Institut für Theoretische Physik und Astrophysik
Julius-Maximilians-Universität Würzburg

Report

How to use Subversion to develop Enzo

Andreas Maier

September 20, 2008

Advisor:
Prof. Dr. J. Niemeyer

Contents

1. Introduction 1

2. Usage 2

A. How to create a new repository 7

B. Tips & tricks 8

C. Using a better `diff` 9

1. Introduction

Subversion (SVN) is a free/open-source version control system. That is, Subversion manages files and directories over time. A tree of files is placed into a central repository. The repository is much like an ordinary file server, except that it remembers every change ever made to your files and directories. This allows you to recover older versions of your data, or examine the history of how your data changed. In

this regard, many people think of a version control system as a sort of "a time machine". Subversion can access its repository across networks, which allows it to be used by people on different computers.

So basically Subversion does the same thing CVS does (Concurrent Versioning System) but has major enhancements compared to CVS. The most important are

- Directory versioning

  CVS only tracks the history of individual files, but Subversion implements a a "virtual" versioned filesystem that tracks changes to whole directory trees over time. Files and directories are versioned.

- True version history

  Since CVS is limited to file versioning, operations such as copies and renames - which might happen to files, but which are really changes to the contents of some containing directory - aren't supported in CVS. Additionally, in CVS you cannot replace a versioned file with some new thing of the same name without the new item inheriting the history of the old - perhaps completely unrelated - file. With Subversion, you can add, delete, copy, and rename both files and directories. And every newly added file begins with a fresh, clean history all its own.

For more information about SVN, the differences and advantages to CVS see [1].

2. Usage

1. Actually we use subversion 1.1.3 on our server virgo. To use subversion without problems make sure you have installed a compatible version of subversion on your local computer.

2. Make a working copy of the enzo repository (this procedure is called checkout)[1]:

   ```
   > svn checkout svn+ssh://virgo/subversion/enzo/enzowue
   ```

3. Make changes to your working directory (editing files, creating, deleting files, etc.)

[1] If you want to access our repository from a computer not in our astro-domain (for example from `altix2.lrz-muenchen.de`) you have to use `svn+ssh://[user]@virgo.astro.uni-wuerzburg.de/subversion/enzo/enzowue`, where `[user]` is your username in our astro-domain. You will be asked for your password three times, so don't give up too early.

4. Check the status of your working directory

```
> svn status
? Make.mach.astrowue
```

Be careful, `svn status` shows the status of the actual directory + subdirectories. That means it will only show the complete status of your working copy, if you are in the root directory of your working copy. The following list shows how to intepret the output of `svn status`:

```
? foo.o              # svn doesn't manage foo.o
A stuff/loot/bloo.h  # this file is scheduled for addition
L abc.c              # svn has a lock in its .svn directory for abc.c
M bar.c              # the content in bar.c has local modifications
M baz.c              # baz.c has property but no content modifications
X 3rd_party          # this dir is part of an externals definition
! some_dir           # svn manages this, but it's either missing or incomplete
~ qux                # versioned as file/dir/link, but type has changed
I .screenrc          # svn doesn't manage this, and is configured to ignore it
A + moved_dir        # added with history of where it came from
M + moved_dir/README # added with history and has local modifications
D stuff/fish.c       # this file is scheduled for deletion
```

In our case we have added a file `Make.mach.astrowue`, but haven't told svn that we want to add this file to the repository.

5. Adding files to the repository

```
> svn add Make.mach.astrowue
> svn status
A Make.mach.astrowue
```

Actually we haven't added the file to the repository, but we only have marked the file `Make.mach.astrowue` to be added to the repository the next time we will commit all our changes to the repository.

6. Clean up your working directory before commiting changes to the repository

Before commiting changes to the repository one should clean up the working directory. Due to calls of `./configure` or `make` a lot of additional files are generated which should not be sent to the repository. [2] To delete these files we call

```
> make distclean
> ./svnclean
```

[2] As long as they are not added by `svn add`, subversion doesn't submit these files to the repository anyway. But to be not confused when checking the status of your working copy one should delete these files.

in the root directory of our working copy of Enzo. The script `svnclean` is a special script written by me and deletes all files that are left over by `make distclean`. After executing these two commands we really have a virginal Enzo directory exept for our own changes.

7. Include the changes of others in your working copy

   You can only commit changes to the repository, if the revision of your working copy is the same as the revision of the repository. If somebody already submitted changes to the repository after you checked out your working copy the revision of your working copy will be lower than the revision of the repository. So you have to update your working copy by

   ```
   > svn update
   U INSTALL
   G README
   C Grid.h
   ```

   But have no fear: Subversion will not delete your changes in your working copy. It will merge the changes from the repository into your working directory in several ways indicated by `C`, `G`, `U`.

   The `U` and `G` codes are no cause for concern; those files cleanly absorbed changes from the repository. The files marked with `U` contained no local changes but were `U`pdated with changes from the repository. The `G` stands for mer`G`ed, which means that the file had local changes, but the changes coming from the repository didn't overlap with the local changes.

   But the `C` stands for conflict. This means that the changes from the repository overlapped with your own, and you have to manually edit these files to resolve the conflict in order to be allowed to commit to the repository. This means you have to decide which of the changes should be included in the next revision of the repository. Subversion helps you in this process:

   - For every conflicted file, Subversion places up to three extra unversioned files in your working copy:

     `filename.mine`

     This is your file as it existed in your working copy before you updated your working copy — that is, without conflict markers. This file has your latest changes in it and nothing else.

     `filename.rOLDREV`

     This is the file that you checked out before you made your changes.

`filename.rNEWREV`

This is the file that your Subversion client just received from the server when you updated your working copy.

Here `OLDREV` is the revision number of the file in your .svn directory and `NEWREV` is the actual revision number of the repository.

- If Subversion considers the file to be of a mergable type (these are all kinds of textfiles, but no binary files like pictures etc.), it places conflict markers — special strings of text which delimit the "sides" of the conflict — into the file to visibly demonstrate the overlapping areas.

If you open the conflicting file you will therefore see something like

```
....
<<<<<<< .mine
your changes
=======
changes from others
(included in Revision 20 from the repository
you just recieved)
>>>>>>> .r20
....
```

If you are sure, that your changes are right and the other changes are wrong you can simpy delete the other changes and the conflict markers. If not, you should communicate with the others and then merge your changes into their changes or even discard your changes. After doing this (and deleting the conflict markers, of course) you should first test the changed file by compiling and running Enzo again, to make sure everything still works. If everything is ok, you must tell Subversion that the conflict in the file (here `Grid.h`) is resolved by calling

```
> svn resolved Grid.h
```

After completing this procedure for every conflicting file you should clean up your working directory again (see 6) and do another call of `svn update` to make sure, no conflicts are left.

8. Last checks before commiting changes

   Before finally commiting your changes to the repository you should call

```
>svn status
A Make.mach.astrowue
M Grid.h
M unkown.C
```

If you cannot remember, what changes you did for example to the file `unkown.h` you can call

```
> svn diff
```

This will show you all changes you made to working copy compared to the last revision you received from the repository. Now, if you are sure that your changes to `unkown.h` were unnecessary or just stupid, you can revert this file back to the unchanged version from the repository by calling

```
> svn revert unkown.h
```

If after calling `svn status` again everything is nice and clear you are finally ready to commit your changes to the repository.

9. Submit your changes to the repository

   ```
   > svn commit -m "comments about my changes"
   ```

   After commiting one can check if everything is ok by looking at the history of all the changes made to the repository.

10. Show the history of changes to the repository

    ```
    > svn log -v
    ```

    This will show all the changes plus additional changes to the paths (`-v`) of the repository. If you want to see only the changes made to lets say revision 3 add the option `-r 3` to the command. If everything is ok you should update your own working copy, so subversion knows that your changes are now offical part of the actual version of the repository. If you don't do that `svn status` will still show your added or changed files, because it will compare it with the old version you formerly checked out.

11. Update your working copy

    ```
    > svn update
    ```

Congratulations! You have submitted your first changes to Enzo. You can now continue by starting again at point 3.

A. How to create a new repository

To create a new repository in our subversion system, your user account must be member of the group `svn`. If this is the case you can install a new repository with the following commands

- Create a new directory on our subversion server virgo

  ```
  > ssh -l [user] virgo.astro.uni-wuerzburg.de
  > cd /subversion
  > mkdir project
  ```

  where `[user]` is your username and `project` should be a resonable name for your new repository.

- Make the directory `project` a subversion directory

  ```
  > svnadmin create --fs-type fsfs project
  ```

  Subversion will generate some sub-directories and files in `project`. The option `--fs-type` makes sure that subversion will use the file system as system for data storage an not the obsolete Berkeley database[3].

- Now you have to change the access rights to the repository. Otherwise only you are allowed to commit changes and not other users of the group `svn`. So change the group of the repository so all members of the group `svn` have access to it

  ```
  > chgrp -R svn project
  ```

  Everybody else (`others`) must not have access to the directory

  ```
  > chmod -R o-rwx project
  ```

  The members of the group `svn` must be allowed to read and write

  ```
  > chmod -R g+rw project
  ```

  To allow the subversion system special access to the log files, you also have to do the following

  ```
  > chmod g+s project/db
  ```

[3] See [1, Chapter 5. Repository Administration].

- If the subversion client is installed on your computer[4] you can now fill the repository with your data[5]

```
> svn import dir svn+ssh://[svnserver]/subversion/project/dir \
  -m "First version"
```

- You can now checkout the data from the newly created repository using

```
> svn checkout svn+ssh://[svnserver]/subversion/project
```

  and work with the repository as explained in section 2.

If you want to remove your repository just use the standard remove command

```
> rm -r project
```

B. Tips & tricks

- To show possible options for a svn command, e.g `checkout` type

```
> svn help checkout
```

- To checkout a specific revision of ENZO, e.g. 10, type

```
> svn checkout -r 10 svn+ssh://virgo/subversion/enzo/enzowue
```

- To show the differences between specific revisions of one file, e.g. show the difference between revision 4 and revision 9 of `Grid.h`, type

```
> svn diff -r 4:10 Grid.h
```

- One can show the differences between two files in a very nice graphical way by using the KDE tool `kompare`. Just pipe the output of the `svn diff` to `kompare` like[6]

```
> svn diff -r 4:10 Grid.h | kompare -o -
```

  and you will get an output like you can see in figure 1.

[4]Check it calling `which svn` from the shell.
[5]Remember: For our system `[svnserver]` is `[user]@virgo.astro.uni-wuerzburg.de`.
[6]Be sure not to forget the last minus sign!

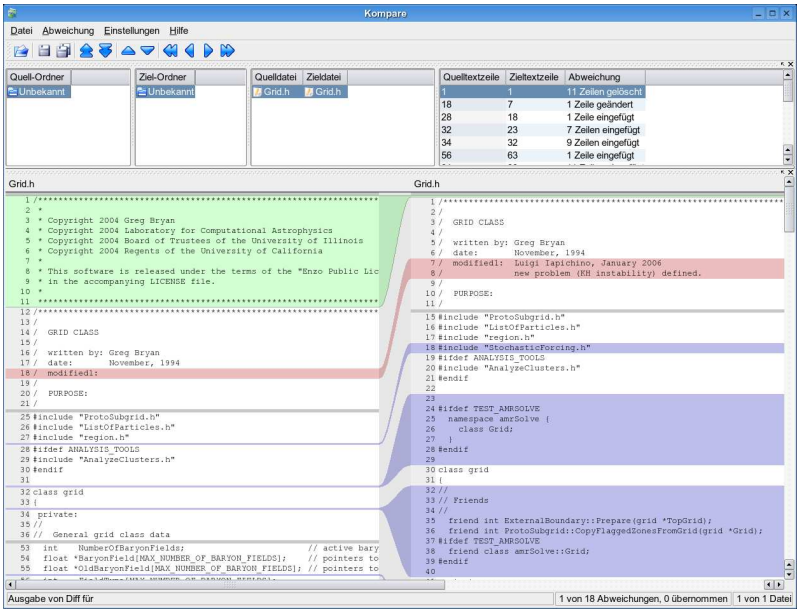

Figure 1: The KDE tool `kompare`. Newly added lines are shown in blue, modified lines are shown in red, and deleted lines are shown in green.

C. Using a better `diff`

It is an annoying feature of the standard `svn diff` command, that it reports also differences in whitespace, that means even if you or your editor just changes a tab into spaces or vice versa `diff` will show you these differences, although they make no difference for compiling the code. For example, if you type

```
> svn diff -r 1:28 Grid_ProjectSolutionToParentGrid.C
```

you will see the following output

```
...

-  if (ProcessorNumber == MyProcessorNumber)
-    for (field = 0; field < NumberOfBaryonFields; field++) {
-      skipi = skipj = skipk = 1;
-      float weight = RelativeVolume;

...

+       if (ProcessorNumber == MyProcessorNumber)
+               for (field = 0; field < NumberOfBaryonFields; field++)
+               {
+                       skipi = skipj = skipk = 1;
+                       float weight = RelativeVolume;

...
```

The standard `svn diff` does not allow to change this behaviour. Fortunately with `svn` you can also use the GNU `diff` command, which is on most system available

under `\usr\bin\diff`. Typing `man diff` shows you, that we need to use the GNU `diff` with the option `--ignore-space-change` or equivalent `-b`. To use it with `svn` to show only the non-space-changes between revision 1 and 28 of the file `Grid_ProjectSolutionToParentGrid.C` type

```
> svn diff --diff-cmd diff -x -b
           -r 1:28 Grid_ProjectSolutionToParentGrid.C
```

This yields the following output

```
...

180c180,181
<     for (field = 0; field < NumberOfBaryonFields; field++) {
---
>               for (field = 0; field < NumberOfBaryonFields; field++)
>               {

...
```

Now the lines which changes in space are not shown anymore. But still it does show the movement of the curly bracket from one line to another, although this doesn't make any difference when compiling the code. Unfortunately this cannot be avoided, because GNU `diff` as `svn diff` can only compare text files line by line and do not parse the content.

References

[1] Sussman, Fitzpatrick, Pilato: Version Control with Subversion, http://svnbook.red-bean.com/en/1.1/svn-book.pdf,2005

[2] Jahre, D.: Subversion - Versionsverwaltungssystem (Was kommt nach CVS?), http://www.clug.de/vortraege/subversion.pdf