An Introduction to the Revision Control System

Walter F. Tichy

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

ABSTRACT

The Revision Control System (RCS) manages software libraries. It greatly increases programmer productivity by centralizing and cataloging changes to a software project. This document describes the benefits of using a source code control system. It then gives a tutorial introduction to the use of RCS.

Functions of RCS

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc. It greatly increases programmer productivity by providing the following functions.

1. RCS stores and retrieves multiple revisions of program and other text. Thus, one can maintain one or more releases while developing the next release, with a minimum of space overhead. Changes no longer destroy the original -- previous revisions remain accessible.
   a. Maintains each module as a tree of revisions.
   b. Project libraries can be organized centrally, decentralized, or any way you like.
   c. RCS works for any type of text: programs, documentation, memos, papers, graphics, VLSI layouts, form letters, etc.
2. RCS maintains a complete history of changes. Thus, one can find out what happened to a module easily and quickly, without having to compare source listings or having to track down colleagues.
   a. RCS performs automatic record keeping.
   b. RCS logs all changes automatically.
   c. RCS guarantees project continuity.
3. RCS manages multiple lines of development.
4. RCS can merge multiple lines of development. Thus, when several parallel lines of development must be consolidated into one line, the merging of changes is automatic.
5. RCS flags coding conflicts. If two or more lines of development modify the same section of code, RCS can alert programmers about overlapping changes.
6. RCS resolves access conflicts. When two or more programmers wish to modify the same revision, RCS alerts the programmers and makes sure that one change will not wipe out the other one.
7. RCS provides high-level retrieval functions. Revisions can be retrieved according to ranges of revision numbers, symbolic names, dates, authors, and states.
8. RCS provides release and configuration control. Revisions can be marked as released, stable, experimental, etc. Configurations of modules can be described simply and directly.
9. RCS performs automatic identification of modules with name, revision number, creation time, author, etc. Thus, it is always possible to determine which revisions of which modules make up a

given configuration.

10. Provides high-level management visibility. Thus, it is easy to track the status of a software project.
    a. RCS provides a complete change history.
    b. RCS records who did what when to which revision of which module.
11. RCS is fully compatible with existing software development tools. RCS is unobtrusive -- its interface to the file system is such that all your existing software tools can be used as before.
12. RCS' basic user interface is extremely simple. The novice only needs to learn two commands. Its more sophisticated features have been tuned towards advanced software development environments and the experienced software professional.
13. RCS simplifies software distribution if customers also maintain sources with RCS. This technique assures proper identification of versions and configurations, and tracking of customer changes. Customer changes can be merged into distributed versions locally or by the development group.
14. RCS needs little extra space for the revisions (only the differences). If intermediate revisions are deleted, the corresponding differences are compressed into the shortest possible form.

Getting Started with RCS

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

```
ci  f.c
```

This command creates f.c,v, stores f.c into it as revision 1.1, and deletes f.c. It also asks you for a description. The description should be a synopsis of the contents of the file. All later checkin commands will ask you for a log entry, which should summarize the changes that you made.

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

```
co  f.c
```

This command extracts the latest revision from f.c,v and writes it into f.c. You can now edit f.c and check it in back in by invoking:

```
ci  f.c
```

Ci increments the revision number properly. If *ci* complains with the message

```
ci error: no lock set by <your login>
```

then your system administrator has decided to create all RCS files with the locking attribute set to ‘‘strict’’. With strict locking, you you must lock the revision during the previous checkout. Thus, your last checkout should have been

```
co  -l  f.c
```

Locking assures that you, and only you, can check in the next update, and avoids nasty problems if several people work on the same file. Of course, it is too late now to do the checkout with locking, because you probably modified f.c already, and a second checkout would overwrite your changes. Instead, invoke

```
rcs  -l  f.c
```

This command will lock the latest revision for you, unless somebody else got ahead of you already. If someone else has the lock you will have to negotiate your changes with them.

If your RCS file is private, i.e., if you are the only person who is going to deposit revisions into it, strict locking is not needed and you can turn it off. If strict locking is turned off, the owner off 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
```

You can set the locking to strict or non-strict on every RCS file.

If you do not want to clutter your working directory with RCS files, create a subdirectory called RCS in your working directory, and move all your RCS files there. RCS commands will look first into that directory to find needed files. All the commands discussed above will still work, without any change*.

To avoid the deletion of the working file during checkin (should you want to continue editing), invoke

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

This command checks in f.c as usual, but performs an additional checkout with locking. Thus, it saves you one checkout operation. There is also an option **-u** for *ci* that does a checkin followed by a checkout without locking. This is useful if you want to compile the file after the checkin. Both options also update the identification markers in your file (see below).

You can give *ci* the number you want assigned to a checked in revision. Assume all your revisions were numbered 1.1, 1.2, 1.3, etc., and you would like to start release 2. The command

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

assigns the number 2.1 to the new revision. From then on, *ci* will number the subsequent revisions with 2.2, 2.3, etc. The corresponding *co* commands

```
co  -r2  f.c      and      co  -r2.1  f.c
```

retrieve the latest revision numbered 2.x and the revision 2.1, respectively. *Co* without a revision number selects the latest revision on the "trunk", i.e., the highest revision with a number consisting of 2 fields. Numbers with more than 2 fields are needed for branches. For example, to start a branch at revision 1.3, invoke

```
ci  -r1.3.1  f.c
```

This command starts a branch numbered 1 at revision 1.3, and assigns the number 1.3.1.1 to the new revision. For more information about branches, see *rcsfile*(5).

Automatic Identification

RCS can put special strings for identification into your source and object code. To obtain such identification, place the marker

```
$Header$
```

into your text, for instance inside a comment. RCS will replace this marker with a string of the form

```
$Header:  filename  revisionnumber  date  time  author  state $
```

You never need to touch this string, because RCS keeps it up to date automatically. To propagate the marker into your object code, 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, even object code. Thus, *ident* helps you to find out which revisions of which modules were used in a given program.

You may also find it useful to put the marker

```
$Log$
```

into your text, inside a comment. This marker accumulates the log messages that are requested during checkin. Thus, you can maintain the complete history of your file directly inside it. There are several additional identification markers; see *co* (1) for details.

---

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

How to combine MAKE and RCS

If your RCS files are in the same directory as your working files, you can put a default rule into your makefile. Do not use a rule of the form .c,v.c, because such a rule keeps a copy of every working file checked out, even those you are not working on. Instead, use this:

```
.SUFFIXES: .c,v

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

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

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

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

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

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

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

Additional Information on RCS

If you want to know more about RCS, for example how to work with a tree of revisions and how to use symbolic revision numbers, read the following paper:

Walter F. Tichy, ‘‘Design, Implementation, and Evaluation of a Revision Control System,’’ in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

Taking a look at the manual page *RCSFILE*(5) should also help to understand the revision tree permitted by RCS.