

# Practical CVS Tutorial

Stephen Soltesz

August 9, 2004

The purpose of this tutorial is to provide practical examples of how to use CVS during a typical source development cycle. The complexity of the CVS interface and underlying structure tends to prevent casual use of the system and prevents us from taking full advantage of the multi-user, parallel development features. With the examples provided in this tutorial, it will be possible to use CVS as just another tool, simplifying back porting bug fixes, maintenance of released packages, and safely going about large scale code changes (such as a platform port, architectural changes, or adding experimental code). Those using CVS after this tutorial will be able to enjoy some of its more mysterious benefits with as little trouble as possible.

Being a tutorial, or 'how to', only necessary procedures are provided. Those wishing to understand the nuance, history, and detail of CVS are invited to read the manual.

## 1 The Environment

In the beginning there is nothing. No package, no directories, no source, nothing.

First, CVS needs a directory in which to create a repository. The repository is a special directory managed by CVS that will hold all files you want CVS to manage, as well as all changes made to these files. The directory may be reached through your local filesystem or remotely via `ssh`. In both cases, the environment variable `CVSROOT` must be set to specify the repository location.

If the repository is accessible through the filesystem, `CVSROOT` maybe set to the absolute path to the CVS directory. For instance, a workstation in the UT COMPUTER SCIENCE lab using the LOCI CVS repository could set `CVSROOT` as:

```
export CVSROOT=/cvs/homes/
```

However, for remote access from laptops or non-NFS mounted workstations, a different variation makes this possible via SSH.

```
export CVSROOT=:ext:username@enterprise.cs.utk.edu:/cvs/homes
export CVS_RSH=ssh
```

Be sure to replace *username* with your username. To experiment on your own and become more comfortable with the CVS commands, you may set `CVSROOT` to a directory in your `$HOME` area without any danger of breaking a shared repository. Once you have chosen and set `CVSROOT`, run `cvs init` to initialize the repository. This needs to be done only once, but running the command multiple times will not hurt anything.

## 2 Adding a package to your CVS repository

If you have write permission to your `CVSROOT`, you may add a module to the CVS repository using `cvs import`. This operation is illustrated figure 1. (A CVS module is another word for a source package name.)

For instance, "ibp", "lors", "libexnode" are all examples of modules currently in the LOCI CVS repository. "Module name" or "package name" may be used interchangeably.)

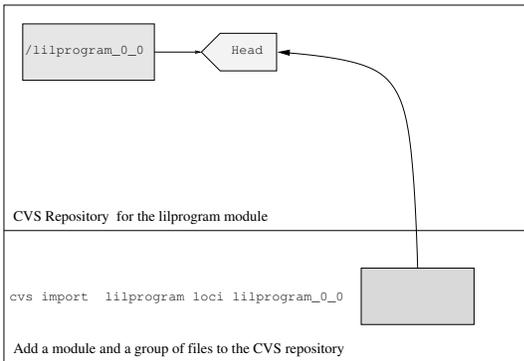


Figure 1: Import (legend is available page 9)

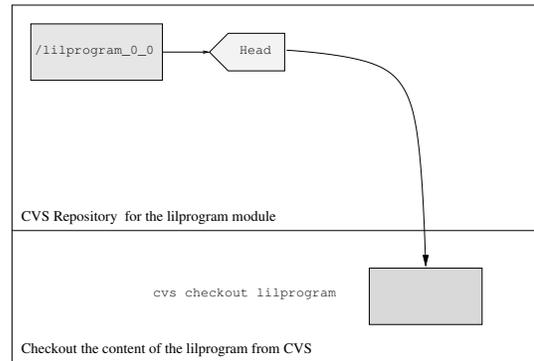


Figure 2: Checkout

```
UNIX> cvs import <package name> <vendor tag> <symbolic tagname>
```

The vendor tag is not important for now. But you must have something in its place.

CVS does not allow tag names to contain any spaces, dashes, or periods. This is unfortunate, because we are used to naming packages with dashes and periods, `lors-0.80`, for example. However, this name would become `lors_0.80`, if it were to be used as a tag name.

From the PWD of the package you wish to import into CVS (not below the directory; in the directory), run the `import` command.

```
UNIX> cat source.c
#include <stdio.h>
int main()
{
    printf("Hello broccoli.\n");
}
UNIX> cvs import lilprogram loci lilprogram_0_0
N lilprogram/source.c

No conflicts created by this import
```

In this example, I have a single C file named `source.c`. CVS prepends the package name to each file or directory imported. The 'N' signifies that this file is new. And, assuming you've not chosen a package name that is already in the CVS repository, the command will exit without an error.

### 3 Retrieving a module from CVS

To begin working from CVS you must retrieve the code you just imported by running `cvs checkout`. This operation is illustrated figure 2. Move to a directory where you wish to place the working directory of your CVS package.

```
UNIX> cvs checkout lilprogram
cvs checkout: Updating lilprogram
U lilprogram/source.c
```

In this example, CVS creates a directory named `lilprogram`, and populates it with `source.c`. The 'U' means CVS has *updated* this file. You may now change to the new directory and begin editing and saving changes.

## 4 Updating your working directory and Committing changes

Once you modify and save your source files, you will wish to commit them back to the CVS repository eventually. Before doing this, it is important to run `cvs update`. This operation is illustrated figure 3.

When you are working with more than one developer, they may have committed changes since the last time you checked out or updated your working directory. Update will first look in the CVS repository and compare your files to those in CVS. If there have been additions committed by others, then CVS brings the files in your working directory up to date with the content of the repository. If you try to commit your changes before updating, CVS will warn you if your code is not up to date with the code in the repository. But, updating first saves you a step and allows you to verify that your code still works with the updates of others.

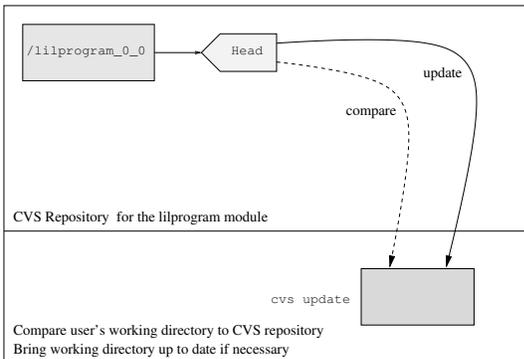


Figure 3: Compare and Update

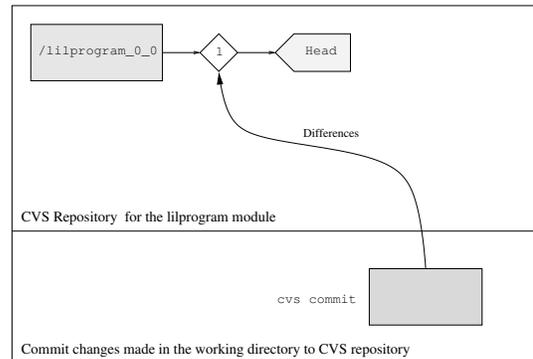


Figure 4: Commit

In my `lilprogram`, I realize that broccoli is not a friendly vegetable, and I would rather say "Go away" than "Hello". I make this change, and run `cvs update`.

```
UNIX> cvs update
cvs update: Updating .
M source.c
```

The 'M' tells me that `source.c` was modified by me and no other changes were made by anyone else. It is now safe to commit the changes. This operation is illustrated figure 4.

```
UNIX> cvs commit source.c
Checking in source.c;
/Users/soltesz/cvsroot/example/source.c,v <-- source.c
new revision: 1.2; previous revision: 1.1
done
```

This commit introduces the first revision in the history of `source.c` in the main branch.

## 5 Branching and Tagging

So far, we have assumed that there is only one branch in CVS, and every `checkout`, `update` or `commit` operates on the most current revision in that branch (also known as the "head" of the branch). One of the

strengths of CVS is its ability to manage independent, parallel development branches within a single module (or package).

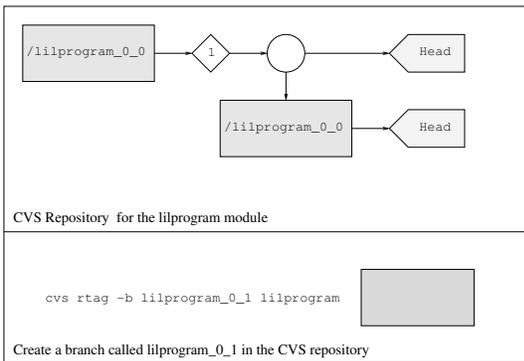


Figure 5: Create a new development branch.

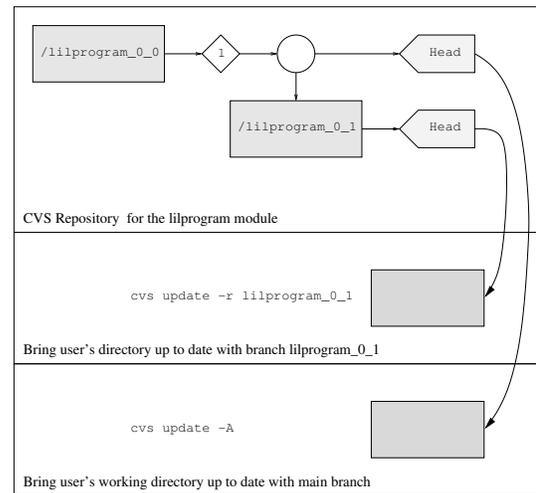


Figure 6: Update user's directory in function of the chosen development branch.

Examples when branching the versions of a package may be beneficial are at the time of a package release, during code porting, or when experimental features are added that may be unstable or otherwise unsuitable for the main branch. For releases, development can continue on the main branch, but if several weeks later bugs are discovered in the release, it may be difficult to recreate a new release and add the bug fixes in the CVS repository. As for porting and experimental features, once the new code is stable, it can be brought back into the main branch.

The important point to recognize is that release maintenance and experimental development can be done inside and with the assistance of CVS, rather than outside of CVS.

To have an understanding of what happens when a branch is created in CVS, imagine that a new directory is created in the repository and a copy is made of the main branch into this new directory. The resulting two directories are entirely independent of one another. Changes committed to one will not show up in the other unless you ask CVS to do this.

This procedure is identical to what you would end up doing if you were adding a fix to a bug in a release available only in the distributed `tar.gz`. You would unpack the archive and add your fixes, entirely independent of the CVS repository. When you do this, you must now return to the CVS repository and make the fixes or changes a second time to commit them. The difference is, of course, that CVS could manage the separate directories and releases for you, allowing you to have a single working directory for all of these tasks if you wish.

`cvs rtag` expects the module name and the name of the branch you are creating. Because the branch name is a tag, the same rules for apply to branches as apply to tag names.

If I wished to create a branch in the `lilprogram` to allow me to work on translating the message I would do this (illustrated figure 5):

```
UNIX> cvs rtag -b lilprogram_0_1 lilprogram
cvs rtag: Tagging lilprogram
```

After the branch command, there are two branch heads in the repository. At this moment, they contain the same information. However, each branch is now independent of the other (See illustration figure 6).

Once a branch has been created I can make my working directory mirror either branch using `cvs update`. In the first example, the new branch is brought into the user's working directory. This step is necessary

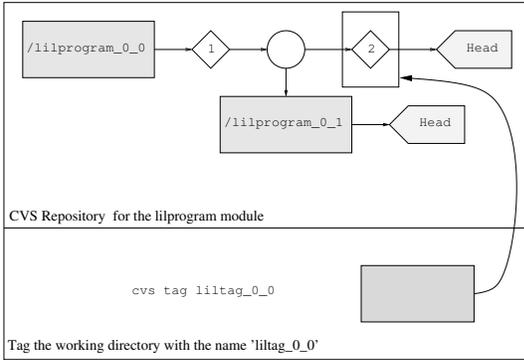


Figure 7: Tag the working directory.

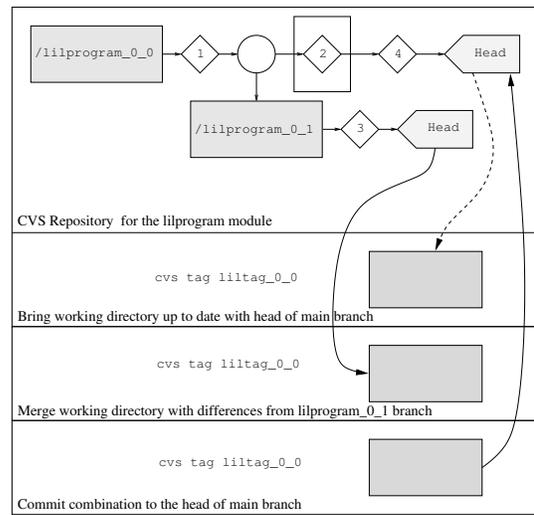


Figure 8: Update, merge and commit.

before you can commit changes or additions to this branch. Now you may update and commit and CVS will know that you are working on the lilprogram\_0\_1 branch.

```

UNIX> cvs update -r lilprogram_0_1
cvs update: Updating .
UNIX> cvs status source.c
cvs status source.c
=====
File: source.c          Status: Up-to-date

Working revision:      1.2      Wed Jun  4 00:00:21 2003
Repository revision:  1.2      /Users/soltesz/cvsroot/lilprogram/source.c,v
Sticky Tag:           lilprogram_0_1 (branch: 1.2.2)
Sticky Date:          (none)
Sticky Options:       (none)

```

The 'Sticky Tag' is lilprogram\_0\_1, showing that it is in fact from the lilprogram\_0\_1 branch. To revert to the main branch:

```

UNIX> cvs update -A
cvs update: Updating .
UNIX> cvs status source.c
=====
File: source.c          Status: Up-to-date

Working revision:      1.2      Wed Jun  4 00:05:16 2003
Repository revision:  1.2      /Users/soltesz/cvsroot/lilprogram/source.c,v
Sticky Tag:           (none)
Sticky Date:          (none)
Sticky Options:       (none)

```

The 'Sticky Tag' is now 'none', which is appropriate for the head of the main branch. Using cvs update -A, files in your working directory will be updated or removed when needed. As long

as you have committed any changes you've made to the other branch, nothing will be lost.

A user may also assign symbolic Tags to any revision in the CVS repository. Tags are static pointers, while branches are functional development trees. Tags can be turned into branches.

My second committed change will be to replace broccoli with celery.

```
UNIX> cat source.c
#include <stdio.h>
int main()
{
    printf("Go away celery.\n");
}
UNIX> cvs commit
\$ cvs ci source.c
Checking in source.c;
/Users/soltesz/cvsroot/lilprogram/source.c,v <-- source.c
new revision: 1.3; previous revision: 1.2
done
UNIX> cvs tag liltag\0\0
cvs tag: Tagging .
T source.c
```

The head of the main branch is now different from the head of the `lilprogram_0_1` branch. I can verify this by updating my working directory to the `lilprogram_0_1` branch and checking `source.c` (see also figure 7).

```
UNIX> cvs update -r lilprogram_0_1
cvs update: Updating .
U source.c
UNIX> cat source.c
#include <stdio.h>
int main()
{
    printf("Go away broccoli.\n");
}
```

## 6 Merging Branches

In the `lilprogram_0_1` branch, I will translate my message to German, and commit this change.

```
UNIX> cat source.c
#include <stdio.h>
int main()
{
    printf("Gehen brokkoli weg.\n");
}
UNIX> cvs commit source.c
Checking in source.c;
/Users/soltesz/cvsroot/lilprogram/source.c,v <-- source.c
new revision: 1.2.2.1; previous revision: 1.2
done
```

While there may be more than one branch, it may ultimately be desirable to bring these branches back together. For instance, if the branch held the code needed to run on windows, this branch could be merged

with the main branch, and any conflicts resolved.

To do this, make sure your working directory is the head of the main branch (or which ever branch you're merging into). Using `cvs update -j <branchname>`, where `branchname` is the name used during the `'rtag'` command (see figure 8). Where there are conflicts CVS will print a warning, and surround the differences in the file with identifying marks.

If I wish to merge the branches of the `lilprogram`, this is the message and content of `source.c`:

```
UNIX> cvs update -A
cvs update: Updating .
U source.c
UNIX> cvs update -j lilprogram_0_1
cvs update: Updating .
RCS file: /Users/soltesz/cvsroot/lilprogram/source.c,v
retrieving revision 1.2
retrieving revision 1.2.2.1
Merging differences between 1.2 and 1.2.2.1 into source.c
rcsmerge: warning: conflicts during merge
UNIX> cat source.c
#include <stdio.h>
int main()
{
<<<<<<< source.c
    printf("Go away celery.\n");
=====
    printf("Gehen brokkoli weg.\n");
>>>>>>> 1.2.2.1
}
```

It is the user's responsibility to resolve these conflicts before committing the file back to the main branch. In this example, I might make the message printed conditional on a command line argument or simply combine the messages into a single print statement.

## 7 Extra Good Stuff

### 7.1 add

After an import, or just in the course of development, there may be new files introduced to the project. To add a single file to the module in the CVS repository you can use `cvs add`, and next `cvs commit` to add the file permanently.

```
UNIX> cvs add header.h
cvs add: scheduling file 'header.h' for addition
cvs add: use 'cvs commit' to add this file permanently

UNIX> cvs commit header.h
RCS file: /Users/soltesz/cvsroot/lilprogram/header.h,v
done
Checking in header.h;
/Users/soltesz/cvsroot/lilprogram/header.h,v <-- header.h
initial revision: 1.1
done
```

## 7.2 remove

The reverse case is when a file has outlived its usefulness and is no longer relevant or needed by the other source files. In these cases we can remove the file from the CVS repository with `cvs remove`. First remove the file, then tell CVS it has been removed.

```
UNIX> rm header.h
UNIX> cvs remove header.h
cvs remove: scheduling 'header.h' for removal
cvs remove: use 'cvs commit' to remove this file permanently
```

To commit the change, run `cvs commit`.

## 7.3 diff

Sometimes after running `cvs update` and before you commit your changes, a file shows up as modified, but you don't remember what changes were made. In this case, you may examine the differences between your copy and the version in CVS using the 'diff' command. For instance, if I added a print statement to `source.c` and forgot, I could run `cvs diff source.c` and see what was added.

```
UNIX> cvs diff source.c
Index: source.c
=====
RCS file: /Users/soltesz/cvsroot/lilprogram/source.c,v
retrieving revision 1.3
diff -r1.3 source.c
10a11
>
27a29,30
>
>      fprintf(stderr, "Hello World\n", s);
```

## 7.4 export

The code is ready for distribution. All bugs are fixed and the build process is perfect.

If we just tar up the CVS working directory, we will have all the extra files from your testing and the CVS directories which don't really belong in a polished distribution. To have a clean export of the CVS repository without the extra cruft, `cvs export` does the job.

`cvs export` is like `checkout`, except it does not create a CVS working directory, only the source files.

From our example earlier, if I wished to create a `lilprogram` distribution of the german translation, I could do the following:

```
UNIX> cvs export -r lilprogram_0_1 lilprogram
cvs export: Updating lilprogram
U lilprogram/source.c
```

This version contains only the German translation and none of the extra CVS directories are included. `annotate`

You're looking at the code, and discover a bug. A real jaw-dropper. "Who would write such a piece of code? and commit it to the repository? It couldn't have been me!"

Well, with `cvs annotate` you can answer this question easily. The output from `annotate` includes the username who last committed each line of a source file.

```
UNIX> cvs annotate source.c
Annotations for source.c
*****
1.1      (soltesz 04-Jun-03): #include <stdio.h>
1.1      (soltesz 04-Jun-03): int main()
1.1      (soltesz 04-Jun-03): {
1.3      (soltesz 04-Jun-03):     printf("Go away celery.\n");
1.1      (soltesz 04-Jun-03): }
```

This is a double edged sword of course. So be nice, lest it is you who commit the next bug :-)

## 8 Conclusion

If there are few things to remember here they are. Do not forget to *update* your working directory before starting to work on a project, and never forget to *update* again before *committing* your changes. Now you feel comfortable with CVS ? Ok, here they are few shortcuts. **checkout** can be abbreviated by **co**, **update** by **upd**, and **commit** by **ci**.

### Useful link(s)

- A quick reference card about CVS is available here: <http://tnerual.eriogerg.free.fr/cvs.html>

