RCS—A System for Version Control

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

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. RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging and identification of revisions, and it provides selection mechanisms for composing configurations. This paper introduces basic version control concepts and discusses the practice of version control using RCS. For conserving space, RCS stores deltas, i.e., differences between successive revisions. Several delta storage methods are discussed. Usage statistics show that RCS's delta storage method is space and time efficient. The paper concludes with a detailed survey of version control tools.

Keywords: configuration management, history management, version control, revisions, deltas.

1. Introduction

Version control is the task of keeping software systems consisting of many versions and configurations well organized. The Revision Control System (RCS) is a set of UNIX commands that assist with that task.

RCS' primary function is to manage *revision groups*. A revision group is a set of text documents, called *revisions*, that evolved from each other. A new revision is created by manually editing an existing one. RCS organizes the revisions into an ancestral tree. The initial revision is the root of the tree, and the tree edges indicate from which revision a given one evolved. Besides managing individual revision groups, RCS provides flexible selection functions for composing configurations. RCS may be combined with MAKE[1], resulting in a powerful package for version control.

RCS also offers facilities for merging updates with customer modifications, for distributed software development, and for automatic identification. Identification is the 'stamping' of revisions and configurations with unique markers. These markers are akin to serial numbers, telling software maintainers

An earlier version of this paper was published in *Software—Practice & Experience* **15**, 7 (July 1985), 637-654.

unambiguously which configuration is before them.

RCS is designed for both production and experimental environments. In production environments, access controls detect update conflicts and prevent overlapping changes. In experimental environments, where strong controls are counterproductive, it is possible to loosen the controls.

Although RCS was originally intended for programs, 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.

This paper discusses the practice of version control using RCS. It also introduces basic version control concepts, useful for clarifying current practice and designing similar systems. Revision groups of individual components are treated in the next three sections, and the extensions to configurations follow. Because of its size, a survey of version control tools appears at the end of the paper.

2. Getting started with RCS

Suppose a text file *f.c* is to be placed under control of RCS. Invoking the check-in 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 check-in 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, execute the check-out command:

co f.c

This command extracts the latest revision from the revision group *f.c,v* and writes it into *f.c*. The file *f.c* can now be edited and, when finished, checked back in with *ci*:

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 check-in 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, the revision should have been locked during the check-out by using the option *−l*:

co $-l$ *f.c*

Of course it is too late now for the check-out with locking, because *f.c* has already been changed; checking out the file again would overwrite the 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 check-out is sometimes useful for backing up to the previous revision.) To be able to proceed with the check-in in the present case, first execute

rcs $-l$ *f.c*

This command retroactively locks the latest revision, unless someone else locked it in the meantime. In this case, the two programmers involved have to negotiate whose modifications should take precedence.

If an RCS file is private, i.e., if only the owner of the file is expected to deposit revisions into it, the strict locking feature is unnecessary and may be disabled. If strict locking is disabled, the owner of the RCS file need not have a lock for check-in. For safety reasons, 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.

To reduce the clutter in a working directory, all RCS files can be moved to a subdirectory with the name *RCS*. RCS commands look first into that directory for RCS files. All the commands presented above work with the *RCS* subdirectory without change.†

It may be undesirable that *ci* deletes the working file. For instance, sometimes one would like to save the current revision, but continue editing. Invoking

ci $-l$ *f.c*

checks in *f.c* as usual, but performs an additional check-out with locking afterwards. Thus, the working file does not disappear after the check-in. Similarly, the option $-u$ does a check-in followed by a check-out without locking. This option is useful if the file is needed for compilation after the check-in. Both options update the identification markers in the working file (see below).

Besides the operations *ci* and *co*, RCS provides the following commands:

ident	extract identification markers
rcs	change RCS file attributes
rcsclean	remove unchanged working files (optional)
rcsdiff	compare revisions
rcsfreeze	record a configuration (optional)
rcsmerge	merge revisions
rlog	read log messages and other information in RCS files

† 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.

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, place the marker

Id

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

$Id: filename revisionnumber date time author state locker $

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

static char rcsid[] = "Id";

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 a complete and unambiguous component list, from 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 Id. The marker

Log

has a similar function. It accumulates the log messages that are requested during check-in. Thus, one can maintain the complete history of a revision directly inside it, by enclosing it in a comment. Figure 1 is an edited version of a log contained in revision 4.1 of the file *ci.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  1983/05/10 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  1983/02/15 15:25:44  wft
 * Added call to fastcopy() to copy remainder of RCS file.
 *
 * Revision 3.8  1983/01/14 15:34:05  wft
 * Added ignoring of interrupts while new RCS file is renamed;
 * avoids deletion of RCS files by interrupts.
 *
 * Revision 3.7  1982/12/10 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.

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; the 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 check-ins 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 a tree has 5 revisions grouped in 2 releases, as illustrated in Figure 2. Revision 1.3, the last one of release 1, is in operation at customer sites, while release 2 is in active development.

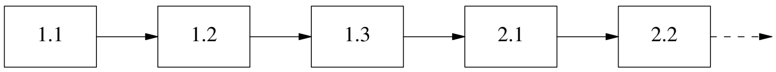

Figure 2. A slender revision tree.

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 an extra revision to be spliced in between 1.3 and 2.1, since that would not reflect the actual development history. Instead, 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 to allow for another branch at 1.3, say 1.3.2. Revisions on the second branch would be numbered 1.3.2.1, 1.3.2.2, and so on. The following steps create branch 1.3.1 and add revision 1.3.1.1:

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 tree of Figure 2 into the one in Figure 3. 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).

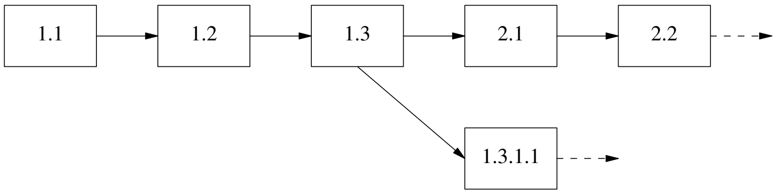

Figure 3. A revision tree with one side branch

Distributed development and customer modifications

Assume a situation as in Figure 2, where revision 1.3 is in operation at several customer sites, while release 2 is in development. Customer sites should use RCS to store the distributed software. However, customer modifications should not be placed on the same branch as the distributed source; instead, they should be placed on a side branch. When the next software distribution arrives, it should be appended to the trunk of the customer's RCS file, and the customer can then merge the local modifications back into the new release. In the above example, a customer's RCS file would contain the following tree, assuming that the customer 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.

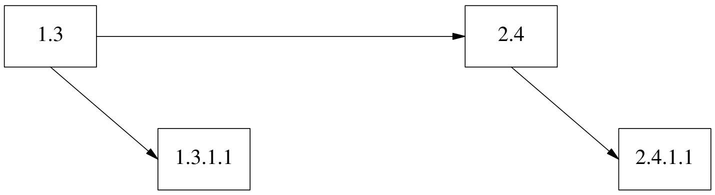

Figure 4. A customer's revision tree with local modifications.

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 the assignment for some reason. In the meantime, another person must perform another modification immediately. In that case, the second person should check-out 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 check-in 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 check-in. 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., as differences between revisions. The user interface completely hides this fact.

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 edit commands allowed are insertion and deletion of lines. If a single character in a line is changed, the edit scripts consider the entire line changed. The program *diff*[2] produces a small, line-based delta between pairs of text files. A character-based edit script would take much longer to compute, and would not be significantly shorter.

Using deltas is a classical space-time tradeoff: deltas reduce the space consumed, but increase access time. However, a version control tool should impose as little delay as possible on programmers. Excessive delays discourage the use of version controls, or induce programmers to take shortcuts that compromise system integrity. To gain reasonably fast access time for both editing and compiling, RCS arranges deltas in the following way. The most recent revision on the trunk is stored intact. All other revisions on the trunk are stored as reverse deltas. A reverse delta describes how to go backward in the development history: it produces the desired revision if applied to the successor of that revision. This implementation has the advantage that extraction of the latest revision is a simple and fast copy 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.

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; secondly, apply reverse deltas until the fork revision for the branch is obtained; thirdly, apply forward deltas until the desired branch revision is reached. Figure 5 illustrates a tree with one side branch.

Triangles pointing to the left and right represent reverse and forward deltas, respectively.

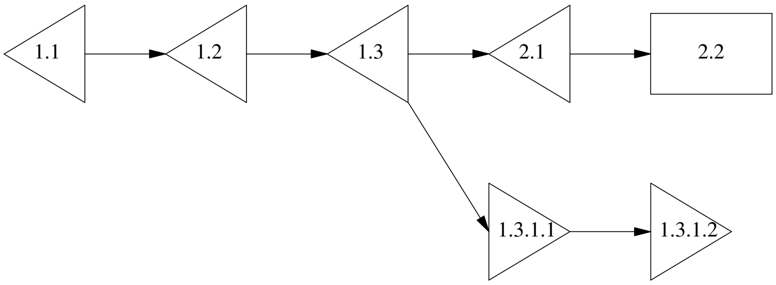

Figure 5. A revision tree with reverse and forward deltas.

Although implementing fast check-out for the latest trunk revision, this arrangement has the disadvantage that generation of other revisions takes time proportional to the number of deltas applied. For example, regenerating the branch tip in Figure 5 requires application of five deltas (including the initial one). Since usage statistics show that the latest trunk revision is the one that is retrieved in 95 per cent of all cases (see the section on usage statistics), biasing check-out time in favor of that revision results in significant savings. However, careful implementation of the delta application process is necessary to provide low retrieval overhead for other revisions, in particular for branch tips.

There are several techniques for delta application. The naive one is to pass each delta to a general-purpose text editor. A prototype of RCS invoked the UNIX editor *ed* both for applying deltas and for expanding the identification markers. Although easy to implement, performance was poor, owing to the high start-up costs and excess generality of *ed*. An intermediate version of RCS used a special-purpose, stream-oriented editor. This technique reduced the cost of applying a delta to the cost of checking out the latest trunk revision. The reason for this behavior is that each delta application involves a complete pass over the preceding revision.

However, there is a much better algorithm. Note that the deltas are line oriented and that most of the work of a stream editor involves copying unchanged lines from one revision to the next. A faster algorithm avoids unnecessary copying of character strings by using a *piece table*. A piece table is a one-dimensional array, specifying how a given revision is 'pieced together' from lines in the RCS file. Suppose piece table PT_r represents revision *r*. Then $PT_r[i]$ contains the starting position of line *i* of revision *r*. Application of the next delta transforms piece table PT_r into PT_{r+1}. For instance, a delete command removes a series of entries from the piece table. An insertion command inserts new entries, moving the entries following the insertion point further down the array. The inserted entries point to the text lines in the delta. Thus, no I/O is involved except for reading the delta itself. When all deltas have been applied to the piece table, a sequential pass through the table looks up each line in the RCS file and copies it to the output file, updating identification markers at the same time. Of course, the RCS file must permit random access, since the copied lines are scattered throughout that file. Figure 6 illustrates an RCS file with two revisions and the corresponding piece tables.

Figure 6 is not available.

Figure 6. An RCS file and its piece tables

The piece table approach has the property that the time for applying a single delta is roughly determined by the size of the delta, and not by the size of the revision. For example, if a delta is 10 per cent of the size of a revision, then applying it takes only 10 per cent of the time to generate the latest trunk revision. (The stream editor would take 100 per cent.)

There is an important alternative for representing deltas that affects performance. SCCS[3], a precursor of RCS, uses *interleaved* deltas. A file containing interleaved deltas is partitioned into blocks of lines. Each block has a header that specifies to which revision(s) the block belongs. The blocks are sorted out in such a way that a single pass over the file can pick up all the lines belonging to a given revision. Thus, the regeneration time for all revisions is the same: all headers must be inspected, and the associated blocks either copied or skipped. As the number of revisions increases, the cost of retrieving any revision is much higher than the cost of checking out the latest trunk revision with reverse deltas. A detailed comparison of SCCS's interleaved deltas and RCS's reverse deltas can be found in Reference 4. This reference considers the version of RCS with the stream editor only. The piece table method improves performance further, so that RCS is always faster than SCCS, except if 10 or more deltas are applied.

Additional speed-up for both delta methods can be obtained by caching the most recently generated revision, as has been implemented in DSEE.[5] With caching, access time to frequently used revisions can approach normal file access time, at the cost of some additional space.

4. Locking: A Controversial Issue

The locking mechanism for RCS was difficult to design. The problem and its solution are first presented in their 'pure' form, followed by a discussion of the complications caused by 'real-world' considerations.

RCS must prevent two or more persons from depositing competing changes of the same revision. Suppose two programmers check out revision 2.4 and modify it. Programmer A checks in a revision before programmer B. Unfortunately, programmer B has not seen A's changes, so the effect is that A's changes are covered up 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), the revision should be checked out and locked, using the $-l$ option on *co*. On subsequent check-in, *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 of making progress with a project. 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 inspect the locked revision while the next one is in preparation. The only operations they cannot do are to lock the revision or to check in the succeeding one. Secondly, check-in operations on other branches in the RCS file are still possible; the locking of one revision does not affect any other revision. Thirdly, revisions are occasionally locked for a long period of time because a programmer is absent or otherwise unable to complete the assignment. If another programmer has to make a pressing change, there are the following three alternatives for making progress: a) find out who is holding the lock and ask that person to release it; b) check out the locked revision, modify it, check it in on a branch, and merge the changes later; c) 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 tolerated under certain circumstances, but will not go unnoticed. Experience has shown that the automatic mail message attaches a high enough stigma to lock breaking, such that programmers break locks only in real emergencies, or when a co-worker resigns and leaves locked revisions behind.

If an RCS file is private, i.e., when a programmer owns an RCS file and does not expect anyone else to perform check-in operations, locking is an unnecessary nuisance. In this case, the 'strict locking feature' discussed earlier may be disabled, provided that file protection is set such that only the owner may write the RCS file. This has the effect that only the owner can check-in revisions, and that no lock is needed for doing so.

As added protection, each RCS file contains an access list that specifies the users who may execute update operations. If an access list is empty, only 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 such as *co*. This approach is consistent with

‡ 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.

the UNIX philosophy of openness, which contributes to a productive software development environment.

5. Configuration Management

The preceding sections described 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, the 'right' revisions from the scanner, the parser, the optimizer and the code generator must be combined. RCS, in conjunction with MAKE, provides a number of facilities to effect a smooth selection.

5.1. RCS Selection Functions

Default 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 on the default branch of each RCS file in the current directory. The default branch is usually the trunk, but may be set to be a side branch. Side branches as defaults are needed in distributed software development, as discussed in the section on the RCS revision tree.

Release based selection

Specifying a release or branch number selects the latest revision in that release or branch. For instance,

*co −r2 *,v*

retrieves the latest revision with release number 2 from each RCS file. This selection is convenient if a release has been completed and development has moved on to the next release.

State and author based selection

If the highest level number within a given release number is not the desired one, the state attribute can help. 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 appropriately, using the *ci* or *rcs* commands. Another alternative is to select a revision by its author, using the *−w* option.

Date based selection

Revisions may also be selected by date. Suppose a release of an entire system was completed and current on March 4, at 1:00 p.m. local time. Then the command

*co −d'March 4, 1:00 pm LT' *,v*

checks out all the components of that release, independent of the numbering. The *−d* option specifies a 'cutoff date', i.e., the revision selected has a check-in date that is closest to, but not after the date given.

Name based selection

The most powerful selection function is based on assigning symbolic names to revisions and branches. In large systems, a single release number or date is not sufficient to collect the appropriate revisions from all groups. For example, suppose one wishes to combine release 2 of one subsystem and release 15 of another. 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 solve this problem. Each RCS file may contain a set of symbolic names that are mapped to numeric revision numbers. For example, assume the symbol *V3* is bound to release number 2 in file *s,v*, and to revision number 15.9 in *t,v*. Then the single command

co −rV3 s,v t,v

retrieves the latest revision of release 2 from *s,v*, and revision 15.9 from *t,v*. In a large system with many modules, checking out all revisions with one command greatly simplifies configuration management.

Judicious use of symbolic revision numbers helps with organizing large configurations. A special command, *rcsfreeze*, assigns a symbolic revision number to a selected revision in every RCS file. *Rcsfreeze* effectively freezes a configuration. The assigned symbolic revision number selects all components of the configuration. If necessary, symbolic numbers may even be intermixed with numeric ones. Thus, *V3.5* in the above example would select revision 2.5 in *s,v* and branch 15.9.5 in *t,v*.

The options *−r*, *−s*, *−w* and *−d* may be combined. If a branch is given, the latest revision on that branch satisfying all conditions is retrieved; otherwise, the default branch is used.

5.2. Combining MAKE and RCS

MAKE[1] is a program that processes configurations. It is driven by configuration specifications recorded in a special file, called a 'Makefile'. 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 extended with an auto-checkout feature for RCS.* When a certain file to be

* This auto-checkout extension is available only in some versions of MAKE, e.g. GNU MAKE.

processed is not present, MAKE attempts a check-out operation. If successful, MAKE performs the required processing, and then deletes the checked out file to conserve space. The selection parameters discussed above can be passed to MAKE either as parameters, or directly embedded in the Makefile. MAKE has also been extended to search the subdirectory named *RCS* for needed files, rather than just the current working directory. However, if a working file is present, MAKE totally ignores the corresponding RCS file and uses the working file. (In newer versions of MAKE distributed by AT&T and others, auto-checkout can be achieved with the rule DEFAULT, instead of a special extension of MAKE. However, a file checked out by the rule DEFAULT will not be deleted after processing. *Rcsclean* can be used for that purpose.)

With auto-checkout, RCS/MAKE can effect a selection rule especially tuned for multi-person software development and maintenance. In these situations, programmers should obtain configurations that consist of the revisions they have personally checked out plus the latest checked in revision of all other revision groups. This schema can be set up as follows.

Each programmer chooses a working directory and places into it a symbolic link, named *RCS*, to the directory containing the relevant RCS files. The symbolic link makes sure that *co* and *ci* operations need only specify the working files, and that the Makefile need not be changed. The programmer then checks out the needed files and modifies them. If MAKE is invoked, it composes configurations by selecting those revisions that are checked out, and the rest from the subdirectory *RCS*. The latter selection may be controlled by a symbolic revision number or any of the other selection criteria. If there are several programmers editing in separate working directories, they are insulated from each other's changes until checking in their modifications.

Similarly, a maintainer can recreate an older configuration by starting to work in an empty working directory. During the initial MAKE invocation, all revisions are selected from RCS files. As the maintainer checks out files and modifies them, a new configuration is gradually built up. Every time MAKE is invoked, it substitutes the modified revisions into the configuration being manipulated.

A final application of RCS is to use it for storing Makefiles. Revision groups of Makefiles represent multiple versions of configurations. Whenever a configuration is baselined or distributed, the best approach is to unambiguously fix the configuration with a symbolic revision number by calling *rcsfreeze*, to embed that symbol into the Makefile, and to check in the Makefile (using the same symbolic revision number). With this approach, old configurations can be regenerated easily and reliably.

6. Usage Statistics

The following usage statistics were collected on two DEC VAX-11/780 computers of the Purdue Computer Science Department. Both machines are mainly used for research purposes. Thus, the data reflect an environment in which the majority of projects involve prototyping and advanced software development, but relatively little long-term maintenance.

For the first experiment, the *ci* and *co* operations were instrumented 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. Table I summarizes the results.

Operation	Total operations	Total deltas applied	Mean deltas applied	Operations with >1 delta		Branch operations	
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 I. Statistics for *co* and *ci* operations.

The first two lines show statistics for check-out and check-in; the third line shows the combination. Recall that *ci* performs an implicit check-out to obtain a revision for computing the delta. In all measures presented, the most recent revision (stored intact) counts as one delta. The number of deltas applied represents the number of passes necessary, where the first 'pass' is a copying step.

Note that the check-out operation is executed more than twice as frequently as the check-in operation. The fourth column gives the mean number of deltas applied in all three cases. For *ci*, the mean number of deltas applied is less than one. The reasons are that the initial check-in requires no delta at all, and that the only time *ci* requires more than one delta is for branches. Column 5 shows the actual number of operations that applied more than one delta. The last column indicates that branches were not used often.

The last three columns demonstrate that the most recent trunk revision is by far the most frequently accessed. For RCS, check-out of this revision is a simple copy operation, which is the absolute minimum given the copy-semantics of *co*. Access to older revisions and branches is more common in non-academic environments, yet even if access to older deltas were an order of magnitude more frequent, the combined average number of deltas applied would still be below 1.2. Since RCS is faster than SCCS until up to 10 delta applications, reverse deltas are clearly the method of choice.

The second experiment, conducted in March of 1984, involved surveying the existing RCS files on our two machines. The goal was to determine the mean number of revisions per RCS file, as well as the space consumed by them. Table II shows the results. (Tables I and II were produced at different times and are unrelated.)

	Total RCS files	Total revisions	Mean revisions	Mean size of RCS files	Mean size of revisions	Overhead
All files	8033	11133	1.39	6156	5585	1.10
Files with ≥ 2 deltas	1477	4578	3.10	8074	6041	1.34

Table II. 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 per cent 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, all measures with those files that contained 2 or more revisions were recomputed. 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 per cent extra space, or 16 per cent per extra revision. Rochkind[3] 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 cent per extra revision). In a later paper, Glasser[6] observed an average of 7 revisions per group in a single, large project, but provided no overhead figure. In his paper on DSEE[5], Leblang reported that delta storage combined with blank compression results in an overhead of a mere 1−2 per cent per revision. Since leading blanks accounted for about 20 per cent of the surveyed Pascal programs, a revision group with 5−10 members was smaller than a single cleartext copy.

The above observations demonstrate clearly that the space needed for extra revisions is small. With delta storage, the luxury of keeping multiple revisions online is certainly affordable. In fact, introducing a system with delta storage may reduce storage requirements, because programmers often save back-up copies anyway. Since back-up copies are stored much more efficiently with deltas, introducing a system such as RCS may actually free a considerable amount of space.

7. Survey of Version Control Tools

The need to keep back-up copies of software arose when programs and data were no longer stored on paper media, but were entered from terminals and stored on disk. Back-up copies are desirable for reliability, and many modern editors automatically save a back-up copy for every file touched. This strategy is valuable for short-term back-ups, but not suitable for long-term version control, since an existing back-up copy is overwritten whenever the corresponding file is edited.

Tape archives are suitable for long-term, offline storage. If all changed files are dumped on a back-up tape once per day, old revisions remain accessible. However, tape archives are unsatisfactory for version control in several ways. First, backing up the file system every 24 hours does not capture intermediate revisions. Secondly, the old revisions are not online, and accessing them is tedious and time-consuming. In particular, it is impractical to compare several old revisions of a group, because that may require mounting and searching several tapes. Tape archives are important fail-safe tools in the event of catastrophic disk failures or accidental deletions, but they are ill-suited for version control. Conversely, version control tools do not obviate the need for tape archives.

A natural technique for keeping several old revisions online 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 technique, available as an option in DEC's VMS operating system, turns out to be inadequate for version control. First, it is prohibitively expensive in terms of storage costs, especially since no data compression techniques are employed. Secondly, indiscriminately storing every change produces too many revisions, and programmers have difficulties distinguishing them. The proliferation of revisions forces programmers to spend much time on finding and deleting useless files. Thirdly, most of the support functions like locking, logging, revision selection, and identification described in this paper are not available.

An alternative approach is to separate editing from revision control. The user may repeatedly edit a given revision, until freezing it with an explicit command. Once a revision is frozen, it is stored permanently and can no longer be modified. (In RCS, freezing a revisions is done with *ci*.) Editing a frozen revision implicitly creates a new one, which can again be changed repeatedly until it is frozen itself. This approach saves exactly those revisions that the user considers important, and keeps the number of revisions manageable. IBM's CLEAR/CASTER[7], AT&T's SCCS[3], CMU's SDC[8] and DEC's CMS[9], are examples of version control systems using this approach. CLEAR/CASTER maintains a data base of programs, specifications, documentation and messages, using deltas. Its goal is to provide control over the development process from a management viewpoint. SCCS stores multiple revisions of source text in an ancestral tree, records a log entry for each revision, provides access control, and has facilities for uniquely identifying each revision. An efficient delta technique reduces the space consumed by each revision group. SDC is much simpler than SCCS because it stores not more than two revisions. However, it maintains a complete log for all old revisions, some of which may be on back-up tape. CMS, like SCCS, manages tree-structured revision groups, but offers no identification mechanism.

Tools for dealing with configurations are still in a state of flux. SCCS, SDC and CMS can be combined with MAKE or MAKE-like programs. Since flexible selection rules are missing from all these tools, it is sometimes difficult to specify precisely which revision of each group should be passed to MAKE for building a desired configuration. The Xerox Cedar system[10] provides a 'System Modeller' that can rebuild a configuration from an arbitrary set of module revisions. The revisions of a module are only distinguished by creation time, and there is no tool for managing groups. Since the selection rules are primitive, the System Modeller appears to be somewhat tedious to use. Apollo's DSEE[5] is a sophisticated software engineering environment. It manages revision groups in a way similar to SCCS and CMS. Configurations are built using 'configuration threads'. A configuration thread states which revision of each group named in a configuration should be chosen. A configuration thread may contain dynamic specifiers (e.g., 'choose the revisions I am currently working on, and the most recent revisions otherwise'), which are bound automatically at build time. It also provides a notification mechanism for alerting maintainers about the need to rebuild a system after a change.

RCS is based on a general model for describing multi-version/multi-configuration systems[11]. The model describes systems using AND/OR graphs, where AND nodes represent configurations, and OR nodes represent version groups. The model gives rise to a suit of selection rules for composing configurations, almost all of which are implemented in RCS. The revisions selected by RCS are passed to MAKE for configuration building. Revision group management is modelled after SCCS. RCS retains SCCS's best features, but offers a significantly simpler user interface, flexible selection rules, adequate integration with MAKE and improved identification. A detailed comparison of RCS and SCCS appears in Reference 4.

An important component of all revision control systems is a program for computing deltas. SCCS and RCS use the program *diff*[2], which first computes the longest common substring of two revisions, and then produces the delta from that substring. The delta is simply an edit script consisting of deletion and insertion commands that generate one revision from the other.

A delta based on a longest common substring is not necessarily minimal, because it does not take advantage of crossing block moves. Crossing block moves arise if two or more blocks of lines (e.g., procedures) appear in a different order in two revisions. An edit script derived from a longest common substring first deletes the shorter of the two blocks, and then reinserts it. Heckel[12] proposed an algorithm for detecting block moves, but since the algorithm is based on heuristics, there are conditions under which the generated delta is far from minimal. DSEE uses this algorithm combined with blank compression, apparently with satisfactory overall results. A new algorithm that is guaranteed to produce a minimal delta based on block moves appears in Reference 13. A future release of RCS will use this algorithm.

Acknowledgements: Many people have helped make RCS a success by contributed criticisms, suggestions, corrections, and even whole new commands (including manual pages). The list of people is too long to be reproduced here, but my sincere thanks for their help and goodwill goes 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 check-in, *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 check-in.

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 check-in 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, check-in 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, check-in 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 standard 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, unlocks, 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.

rcsclean **– clean working directory**

Rcsclean removes working files that were checked out but never changed.*

rcsdiff **– compare revisions**

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

rcsfreeze **– freeze a configuration**

Rcsfreeze assigns the same symbolic revision number to a given revision in all RCS files. This command is useful for accurately recording a configuration.*

rcsmerge **– merge revisions**

Rcsmerge merges two revisions, *rev1* and *rev2*, with respect to a common ancestor. A 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 *rev1* is the differing revision, the segment in *rev1* replaces the corresponding segment of *rev2*. Type (c) indicates an overlapping change, is flagged as an error, and requires user intervention to select the correct alternative.

rlog **– read log messages**

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

* The *rcsclean* and *rcsfreeze* commands are optional and are not always installed.

References

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

2. Hunt, James W. and McIlroy, M. D., "An Algorithm for Differential File Comparison," 41, Computing Science Technical Report, Bell Laboratories (June 1976).

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

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

5. Leblang, David B. and Chase, Robert P., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices,* 19, 5, pp. 104-112 (May 1984). Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments..

6. 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.

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

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

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

10. Lampson, Butler W. and Schmidt, Eric E., "Practical Use of a Polymorphic Applicative Language" in *Proceedings of the 10th Symposium on Principles of Programming Languages,* pp. 237-255, ACM (January 1983).

11. 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).

12. Heckel, Paul, "A Technique for Isolating Differences Between Files," *Communications of the ACM,* 21, 4, pp. 264-268 (April 1978).

13. Tichy, Walter F., "The String-to-String Correction Problem with Block Moves," *ACM Transactions on Computer Systems,* 2, 4, pp. 309-321 (Nov. 1984).