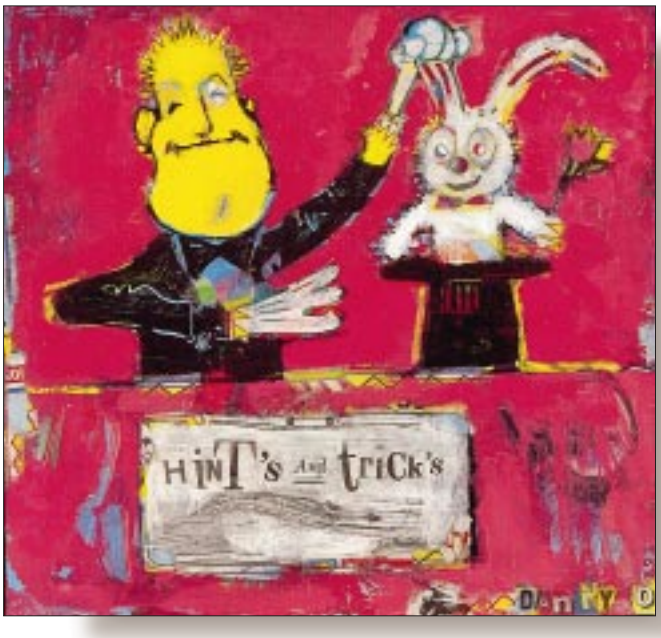
by Jeffreys Copeland and Haemer



Practical CVS, Part 2

Jeffrey Copeland *(copeland@alumni.caltech.edu) is a member of the technical staff at QMS' R&D group in Boulder, CO. He's been a software consultant to the Hugo award administrators for several years. He spends his spare time raising children and cats.*

Jeffrey S. Haemer *(jsh@usenix.org) now works for QMS, too, and is having a great time. Before he worked for QMS, he operated his own consulting firm, and did a lot of other things, like everyone else in the software industry.*

For several articles, we have been developing a theme. Three months ago, we discussed how to back up individual files with `cp`. (If you didn't read the article, you're probably thinking, "How you write an entire article about `cp`?" Utilities from the Free Software Foundation, like love, are many-splendor'd things.) Next, we progressed to Revision Control System (RCS), which lets you maintain a detailed revision history of a file. Last month, we discussed the basics of Concurrent Versions System (CVS), which extends RCS to let you manage the full complexity of entire product releases.

Significantly, all of these are free and available from the Net: CVS is available at `http://www.loria.fr/~molli/cvs-index.html`; and the current version of RCS can be found in your favorite GNU tools directory, `ftp://prep.ai.mit.edu/pub/gnu/rcs-5.7.tar.gz`, for example.

This month, we'll sketch some of the more sophisticated features CVS offers to multi-developer, multisite, multirelease products.

Keeping Tags on Releases

Students and professors sometimes ask us how professional programming differs from programming for courses. One unique feature of professional programming is that because products evolve as time goes on, so much of it deals with multiple releases: preparing releases, documenting releases, freezing releases, fixing bugs in releases, upgrading releases, merging features from one release into another and so on. Academic software engineering, in contrast, often focuses on development toward a defined goal. When the "final version" of the software is finished, so is the project.

One thing that CVS provides to help manage product releases is *tags*.

Like RCS, on which it's built, CVS lets developers check out particular file revisions:

```
cvs co -r 30.2 main.c
```

The convenience of symbolic names was not lost on the authors of CVS, and so

```
cvs co -r prometheus main.c
```

will check out the revision named "prometheus." Not only is this easier to remember, but you can use a single name to check out an entire release. Assuming we have a module named "ll," instead of saying this:

```
cd ll
cvs co -r 100.43 README
cvs co -r 30.2 main.c
cvs co -r 12.1.9.6 abracadabra.pl
...
```

we can say

```
cvs co -r prometheus ll
```

assuming you've already attached the tag "prometheus" to release 100.43 of `README`, release 30.2 of `main.c` and so on.

And how do you attach tags to versions?

```
cvs tag prometheus ll
```

Like most other CVS commands, `cvs tag` understands trees and will traverse the entire hierarchy that you name, tagging every file in it. As an aside, you can block this file-tree walk by invoking it as `cvs tag -l`.

CVS uses the same flag in the same way for different commands wherever it's sensible, and the `-l` (local) flag also prevents subdirectory traversal for `checkout`, `commit`, `diff`, `export`, `remove`, `rdiff`, `rtag`, `status` and `update`.

This brings us to an interesting distinction: tags versus sticky tags.

Implementing Sticky Tags

There are two reasons to name a release: (a) because you want to work on it and (b) because you don't. For example, if I'm running a project and announce a code freeze, I probably want a label for all the file versions that make up that frozen release. This label provides a snapshot of the code at the point of the freeze.

Tags are the tool of choice here, and `cvs freeze` is a synonym for `cvs tag`. (Actually, many CVS commands have mnemonic synonyms. Try `cvs --help-synonyms` for a list.)

On the other hand, when I release products to the field, doing maintenance on those products means being able to check out old versions of files, revise them, and check those revised versions back into the repository without interfering with main-line development.

RCS permits this kind of work on individual files with "branch deltas." If the main-line development version of `foo.c` is revision 20.17, then checking out that version and checking it back in produces revision 20.18. If, however, you also need to fix a bug in an older version, say, revision 16.35, you can check out that revision with the command `co -r 16.35 foo.c`. Modifying the file and checking it back in produces a branch, with revision 16.35.1.1, and further work on that branch can produce 16.35.1.2, 16.35.1.3 and so on. The branch this creates is 16.35.1, and you can request the most recent revision of that branch by just typing `co -r 16.35.1 foo.c`

This sort of maintenance work is frequent for real software releases, so CVS lets you tag an entire collection of particular files for future checkout and maintenance. If you have a particular suite of file versions checked out, the command `cvs tag -b prometheus_maintenance` puts a "sticky tag" on all the modules in the current directory hierarchy. Not only will this let you check out the tagged version in the future–say with `cvs co -r prometheus_maintenance ll`–but when you check any changes back in, they will go in as a branch off of the appropriate revision of each of the changed files. Future checkouts of this same tagged version will get the most recent file revisions associated with that label. The tag "sticks to" the tip of the branch and grows with it.

There is a second sense in which the tag is sticky: If you check out a version with a sticky tag, chances are you're doing maintenance, so CVS remembers that you're working with this product release. Thus, a typical cycle looks like this:

```
# begin by checking out the
# "prometheus_maintenance" version
# of module "ll".
cvs co -r prometheus_maintenance ll
cd ll
   # work to upgrade some of the files,
...
   # save the changes in the branch
cvs ci
   # work to upgrade more files
   # in the same branch
...
   # pick up everyone else's
   # changes to this branch
cvs update
...   # and so on
```

To return to working on the main-line, you can reset the sticky tags with

```
cvs update -A
```

or you can simply discard the `prometheus_maintenance` tree and start from scratch with a new `cvs co`.

Administrative Files

Try this:

```
cvs co CVSROOT
ls CVSROOT
```

As we discussed last time, when you create a tree with `cvs init`, CVS automatically creates a suite of administrative files

in the directory `$CVSROOT/CVSROOT`. (If we were designing CVS, we'd pick a better name. If we were designing CVS, we'd have to implement it and maintain it. Actually, the name's not all that bad.)

Because the files are kept in the repository itself, we can check them out, modify them and check them back in. Figure 1 shows us the files in `$CVSROOT/CVSROOT`. Notice that `$CVSROOT/CVSROOT` contains both an RCS version *and* a checked-out master version of each file. The checked-out master is the administrative file that governs CVS' behavior. The RCS version is the repository version that contains that file's entire history. Whenever you check in a new revision of one of these files, CVS automatically updates the checked-out master with your changes.

One file, `history`, has no RCS version. This isn't a file that you can edit. It's just a record CVS keeps of what's been done. Each of the other files lets you configure how CVS behaves in interesting ways. Here are some highlights:

• `commitinfo` lets you specify what sorts of sanity checks should be made on files at check-in time. On our system, we run all files through a program that does a suite of fairly simple, but still useful, checks. For example, we require that all files have a "Header" or "Id" line (see the man page for the RCS `co` command). If they fail the check, CVS warns us of the problem, and the check-in fails until we fix it.

• `editinfo` lets you specify sanity checks on the log comments. Ours makes you say something. Anything. No more empty log comments in our tree, but we still get comments like "Fixed a bug."

• `modules` is a key file because it helps CVS users manipulate collections of files as entities. Some of this is well-documented, but some isn't. For example, if you have three top-level modules, called "curly," "larry" and "moe," each of which you can check out individually, you can make a sort of compound module with an entry like this in the modules file:

```
stooges   stooges & curly & larry & moe
```

This will check out the `stooges` module, but then also check out `curly`, `larry` and `moe`, making each of them a subdirectory under `stooges`.

• `rcsinfo` lets you specify log entry templates. If you have a form–or 40 forms, a different one for every module–that you want developers to use when they change files, this is the place to specify them.

These files, and the other administrative files, are well-documented internally. We encourage you to check out a copy and read through them.

Figure 1. The $CVSROOT/CVSROOT Directory

```
$ ls -C $CVSROOT/CVSROOT
checkoutlist     cvswrappers,v  loginfo,v   rcsinfo     verifymsg,v
checkoutlist,v   editinfo       modules     rcsinfo,v
commitinfo       editinfo,v     modules,v   taginfo
commitinfo,v     history        notify      taginfo,v
cvswrappers      loginfo        notify,v    verifymsg
```

Miscellaneous Tricks

The title of this column is "Practical CVS," and we want to leave you with a few useful tricks that we couldn't figure out where to fit in earlier:

• CVS understands dates. The flag `-D`, which lets you specify a date, is often a useful alternative to `-r`, which makes you specify a revision. The variety of acceptable date formats is surprising. For example, if you broke your product sometime in the last two weeks, but you don't know how or when, you can get back a working version like this:

```
cvs update -D 'a fortnight ago'
```

and work your way forward. Or, you can just see the changes:

```
cvs diff -D 'a fortnight ago'
```

• CVS understands the Internet. If you have networked development machines, CVS will work across the Net. We do not just mean that you can remote mount `$CVSROOT`. We have a CVS repository in Mobile, AL. Working on machines in Boulder, CO, we routinely work with modules from the Mobile, AL, repository like this:

```
CVSROOT=jsh@moe.qms.com:/proj_storage/cvsroot
export CVSROOT
cvs co mvp
```

The first line says the CVS repository is on the Mobile machine `moe.qms.com` in directory `/proj_storage/cvsroot`, and we want to work with this repository as user `jsh`. The second line uses the local CVS program as a client and the remote CVS program as a server, and checks out a copy across the Net. All subsequent CVS commands within our checked-out copy know that the repository is remote, and just work. (Note: Your `.rhosts` file must be set up correctly on the remote machine.)

If performance is a problem, we use the `-z` flag, which automatically gzips commands and data at one end, and gunzips them at the other. You also need to give it a degree of compression, like this:

```
cvs -z 5 update
```

• CVS understands the idea of merging changes into the main-line. We often want to merge our changes from branches into the main-line development tree. Sometimes, these are bug-fixes to field releases that we want to fold into the main-line. Other times, we create branches to do experimental development. If the experiment is a success, we add it to our product.

A traditional, and tedious, way to approach problems like this is with `diff`. CVS provides a labor-saving `-j` flag that does much of the work for us. A single `-j` flag joins changes from the named revision into the current version. For example, using the command below, we join changes from the `prometheus_maintenance` revision

to the tree we're currently working on:

```
cvs update -j prometheus_maintenance
```

To merge changes from the top-of-the-tree into the version you have checked out use

```
cvs update -j HEAD
```

The manual will also show you how to use a pair of `-j` flags to merge a specific pair of revisions, and even to remove all changes between a pair of revisions.

• CVS understands changes in the hierarchy. If you remove a file, you expect to stop seeing it. Well, at least up to the point that you need to reproduce last week's build. The command

```
cvs rm foo.c
```

schedules a file for removal, and the next check-in "removes" it. Under the covers, CVS actually moves the underlying RCS file into a subdirectory within the CVS repository, called the "Attic." When you ask for old versions, CVS will go into the Attic and find them, where they're stored with their entire revision history intact.

On the other side of the coin, if we want to add a new file,

```
cvs add
```

does the trick. A subsequent `cvs co` of a version before the file existed will omit that file from the check-out.

Directories are more complicated. When we've removed all the files in a directory with `cvs rm`, we're left with a bunch of empty directories. We can prune those dead branches with

```
cvs update -P
```

When someone else has created new directories that we want to bring in, we can fall back on `cvs co`, but we can also use

```
cvs update -d
```

• Using diffs. When we're checking things in, we find it useful to have the `diffs` around to help us write our log comments:

```
cvs diff > /tmp/DIFFS 2>&1
cvs ci
```

By now, you should have an overview of how CVS works, and some tricks to make your use of it more productive. Next time, we'll begin discussing the problem of printing over the Net, and how our old UNIX standby `lpr` and its daemon `lpd` are insufficient to the task. Until then, happy trails.