Chapter 9

Third-Party Tools that Work with CVS

What Are Third-Party Tools?

Around all successful open source programs, third-party tools sooner or later pop up, sometimes as fast as mushrooms. Many people have written programs to augment CVS. We call these "third-party tools" because they have their own maintainers, separate from the CVS development team. Most of these programs are not distributed with CVS, although some are. This chapter covers third-party tools that we have found useful, but that are not distributed with CVS.

Although there are some very popular and widely used non–command-line or non-Unix interfaces to CVS (we list download sites for these in Chapter 3), we do not discuss most of them in this chapter. Their popularity makes it easy to find out more about them from mailing lists and newsgroups. One exception to this is the Emacs pcl-cvs interface, which is very useful, but sometimes tricky to install. We'll start there.

pcl-cvs: An Emacs Interface to CVS

Depends on: Emacs, Elib

URLs:

- **www.cvshome.org/dev/addons.html**
- **ftp://ru.cs.yale.edu/pub/monnier/pcs-cvs/**

Authors: Per Cederqvist and Stefan Monnier (current maintainer)

pcl-cvs is one of two Emacs/CVS interfaces. The other is the native VC (Version Control) interface built into Emacs. We prefer pcl-cvs because it was written exclusively for CVS and therefore works smoothly with the CVS way of doing things. VC, on the other hand, was designed to work with several different back-end version control systems—RCS and SCCS, as well as CVS—and is not really "tuned" for CVS. For example, VC presents a file-based rather than a directory-based interface to revision control.

The advantages of pcl-cvs are strong enough that many people choose to download and install it rather than use VC. Unfortunately, pcl-cvs has one disadvantage: It can be a bit tricky to install (much of this section is devoted to overcoming possible installation hurdles.

Although the rest of these instructions use examples from a version 1.09 distribution, they should apply to later versions as well.

Installing pcl-cvs

If you don't normally deal with Emacs installation and site-maintenance issues, the pcl-cvs installation procedure might seem a bit daunting. A little background on how Emacs works might help.

Most higher-level Emacs features are written in a language called "Emacs Lisp" (Emacs itself is essentially an interpreter for this language). People add new features to Emacs by distributing files of Emacs Lisp code. pcl-cvs is written in this language, and it depends on a library of useful, generic Emacs Lisp functions called "Elib" (also written in part by Per Cederqvist, but distributed separately from pcl-cvs).

Elib is not included in the regular Emacs distribution (at least not FSF Emacs; we don't know about XEmacs), so you might have to download and install it yourself before you can use pcl-cvs. You can get it from **ftp://ftp.lysator.liu.se/pub/emacs/elib-1.0.tar.gz**. The package contains installation instructions. You should use only version 1.0 of Elib.

Once Elib is installed, you're ready to build and install pcl-cvs. These instructions apply both to version 1.50 and the 2.x series (although you should check the NEWS and INSTALL files in newer distributions to see what's changed).

First, unpack pcl-cvs:

```
yarkon$ zcat pcl-cvs-1.09.tar.gz | tar xvf -
pcl-cvs-1.50/
pcl-cvs-1.50/README
pcl-cvs-1.50/NEWS
pcl-cvs-1.50/INSTALL
pcl-cvs-1.50/ChangeLog
pcl-cvs-1.50/pcl-cvs.el
pcl-cvs-1.50/pcl-cvs.texinfo
pcl-cvs-1.50/compile-all.el
pcl-cvs-1.50/pcl-cvs-lucid.el
```

```
pcl-cvs-1.50/pcl-cvs-startup.el
pcl-cvs-1.50/pcl-cvs.info
pcl-cvs-1.50/Makefile
pcl-cvs-1.50/texinfo.tex
```

Next, go into the source tree's top level:

```
yarkon$ cd pcl-cvs-1.05/
```

A Makefile is supplied there. According to the instructions in the INSTALL file, you're supposed to edit a few paths at the top of the Makefile and then run:

```
yarkon$ make install
```

If that works, great. However, this sometimes results in an error (the pcl-cvs code itself is very portable, but its installation procedures sometimes are not). Do this if you get an error:

```
yarkon$ make clean
yarkon$ make
```

If all goes well, these commands accomplish a significant part of the installation by byte-compiling all of the Emacs Lisp files. (Byte-compiling converts a file of human-readable Emacs Lisp code—an .el file—into a more compact and efficient representation—an .elc file. Emacs can load and run an .elc file with better performance than they can a plain .el file.)

We'll proceed as though the byte-compilation stage succeeded. If the byte compilation does not appear to succeed, don't worry: The .elc files are a luxury, not a necessity. They improve performance slightly, but you can run pcl-cvs from the raw .el files with no problem.

If the **make install** failed, the next step is to get the Emacs Lisp (whether .el or .elc) into a directory where Emacs can load it automatically. Emacs has a designated directory on the system for locally installed Lisp. To find this directory—it will have a file named "default.el" in it—check the following locations, in this order:

1. /usr/share/emacs/site-lisp/
2. /usr/local/share/emacs/site-lisp/
3. /usr/lib/emacs/site-lisp/
4. /usr/local/lib/emacs/site-lisp/

Once you've found your site-lisp directory, copy all of the Lisp files to it (you might have to do this step as root):

```
yarkon# cp -f *.el *.elc /usr/share/emacs/site-lisp/
```

The last step is to tell Emacs about the entry points to pcl-cvs (the main one being the function **cvs-update**), so it will know to load the pcl-cvs code on demand. Because Emacs always reads the default.el file when it starts up, that's where you need to list the pcl-cvs entry points. Fortunately, pcl-cvs provides the necessary content for default.el. Simply put the contents of pcl-cvs-startup.el into default.el (or perhaps into your .emacs, if you're just installing this for yourself) and restart your Emacs.

You might also want to copy the .info files into your info tree and add pcl-cvs to the table of contents in the dir file.

Using pcl-cvs

Once installed, pcl-cvs is very easy to use. You just run the function **cvs-update**, and pcl-cvs brings up a buffer showing which files in your working copy have been modified or updated. From there, you can commit, do diffs, and so on.

Because cvs-update is the main entry point, we suggest that you bind it to a convenient key sequence before going any further. We have it bound to Ctrl+C+V in our .emacs:

```
(global-set-key "\C-cv" cvs-update)
```

Otherwise, you can run it by typing "M-x cvs-update" (also known as "Esc-x cvs-update").

When invoked, cvs-update runs **cvs update** as if it is in the directory of the file in the current buffer—just as if you typed **cvs update** on the command line in that directory. Here's an example of what you might see inside Emacs:

```
PCL-CVS release 1.05 from CVS release $Name:  $.
Copyright (C) 1992, 1993 Per Cederqvist
Pcl-cvs comes with absolutely no warranty; for details consult the manual.
This is free software, and you are welcome to redistribute it under certain
conditions; again, consult the TeXinfo manual for details.
 Modified ci README.txt
 Modified ci fish.c
---------- End ----
```

Two files have been locally modified (some versions of pcl-cvs show the subdirectories where the files are located). The next logical action is to commit one or both of the files, which is what the **ci** on each line means. To commit one of them, go to its line and type "c". You are brought to a log message buffer, where you can type a log message that's as long as you want (real log message editing is the major advantage of pcl-cvs over the command line). Press Ctrl+C twice when done to complete the commit.

If you want to commit multiple files at once, sharing a log message, first use **m** to mark the files that you intend to commit. An asterisk appears next to each file as you mark it:

```
PCL-CVS release 1.05 from CVS release $Name:  $.
Copyright (C) 1992, 1993 Per Cederqvist
Pcl-cvs comes with absolutely no warranty; for details consult the manual.
This is free software, and you are welcome to redistribute it under certain
conditions; again, consult the TeXinfo manual for details.
* Modified ci README.txt
* Modified ci fish.c
---------- End ----
```

Now when you type "c" anywhere, it applies to all (and only) the marked files. Write the log message and commit them by pressing Ctrl+C twice as before.

You can also type "d" to run **cvs diff** on a file (or on marked files) and "f" to bring a file into Emacs for editing. Other commands are available; type Ctrl+H+M in the update buffer to see what else you can do.

Error Handling in pcl-cvs

The pcl-cvs program has historically had an odd way of dealing with error and informational messages from CVS (although this might be corrected in the latest versions). When it encounters a message from CVS that it doesn't know about, the program gets hysterical and throws you into a mail buffer, ready to send a pregenerated bug report to the author of pcl-cvs. Unfortunately, among the CVS messages that pcl-cvs might not know about are the ones associated with conflicting merges, which, although not common, certainly do occur from time to time.

If pcl-cvs suddenly dumps you into a mail buffer, don't panic. Read over the contents of the buffer carefully—the offending CVS output should be in there somewhere. If it looks like a merge, you can just get rid of the mail buffer and rerun **cvs-update**. It should now succeed, because CVS won't output any merge messages (because the merge has already taken place).

cvsutils: General Utilities for Use with CVS

Depends on: Perl

URL: **http://www.red-bean.com/cvsutils/**

Authors: Tom Tromey (original author) and Pavel Roskin (current maintainer)

The suite of small programs called "cvsutils" generally (although not always) performs "offline" operations in the CVS working copy. Offline operations are those that can be done without contacting the repository, while still leaving the working copy in a consistent state for the next time the repository is contacted. Offline behavior can be extremely handy when your network connection to the repository is slow or unreliable.

We list the cvsutils programs here in approximate order of usefulness (according to our opinion), with the more useful ones coming first. Coincidentally, this also arranges them by

safety. Safety is an issue because some of these utilities can, in their normal course of operation, cause you to lose local modifications or files from your working copy. Therefore, read the descriptions carefully before using these utilities.

Note

This documentation is accurate as of version 0.2.0. Be sure to read the README file in any later versions for more up-to-date information.

Cervisia

Danger level: Low

Contacts repository: Yes

Cervisia is a graphical front end for the CVS client with the following features:

- Updating or retrieving the status of a working directory or single files. Files are displayed in different colors depending on their status, and the shown files can be filtered according to their status.
- Common operations, such as adding, removing, and committing files.
- Advanced operations, such as adding and removing watches, editing and unediting files, locking and unlocking.
- Checking out and importing modules.
- Graphical **diff** against the repository and between different revisions.
- Blame-annotated view of a file.
- View of the log messages in tree and list form.
- Resolution of conflicts in a file.
- Tagging and branching.
- Updating to a tag, branch, or date.
- A ChangeLog editor coupled with the commit dialog.

This is a very useful utility for people who prefer the GUI approach to everything. In our extended use it proved quite stable and reliable. You can download this utility from **http://cervisia.sourceforge.net/**. Cervisia is distributed freely under the Q Public License.

cvsu

Danger level: None

Contacts repository: No

cvsu does an offline **cvs update** by comparing the timestamps of files on disk with their timestamps recorded in CVS/Entries. You can thus tell which files have been locally modified and which files are not known to be under CVS control. Unlike **cvs update**, **cvsu** does not bring down changes from the repository.

Although it can take various options, **cvsu** is most commonly invoked without any options:

```
yarkon$ cvsu
? ./bar
? ./chapter-10.html
M ./chapter-10.sgml
D ./out
? ./safe.sh
D ./tools
```

The left-side codes are like the output of **cvs update**, except that **D** means directory. This example shows that chapter-10.sgml has been modified locally. What the example doesn't show is that **cvsu** ran instantly, whereas a normal **cvs update** would have required half a minute or so over our slow modem line.

Run

```
yarkon$ cvsu --help
```

to see a list of options.

cvsdo

Danger level: Low to none

Contacts repository: No

cvsdo can simulate the working copy effects of **cvs add** and **cvs remove**, but without contacting the repository. Of course, you'd still have to **commit** the changes to make them take effect in the repository, but at least you can speed up the **add** and **remove** commands themselves this way. Here's how to use it:

```
yarkon$ cvsdo add <i>FILENAME</i>
```

or

```
yarkon$ cvsdo remove <i>FILENAME</i>
```

To see a list of further options, run:

```
yarkon$ cvsdo --help
```

cvschroot

Danger level: Low

Contacts repository: No

cvschroot deals with a repository move by tweaking the working copy to point to the new repository. This is useful when a repository is copied en masse to a new location. When that happens, none of the revisions are affected, but the CVS/Root (and possibly the CVS/Repository) file of every working copy must be updated to point to the new location. Using **cvschroot** is a lot faster than checking out a new copy. Another advantage is that it doesn't lose your local changes.

Usage:

```
yarkon$ cvschroot NEW_REPOS
```

For example:

```
yarkon$ cvschroot :pserver:newuser@newhost.wherever.com:/home/cvs/myproj
```

cvsrmadm

Danger level: Low to medium

Contacts repository: No

This removes all of the CVS/ administrative subdirectories in your working copy, leaving behind a tree similar to that created by **cvs export**. Although you won't lose any local changes by using **cvsrmadm**, your working copy will no longer be a working copy.

Use with caution.

cvspurge

Danger level: Medium

Contacts repository: No

This removes all non–CVS-controlled files in your working copy. It does not undo any local changes to CVS-controlled files.

Use with caution.

cvsdiscard

Danger level: Medium to high

Contacts repository: Maybe

This is the complement of **cvspurge**. Instead of removing unknown files while keeping your local changes, **cvsdiscard** undoes any local changes (replacing those files with fresh copies from the repository) and keeps unknown files.

Use with extreme caution.

cvsco

Danger level: High

Contacts repository: Maybe

cvsco is the union of **cvspurge** and **cvsdiscard**. It undoes any local changes and removes unknown files from the working copy.

Use with truly paranoid caution.

cvs2cl.pl: Generate GNU-Style ChangeLogs from CVS Logs

Depends on: Perl

URL: **www.cvshome.org/dev/addons.html**

cvs2cl.pl condenses and reformats the output of **cvs log** to create a GNU-style ChangeLog file for your project. ChangeLogs are chronologically organized documents showing the change history of a project, with a format designed specifically for human-readability (see the following examples).

The problem with the **cvs log** command is that it presents its output on a per-file basis, with no acknowledgment that the same log message, appearing at roughly the same time in different files, implies that those revisions were all part of a single commit. Thus, reading over log output to get an overview of project development is a hopeless task—you can really see the history of only one file at a time.

In the ChangeLog produced by **cvs2cl.pl**, identical log messages are unified, so that a single commit involving a group of files shows up as one entry, as shown in this example:

```
yarkon$ cvs2cl.pl -r
cvs log: Logging .
cvs log: Logging a-subdir
cvs log: Logging a-subdir/subsubdir
cvs log: Logging b-subdir
yarkon$ cat ChangeLog
...
2001-07-29 05:44  jrandom
```

```
   * README (1.6), hello.c (2.1), a-subdir/whatever.c (2.1),
   a-subdir/subsubdir/fish.c (2.1): Committing from pcl-cvs 2.9, just
   for kicks.

2001-06-12 22:48  jrandom

   * README (1.5): [no log message]

2001-07-1119:34  jrandom

   * README (1.4): trivial change
...
yarkon$
```

The first entry shows that four files were committed at once, with the log message, "Committing from pcl-cvs 2.9, just for kicks." (The **-r** option was used to show the revision number of each file associated with that log message.)

Like CVS itself, **cvs2cl.pl** takes the current directory as an implied argument, but it acts on individual files if it's given file name arguments. Following are a few of the most commonly used options.

-h, --help

Show usage (including a complete list of options).

-r, --revisions

Show revision numbers in output. If used in conjunction with **-b**, branches are shown as ***BRANCHNAME.N***, where **N** is the revision on the branch.

-t, --tags

Show tags (symbolic names) for revisions that have them.

-b, --branches

Show the branch name for revisions on that branch. (See also **-r**.)

-g *OPTS*, --global-opts *OPTS*

Pass ***OPTS*** as global arguments to **cvs**. Internally, **cvs2cl.pl** invokes **cvs** to get the raw log data; thus, ***OPTS*** are passed right after the **cvs** in that invocation. For example, to achieve quiet behavior and compression, you can do this:

```
yarkon$ cvs2cl.pl -g "-Q -z3"
```

-l *OPTS*, --log-opts *OPTS*

Similar to **-g**, except that ***OPTS*** are passed as command options instead of global options. To generate a ChangeLog showing only commits that happened between July 26 and August 15, you can do this:

```
yarkon$ cvs2cl.pl -l "'-d2001-06-12<2001-02-24'"
```

Notice the double-layered quoting—this is necessary in Unix because the shell that invokes **cvs log** (inside **cvs2cl.pl**) interprets the "**<**" as a shell redirection symbol. Therefore, the quotes have to be passed as part of the argument, making it necessary to surround the whole thing with an additional set of quotes.

-d, --distributed

Put an individual ChangeLog in each subdirectory, covering only commits in that subdirectory (as opposed to building one ChangeLog that covers the directory in which **cvs2cl.pl** was invoked and all subdirectories underneath it).

cvslock: Lock Repositories for Atomicity

Depends on: C compiler for installation; nothing for runtime

URL: **ftp://riemann.iam.uni-bonn.de/pub/users/roessler/cvslock/**

This program locks a CVS repository (either for reading or writing) in the same way that CVS does, so that CVS will honor the locks. This can be useful when, for example, you need to make a copy of the whole repository and want to avoid catching parts of commits or other people's lockfiles.

The **cvslock** distribution is packaged extremely well and you can install it according to the usual GNU procedures. Here's a transcript of an install session:

```
yarkon$ zcat cvslock-0.1.tar.gz | tar xvf -
cvslock-0.1/
cvslock-0.1/Makefile.in
cvslock-0.1/README
cvslock-0.1/COPYING
cvslock-0.1/Makefile.am
cvslock-0.1/acconfig.h
cvslock-0.1/aclocal.m4
cvslock-0.1/config.h.in
cvslock-0.1/configure
cvslock-0.1/configure.in
cvslock-0.1/install-sh
cvslock-0.1/missing
```

```
cvslock-0.1/mkinstalldirs
cvslock-0.1/stamp-h.in
cvslock-0.1/cvslock.c
cvslock-0.1/cvslock.1
cvslock-0.1/snprintf.c
cvslock-0.1/cvslssh
cvslock-0.1/VERSION
yarkon$ cd cvslock-0.1
yarkon$ ./configure
 ...
yarkon$ make
gcc -DHAVE_CONFIG_H -I. -I. -I.   -g -O2 -c cvslock.c
gcc -g -O2  -o cvslock  cvslock.o
yarkon$ make install
 ...
yarkon$
```

(Note that you might have to do the **make install** step as root.)

Now, **cvslock** is installed as /usr/local/bin/cvslock. When you invoke it, you can specify the repository with **-d** or via the **$CVSROOT** environment variable, just as with CVS itself (the following examples use **-d**). Its only required argument is the name of the directory to lock, relative to the top of the repository. It will lock that directory and all of its subdirectories. In this example, there are no subdirectories, so **cvslock** creates only one lockfile:

```
yarkon$ ls /usr/local/newrepos/myproj/b-subdir/
random.c,v
yarkon$ cvslock -d /usr/local/newrepos  myproj/b-subdir
yarkon$ ls /usr/local/newrepos/myproj/b-subdir/
#cvs.rfl.cvslock.yarkon.27378  random.c,v
yarkon$ cvslock -u -p 27378 -d /usr/local/newrepos  myproj/b-subdir
yarkon$ ls /usr/local/newrepos/myproj/b-subdir/
random.c,v
yarkon$
```

Notice that when we cleared the lock (**-u** for "unlock"), we had to specify **-p 27378**. That's because **cvslock** uses Unix process IDs when creating lockfile names to ensure that its locks are unique. When you unlock, you have to tell **cvslock** which lock instance to remove, even if there's only one instance present. Thus, the **-p** flag tells **cvslock** which previous instance of itself it's cleaning up after (you can use **-p** with or without **-u**, though).

If you're going to be working in the repository for a while, doing various operations directly in the file system, you can use the **-s** option to have **cvslock** start up a new shell for you. It then consults the **$SHELL** environment variable in your current shell to determine which shell to use:

```
yarkon$ cvslock -s -d /usr/local/newrepos myproj
```

The locks remain present until you exit the shell, at which time they are automatically removed. You can also use the **-c** option to execute a command while the repository is locked. Just as with **-s**, the locks are put in place before the command starts and are removed when it's finished. In the following example, we lock the repository just long enough to display a listing of all of the lockfiles:

```
yarkon$ cvslock -c 'find . -name "*cvslock*" ' -d /usr/local/newrepos myproj
cvslock: '/usr/local/newrepos/myproj' locked successfully.
cvslock: Starting 'find . -name "*cvslock*" -print'...
./a-subdir/subsubdir/#cvs.rfl.cvslock.yarkon.27452
./a-subdir/#cvs.rfl.cvslock.yarkon.27452
./b-subdir/#cvs.rfl.cvslock.yarkon.27452
./#cvs.rfl.cvslock.yarkon.27452
yarkon$ find /usr/local/newrepos/myproj -name "*cvslock*" -print
yarkon$
```

The command (the argument to the **-c** option) is run with the specified repository directory as its working directory.

By default, **cvslock** creates read-locks. You can tell it to use write-locks instead by passing the **-W** option. (You can pass **-R** to specify read-locks, but that's the default anyway.) Always remove any locks when you're finished, so that other users' CVS processes don't wait needlessly.

Note that you must run **cvslock** on the machine where the repository resides—you cannot specify a remote repository. (For more information, run **man cvslock**, which is a manual page installed when you ran **make install**.)

Other Packages

Many other third-party packages are available for CVS. Following are pointers to some of these.

Jalindi Igloo

Jalindi Igloo is a program that allows you to connect Microsoft Visual Studio and other IDEs directly to a CVS repository. The program is completely free and can be used in any way you like.

The product consists of a file called igloo.dll, which is a connectivity module between Visual Studio or any SCCAPI-compliant IDEs, and the cvs2ntslib.dll that is supplied with CVS for NT.

The product is more useful working alongside CVS or WinCVS. To use this product, first check out your files to a sandbox. Then choose Add To Source Control within Visual Studio, and it will automatically recognize that it is a CVS project. If you do not have a CVS repository or module, it will give you a chance to connect to or create one.

You can download this utility from **www.jalindi.com/igloo/**.

CVSUp (Part of the FreeBSD Project)

CVSUp is an efficient generic mirroring tool with special built-in support for mirroring CVS repositories. The FreeBSD operating system uses it to distribute changes from its master repository, so users can keep up to date conveniently.

For more information on CVSUp in general, check out **www.polstra.com/projects/freeware/CVSup/**. For its use in FreeBSD in particular, see **www.freebsd.org/handbook/synching.html#CVSUP**.

CVSWeb: A Web Interface to CVS Repositories

CVSWeb provides a Web interface to browsing CVS repositories. A more accurate name might be "RCSWeb," because what it actually does is allow you to browse revisions directly in a repository, viewing log messages and diffs. Although we've never found it to be a particularly compelling interface, we have to admit that it is intuitive enough and a lot of sites use it.

Although Bill Fenner originally wrote the software, the version most actively under development right now seems to be Henner Zeller's, at **http://linux.fh-heilbronn.de/~zeller/cgi/cvsweb.cgi/**.

You might also want to visit Fenner's original site at **www.freebsd.org/~fenner/cvsweb/** and possibly Cyclic Software's summary of the CVSWeb scene at **www.cvshome.org/dev/addons.html**.

Finally, if you'd like to see CVSWeb in action, you can browse a good example at **http://sourceware.cygnus.com/cgi-bin/cvsweb.cgi/**.

The CVS contrib/ Directory

As we mentioned in Chapter 3, a number of third-party tools are shipped with CVS and are collected in the contrib/ directory. Although we are not aware of any formal rule for determining which tools are distributed with CVS, there might be an effort under way to gather most of the widely used third-party tools and put them in contrib/ so people know where to find them. Until that happens, the best way to find such tools is to look in contrib/, look at various CVS Web sites, and ask on the mailing list.

Writing Your Own Tools

CVS can at times seem like a bewildering collection of improvised standards. There's RCS format, various output formats (history, annotate, log, update, and so on), several repository administrative file formats, working copy administrative file formats, the client/server protocol, the lockfile protocol.... (Are you numb yet? We could keep going, you know.)

Fortunately, these standards remain fairly consistent from release to release—so if you're trying to write a tool to work with CVS, you at least don't have to worry about hitting a moving target. For every internal standard, there are usually a few people on the **info-cvs@gnu.org** mailing list who know it extremely well (several of them helped us out during the writing of this book). There is also the documentation that comes with the CVS distribution (especially doc/cvs.texinfo, doc/cvsclient.texi, and doc/RCSFILES). Finally, there is the CVS source code itself, the last word on any question of implementation or behavior.

With all of this at your disposal, there's no reason to hesitate. If you can think of some utility that would make your life with CVS easier, go ahead and write it—chances are other people have been wanting it, too. Unlike a change to CVS itself, a small, standalone external utility can get wide distribution very quickly, resulting in quicker feedback for its author and faster bug fixes for all of the users.