CVS in 20 (or so) Pages*

Gal Aviel (Email: galaviel@yahoo.com)

Revision : 1.4
25 Jan 2004

CVS is probably the most common version control system in the world. It's easy to use, stable, open source, and . . . free. This document provides easy step by step instructions and examples on how to introduce a legacy (under SCCS or no version control at all) project tree into CVS. This document also describes daily and advanced CVS usage, together with working examples, each and every step of the way. Note: This document is heavily based on "Version Management with CVS for CVS 1.11.2" by Per Cederqvist.

*Inspiration for document name from "Perl in 20 pages" written by Russell Quong, see http://www.best.com/~quong/perlin20/.

Contents

1 About This Document **5**
- 1.1 General . . . 5
- 1.2 Copyright . . . 5
- 1.3 License . . . 5
- 1.4 Bugs . . . 5

2 General CVS Concepts **6**
- 2.1 The Repository and the Working directory . . . 6
- 2.2 Revision Vs. Version . . . 6
- 2.3 Modules . . . 6
- 2.4 Recursion . . . 6
- 2.5 Multiple Developers . . . 7
- 2.6 RCS . . . 7
- 2.7 Tags . . . 7
- 2.8 Branches . . . 7

3 How To Introduce an Arbitrary Source Tree into CVS **8**
- 3.1 Deciding on your Directory Structure . . . 8
- 3.2 Doing some Cleanups . . . 8
- 3.3 Setting CVSROOT Environment Variable . . . 8
- 3.4 Creating a new Empty Repository . . . 8
- 3.5 Importing SCCS Files . . . 9
  - 3.5.1 SCCS to RCS . . . 9
  - 3.5.2 Copy Your RCS converted files into $CVSROOT . . . 10
- 3.6 Importing Plain Files . . . 10
- 3.7 Turning Watch Feature On - For Repository (one time operation) . . . 10
- 3.8 That's it ! . . . 11

4 Common CLI Usage **12**
4.1 Getting a Private Working Copy . . . 12
4.1.1 Get Latest . . . 12
4.1.2 Get According to Date/Time . . . 13
4.1.3 Get According to Tag . . . 13
4.2 Creating and Moving (Updating) Tags . . . 13
4.2.1 Creating a Tag . . . 14
4.2.2 Moving a Tag . . . 14
4.3 Make Sure Nobody is Editing the File . . . 14
4.4 Open File For Editing . . . 15
4.5 Abandon File Being Edited . . . 15
4.6 Commit Your Changes . . . 15
4.7 Modifying your Commit Comment after commit . . . 16
4.8 Remove Your Private Working Copy . . . 16
4.9 Getting The Status of your Files . . . 16
4.10 Browsing Log Messages . . . 17
4.11 Add a File or Directory to CVS . . . 18
4.11.1 Handling Binary Files . . . 19
4.12 Remove a File from CVS . . . 19
4.13 Comparing Two Tags/Dates . . . 21
4.13.1 The manual way . . . 21

5 Advanced CLI Usage **22**
5.1 Defining Modules . . . 22
5.2 Exporting Sources for Delivery (according to Tag/Date) . . . 22
5.3 Undeleting a File . . . 22
5.4 Append Revision History to a File . . . 22
5.5 Delete a Revision or Revision Range . . . 23
5.6 Editing CVS Administrative files . . . 24
5.7 Automatically Updating a Browse Directory (+update log) . . . 24
5.8 Ignoring Temp. Files . . . 25

6 CVS Weakness **25**
6.1 Symbolic Links 25
6.2 Effective Handling of Binary files 25

1 About This Document

1.1 General

There are many CVS documentation out there, so you're probably wondering (as I still am) weather the world needs yet another one. The reason that I wrote this document, is that I could not find any CVS documents that were detailed to the exact (moderate) degree that I was looking for. Existing docs were either too detailed (100+ pages) or too concise (the various CVS command references that are out there). I was looking for something in between. In addition, non of the documents was "cookbook" oriented, i.e. presented the user with a clear procedure to follow. Hopefully this document will achieve the these two goals.

1.2 Copyright

Written and Copyright (C) 2004 by Gal Aviel.

1.3 License

This document is published under GNU GPL version 2.

1.4 Bugs

Corrections ? Suggestions ? please contact me at galaviel@yahoo.com

Preliminary

2 General CVS Concepts

This section will briefly discuss several essential CVS concepts, which hold true, no matter what the CVS command you're using.

2.1 The Repository and the Working directory

The repository is the "vault" where all the source files are safely kept. Physically, the Repository is a directory of your choice in your file system. Never should you attempt to directly edit any files in the Repository. Usually, in order to work with CVS, the `$CVSROOT` env. variable has to be set, and has to point to the Repository.

The working directory however, is where the real editing of files occurs. The working directory can be any arbitrary directory in your file system, although many people prefer to use a known location, such as `~/cvs_workdir/`. Normally, a developer would change into the working directory, get (check-out) some source files from the repository, edit them, and then commit (check-in) his changes back into the Repository.

2.2 Revision Vs. Version

A **revision** refers to a single file (e.g. revision 1.12 of myfile.c) whereas a **version** is associated with a collection of files (e.g. a software product), each file having its own specific revision. The concept of version is closely associated with the tag concept (see tags in section (2.7) below).

2.3 Modules

A CVS module defines a mapping between a collection of CVS repository items (files, directories) and the current directory. For example, the `verilog` module could define a mapping between `$CVSROOT/lang/verilog` and `./lang/verilog`. Whenever the `cvs get verilog` command is issued, cvs creates the `./lang/verilog` directory, and copies all files under `$CVSROOT/lang/verilog` to the latter location.

2.4 Recursion

CVS is recursive by default, meaning that if a directory is specified as an argument, CVS will operate on everything below and including the specified directory.

Preliminary

2.5 Multiple Developers

Unlike the old SCCS version control system, which only permits one developer to edit a file at any given time, CVS allows for multiple developers to work concurrently on the same file. At commit time, CVS can automatically merge the changes developer A has made with the changes that developer B has performed. In case CVS cannot figure out a clean merge, it prompts the user.

It is generally recommended (especially when the number of developers is small) to adopt the SCCS approach and to dis-allow concurrent editing of any file by more than one developer at a time. In order to enforce such exclusive editing regime, the `cvs editors` and `cvs edit` commands can be used (see subsections (4.3) and (4.4)).

2.6 RCS

CVS uses RCS internally for file-level revision control.

2.7 Tags

A CVS Tag is a symbolic name for a certain revision of a file. It is an association between a symbolic name (a string, assigned by the developer) and a certain revision of a file (e.g. revision 1.12). Since it is possible to tag many files at once (recall that CVS commands are recursive by default), the tag feature can be used to create a "virtual" snapshot of the source files, without actually wasting any disk space, that is otherwise required in order to copy all the required source files to a different directory.

2.8 Branches

The CVS manual can answer this one better than me:

> **What branches are good for**
> Suppose that release 1.0 of tc has been made. You are continuing to develop tc, planning to create release 1.1 in a couple of months. After a while your customers start to complain about a fatal bug. You check out release 1.0 (see section Tags-Symbolic revisions) and find the bug (which turns out to have a trivial fix). However, the current revision of the sources are in a state of flux and are not expected to be stable for at least another month. There is no way to make a bug-fix release based on the newest sources.
> The thing to do in a situation like this is to create a branch on the revision trees for all the files that make up release 1.0 of tc. You can then make modifications to the branch without disturbing the main trunk. When the modifications are finished you can elect to either incorporate them on the main trunk, or leave them on the branch.

Preliminary

3 How To Introduce an Arbitrary Source Tree into CVS

This section will briefly explain the procedure that each developer should follow, in order to introduce directory trees into CVS. It describes introducing an existing project under SCCS version control into CVS, as well as introducing a project tree which is not under any version control at all.

3.1 Deciding on your Directory Structure

Prior to introducing the any source tree into CVS, decide on how your directory tree will look like, and make the changes required. After introduction to CVS, it becomes much harder (although not impossible) to move or rename directories.

3.2 Doing some Cleanups

Before even starting, cleanup your directories from unnecessary files.

3.3 Setting CVSROOT Environment Variable

In order to work with CVS, the $CVSROOT need to be set that it points to the repository. For example:

```
setenv CVSROOT ~/my_cvs_repository
```

3.4 Creating a new Empty Repository

Do this by issuing the command

```
cvs -d ~/my_cvs_repository init
```

. This will create the `$CVSROOT/CVSROOT` directory and a whole bunch of admin. files under it.

Preliminary

3.5 Importing SCCS Files

In case your existing source files are currently under SCCS control, you can convert them to RCS (the internal file-level revision control program that CVS uses), and then copy the resulting RCS files to the repository. Follow the instructions on section (3.5.1). The conversion process will conserve the entire SCCS history of the file, almost nothing is lost.

3.5.1 SCCS to RCS

In order to convert your existing SCCS files into the RCS format, use the `sccs2rcs.in` script by Ken Cox, which is available in the `contrib` subdirectory of the CVS distribution. From the script's header:

```
Sccs2rcs is a script to convert an existing SCCS history into an
RCS history without losing any of the information contained therein.
It will NOT delete or alter your ./SCCS history under any circumstances.
Run in a directory where ./SCCS exists and where you can create ./RCS.
Date, time, author, comments, branches, are all preserved.
After everything finishes, ./SCCS will be moved to ./old-SCCS.
```

The actual actions that you must perform are:

1. `cd /my_old_sccs_files/`
2. `source ~/downloads/cvs-1.11.1p1/contrib/sccs2rcs.in`
3. Optional: enter description for each file when prompted.
4. Accept default keyword translation when prompted.
5. Watch the output for any errors.

Preliminary

3.5.2 Copy Your RCS converted files into $CVSROOT

The final step is moving the newly created `RCS/` subdir to your CVS repository. Before copying the files, create the directories to host them. The old (pre entering the source code tree into CVS) directory tree structure should be reconstructed under $CVSROOT. For example, the lang project files in `~/cvs_workdir/lang` you should `mkdir $CVSROOT/cvs_workdir/lang`. Next, copy the files, by `cp ~/cvs_workdir/lang/RCS/* $CVSROOT/cvs_workdir/lang`. *Please note that this is one of the few times you are allowed to modify the CVS Repository directly !*

3.6 Importing Plain Files

In case your existing source files are not under any version control system, issue the `cvs import` command from the root of the directory tree you want to import. For example, if you want to add the `lang` subdirectory and everything under it to the Repository, issue the following command:

```
cd lang ; cvs import -m "my import log message" lang myvendortag myreleasetag
```

3.7 Turning Watch Feature On - For Repository (one time operation)

In order to work in exclusive editing mode (only one developer is allowed to edit a file at one time), the watch feature must to be turned on. When watch is turned on, as developers check out (get) sources, they will get Read Only copies of the files, reminding them to use '`cvs edit`' command (see (4.4)) before doing any editing. In order to activate the watch feature, first get the subtree you want to watch. As all CVS commands work recursively by default, the following command will turn watch for the entire `lang` subtree:

Preliminary

```
cd ~/cvs_workdir
cvs get lang
[files are RW]
cvs watch on lang
cvs release -d lang
[on the next 'cvs get', files will be checked out Read Only]
cvs get lang
[issue 'ls -la' to make sure files are RW, then release]
cvs release -d lang
```

3.8 That's it !

You're ready to begin working with the CVS. Proceed to Next Chapter.

Preliminary

4 Common CLI Usage

This section explains how the average developer should use CVS for the day to day operations, using the CLI (Command Line Interface). Generally speaking, the typical CVS work flow is as follows:

1. Getting a private copy of the source files. This private copy is based on tag (see section (4.2)) or just the latest version of all the files (default).
2. Edit the files. Changes made are local, other developers do not see them yet.
3. Commit the changes back to the Repository. At this time, other developers can see the change, and have the option of update their local copy. Note that CVS supports automatic commit notification by email and other means so that other developers get immediate notification.
4. Release the local copy of the sources.
5. Back to (1) above.

4.1 Getting a Private Working Copy

4.1.1 Get Latest

In order to get a copy of the latest source files, goto your private work directory, optionally create an sub-directory for the Activity which you are about to perform (e.g. bug-fix #211), and get the sources from CVS:

```
mkdir ~/cvs_workdir/bug-fix211
cd ~/cvs_workdir/bug-fix211
cvs get lang/verilog
cd ./lang/verilog/
[edit files]
```

Preliminary

4.1.2 Get According to Date/Time

It is possible to get all changes that were committed no latter than a specified date. This is accomplished using the `-D` option to `cvs get`:

```
[get all files under 'lang' module, as they were at Sep 24 2003 20:25]
cvs get -D "24 Sep 2003 20:25" lang
[get the project from 1 hour ago]
cvs get -D "1 hour ago" lang
```

CVS accepts a wide verity of date specification formats. See "common command options" in the CVS manual.

4.1.3 Get According to Tag

Sometimes It's useful to get specific version of a subproject, that were specifically selected (tagged) by the a developer as suitable for some purpose. For example, a developer might choose create a tag called FREEZE, which holds a collection of revisions of the source files, that the developer feels are ready for integration. The project integrator would then checkout only these carefully selected revisions, in order to create a freeze of the overall project and perhaps create a new build of the software product.

In order to checkout according to tag, issue the following command:

```
[get all files in 'lang' modules that are tagged with LANG_FREEZE tag]
cvs get -rLANG_FREEZE lang
```

4.2 Creating and Moving (Updating) Tags

For tagging, you must first checkout the module/files you want to tag. Please note that unlike modifying the sources locally, the tag operation effects the repository immediately, and everybody can see your new tag. Also note, that tagging (like any other CVS operation) is recursive, therefor when tagging a directory, everything beneath it is also tagged.

One surprising aspect of the tag subcommand, is that it tags the checked-in revisions, which may differ from your locally modified files. Because of this behavior, it is recommended to use the '`-c`' flag when tagging, thus verifying that the files you are about to tag are not locally modified.

4.2.1 Creating a Tag

Let's see the code:

```
[get the lang module]
cvs get lang
[tag lang directory with as "LANG_PRE_BUGFIX_231",'-c' flag makes sure none of the files
are locally modified]
cvs tag -c LANG_PRE_BUGFIX_231 lang
[can now safely start making changes, because any disaster can be reverted back ...]
```

4.2.2 Moving a Tag

It is generally useful to keep several constant tags for a module, such as STABLE and LATEST, and update those tags during the development process. These constant tags can be used by the build environment in order to build (compile) the latest version of the product. The following code shows how to update the LATEST tag:

```
[update the 'LANG_LATEST' tag of 'lang' module, so it points to the currently checked out
revisions of the files under 'lang' subdir]
cvs tag -F LANG_LATEST lang
```

4.3 Make Sure Nobody is Editing the File

In order to conform to the exclusive editing regime, the developer must verify, prior to editing a file, that no other developer is already editing it:

```
> cd cvs_workdir
> cvs get lang
[find out who's editing what under 'lang' subtree]
> cvs editors
lang/perl/myperl.pl gala Sun Nov 24 12:52:54 2003 GMT lxr220 /home/gala/cvs_workdir/lang/perl/myperl.pl
```

The output of 'cvs editors' reveals that user gala is editing the repository file lang/perl/myperl.pl from lxr220 workstation, and that gala's local/private copy is located at /home/gala/cvs_workdir/lang/perl/myperl.pl (the date field I don't know what it means ...).

4.4 Open File For Editing

In order to start modifying a file, the developer must issue the following commands:

```
[after getting the lang subtree and verifying that nobody is editing]
> cd ./cvs_workdir/lang/perl/
> cvs edit myperl.pl
[note that myperl.pl changed to RW]
[Edit the file ...]
```

4.5 Abandon File Being Edited

Things got messy ? want to start fresh again ? The following command abandon all the work that was done on the file(s), and reverts them back to the repository revision(s) on which they are based:

```
[myperl.pl is Read-Only now]
cvs edit myperl.pl
[myperl.pl is now RW]
[edit edit edit ...  make a mess out of this file ...]
[want to start fresh !]
cvs unedit myperl.pl
[file is again Read Only, and is reverted back to the Repository
Revision on which it was based]
```

4.6 Commit Your Changes

Once your changes compile, you want to commit them back to the repository, and create a delta:

```
cvs commit -m “I made this and that change” myperl.pl
```

By the way, if you don’t use the `-m` switch, then CVS will invoke your `$EDITOR` and prompt you to enter a log message for the delta. Some people find that writing the comment in NEdit is nicer than writing it on the command line, especially if the log message is lengthy.

4.7 Modifying your Commit Comment after commit

TBD but it can be done.

4.8 Remove Your Private Working Copy

Once you have finished with the activity at hand, you’d want to erase your private copy in an orderly fashion. Using the below `release` CVS sub-command will also make sure that you didn’t forget to commit something.

```
[cd to the top-most directory where your CVS subtree begins]
cd /home/gala/cvs_workdir/
cvs release -d lang
```

4.9 Getting The Status of your Files

The CVS `status` sub-command, informs you of the status of your checked-out file, relative to the Repository. Based on the operations you have performed on a checked out file, and what operations others have performed on that file in the repository, the status of a file can have the following values:

Up-to-date The file is identical with the latest revision in the repository.

Locally Modified You have edited the file, and have not yet committed your changes

Locally Added You have added the file with CVS `add` sub-command, and have not yet committed your changes (see adding files in sub-sec (4.11)).

Preliminary

Locally Removed You have removed the file with CVS `remove` sub-command, and have not yet committed your changes (see removing files in sub-sec (4.12)).

Needs Checkout Someone has committed a newer version to the repository; use CVS `update` sub-command to update.

Needs Merge Someone has committed a newer revision to the repository, and you have also made local modifications to the file. Note: this status should not normally occur, when working in an exclusive editing mode, where only one developer is allowed to edit a file at any given time.

File had conflicts on merge In case the status was "Needs Merge" and you tried CVS `update` to merge, it could be that the merge failed, and you will get the current status. You need to resolve the merge manually.

Unknown CVS does not know anything about this file. This could happen, if you created a new file and did not yet run the CVS `add` sub-command.

Below is the output from the `cvs update` command. It reveals that two verilog files differ from the repository revision on which they are based, while the 3rd file is unknown to CVS.

```
cvs -qn update
M myfile1.v
M myfile2.v
?  verilog.log
```

4.10 Browsing Log Messages

Use the `cvs log` subcommand to view the commit logs of your files, and other information. By default, `cvs log` prints everything when invoked without any flags. All other options/switches limit/filters the amount of information displayed by `cvs log`. Useful options to `cvs log` include showing check-in's done by a specific user, or check-in's that occurred in a range of dates or revisions or tags:

Preliminary

```
[works on the current directory by default, print _all_ the
available info on each file]
> cvs log (or cvs log filename for a specific file)
[print only log messages for revisions 1.12 to 1.19]
> cvs log -r1.12:1.19 filename
[print only log messages for TAG1 to TAG2]
> cvs log -rTAG1:TAG2 filename
[print only log messages of commits by user 'gala']
> cvs log -wgala filename
```

4.11 Add a File or Directory to CVS

Adding a file or directory is easy:

1. First, checkout a copy of the sources.
2. create the file or directory.
3. Use CVS `add` sub-command on the file:

```
> cvs add testbench.v
cvs add:  scheduling file 'testbench.v' for addition
cvs add:  use 'cvs commit' to add this file permanently
```

4. Use CVS `commit` sub-command to actually check in the file into the repository.

```
>cvs commit -m "Initial version of file" testbench.v
```

5. Tell CVS to watch the new file (required for exclusive editing). You can probably do this for just the single file; but here I renew the watch on all the files in lang module:

eLibrary

```
[cd up to where lang dir is a subdir]
>cd ..
>cvs watch on lang
>cd lang
[continue working]
```

4.11.1 Handling Binary Files

When adding a binary file, be sure to use the '-kb' switch, i.e.

```
cvs add -kb filename
```

If you already added the file but forgot to use the '-kb' switch, run the following command:

```
cvs admin -kb filename
cvs update -A filename
```

4.12 Remove a File from CVS

Removing a file which is under CVS control is easy, and has the nice side effect of being able to retrieve old revisions. Follow these steps:

1. First, make sure that you don't have any uncommitted modifications you've made to the file, which will be lost once you remove it. Do this using the e.g. `update` or `status` CVS sub-command:

Preliminary

```
>cvs status testbench.v
===================================================================
File:  testbench.v Status:  Up-to-date
Working revision:  1.1 Wed Oct 23 18:26:22 2003
Repository revision:  1.1
/cvs_repository/lang/verilog/testbench.v,v
Sticky Tag:  (none)
Sticky Date:  (none)
Sticky Options:  (none)
```

2. Remove the file (using `rm` shell command) from your working copy of the CVS directory:

```
>\rm testbench.v
rm:  testbench.v:  override protection 444 (yes/no)?  yes
```

3. Tell CVS that you want to remove the file using the CVS `remove` sub-command:

```
>cvs remove testbench.v
cvs remove:  scheduling ‘testbench.v’ for removal
cvs remove:  use ’cvs commit’ to remove this file permanently
```

4. Use CVS `commit` sub-command to actually perform the removal of the file from the repository:

```
>cvs commit -m "removed testbench.v file, no longer needed”
cvs commit:  Examining .
Removing testbench.v;
/cvs_repository/lang/testbench.v,v <-- testbench.v
new revision:  delete; previous revision:  1.1
done
```

4.13 Comparing Two Tags/Dates

Sometimes it is useful to compare two arbitrary points in the development cycle of a CVS module. Either one of these points can be a CVS tag, or a date (down-to the minute).

4.13.1 The manual way

Use the `cvs rdiff`[1] to know what changed between two releases/tags:

```
>cvs rdiff -s -r LANG_PRE_BUGFIX231 -r LANG_POST_BUGFIX231 lang
cvs rdiff:  Diffing lang
File lang/verilog/file1.v changed from revision 1.1 to 1.2
File lang/verilog/file2 changed from revision 1.1 to 1.2
File lang/verilog/testbench.v changed from revision 1.54 to 1.56
```

The above command compares the revisions of all the files in the *lang* module, between tag LANG_PRE_BUGFIX231 and LANG_POST_BUGFIX231 tag. The '-s' options denotes that the output format will be 'summary', rather than actual diff format. For each of the files which have changed, you can further view the all the log messages between these two revisions, by using `cvs log`:

```
>cvs log -rLANG_PRE_BUGFIX231:LANG_POST_BUGFIX231 testbench.v
[long header removed ...]
total revisions:  60; selected revisions:  3
description:
----------------------------
revision 1.56 date:  2003/01/20 12:54:59; author:  gala; state:  Exp; lines:  +70 -241 added
signalscan dumping.
----------------------------
revision 1.55 date:  2003/01/20 11:38:49; author:  gala; state:  Exp; lines:  +44 -10 added case
to handle DPR.
----------------------------
revision 1.54 date:  2003/01/08 14:30:18; author:  gala; state:  Exp; lines:  +4 -4 comment
change.
=============================================================================
```

[1] The 'r' in `cvs rdiff` stands for 'release diff', as oppose to the regular `cvs diff` which diff's between two revisions of the same file.

5 Advanced CLI Usage

This section will describe the less-common and more destructive CVS commands, so please be careful !

5.1 Defining Modules

This is covered pretty well in the CVS Manual (see “The Cederqvist” at cvs homepage `www.cvshome.org`).

5.2 Exporting Sources for Delivery (according to Tag/Date)

The CVS `export` sub-command is a variant of the `checkout` sub-command: use it to create a copy of the sources, but without the CVS administrative files, that are created under ./CVS in your checkout directory. This is good for freeze of release to 3rd party/off site, since it contains only the sources, without any unwanted files. The `export` sub-command requires a tag or a date, so that you can reproduce the release at any time.

```
cvs export -r POST_BUGFIX231 lang
```

It is also possible to export by date, which will export the most recent revision of each file no later that the date specified:

```
cvs export -D 261103 lang
```

5.3 Undeleting a File

You can take back any shell `rm filename` followed by `cvs remove filename`, by just typing `cvs add filename`. Also, you can take back any shell `rm filename` by `cvs update filename`.

5.4 Append Revision History to a File

CVS has a number of special keyword such as `$Log` or `$Revision: 1.4 $`, that when appearing in a file, get expanded to their actual values at commit time. This allows, for example, for the complete revision history to appear inside a file, as shown below. The complete list of CVS keywords appears in the

Preliminary

```
/*
$Log:  cvs_in_20_pages.lyx,v
$Revision 1.6 2003/01/28 12:49:25 gal
indented revision history adding code
Revision 1.5 2003/01/27 17:56:29 gal
bugfix to revision removal
Revision 1.4 2003/01/22 13:46:46 gal
(no log message)
...
*/
```

In order to enable this feature, add the following text to your file (preferably at the beginning of the file):

```
/*
$Log:␣$
*/
```

5.5 Delete a Revision or Revision Range

Sometimes, you would like to discard some deltas you've made. You can delete just one revision, or a revision range. For deleting a specific Revision (usually the latest):

```
cvs admin -o1.49 testbench.v
```

After deletion, you must manually update the file in *each and every* directory where the file is checked out by using

```
cvs update testbench.v
```

Please note that currently, directories which mirror the latest project (see (5.7)), will not be updated automatically - this is still not supported. You have to cd manually and perform the latter command yourself. However, since deleting a revision is (should not be) a common operation, then hopefully this is not too much of a drawback.

5.6 Editing CVS Administrative files

The editing of CVS administrative files (which should be done only by CVS ADMIN) is done as follows (`loginfo` file is just an example):

```
cvs get CVSROOT
cd CVSROOT
cvs edit loginfo
[edit away at file loginfo]
cvs commit loginfo
[popup your favorite $EDITOR for log message]
```

5.7 Automatically Updating a Browse Directory (+update log)

Sometimes it is useful to have the latest project files exported to some directory, and updated automatically (every time a commit is made). This task can be done in two steps. First, alias the cvs command such that the new command will define a new so-called 'cvs user variable'. For example, in your .cshrc file, alias the cvs command as follows:

```
# define a cvs user variable, which can be
# referred to in cvs administrative files.
alias cvs "cvs -s CVS_BROWSE_DIR=/public/my_browse_dir"
```

Next, edit the `loginfo` CVS administrative file, to include the following line (all the text must appear in one line):

```
^lang (echo "******************* "; date; cat; (sleep 2; cd ${=CVS_BROWSE_DIR}/lang;
cvs -q update -d) &) >> $CVSROOT/CVSROOT/updatelog 2>&1
```

A welcome side effect of the above `loginfo` line you've just added, is that from now on, the low of all the automatic updates in the browse directory, will be appended to file `$CVSROOT/CVSROOT/updatelog`.

5.8 Ignoring Temp. Files

Sometimes, when updating/importing/releasing a CVS working directory , CVS complains about various temporary files (compiler *.o object files and such). This can be a bit annoying, and it maybe desirable to instruct CVS to automatically ignore such files. This can be accomplished by placing the following line in `cvsignore` administrative file:

```
verilog.log
```

The syntax of the `cvsignore` file is rather simple. Each line in the file can contains a space separated list of filenames or file patterns[2] (with wild cards) that CVS should ignore.

6 CVS Weakness

There are several areas where CVS has some room for improvement . . . they will be listed here.

6.1 Symbolic Links

There is currently no native support within CVS for handling symbolic links. This limitation of CVS can be worked around, as CVS allows for hooks, in the form of user specified script which are run at check-in/out time. Therefor, all that is needed, is a check-in script that saves all the symbolic links into a plan text file can be easily written, and a check-out script which rebuilds the symbolic links according to the content of that text file.

6.2 Effective Handling of Binary files

Currently, CVS does not handle binary files in an effective manner, storage-wise. If a binary file (which, by the way, has to be pre-declared as such) is changed, CVS replaces the entire file. CVS does not employ a no smart binary diff; if a single byte of the file is changed from say revision 1.11 to revision 1.12, then the cost in terms of storage of revision 1.12 is the entire size of the file.

[2] file patterns in sh(1) syntax.

Preliminary

References

[1] “The Cederqvist”: The Official CVS Manual http://www.cvshome.org/docs/manual.

Preliminary