Using CVS to Manage Source RTL

6.375 Tutorial 2
February 1, 2008

In this tutorial you will gain experience using the Concurrent Versions System (CVS) to manage your source RTL. You will be using CVS to submit your lab assignments, but more importantly CVS will make it much easier for groups of students to work on the same project simultaneously. CVS keeps track of the changes each user makes to the source RTL - this complete version history enables users to monitor what changes other users are making (each change includes log information), tag working versions of the design (so that users can always revert back to a working version), and resolve conflicts (when two users change the same bit of RTL at the same time).

CVS stores all version information in a central *repository*. Users cannot access the repository directly, but must instead use a special set of CVS commands. Users begin by *adding* the initial version of their source RTL to the repository. Users can then *checkout* the most current version of the source RTL into a private *working directory*. Local changes to the working directory are not stored in the repository until the user does a *commit*. Each commit includes a log message so that other users can easily read what has changed. If multiple users are changing the source RTL at the same time, then the state of the repository might be different from when the user performed the original checkout. At any time, a user can do an *update* which brings the checked out working directory in sync with the global repository. Sometimes an update will identify a *conflict*. A conflict indicates that two users have made changes to the same bit of of RTL. Users must resolve conflicts by hand - essentially they must choose whose changes should be permanent. Figure 1 illustrates the relationship between the central repository and each user's personal working directory.

In this tutorial we will create a test directory and a test file, add them to the repository, make some changes, commit these changes, and then emulate issues which arise when multiple users change the same file at the same time. Finally, we will learn how to use the ViewCVS program to browse the repository, read log messages, and track changes.

You can find more information in the CVS user guide (`cvs-user-guide.pdf`) located in the course locker (`/mit/6.375/doc`).

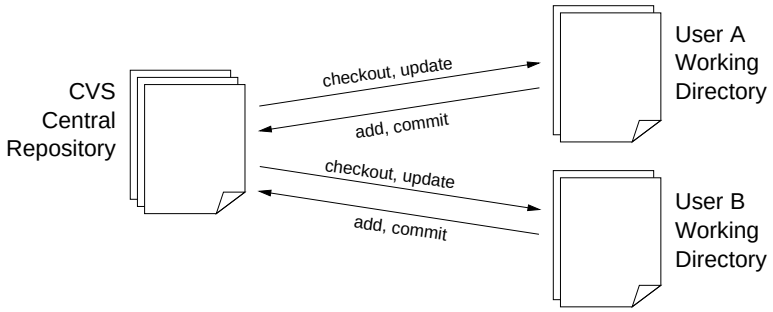

Figure 1: Basic CVS model. Users checkout versions from the central repository into private working directories.

Getting started

Before using the 6.375 toolflow you must add the course locker and run the course setup script with the following two commands. The course setup script will set the `CVSROOT` environment variable to `/mit/6.375/cvsroot`. The CVS commands use this environment variable to determine the location of the repository.

```
% add 6.375
% source /mit/6.375/setup.csh
```

Adding Directories and Files to CVS

In this section we will look at how to add directories and files to the repository. Figure 2 shows the timeline of cvs commands involved in this section.

All of your work will be in either your student CVS directory or your group's project CVS directory. For this tutorial we will be working in your student CVS directory. The very first step is to checkout this directory. All cvs commands are of the form `cvs <command>`. For example, assuming your user-name is `cbatten` the following commands will checkout your student CVS directory.

```
% mkdir tut2
% cd tut2
% cvs checkout 2008s/students/cbatten
% cd 2008s/students/cbatten
```

The following commands first create a test directory and test file, and then add them to the cvs repository.

```
% mkdir cvstest
% cvs add cvstest
% cd cvstest
% echo ‘‘Fred : 617-555-0123’’ > phone-list.txt
% cvs add phone-list.txt
```

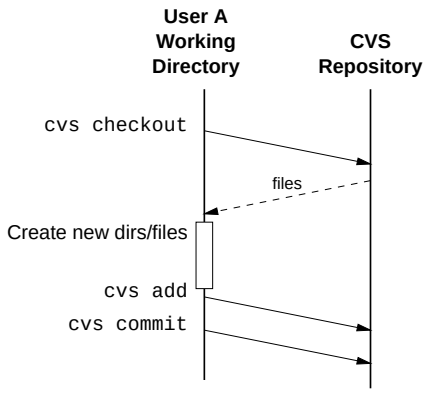

Figure 2: Timeline for checkout, add, and commit.

It is very important to note that the `cvs add` command is not recursive. In other words you must add each file individually. We can use the `find` and `xargs` programs to emulate a recursive add. The following command will add the given `<directory>` and all files within that directory to the CVS repository.

```
% find <directory> | xargs cvs add
```

Although the directory and file are *added* to the repository they are not actually *in* the repository until we commit them. Adding directories and files simply lets CVS know that we want the versioning system to track these files. We need to *commit* the directory and file before they are permanently in the repository. You can use the following command to check the current status of all files.

```
% pwd
tut2/2008s/students/cbatten/cvstest
% cvs status
```

The status information should reflect that `phone-list.txt` has been locally added. We now need to commit our new files to the repository. We can do this with the following commands.

```
% pwd
tut2/2008s/students/cbatten/cvstest
% cd ..
% cvs commit cvstest
```

The `cvs commit` command takes a list of files and directories as an argument. Unlike the `cvs add` command, the `cvs commit` command is recursive, so committing a directory will effectively commit all files (and subdirectories) within the directory. If you do not specify any files or directories, then CVS will commit the current directory.

After executing the `cvs commit` command, you will be able to enter a log message using the simple Pico text editor. Use the `cvs status` command again to verify that the `phone-list.txt` file has now been committed into the repository. The status information lists a revision number. Every change is given a unique revision number. These numbers are assigned by the CVS system, and users should usually avoid working with revision numbers directly.

Our final step is to delete our working directory. It is essential that we always keep in mind the difference between what is in the repository versus what is in the repository. As long as all our new files have been added, and all our changes have been committed then there is nothing wrong with deleting our working directory and doing a clean checkout.

```
% pwd
tut2/2008s/students/cbatten
% cd ../../..
% rm -rf 2008s
```

Making and Committing Changes

In this section we will checkout the `cvstest` directory, make a change, and then commit the change back into the repository. Figure 3 shows the timeline of cvs commands involved in this section. Our first step is to do a clean checkout of the `cvstest` directory. The `cvs checkout` command is recursive, so by checking out the `cvstest` directory we also checkout all files and subdirectories contained within `cvstest`.

```
% pwd
tut2
% cvs checkout 2008s/students/cbatten/cvstest
```

Now let's say Fred's phone number has changed.

```
% cd 2008s/students/cbatten/cvstest
% perl -i -pe 's/Fred.*/Fred : 617-555-4567/' phone-list.txt
```

You can use the `cvs diff` command to see how your current working directory differs from the repository. For example, if you use the `cvs diff` command in the `cvstest` directory, it will show that you have updated Fred's phone number.

```
% pwd
2008s/students/cbatten/cvstest
% cvs diff
Index: phone-list.txt
retrieving revision 1.1
1c1
< Fred : 617-555-0123
---
> Fred : 617-555-4567
```

And finally we commit our changes with an appropriate log message.

```
% cvs commit
```

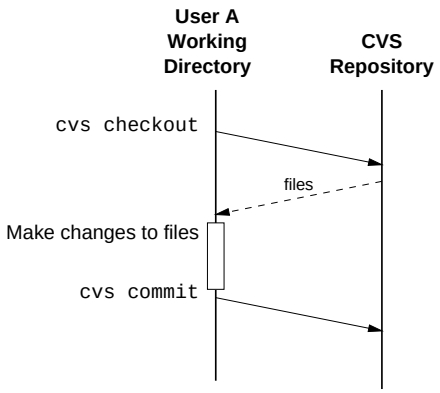

Figure 3: Timeline for checkout, modify, and commit.

We are done making our changes and everything has been committed so we can now delete our working directory.

```
% pwd
tut2/2008s/students/cbatten/cvstest
% cd ../../../..
% rm -rf 2008s
```

Updating the Working Directory

In this section we will see how CVS can help multiple users work on the same project at the same time. Figure 4 shows the timeline of cvs commands involved in this section. To emulate two users we will checkout our `cvstest` directory into two different working directories. We begin with User A using the following commands to checkout the current `cvstest` directory.

```
% pwd
tut2
% mkdir userA
% cd userA
% cvs checkout 2008s/students/cbatten/cvstest
% cd ..
```

Let's say User B now checks out the current `cvstest` directory, adds a new file which contains email addresses, and adds information about a new person named Jane.

```
% pwd
tut2
% mkdir userB
% cd userB
% cvs checkout 2008s/students/cbatten/cvstest
% cd 2008s/students/cbatten/cvstest
% echo ‘‘Fred : fred@mit.edu’’ > email-list.txt
% echo ‘‘Jane : 617-555-0021’’ >> phone-list.txt
% echo ‘‘Jane : jane@mit.edu’’ >> email-list.txt
% cat phone-list.txt
Fred : 617-555-4567
Jane : 617-555-0021
% cat email-list.txt
Fred : fred@mit.edu
Jane : jane@mit.edu
% cvs add email-list.txt
% cvs commit
```

User B's final `cvs commit` command commits both his changes to the `phone-list.txt` file as well as his newly added `phone-list.txt` file. Notice that at this point in time, User A's working directory is out-of-date (it does not contain the new `email-list.txt` file nor does it contain the updated `phone-list.txt` file). User A can use the `cvs status` command to realize that her working directory is out-of-date with respect to the central repository.

```
% pwd
tut2/userB/2008s/students/cbatten/cvstest
% cd ../../../../../userA/2008s/students/cbatten/cvstest
% cvs status
cvs status: Examining .
===================================================================
File: phone-list.txt    Status: Needs Patch

  Working revision:    1.2  Sun Feb 19 21:20:50 2008
  Repository revision: 1.3  .../2008s/students/cbatten/cvstest/phone-list.txt,v
  ...
```

Notice that the status of the `phone-list.txt` file is `Needs Patch`, and that the version of the file in the repository is later than the version in the working directory. User A can use the `cvs update` command to bring her working directory in sync with the central repository.

```
% pwd
tut2/userA/2008s/students/cbatten/cvstest
% ls
CVS phone-list.txt
% cat phone-list.txt
Fred : 617-555-4567
% cvs update
% ls
CVS email-list.txt phone-list.txt
% cat phone-list.txt
Fred : 617-555-4567
Jane : 617-555-0021
% cat email-list.txt
Fred : fred@mit.edu
Jane : jane@mit.edu
```

If User A uses `cvs status` again, she will see that her files are now up-to-date with respect to the central repository. As another example, User A can use the following commands to delete a file and then restore that file to the most current version in the repository.

```
% pwd
tut2/userA/2008s/students/cbatten/cvstest
% rm -rf phone-list.txt
% cvs update
```

If User A really wants to delete the file from the repository, then she can first delete the file from the working directory and then use the `cvs remove` command. These removals are not visible to other users until User A does a `cvs commit`.

```
% pwd
tut2/userA/2008s/students/cbatten/cvstest
% rm -rf email-list.txt
% cvs remove email-list.txt
% cvs commit
```

CVS does not provide explicit commands for renaming or moving files, so users must instead use a combination of `cvs remove` and `cvs add` to preform these operations.

If User B now does a `cvs update`, his copy of `email-list.txt` will also be removed. Let's finish up by deleting the working directories.

```
% pwd
tut2/userA/2008s/students/cbatten/cvstest
% cd ../../../../..
% rm -rf userA
% rm -rf userB
```

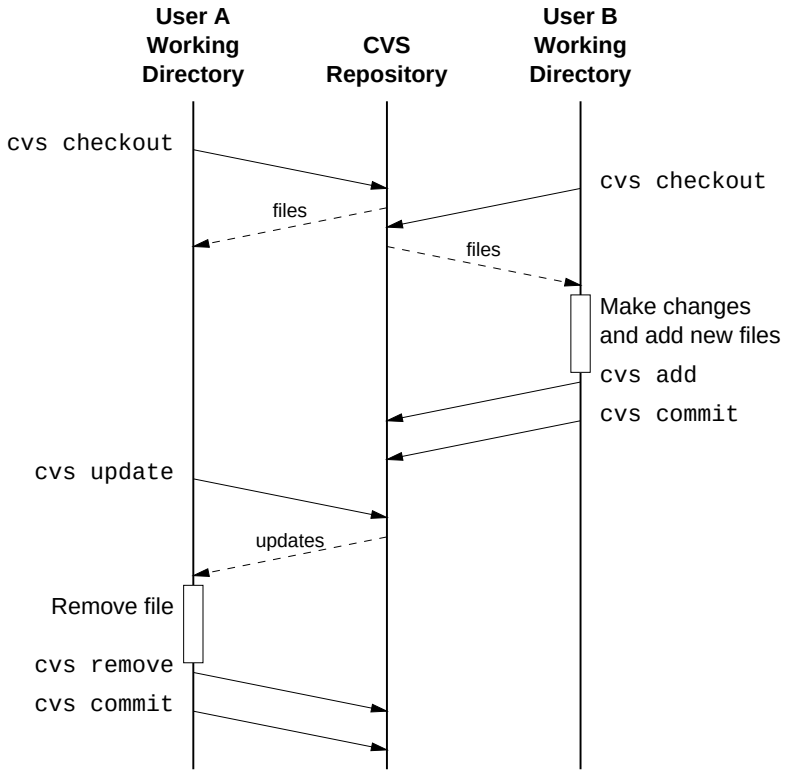

Figure 4: Timeline for two users with no conflicts

Resolving Conflicts

In the previous section we examined what happens when two users simultaneously modify files in the same project. Notice however, that in the previous example, the two users never modified the same line of a file at the same time. In this section we will examine how to handle these *conflicts*. Figure 5 shows the timeline of cvs commands involved in this section.

Let's begin with both User A and User B doing a standard checkout.

```
% pwd
tut2
% mkdir userA
% cd userA
% cvs checkout 2008s/students/cbatten/cvstest
% cd ..
% mkdir userB
% cd userB
% cvs checkout 2008s/students/cbatten/cvstest
% cd ..
```

Now let's assume that User A changes Fred's phone number as follows.

```
% pwd
tut2
% cd userA/2008s/students/cbatten/cvstest
% perl -i -pe 's/Fred.*/Fred : 617-555-3333/' phone-list.txt
% cvs commit
```

Now assume that User B changes Fred's phone number to a different number.

```
% pwd
tut2/userA/2008s/students/cbatten/cvstest
% cd ../../../../../userB/2008s/students/cbatten/cvstest
% perl -i -pe 's/Fred.*/Fred : 617-555-5555/' phone-list.txt
% cvs commit
cvs commit: Examining .
cvs commit: Up-to-date check failed for `phone-list.txt'
cvs [commit aborted]: correct above errors first!
```

Notice that User B has received an error when trying to commit his changes. The CVS repository has realized that User B's change conflicts with User A's original change. So User B must do a `cvs update` first.

```
% pwd
tut2/userB/2008s/students/cbatten/cvstest
% cvs update
cvs update: Updating .
RCS file: .../2008s/students/cbatten/cvstest/phone-list.txt,v
retrieving revision 1.3
```

```
retrieving revision 1.4
Merging differences between 1.3 and 1.4 into phone-list.txt
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in phone-list.txt
C phone-list.txt
```

The CVS messages during the update indicate that there is a conflict in `phone-list.txt`. A conflict simply means that there is a file in the current working directory which has lines which differ from what is in the repository. CVS tries to merge files so that only true conflicts are reported. In other words, if two people change different parts of the same file, then usually CVS will be able to merge the two versions without a conflict. In this case both User A and User B changed the exact same line, so CVS does not know which modification should take priority. CVS requires that users resolve conflicts by hand. After doing a `cvs update`, User B's copy of `phone-list.txt` will look like this.

```
% pwd
tut2/userB/2008s/students/cbatten/cvstest
% cat phone-list.txt
<<<<<<< phone-list.txt
Fred : 617-555-5555
=======
Fred : 617-555-3333
>>>>>>> 1.4
Jane : 617-555-0021
```

The content involved in the conflict is delimited by `<<<<<<<` and `>>>>>>>`. User B then needs to resolve the conflict by editing `phone-list.txt`. Let's assume that User B wants to override User A's earlier change. Edit `phone-list.txt` so that it looks like this.

```
% pwd
tut2/userB/2008s/students/cbatten/cvstest
% cat phone-list.txt
Fred : 617-555-5555
Jane : 617-555-0021
```

Now if User B does a `cvs commit` it will succeed and essentially User A's changes will have been overridden. User A can then do a `cvs update` to synchronize her working directory. A common rule of thumb is to always do an update (to catch conflicts) before doing a commit. Let's finish up by deleting the working directories.

```
% pwd
tut2/userB/2008s/students/cbatten/cvstest
% cd ../../../../..
% rm -rf userA
% rm -rf userB
```

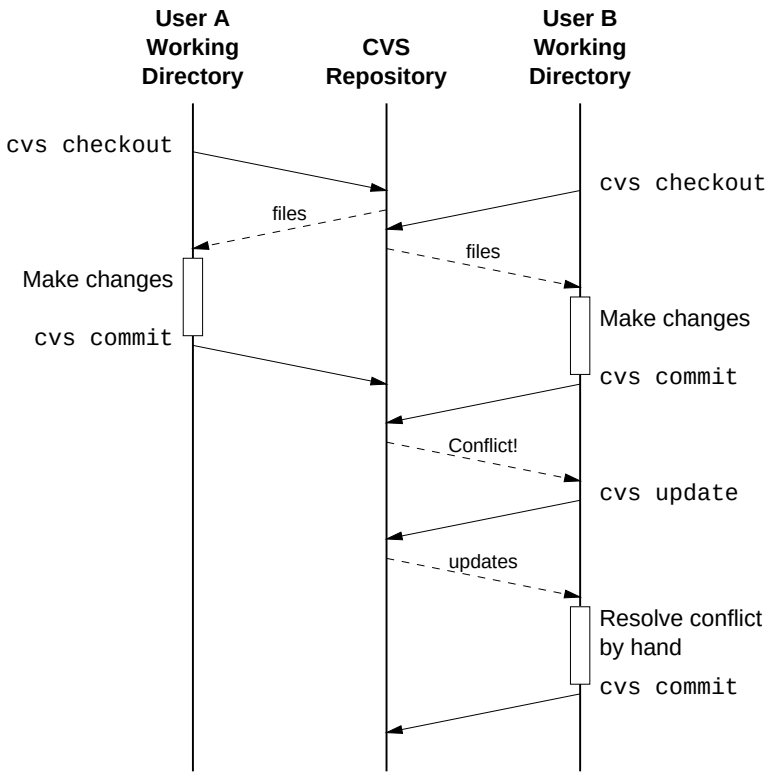

Figure 5: Timeline for two users with conflicts

Using Tags

In the previous sections we have learned how to use CVS to manage various versions of our source RTL. We have primarily focused on how to manipulate the most current version in the repository. Sometimes when we reach a milestone, we want to mark a version so that we can retrieve it at a later time even if we have already made (and committed) many other changes. We can do this with CVS tags. A tag is simply a symbolic name we give to a specific version of several files.

The following commands checkout the `cvstest` directory, add two new files, and then tags everything. It is important to always do a `cvs update` and a `cvs commit` before doing a `cvs tag` since the tag operation works on the most current version in the repository. If you have made changes in your local directory which are not yet committed then this could incorrectly tag the files. The `-c` option to the `cvs tag` command will cause CVS to abort if there are any locally modified files which are not committed yet.

```
% pwd
tut2
% cvs checkout 2008s/students/cbatten/cvstest
% cd 2008s/students/cbatten/cvstest
% echo ‘‘Fred : fred@mit.edu’’ > email-list.txt
% echo ‘‘Jane : jane@mit.edu’’ >> email-list.txt
% echo ‘‘Fred : 32-G736’’ > office-list.txt
```

```
% echo ‘‘Jane : 32-G738’’ >> office-list.txt
% cvs add email-list.txt
% cvs add office-list.txt
% cvs update
% cvs commit
% cvs tag -c example-tag
```

The above commands tag all three files (`phone-list.txt`, `email-list.txt`, and `office-list.txt`) with the symbolic tag `example-tag`. We can now retrieve this version of the files at any time by using this tag. To see how this work's let's first commit some additional changes.

```
% pwd
tut2/2008s/students/cbatten/cvstest
% echo ‘‘Sara : 617-555-0234’’ >> phone-list.txt
% echo ‘‘Sara : sara@mit.edu’’ >> email-list.txt
% echo ‘‘Sara : 32-G736’’ >> office-list.txt
% cvs update
% cvs commit
```

Now let's delete the working directory and try checking out two different versions of the files.

```
% pwd
tut2/2008s/students/cbatten/cvstest
% cd ../../../..
% rm -rf 2008s
% mkdir tagged-version
% cd tagged-version
% cvs checkout -r example-tag 2008s/students/cbatten/cvstest
% cd ..
% mkdir current-version
% cd current-version
% cvs checkout 2008s/students/cbatten/cvstest
% cd ..
% cat tagged-version/2008s/students/cbatten/cvstest/phone-list.txt
Fred : 617-555-3333
Jane : 617-555-0021
% cat current-version/2008s/students/cbatten/cvstest/phone-list.txt
Fred : 617-555-3333
Jane : 617-555-0021
Sara : 617-555-0234
```

The checked out version in the `tagged-version` directory corresponds to the `example-tag` symbolic tag, while the version in the `current-version` directory corresponds to the most recent version in CVS. Tagging is particularly useful when you reach a working milestone. You can tag your project and then at any time you can go back and retrieve the working version as of that milestone. Let's finish up by deleting the working directories.

```
% pwd
tut2
% cd ..
% rm -rf tut2
```

Using ViewCVS to Browse the Repository

The previous sections illustrated how to add, checkout, update, and commit files using CVS. It is often useful to browse the repository to see what changes other users have made. For example, assume one evening your RTL is working wonderfully. A group member makes some changes the next day and checks them in. Now the RTL starts failing some of your tests. You can browse the CVS repository to see what changes your partner made, and thus try and identify what is causing the failures. Although it is possible to browse the repository using cvs commands directly, it is much more convenient to use a graphical front-end to the repository. In 6.375, we will be using ViewCVS to browse the repository. The following command will launch ViewCVS and display the configuration dialog box (see Figure 6).

```
% start-viewcvs &
```

Click on the *Open Browser* button. After a few seconds a Mozilla browser should appear, and it should automatically point to `http://localhost:6375/viewcvs`. Browse the repository to the `cvstest` directory we have been working on in this tutorial. You should see the three files (`phone-list.txt`, `email-list.txt`, and `office-list.txt`). If you click on the drop-down *Show files using tag:* menu at the bottom of the page you can choose which tag to browse. If you click on one of the files you can see a complete version history of that file. The history lists the time, user, and log message for every version. Furthermore, you can actually view each version or compare the differences between versions.

If more than one user on a specific machine is using ViewCVS at the same time, then the configuration dialog box may hang saying "Starting server ...". In this case you simply need to start ViewCVS on a different (currently unused) port using the `-p` command line option. Try several ports around 6375 until the configuration dialog box indicates you were able to setup a localhost server.

```
% start-viewcvs -p 6376 &
```

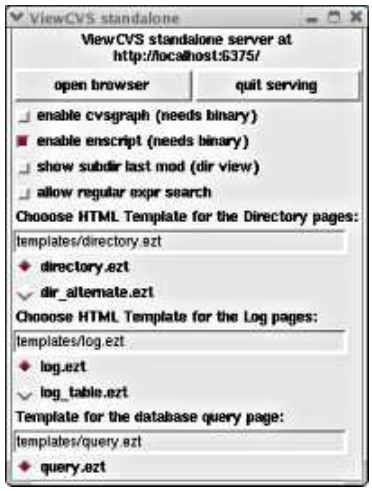

Figure 6: ViewCVS Configuration Dialog Box

Review

In this tutorial you have learned how to use CVS to manage your source RTL. You have learned how to add new files to the repository, checkout files from the repository, and commit local changes into the repository. You have also learned how to use ViewCVS to browse the CVS repository. The following table lists the cvs commands that you will use most frequently in this course.

`cvs checkout`	Checks out files from the repository into the working directory
`cvs add`	Adds new files to repository (must commit after adding)
`cvs commit`	Commits files in working directory into the repository
`cvs update`	Brings working directory in synch with respect to repository
`cvs remove`	Removes file from CVS (must use `rm` first)
`cvs status`	Shows status of files in working dir compared to current version in repo
`cvs diff`	Shows how files in working dir differ from current version in repo
`cvs tag`	Adds a symbolic tag for current version (always use with `-c`)