Version control with cvs

An introduction

Markus Bjartveit Krüger

markusk@pvv.org

http://www.pvv.org/~markusk/versioncontrol.pdf

CVS

Concurrent **V**ersions **S**ystem, originally written as a set of shell scripts by Dick Grune in 1986, made into a C program in 1989 by Brian Berliner with the aid of Jeff Polk.

CVS is a widely used version control system for projects of all sizes. CVS was originally built on RCS, and although CVS no longer uses RCS internally, CVS still behaves like RCS in many ways.

FreeBSD and Netscape are two of the larger software projects that uses CVS.

Features of cvs

- Store and retrieve multiple revisions of text.
- Maintain a complete history of changes.
- Maintain a tree of revisions.
- Merge revisions and resolve conflicts.
- Control releases and configurations.
- Automatically identify each revision with name, revision number, creation time, author, etc.

- Run scripts when checking files in or out, e.g. mailing the project group about changes, indenting code properly, and so on.

- Client/server.

- Allows several developers to edit a file at the same time.

Version, release or revision?

The word *version* is used both to describe the various stages of a file during development and to identify *releases* of a software product (Windows NT 4.0, Netscape 4.5, and so on).

To avoid confusion, we will use the term *revision* for files. Release numbering is separate from revision numbering, and is done through *symbolic revisions*.

A simple session with cvs

Check out a working copy of module hello

```
$ cvs checkout hello
cvs checkout:  Updating hello
U hello/Makefile
U hello/hello.c
```

Fetch updates made by others

```
$ cvs update
cvs update:  Updating .
U hello.c
```

Make changes and commit them

```
$ cvs commit
/home/markusk/cvs/hello/hello.c,v <-- hello.c
new revision:  1.3; previous revision:  1.2
done
```

Command line help

To list all available global options:

```
cvs --help-options
```

To list all available commands:

```
cvs --help-commands
```

To display usage information for a command:

`cvs -H` *command*

Specifying revisions

Most CVS commands take the command option `-r`*rev*, which can be used to specify which revision to check out, what revision number to use when checking in, and so on.

```
cvs update -r1.4 file
cvs commit -r2.0 file
cvs log -r1.2:1.4 file
```

Another common option is `-D`*date*, which tells CVS to use a revision from the given date. This can be combined with `-z` to specify local time (default is UTC).

```
cvs update -D'1999-01-01 13:00' file
cvs log -D'1998-07-06<1999-01-01' file
```

The repository

CVS modules and administrative files are stored in a *repository*, either on the local machine or via a network.

You can specify where the repository is with the `-d` global option,

```
cvs -d /usr/local/cvsroot checkout foo
```

or by setting the `$CVSROOT` environment variable (bash example),

```
CVSROOT=/usr/local/cvsroot
export CVSROOT
cvs checkout foo
```

A working copy remembers its repository. When running CVS commands within a working copy, you do not need to specify a repository.

Checking out a working copy

Each developer has his or her own working copy checked out of the repository. When you start working on a project, you check it out with

```
cvs checkout foo
```

This creates the directory `foo` and checks out a working copy to this directory. If you want the directory to have some other name, you can use the `-d` option:

```
cvs checkout -d foobar foo
```

You normally check out a project only once. Changes are fetched using `cvs update`.

Committing changes

You register changes you have made in the repository with

```
cvs commit [file]
```

If no files are specified, all changes in the current directory and its subdirectories are commited. An easy way to commit all changes is to change to the base directory of the project and run `cvs commit` from there.

When committing, CVS prompts for a log message. It is important to write descriptive log messages in order to discover when what changes was made.

Updating your working copy

Use `cvs update` to update your working copy with the changes others have made, or to fetch a specific revision.

```
cvs update file
cvs update -r1.12 file
```

If no file is specified, CVS updates all files in the current directory and its subdirectories.

`cvs update` prints a line for each file, preceded by one character telling the status of the file.

U,P File was updated.

A File has been added to your working copy, and will be added to the repository when you `commit`.

R File has been removed from your working copy, and will be removed from the repository when you `commit`.

M File has been changed by you since your last commit, and there may have been merged changes from the repository.

C There is a conflict between your changes to the files and changes in the repository.

? File is in your working directory, but does not exist in the repository.

Merging and conflicts

CVS allows several developers to modify a file at the same time. When developers commit their files or update their working copies, the changes are merged together.

If changes are overlapping, CVS reports this as a merge conflict and requires the user to resolve the conflicts before commiting the file. Conflicts are marked like this:

```
int main(int argc, char** argv)
{
<<<<<<< main.c
    if (argc != 1)
=======
    if (argc != 2)
>>>>>>> 1.4
        usage();
```

This shows a conflict in the file `main.c`, where the latest revision in the repository is 1.4. To resolve the conflict, edit the code between the `<<<<<<<`, `=======`, and `>>>>>>>` markers, remove the markers, and commit.

Adding and removing files

To add new files to a project, first create the files in your working copy, then run

`cvs add` *files*

To remove files, first remove the files from your working directory, then run

`cvs remove` *files*

The files will be added to or removed from the repository the next time you run `cvs commit`. (Actually, CVS never really removes files from the repository, it marks them as dead instead.)

Adding directories

To add a directory:

```
mkdir dir
cvs add dir
```

This adds only the directory, if there are files or subdirectories you must add these explicitly. If you want to add a new directory hierarchy, you can use `cvs import` instead.

CVS does *not* automatically create new directories when updating. To fetch new directories that have been added to the module, use the `-d` option:

```
cvs update -d foo
```

Removing directories

You cannot remove a directory directly. Instead, you must remove all files in it with `cvs remove`, then run

```
cvs update -P
```

The `-P` option tells CVS to prune away empty directories.

Moving files and directories

Moving files is normally done like this:

```
mv oldname newname
cvs remove oldname
cvs add newname
```

Directories are moved by adding the directory, moving all files from the old to the new directory and removing the directory with `cvs update -P`.

This way of moving files is safe, but has the drawback that the new file loses its change log. The change log is still available through the old file name. See the CVS documentation for ways to move a file and keep the change log.

Ignoring files

Often, your working directory will contain temporary files (object files, executables, etc.) that you do not want to add to the repository.

To make CVS ignore a file, edit the `.cvsignore` file in the directory containing the file. Each line of .cvsignore specifies a file or a filename wildcard to ignore.

You can also make CVS ignore files by editing the `cvsignore` administrative file.

Symbolic revisions

Typically, the files in a repository will have very different revision numbers. This means you can't fetch revision 1.4 of each file and hope to get a snapshot of the module at a given time; revision 1.4 of `main.c` could be a month old, with many changes between then and now, while revision 1.4 of `Makefile` might not even exist.

One way around this is to specify a date when checking out or updating your working copy:

```
cvs checkout -D 1999-03-15 00:01
```

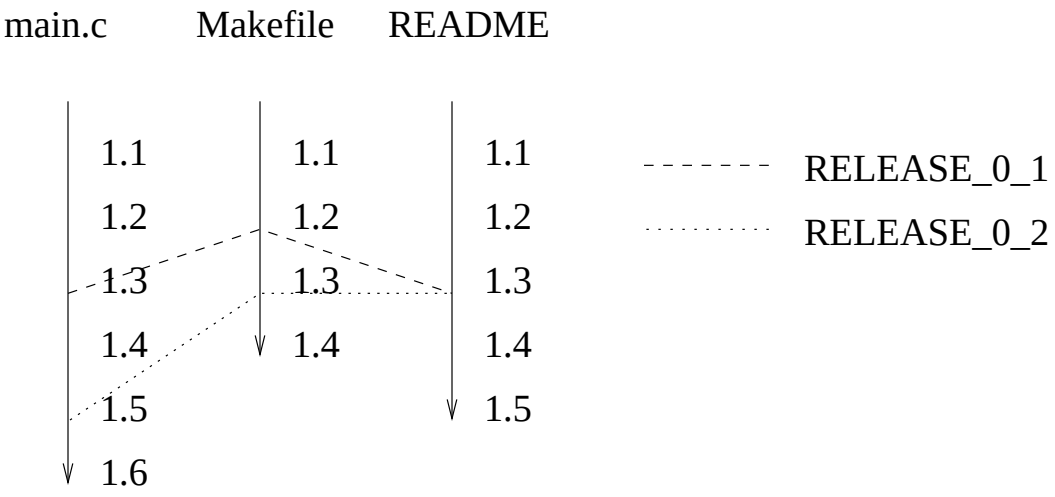
will check out a snapshot of the repository from midnight, March 15th.

A better way is to *tag* the repository with a symbolic name, and use this symbolic name later on for checkouts and such.

```
cvs tag REL_0_1
```

This gives all files in the current directory and subdirectories the symbolic tag REL_0_1. Normally, you want to tag all files in the repository, to create a snapshot of the entire module.

The tag is "tied" to the current revision number of each file in your repository. You can move the tag to another revision number by rerunning `tag` with the `-f` option.



You can now check out or update a working copy corresponding to REL_0_1:

```
cvs checkout -rREL_0_1
cvs update -rREL_0_1
```

Status

To view current status of a file (revision, modified or not, newer revision in repository, etc):

`cvs status` *file*

A useful option is `-v` (verbose), to view all tags set for a given file:

`cvs status -v` *file*

Viewing logs

It is highly useful to view the log messages of revisions to track down when a specific change was made. This requires that developers write good log messages.

To view the entire change logs of a file:

`cvs log` *file*

To view log messages from revision 1.1 to 1.3, inclusive:

`cvs log -r1.1:1.2` *file*

To view log messages for symbolic tag `REL_0_1`:

`cvs log -rREL_0_1` *file*

Tracking changes

To view differences between revisions of a file:

```
cvs diff file
cvs diff -r1.1 -r1.15 file
```

To view all differences made in current directory and subdirectories (useful before committing changes for determining a log message and avoid committing unintended changes):

```
cvs diff
```

To show in which revision each line of a file was last changed, and which developer made the change:

```
cvs annotate file
```

Branches

At times, it is useful to have several development branches at the same time:

- Bugs are found in an old version, but the current version has changed so much that it is impractical to make fixes there.

- You are about to add a major new feature, and your changes will affect other developers seriously.

The solution is to make a separate *branch*, so that changes on one branch will not affect other branches. Changes can later be merged between branches (i.e. develop a feature on one branch and merge it into the main branch when it is stable, or merge bug fixes without including other changes).

Creating a branch

From an existing tag (no working copy needed):

```
cvs rtag -b -r REL_1_0 REL_1_0_FIXES foo
```

From your current working copy:

```
cvs tag -b REL_1_0_FIXES
```

Working on a branch

To start working on a branch, use `checkout`:

```
cvs checkout -r REL_1_0_FIXES
```

Or, if you already have a working copy:

```
cvs update -r REL_1_0_FIXES
```

Branch tags are sticky, meaning that later commits go to your current branch, not the main branch, and updates fetches the latest revisions of the branch.

Merging a branch

You can merge changes made on a branch into your working copy by using `update` with the `-j` option.

```
cvs update -j REL_1_0_FIXES
```

After resolving any conflicts that occur, you commit the merged files as usual with `commit`.

Client/server CVS

You can use a CVS repository on a remote machine by specifying the repository as *hostname*:*directory*:

```
cvs -d verden.pvv.ntnu.no:/usr/local/cvs
checkout foo
```

This uses `rsh` to connect to `verden.pvv.ntnu.no` and check out the `foo` module. You do not need to have a CVS server running on the remote machine; CVS starts a server process when connecting.

If you want to use something else than `rsh` to connect, set the `$CVS_RSH` variable to the program you want to use (e.g., `ssh`).

CVS also provides other ways to use a remote server. See the CVS documentation for more information.

Keyword substitution

CVS defines several keyword strings that will be replaced with information from CVS when checking out a file. Some of these strings are:

`$Author$` Login name of the user who checked in the revision.

`$Date$` Date and time the revision was checked in.

`$Id$` Name of CVS file, revision number, date and time, author, state, and locker of file (if locked).

`$Revision$` Revision number of file.

`$Log$` Log message. On each checkin, the latest log message is appended with the prefix of the Log line. This is useful for languages with comments that go to the end of the line.

A common practice is to put a `$Id$` keyword in a comment at the start of the file.

Binary files

Binary files (images, Word documents, external libraries, etc.) in a CVS repository can cause problems if CVS expands keywords in the binary files or tries to merge revisions.

You avoid these problems by specifying that a file is binary:

```
cvs add -kb file
```

If you've already added the file, you can mark it as binary with `cvs admin`:

```
cvs admin -kb file
```

You can specify that CVS should treat all files matching a wildcard (e.g., all `.exe` files, all `.jpg` files) as binary by editing the `cvswrappers` adminstrative file.

Administrative files

There are several administrative files in the CVS repository that tells CVS how to behave. You can edit these files by checking out the `CVSROOT` module, edit the files you want and commit. See the CVS documentation for full descriptions of each of these files.

modules Specifies which modules exist in the repository, allows setting up alias modules and options for running programs when the module is used.

cvswrappers Allows you to transform files on their way in and out of CVS. Useful for specifying binary files, among other things.

commitinfo Checks that a commit is allowed.

verifymsg Evaluates and validates log messages.

editinfo Program to use for editing log messages.

loginfo Program run on complete commits. Can be used for mailing a notice to the project group upon commits.

rcsinfo A fill-out form for log messages.

cvsignore Specifies which files CVS should ignore.

config Various other CVS configuration.

Some CVS tools

pcl-cvs A CVS interface for the editor Emacs, which is bundled with version 21. An older version that works with Emacs 20 and XEmacs is available at
`ftp://flint.cs.yale.edu/pub/monnier/pcl-cvs`

jCVS A CVS client written in Java.
`http://www.jcvs.org/`

CvsGui A collection of CVS interfaces for various platforms, including Windows, Macintosh and X with GTK.
`http://www.wincvs.org`

TortoiseCVS Shell Extension for Windows that integrates CVS with Windows Explorer.
`http://www.tortoisecvs.org`

Eclipse A very good open source extensible IDE, written in Java and primarily for Java development, that has integrated support for CVS.
`http://www.eclipse.org`

Jalindi Igloo Visual Studio bindings for CVS.
`http://www.jalindi.com/igloo/`

For more information...

man pages cvs(1), cvs(5)

info pages info cvs

"Version Management with CVS" The reference manual to CVS, written by Per Cederqvist et al. The manual corresponds to the info pages, and this course builds mainly on it. The manual is available in various formats at the CVS web site:
`http://www.cvshome.org/docs/manual/`

CVS Home The CVS web site, containing source and binary realeases, documentation, tutorials, the FAQ, and pointers to other resources such as mailing lists, newsgroups, and support information.
`http://www.cvshome.org/`

"Open Source Development With CVS" Written by Karl Fogel, previously a member of the CVS development team. As far as I know, this is the only book available that describes CVS in depth. It also contains material on running an open source project. Most of the book is available for free on WWW:
`http://cvsbook.red-bean.com/`

`comp.software.config-mgmt` Newsgroup about software configuration management and version control.

Some alternatives

Subversion An intended replacement for CVS. Currently being developed; release date for version 1.0 still to be decided. Now self-hosted.
`http://subversion.tigris.org/`

BitKeeper A commercial product that also has a free use license. BitKeeper has a more distributed architecture than CVS.
`http://www.bitkeeper.com/`

Perforce
`http://www.perforce.com/`

ClearCase
`http://www.rational.com/products/clearcase/index.jsp`

SourceSafe
`http://msdn.microsoft.com/ssafe/`