# Version control with RCS and CVS

An introduction

Markus Bjartveit Krüger

# RCS

**R**evision **C**ontrol **S**ystem, written by Walter F. Tichy in 1982 and further developed by Paul Eggert.

RCS is mostly used for revision control of single files—scripts, configuration files, and other files that "stand alone". It is not well suited for managing large collections of files.

This course will only cover basic use of RCS. We'll show more advanced revision control with CVS instead.

# Features of RCS

- Store and retrieve multiple revisions of text.
- Maintain a complete history of changes.
- Resolve access conflicts.
- 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.

# 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*.

# Putting files into RCS

RCS maintains an archive file–called a *RCS file*—for each file. These files are normally kept in a directory named `RCS`.

To start using RCS with a file, first create this directory by running `mkdir RCS` in the file's directory, then put the file into RCS with the command

**ci -i -u** *file*

This creates a file named *file*`,v` in the `RCS` directory, and makes the file read-only.

# Editing files with RCS

You edit a file that has been put into RCS like this:

1. Check out a working copy:

   `co -l` *file*

2. Edit the working copy.

3. Check in the new revision:

   `ci -u` *file*

# Checking out a working copy

In order to edit a file, you first need to *check out* a working copy of the file, by running the command

`co -l` *file*

`co` (check out) is the command used to check out RCS copies. The `-l` option tells RCS to lock the archive file so that others cannot edit the file while you are working on it.

# Checking in a new revision

After editing, you need to *check in* the file in order to store your changes in the archive and release the lock so that others can edit the file later.

`ci -u` *file*

`ci` (check in) is used to check in revisions. The `-u` option tells RCS to unlock the file, and also to keep the working copy. (Otherwise, RCS would remove the working copy, which in most cases you don't want RCS to do.)

You will be asked for a log entry when checking in. It is important to write good log entries, so that you can see what changes happened when in the revision history.

# Viewing differences

You can view differences between two revisions with `rcsdiff`:

`rcsdiff` *file*

shows differences between the working copy of *file* and the latest revision that was checked in.

`rcsdiff -r1.3` *file*

shows differences between the working copy and revision 1.3 of *file*.

`rcsdiff -r1.1 -r1.2` *file*

shows differences between revision 1.1 and 1.2 of *file*.

# Viewing the revision log

You can view the log of revision changes with `rlog`:

```
rlog file
```

shows all log messages for *file*, along with other RCS information.

You can specify which revision logs you want to see:

```
rlog -r1.1 file
rlog -r1.1:1.5,1.7 file
```

The first command shows the log entry for revision 1.1, the second shows the log entries for revision 1.1 through 1.5 and also revision 1.7.

# Resolving lock conflicts

It happens quite often that someone forgets to check in a file in RCS after editing it, or forgets to unlock it. In this case, RCS will complain when you try to check out the file.

Conflicts can be resolved like this:

1. Check if the working copy of the file differs from the RCS archive with `rcsdiff`. If it does, make an extra copy of the file (*file*.tmp, for instance).

2. "Steal" the lock with the command

   `rcs -L` *file*

RCS will send a mail to the user who had the lock, informing him that his lock has been broken.

3. If the previous working copy differed from the latest revision, copy it over the current working copy (`mv` *file*`.tmp` *file*) and check it in with an appropriate log message, then check the file out again.

# Specifying revisions

Most RCS commands take the 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.

`co -r1.4` *file*
`ci -r2.0` *file*
`rlog -r1.2:1.4` *file*

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

`co -d'1999-01-01 13:00'` *file*
`rlog -d1998-07-06<1999-01-01` *file*

# Keyword substitution

RCS defines several keyword strings that will be replaced with information from RCS 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 RCS 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.

# . . . and much more

RCS has many more features, but if you require them you are probably better off using CVS instead, which gives a more convenient interface. Some of the RCS features not covered here are

- Symbolic names for revisions
- Creating and merging development branches
- Multiple working copies

More information on RCS can be found in the man pages (rcs(1), ci(1), co(1), ident(1), rcsdiff(1), rcsmerge(1), rlog(1), rcsintro(1)), and in the book *Applying RCS and SCCS* from O'Reilly.

# CVS

**C**oncurrent **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

CVS has all the features of RCS, and also:

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

- CVS operates on logical collections of files (modules), not just single files as RCS does.

- Client/server CVS.

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

# A simple session with CVS

Checking out a working copy of a module named `foo`:

```
cvs checkout foo
```

This creates a directory named `foo` and copies all files and subdirectories of the module `foo` into it. Unlike RCS, which checks out a single file at the time, CVS operates on entire directory trees.

After editing a file in the module, you *commit* your changes with:

```
cvs commit file
```

You can also commit all files in the module by omitting the file name.

```
cvs commit
```

# Structure of CVS commands

All CVS commands are on the form

`cvs` *[global options] command [command options] [command arguments]*

**Global options** are common to all commands. (One exception: the `history` command.)

**Command** is what you want CVS to do. Most commands has a long form and one or more short forms. (E.g. checkout→co, commit→ci.)

**Command options** are specific to the current command.

**Command arguments** are often optional. When they are left out, CVS behaves recursively.

Global options must be given before the command, command options afterwards. Global options mean the same to all commands, command options usually have different meanings for different commands.

`cvs -l co foo` $\neq$ `cvs co -l foo`

# The repository

CVS modules and administrative files are stored in a *repository*. The repository can be on the local machine, or be accessed over a network remotely.

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.

# Setting up a repository

When you are joining an existing project, the repository is probably already in place. If you need to set up one yourself, this is how:

```
cvs -d /usr/local/cvs init
```

This will create the directory `/usr/local/cvs` if it does not exist, and create a repository in it. `cvs init` will not overwrite any files already present in the directory.

# Starting a project with CVS

To set up a new project `foo` from sources in the working directory *wdir* with CVS, use `cvs import`.

```
cd wdir
cvs import foo foo start
```

This creates the directory `foo/rdir` in your repository, and installs all files and subdirectories of your working directory.

The first argument to `cvs import` says where to place your project in the repository.

The second argument is a *vendor tag* saying where the source is from. It's not always useful, but CVS requires it.

The third argument is a *release tag*, used to mark the start of the project.

# 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.

# 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

Unlike RCS, 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 and removing 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
```

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
```

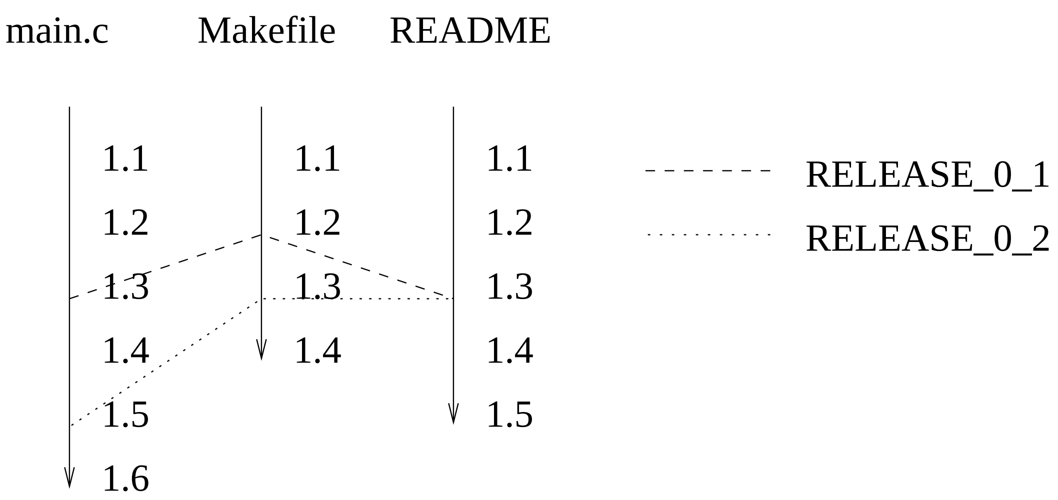
will check out a snapshot of the repository from midnight, March 15th.

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
```

# Sticky tags

When you fetch specific revisions using revision numbers, symbolic names, or dates, CVS gives them a *sticky tag*. This is useful with branches (more on this later on), but not so useful with non-branch tags.

Sticky tags ensure that later commits and updates uses the same tag. For non-branch tags, this means that you are "stuck" on the current revision of each file: updates fetch the same revisions, commits aren't allowed.

To remove sticky tags and revert to the head of the development trunk, use `cvs update -A`.

If what you really want is to replace your current revision of a file with a new file, you can do this with the `-p` option, which sends the contents of the file to standard output.

```
cvs update -r1.1 -p main.c > main.c
```

This replaces your copy of `main.c` with revision 1.1, without making any sticky tags.

# Status, logs, differences

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

```
cvs status file
```

To view change logs of a file (like `rlog` in RCS):

```
cvs log file
cvs log -r1.1 -r1.2 file
cvs log -rREL_0_1 file
```

To view differences between revisions of a file (like `rdiff` in RCS):

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

# Branches

At times, it is useful to have several development branches at the same time. Suppose you've released version 1.0 of your software and started adding features for release 1.1. Then bugs are found in release 1.0, but release 1.1 has already changed so much that it is impractical to make fixes there.

The solution is to make a separate *branch* for your bug fixes, so that release 1.0 can be fixed without disturbing work on release 1.1, and without the new bugs of release 1.1 slipping into release 1.0.

Another reason to branch is when you are going to add a major new feature, and your changes will affect other developers seriously. If you create a new branch, add your feature there, and merge the branch back when it's done, you avoid making the project unstable for everyone else.

Given that release 1.0 has been tagged REL_1_0, you can create a branch with

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

This command does not require a working copy. It creates a branch named REL_1_0_FIXES, rooted at REL_1_0. You can also create a branch from your current working copy:

```
cvs tag -b REL_1_0_FIXES
```

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

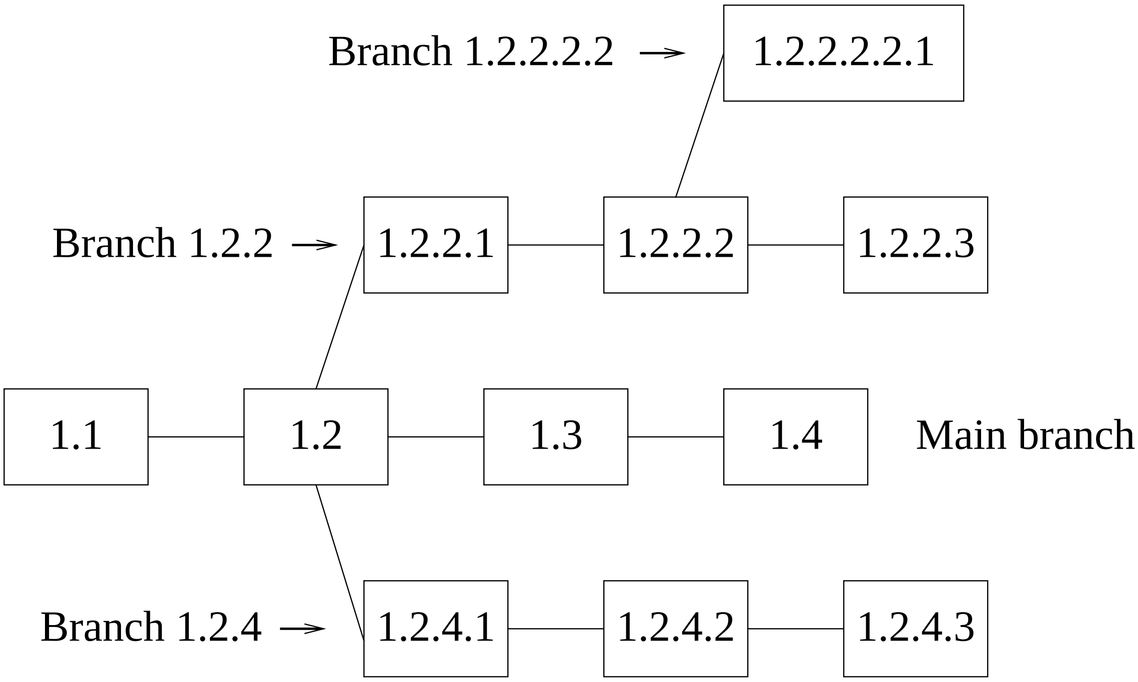
```
cvs checkout -r REL_1_0_FIXES
```

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.

# Branches and revisions




When a branch is made, a branch revision number is created by appending the first unused even integer to the current revision. After the branch number follows the revision number on 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`.

# Merging several times

Sometimes you keep changing a branch after merging it into the main branch. To avoid unneccessary work with the merge, it is convenient to merge only the changes since last merge into the main branch.

The easiest way to do this is to tag the branch each time you merge it, then give two `-j` options to `update`. Only the changes between the two revisions specified are merged.

```
cvs update -jREL_1_0_MERGED -jREL_1_0_FIXES
```

# 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 provides the same keyword substitutions as RCS.

# Binary files

Sometimes you want to include binary files in a CVS repository: images, Word documents, external libraries, etc. This causes 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 form for log messages.

**cvsignore** Specifies which files CVS should ignore.

**config** Various other CVS configuration.

# Releasing a module

If you are done with your working copy, you should notify CVS about it with the `release` command. CVS logs checkouts and releases, which can be viewed with the `history` command. When you release your working copy, CVS will check that you haven't made any changes since last commit. If you want CVS to delete the working copy after releasing the module, give the `-d` option to `release`.

```
cvs release foo
cvs release -d foo
```

# For more information...

This course gives a basic introduction to CVS which should be sufficient for most needs. However, there are many options and commands not covered. 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 also available as HTML and PostScript:
`http://www.loria.fr/~molli/`

**Cyclic Software** Cyclic Software maintains and develops CVS. Their web site contains the latest releases of CVS, and lots and lots of information, add-ons, etc, etc.
`http://www.cyclic.com`

**"Open Source Development With CVS"** This book by Karl Fogel is the only CVS book I know of. It was published quite recently. Most of the book is available for free on WWW:
`http://cvsbook.red-bean.com/`