RCS Cookbook

The Revision Control System (RCS) developed by Walter Tichy at Purdue University makes it easier to manage large software projects by providing facilities for supporting multiple paths of development, maintaining revision histories and naming software revisions. This document provides advice for how to use these facilities, along with enhancements developed here at the Mach Project, to manage software projects.

Managing Multiple Development Paths

One common problem in managing large software projects comes to play when there are several people working on the same sets of files. RCS helps by providing per-file locking to guarantee that no two users modify the same file at the same time. If several users are modifying the same line of development, however, this locking does nothing to guarantee that one user's changes in one file won't interfere with another user's changes in another file.

To combat this problem, RCS provides the notion of a "branch". The central path of development in an RCS project is usually referred to as the trunk or mainline. It represents the set of sources you will get if you do an rcsco operation with no revision modifiers. The versions found on the mainlin have two digit version numbers.

Branches

A branch is a separate line of development carried out in parallel to the main line of development. Versions found on branches have more than two digits in their version numbers. The first N-2 digits correspond to the version from which the branch was made. The first N-1 digits correspond to the version number of the branch. All digits make up the version number of an element on a branch. For example, version 1.2.1.1 is an element in the first branch off of version 1.2. That branch is named 1.2.1. You can make branches off of branches. Each extra level introduces another two digits into the branched element version number.

You refer to branched elements the same way you refer to mainline elements in rcs commands. For example,

    rcsco -r1.2.1.2 foo.c

will check out the second element in the first branch off version 1.2 of foo.c.

If you use a branch name (as opposed to a branched element name), you implicitly refer to the last element on that branch.

Creating branches

You can create branches using the -b option of the rcsci command. The -b option will create a new branch off the revision specified for the rcsci command. For example, the command

```
rcsci -b foo.c
```

will create a branch off the version at the end of the mainlin. The command

```
rcsci -b1.2 foo.c
```

will create a branch off version 1.2 of foo.c.

Naming branches

Branch numbers are hard to deal with, but you can assign symbolic names to branches. Working with branch numbers is strongly discouraged. You can assign a name to a branch when you create the branch by using the -n or -N option of the rcsci command.

For example, a typical command to create and name a branch would be

```
rcsci -b -ndevice_io_fix foo.c
```

This command will create a branch named device_io_fix off the last version on the mainline of foo.c. The name is assigned to the branch (not the branched element), so subsequent commands such as

```
rcsco -l -rdevice_io_fix foo.c
```

will refer to the latest element in that line of development.

Using branches

Now that you can create and name branches, you have a place to do work that won't interfere with the mainline development. Branches work just like the mainline, so you can keep a revision history of work done on the branch, and use rcsdiff to compare elements found on the branch with elements on the mainline or anywhere else in the tree.

Merging

Once you have produced a set of working changes on your branch you will want to incorporate those changes back into the mainline of development. This involves figuring out what is different between your branched version and the mainline version, and of those differences which you wish to keep and which you wish to discard.

The program branchmerge is designed to take two revisions in an RCS file, locate their common ancestor and do a three way merge on them. branchmerge assumes that development may have been taking place in both lines of development but that the development is independent. By making comparisons with a common ancestor, branchmerge can easily determine which changes represent progress in a particular line of develpment.

When changes overlap (both lines of development changed the same range of lines in different ways), branchmerge has no way to determine what the correct action is. By default, branchmerge will print out the conflicting lines and prompt the user for what action to take. In this interactive mode, branchmerge can be instructed to take one or the other of the conflicting changes, both, or both with annotation (both sets of changes are included in the file, enclosed in rows of less than and greater than symbols).

The -l option to branchmerge will lock the target version of the merge. The -D option to branchmerge will enclose changes found on the branch inside conditional compilation directives.

branchmerge can also be used to merge changes from the mainline onto a branch. This lets you continue development on a branch while picking up new functionality from the mainline to keep your branch from getting too far out of date with respect to the mainline. (Actually, branchmerge can be used to merge changes from any two parts of the rcs tree, but by it is most commonly used to move changes to and from the mainline.)

branchmerge merges two versions out of the RCS tree. It does not use the workfile in the current directory. branchmerge creates the merged source in the workfile and will warn you if the workfile is writable (implying that you may be in the process of changing it).

Some example branchmerge commands are,

```
# merge device_io_fix
    branchmerge -rdevice_io_fix foo.c bar.c

# merge with and lock mainline
    branchmerge -l -rdevice_io_fix  foo.c bar.c

# merge the mainline onto your branch; lock your branch
    branchmerge -l "-r<>" -rdevice_io_fix foo.c bar.c

# merge onto the mainline, marking all of your changes
    branchmerge -l -rdevice_io_fix -DMACH_DEVICE foo.c bar.c
```

Configuration

Branching is a good way of organizing information, but it can be inconvenient. It's undesirable to make branches to all elements. It's undesireable to have to remember which elements you've made branches to. The RCS configuration facility lets you specify rules that are used by RCS when it selects which version of an element it is to operate on. Using this facility, it is possible to always select branched versions of an element when they are present.

It is also possible to create a configuration that will represent the state of a project at some particular point in time, providing a stable base upon which you can make changes.

Configuration concepts

A configuration is a set of rules used by RCS when choosing what element to manipulate. When you check out an element and ask for revision 1.2, for example, you are providing a simple rule for RCS to use when making its extraction. The configuration facility lets you provide a series of rules that are each tried in order. The operation is performed on the version selected by the first successfully matching rule.

If the -r option is given on the command line of an rcs rule, its argument is used as the configuration. If no configuration is specified on the command line and the file .rcsconfig exists in the current directory, its contents are used as the configuration. If no -r option is given and the file .rcsconfig does not exist then no explicit configuration is used and operations are performed on the version at the end of the mainline.

Specifying a configuration

Configurations consist of a series of rules separated by newlines or semicolons.

```
<configuration>   ::=    <config-rule>
                         <configuration> ’;’ <config-rule>

<config-rule>     ::=    <rcs-name>
                         <date-spec>
                         <ver-file-spec>
                         <wildcard-spec>

<rcs-name>        ::=    <branch-name>
                         <element-name>
                         <version-number>

<element-name>    ::=    identifier
<branch-name>     ::=    identifier

<version-number>  ::=    number
                         <version-number> ’.’ number

<date-spec>       ::=    <date>
                         <date> <time>
                  ::=    ’<’ ’>’

<date>            ::=    ’<’ number ’/’ number ’/’ number

<time>            ::=    number ’:’ number

<pathname>        ::=    ’.’
                         ’..’
                         ’~’
                         <pathname> ’/’ identifier

<ver-file-spec>   ::=    <pathname>
                  ::=    <pathname>

<wildcard-spec>   ::=    ’-for’ <wildcard> <rcs-name>

<wildcard>        ::=    <pathname>
```

Sample configurations

Examples of configurations might be

```
device_io_fix
<>
```

(take from the branch device_io_fix if it exists, else the mainline.)

```
device_io_fix
../shapshot-03-20-88
```

(take from the branch device_io_fix if it exists, else read the file ../snapshot-03-20-88, find the rcs name and corresponding version.)

```
device_io_fix
```

```
<88/01/21 18:40
```

(take from the branch device_io_fix if it exists, else use the version that was active as of January 21, 1988 at 6:40pm.)

```
rcsco "-rdevice_io_fix;<88/01/21 18:40" myfile.c
```

(same configuration, only given on a command line)

Configuration subtlties

Date specifications apply only to the mainline, since a date specification can be ambiguous given parallel lines of development. The special date "<>" refers to the version at the end of the mainline.

Version Control

Maintaining software releases has traditionally been a problem. Once a software system has been given to the public, development moves forward but bugs tend to be found more in the earlier, released software. As a result, it is convenient to be able to return to a known point in software development. It is convenient both for validation and verification purposes and also for the purposes of applying bug fixes to previous versions for spot releases.

The usual way to save the state in a software project was to take a snapshot of all the files that made up that project at that time. Since RCS provides you with the ability to return to a previous version of a file, it is possible to use RCS to reproduce the state of a software project at some point in time. The only thing required to return to previous state of an RCS project is the ability to describe all of the versions of the elements that went into producing that state.

Naming versions

One way to save the state of some sources is to use the name_version program which assigns a text name to selected versions. name_version uses the configuration facility so it can use complicated rules to choose which version to assign a name to. The rules are taken from the .rcsconfig file by default or can be overridden on the command line.

For example,

```
name_version -nXM13-devio_fix "-rdevice_io_fix;<>" RCS/*,v
```

can be used to assign the name XM13-devio_fix to the configuration that consists of the device_io_fix branch and the mainline. Once assigned, you can use this new name as part of a -r option when checking out files or as part of a configuration.

XXX The -R option can be used to recursively define names for an entire tree.

Names are good because they are a fairly self-describing way to freeze the state of a configuration. Names are bad because there is a (large) finite limit on the number of names you can put into an RCS tree, so indiscriminate use of names could potentially interfere with the operation of the tree. They are also bad because there’s no restriction on who can add or remove names from an RCS tree, so names can be accidentally removed from the tree.

Snapshotting versions

Another way to save the state of an RCS maintained source tree is to save in a file the names of all of the source versions and their associated revision numbers. A file of this format can be used in an RCS configuration rule. A file of this format can be itself saved in RCS and assigned a convenient name. With this facility, all of the information required to reproduce an entire software system is embodied by this single file; different versions of the software system may be described by the different versions of this file.

The snapshot command can be used to save the version numbers of all elements found in one or more directories. The snapshot command uses the configuration facility to select which version of an element to record. The rules are taken from the .rcsconfig file by default or may be overridden on the command line.

The snapshot command writes its output into a file called RCSSNAP, by default. The -f option can be used to make snapshot write to a different file.

For example,

```
snapshot "-rdevice_io_fix;<>" ../{kern,bsd,vm,sun*,net*}
```

If snapshot is run with the -u option, it generates a snapshot which it compares with the current contents of the RCSSNAP file. It takes the result of the difference and generates a script of rcsco commands required to perform an update that will result in the configuration given with the "snapshot -u" command.

The file names that are recorded in the snapshot file must be recognizable by the RCS program as matching the target file name. For configurations where several different subdirectories contain source, the two universal ways to snapshot versions are to use absolute pathnames and to use relative pathnames preceded by a common ".." prefix.

The previous example snapshot command would generate a file with contents that look like

```
../kern/RCS/Makefile,v  1.7.1.2
../kern/RCS/ast.c,v     1.7
../kern/RCS/clock_prim.c,v      1.9
../kern/RCS/device.c,v  1.1
../kern/RCS/device.defs,v       1.1
```

A snapshot command like

```
snapshot "-rdevice_io_fix;<>" /usr1/work/mach/{kern,bsd,vm,sun*,net*}
```

would generate a file with contents that look like

```
/usr1/work/mach/kern/RCS/Makefile,v      1.7.1.2
/usr1/work/mach/kern/RCS/ast.c,v         1.7
/usr1/work/mach/kern/RCS/clock_prim.c,v 1.9
/usr1/work/mach/kern/RCS/device.c,v     1.1
/usr1/work/mach/kern/RCS/device.defs,v  1.1
```

The RCS configuration program uses the "getwd" system call to determine the value of "." when looking for matching file names. As a result, snapshotting through links will generate a file full of names that the configuration program won’t be able to match.

Cooking

There are many different ways these facilities may be used to assist in project management. So far in the Mach project, two main approaches to using these tools have emerged. We'll call these the B method and the D method. Both methods assume that there is a master version of the source which is to be modified. Both methods assume that development is to be done elsewhere (usually on the local disk of a workstation).

The B method

The model for the B method is (to be filled in by Bob Baron)

The D method

The model for the D method is that the development is to proceed exactly as if development were being done on the master sources. The illusion is produced by making a local copy of the source tree structure, replacing RCS subdirectories with links back to the master RCS subdirectories. Therefore, local RCS commands reference the master RCS source, while the checked out sources are kept on the local disk.

The D method lends itself to using RCS configurations for providing a stable base upon which to work. Here are some steps required for setting up for and doing development using the D method:

1) Create a tree with structure as the master source tree.

2) Make links to the master RCS subdirectories for each RCS subdirectory.

3) Make a .rcsconfig link in each directory to a master RCS configuration file (so you can change the entire configuration by changing one file).

4) Choose an RCS configuration upon which to do development

   a) If your configuration doesn't involve working off the mainline, either name or snapshot a base set of sources to work from (or check out an existing snapshot or use an existing name).

   b) If your configuration is going to involve a parallel line of development, choose a branch name.

   c) Enter this configuration into the master RCS configuration file.

6) Check out and lock modules you wish to change (rcsco -l {element}).

7) Make and test modifications.

8) Check in modules, making branches as necessary (rcsci [-b -n{branchname}] {element}).

9) Repeat (6-8) until done.

10) Merge with mainline, locking element mainline version. (branchmerge -r{branchname} -l [-D{define-name}] {element})

11) Test mainline merged version.

12) Check in mainline version (rcsci {element})

Checking modules in with the -u option leaves an unlocked copy in the working directory. This module's last modification date is preserved, ensuring that make won't unnecessarily try to recompile it.

Suggested rules of the road

As noted before, there is no inter-file consistency guaranteed by RCS, so it is undesireable to have two people modifying a particular line of development simultaneously. As a result, you shouldn't modify the mainline without some convention or safeguard (like chmod'ing the RCS tree) to prevent simultaneous access.

If development is to take place over an extended period of time it should be done on branches. The more conservative approach is to root these branches in a stable base.

A little riskier mode of operation would be to do development from branches based on the mainline, and update sources and branched versions periodically to keep up with the mainline development.

Future Directions

One of the big advantages that working with RCS in this fashion gives is precision. It provides a way to precisely name the source versions used to build a software system. Given that you can use the tools orignally used to generate the software system (compilers, etc.) and that all components used in the system are named this naming should allow you to exactly reproduce a system.

A weakness exists in this system, though, since there is no strong link between the checked out versions found in the working directory and the versions in the master source. There is also no strong link between the objects and the sources used to produce them. Ideally, this environment would be set up to work directly from the RCS sources when possible (for all elements not checked out for modification). In addition, the system could associate a list of the versions of all components required to produce an object module with that object module. It could use this information to precisely identify object modules and possibly to share object modules between diverse users.