

Table 1: Statistics for *co* and *ci* operations.

The first 2 lines shows statistics for checkout and checkin, which are the read and write operations on RCS files, respectively. The last line shows the combination. Recall that *ci* performs an implicit checkout to produce a revision for computing the delta. Checkin of the initial revision does not require a delta, and hence no implicit checkout is performed in that case. In all measures presented, the most recent revision (stored intact) counts as one delta.

The checkout operation is executed more than twice as frequently as the checkin operation. Column 4 gives the mean number of deltas applied per operation. Note that for checkin, the mean number of deltas applied is less than 1. The reasons are that initializations require no delta, and that the only time *ci* requires more than 1 delta is for branches. Since branches were not used often (compare column 6), almost all *ci* operations produced just 0 or 1 delta. Column 5 shows the actual number of operations that applied more than 1 delta.

The last three columns demonstrate that the most recent revision is by far the most frequently accessed. For RCS, the checkout for this revision is a simple copying operation, which is the absolut minimum given the copy-semantics of *co*. We expect that in a production environment access to older revisions and branches is more common. However, even if access to older deltas were twice as frequent, reverse deltas still would have a substantial advantage over other arrangements.

The second experiment, conducted in March of 1984, involved surveying the existing RCS files on the 2 machines. The goal was to determine the mean number of revisions per RCS file, as well as the space consumed by them. Table 2 below shows the results.

	total RCS files	total rev's	mean rev's	size of RCS files	size of rev's	overhead
all files	8033	11133	1.39	6156	5585	1.10
> 1 deltas	1477	4578	3.10	8074	6041	1.34

Table 2: Statistics for RCS files.

The mean number of revisions per RCS file is 1.39. Columns 5 and 6 show the mean sizes (in bytes) of an RCS file and of the latest revision of each RCS file, respectively. The "overhead" column contains the ratio of the mean sizes. Assuming that all revisions in an RCS file are approximately the same size, this ratio gives a measure of the space consumed by the extra revisions.

In our sample, over 80% of the RCS files contained only a single revision! The reason is that our systems programmers routinely check in all source files on the distribution tapes, even though they may never touch them again. To get a better indication of how much space savings are possible with deltas, we recomputed all measures with those files that contained 2 or more revisions. Only for those files is RCS necessary. As shown in the second line, the average number of revisions for those files is 3.10, with an overhead of 1.34. This means that the extra 2.10 deltas require 34% extra space, or 16% per extra revision. Rochkind [Roc75a] measured the space consumed by SCCS, and reported an average of 5 revisions per group and an overhead of 1.37 (or about 9% per extra revision). Glasser [Gla78a] reported an average of 7 revisions per group in a certain large project, but provided no overhead figure.

Both Rochkind's and our measurements confirm that the space needed for extra revisions is small. Using delta techniques, the luxury of keeping multiple revisions online becomes affordable. Introducing a system like RCS may actually reduce storage requirements, because programmers often save multiple revisions anyway. Providing delta storage with a system like RCS may therefore free a considerable amount of space.

7. Historical Notes

The need to keep backup copies of software arose when programs and data were no longer stored on paper media, but were entered from terminals and stored on disk. Backup copies were needed for reliability, and editors soon saved a backup copy for every file touched. This strategy is valuable for short-term backups, but not suitable for long-term version control, since an existing backup copy is overwritten whenever the corresponding file is edited again.

As software engineers realized that several older revisions must be kept online for effective software maintenance, a number of approaches evolved. One simple technique is to never delete a file; editing a file simply creates a new file with the same name, but with a different sequence number. This approach turns out to be impractical for several reasons. First, it is prohibitively expensive in terms of storage costs, since it does not use any data compression techniques. Second, indiscriminately keeping every change results in too many revisions and programmers have difficulties telling them apart. The proliferation of revisions forces programmers to spend much time on finding and deleting unwanted ones. Third, most of the support functions like locking, logging, and identification described in this paper are not available.

An alternative approach is to separate editing from revision control. Editing generates a one-level backup as before, but the user must issue an explicit checkin command for permanently storing and freezing a revision. An early system employing this technique was CLEAR/CASTER [Bro70a], which maintains a data base of programs, specifications, documentation, and messages. It even uses a delta technique for storing multiple versions. CLEAR/CASTER was intended to provide control over the development process from a management viewpoint. The software tools SCCS [Roc75a], SDC [Hab79a], and CMS [DEC82a] not only store multiple revisions of source text, but also record a log entry for each revision and prevent conflicting updates. SCCS also provides a simple form of automatic identification by expanding certain markers. Both SCCS and CMS are in widespread use.

Tools that handle multiple versions of configurations are still in a state of flux. For example, the combination of RCS and MAKE is not satisfactory, because MAKE cannot determine

accurately which source revisions and processing steps were used in generating a given derived object. MAKE's reliance on time stamps and naming conventions limits its capabilities. Cooprider [Coo78a] describes an extremely flexible system for version control. Although his system is too complex for practical use, it explores many important concepts for version control. A basic model for describing multi-version/multi-configuration systems appears in [Tic82b]. The model is based on AND/OR graphs, where AND nodes represent configurations, and OR nodes represent version groups. General capabilities of version control systems can be found in the Stoneman Document [Bux80a], which outlines the requirements for Ada programming environments.

Availability: RCS is being distributed with Berkeley Unix 4.2. RCS has been ported to a large number of Unix and Unix-like operating systems. Sites that do not have a Unix 4.2 license may request RCS from the author. The modified MAKE program with the auto-checkout feature is not included in Unix 4.2, and must be obtained from the author.

Acknowledgements: Many people have helped make RCS a success by contributed criticisms, suggestions, and even software. The list of people is too long to be reproduced here, but my sincere thanks go to all of them.

Appendix: Synopsis of RCS Operations

ci – check in revisions

Ci stores the contents of a working file into the corresponding RCS file as a new revision. If the RCS file doesn't exist, *ci* creates it. *Ci* removes the working file, unless one of the options -u or -l is present. For each checkin, *ci* asks for a commentary describing the changes relative to the previous revision.

Ci assigns the revision number given by the -r option; if that option is missing, it derives the number from the lock held by the user; if there is no lock and locking is not strict, *ci* increments the number of the latest revision on the trunk. A side branch can only be started by explicitly specifying its number with the -r option during checkin.

Ci also determines whether the revision to be checked in is different from the previous one, and asks whether to proceed if not. This facility simplifies checkin operations for large systems, because one need not remember which files were changed.

The option -k searches the checked in file for identification markers containing the attributes revision number, checkin date, author, and state, and assigns these to the new revision rather than computing them. This option is useful for software distribution: Recipients of distributed software using RCS should check in updates with the -k option. This convention guarantees that revision numbers, checkin dates, etc., are the same at all sites.

co – check out revisions

Co retrieves revisions according to revision number, date, author, and state attributes. It either places the revision into the working file, or prints it on the std. output. *Co* always expands the identification markers.

ident – extract identification markers

Ident extracts the identification markers expanded by *co* from any file and prints them.

rcs – change RCS file attributes

Rcs is an administrative operation that changes access lists, locks and unlocks revisions, breaks locks, toggles the strict-locking feature, sets state attributes and symbolic revision numbers, changes the description, and deletes revisions. A revision can only be deleted if it is not the fork of a side branch. This restriction is necessary to avoid splitting of the tree into disconnected pieces.

rcsdiff – compare revisions

Rcsdiff compares two revisions and prints their difference, using the Unix tool *diff* [UCB83a]. One of the revisions compared may be checked out. This command is useful for finding out about changes.

rcsmerge – merge revisions

Rcsmerge merges two revision with respect to a third. Suppose *rev1*, *rev2*, and *rev3* are revisions, where *rev1* is a common ancestor of *rev2* and *rev3*. Merging *rev2* and *rev3* with respect to *rev1* has the effect of incorporating all changes between *rev1* and *rev2* into a copy of *rev3*. The merge operation is implemented with a 3-way file comparison, using an adaptation of the Unix tool *diff3* [UCB83a]. 3-way file comparison determines the segments of lines that are (a) the same in all three revisions, or (b) the same in 2 revisions, or (c) different in all three. For all segments of type (b) where *rev2* is the differing revision, the segment of *rev2* replaces the corresponding segment of *rev3*. Type (c) indicates an overlapping change, is flagged as an error, and requires user intervention to select the correct alternative.

rlog – print information about RCS files

Rlog prints the log messages and other information in an RCS file.

References

Bro70a. Brown, H.B., "The Clear/Caster System," *Nato Conference on Software Engineering, Rome*, (1970).

Bux80a. Buxton, John N., *Requirements for Ada Programming Support Environments ("Stoneman")*, US Department of Defense (February 1980).

Coo78a. Cooprider, Lee W., *The Representation of Families of Programmed Systems*, PhD thesis, Carnegie-Mellon University, Department of Computer Science (1978).

DEC82a. DEC,, *Code Management System*, Digital Equipment Corporation (1982). Document No. EA-23134-82

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

Gla78a. Glasser, Alan L., "The Evolution of a Source Code Control System," *Software Engineering Notes* 3(5) p. 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).

Hun76a. Hunt, James W. and McIlroy, M. D., "An Algorithm for Differential File Comparison," Computing Science Technical Report No. 41, Bell Laboratories (June 1976).

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

Tic82b. Tichy, Walter F., "A Data Model for Programming Support Environments and its Application," in *Automated Tools for Information System Design and Development*, ed. Hans-Jochen Schneider and Anthony I. Wasserman,North-Holland Publishing Company, Amsterdam (1982).

Tic82a. Tichy, Walter F., "Design, Implementation, and Evaluation of a Revision Control System," pp. 58-67 in *Proceedings of the 6th International Conference on Software Engineering*, IPS, ACM, IEEE, NBS (September 1982).

UCB83a. UCB,, *UNIX Programmer's Manual, 4.2 BSD*, University of California, Berkeley (Aug. 1983).