

# Subversion workflow guide

Joanne Carr <j.m.carr@open.ac.uk>

January 2010

## Contents

<b>1</b>	<b>Before we start: some definitions, etc.</b>	<b>2</b>
<b>2</b>	<b>UKRmol-in repository layout</b>	<b>3</b>
<b>3</b>	<b>Checking out</b>	<b>3</b>
<b>4</b>	<b>Monitoring and dealing with changes – both in the repository and local</b>	<b>4</b>
4.1	Other people’s changes to the repository . . . . .	4
4.2	Your local changes . . . . .	5
<b>5</b>	<b>Committing your local changes to the repository</b>	<b>6</b>
<b>6</b>	<b>The log file</b>	<b>7</b>
<b>7</b>	<b>Branching</b>	<b>7</b>
<b>8</b>	<b>Merging (also known as Porting)</b>	<b>8</b>
<b>9</b>	<b>Conflicts</b>	<b>11</b>

This document is a guide to the basic day-to-day Subversion workflows. It is by no means comprehensive, and assumes that the reader is already familiar with version control concepts in general and the way that Subversion does things in particular (including differences from other systems such as CVS). See the Subversion manual (<http://svnbook.red-bean.com>) for much more information and the definitive answer. This workflow guide makes use of some excerpts from the manual, particularly definitions: I gratefully acknowledge the authors Ben Collins–Sussman, Brian W. Fitzpatrick and C. Michael Pilato for writing such a useful companion to Subversion and releasing it under the Creative Commons Attribution License (<http://creativecommons.org/licenses/by/2.0/>).

`svn` is the name of the command-line client program for Subversion; `svn` commands are typeset like `this` in this guide.

## 1 Before we start: some definitions, etc.

- Your working copy is the set of directories and files on your local machine that came from the repository originally via the `svn checkout` command (see later).
- Revision numbers in a Subversion repository are ‘global’: they refer to the repository filesystem as a whole rather than its individual subdirectories or files. That is to say, each revision number identifies a particular state of the repository after some committed change. This means that revisions `n` and `m` of a particular file do not necessarily differ, where ‘revision `n` of a file’ is a quick way of saying ‘the file as it exists in revision `n`’.
- There are various revision keywords that can be used in place of a revision number in an `svn` command:
  - `HEAD`: the latest (‘youngest’) revision in the repository of the directory being considered (whether a local working copy or a repository URL).
  - `BASE`: the revision number of an item in a working copy. If the item has been locally modified, this refers to the way the item appears without those local modifications (i.e. the pristine copy immediately after the checkout or most recent update).
  - `COMMITTED`: the most recent revision prior or equal to `BASE` in which an item changed.
  - `PREV`: the revision immediately before the last revision in which an item changed, or `COMMITTED-1`.

The first two keywords are likely to be used more often than the latter two. As distinct from `HEAD`, the three keywords `PREV`, `BASE`, and `COMMITTED` can be used to refer to working copies only – not repository URLs.

- Something very useful to note is that `svn help <cmd>`, where `<cmd>` is one of the `svn` subcommands (e.g. `checkout`, `commit`, `status`), will give a description of the subcommand and its options.

## 2 UKRmol-in repository layout

At the top level of the CCPForge UKRmol-in repository there are the following directories:

- trunk/ The main development stream, sometimes referred to in version control jargon as the ‘main branch’. This contains stable, well tested and fully documented code. Bug fixes go into here (when they’ve been tested). New functionality when complete (and tested...) will be ported from its development branch (see below) into the trunk for incorporation in a future public release.
- dev-branches/ The home for people’s development branches. Coding for new functionality starts in a new branch, thus allowing a work in progress to be maintained under version control without destabilizing the trunk and affecting other users. Your development branch is your own space – you can use whatever working patterns suit you best. The only requirement is that the code is in good order (i.e. thoroughly documented and tested, and complies with the standards set out in our coding guide) in time for the port into trunk. See section 7 for a discussion of branching.
- archive/ The place where old (i.e. before October 2009) versions of the code and more are stored.
- tags/ Contains code taken from trunk/ to be prepared for a public release version.
- project\_guides/ Contains documents for developers, to be read before starting coding, concerning the running of the UKRmol-in project. Our coding standards document lives here (as well as in the Docs section of CCPForge).

Each of the current development streams (trunk/ and individual directories under dev-branches/) contains three subdirectories:

- source/ Contains the source code and Makefile.
- doc/ Contains the user documentation.
- tests/ Contains the test suite appropriate for / compatible with source/ at the same revision.

## 3 Checking out

For the initial checkout, run

```
svn co --username <name> http://ccpforge.cse.rl.ac.uk/svn/ukrmol-in/
```

which gets everything at the HEAD of the repository and puts it in the current working directory, as `./ukrmol-in/<repository-structure>`. Note that either the long name, `checkout`, or the shorthand, `co`, can be used in the command.

You can append a directory path in the repository to the end of the CCPForge URL (for example, `http://ccpforge.cse.rl.ac.uk/svn/ukrmol-in/trunk/source`) to get that directory (and any subdirectories) only. A local directory path can be added as a separate argument at the end of the command to specify a different destination to the current working directory for the checked-out files.

Subversion by default should store your credentials for any subsequent commands, so there should be no further need to include the `--username <name>` option and give your CCPForge password. Note that such caching can be turned off, though.

## 4 Monitoring and dealing with changes – both in the repository and local

### 4.1 Other people's changes to the repository

Suppose that you're reading papers / thinking and so don't change anything in your working copy for a while. This means that your working copy is still identical to BASE. In the meantime, another developer is making and committing changes. You can see what's going on in the repository with:

- `svn diff -r BASE:HEAD`  
which compares the BASE and HEAD revisions, displaying differences between the files at the two revisions in unified diff format.
- `svn status -u`  
which produces a summary of the 'state' of your working copy with respect to HEAD (and would therefore include any local modifications you had made).

Particular things to look for in the output from the status command: an 'M' in the entry for a file means that it is locally modified; a 'C' means it is in a state of conflict (see Section 9), and a '\*' means that the file is out of date with respect to the repository.

The status and diff commands can be issued from any of the subdirectories of your working copy, and will then apply to that directory and its subdirectories only. They can also optionally take a filename at the end to specify the 'target' (rather than being applied to the whole directory).

An additional command that you might find useful is `svn cat`, which allows you to view an earlier revision (specified by the `-r n` option) of a file without your local copy being modified. The output can be redirected, for example:

```
svn cat -r 3 filename > filename.v3
```

You can update your working copy with the changes in the repository since your checkout or last update with

- `svn update`  
The default behaviour is to bring your working copy up-to-date with HEAD; you can specify a particular revision with `svn update -r <number>`.

## 4.2 Your local changes

Now the time comes for you to make some local modifications – to do some coding in your working copy.

You need to keep track of the changes you have made locally with respect to BASE (reminder: the revision you either checked out or to which you most recently updated your working copy):

- `svn status`  
which shows the status of your working copy with respect to BASE. Note that the `-u` flag, which causes the comparison to be made between your working copy and HEAD, is absent here.
- `svn diff`  
which shows the differences between files in your working copy and in BASE.

You also need to deal with other people's changes that have been made in the repository in the meantime (see section 4.1). It's good policy to regularly bring your working copy up-to-date with HEAD (apart from your local modifications, i.e. with your local changes 'on top'). This makes subsequent commits of your own changes into the repository significantly easier (see Section 5). Running `svn update` will either cause local files to absorb changes from the repository 'cleanly' (by either simply updating a file that has no local changes or merging textually non-overlapping repository changes into a locally modified file), or will flag a conflict where the changes from the repository and your own overlap (see Section 9). Note that an update should never be performed 'blind', i.e. without checking by hand the changes that would be applied for 'hidden' conflicts with your own development work via the various `svn diff` commands mentioned above. For instance, the edits that are textually non-overlapping in a merged file as described above may in fact interfere with each other conceptually.

A very useful additional command for examining differences is

- `svn diff -r HEAD`  
which compares your working copy, including all the local changes, with HEAD.

Differences between any two revision numbers, *n* and *m*, in the repository can also be examined via `svn diff -r n:m`.

If you want to throw away your local changes to a file or directory, returning it to the pristine copy corresponding to the BASE revision, then run

- `svn revert <name>`

## 5 Committing your local changes to the repository

Your local edits are complete: they have been thoroughly tested in terms of running a test suite and are compatible with all the concurrent changes in the repository. They are ready to be ‘committed’ to the repository. Note that it’s best practice for a single commit to comprise ONLY related changes, e.g. a collection of edits implementing a new functionality or fixing a single bug. That is to say, new code for distinct purposes should be committed separately. This does not mean, however, that one must wait until the coding of a new feature is complete before making the first commit; incremental commits to your own branch (and, of course, updates / merges from the repository) have many advantages for both you and other developers.

If your changes involve adding / deleting files, then you need to let Subversion know about this before committing:

```
svn add <filename>
```

```
svn delete <filename>
```

Note that the delete command also immediately removes the file locally.

To commit all your local changes to the repository:

- `svn commit`

Or, to commit changes in one file only, run `svn commit <filename>`

You’ll be prompted to supply a message for the log as part of the commit: it’s important that you provide something descriptive and helpful. For the UKRmol-in project, a commit will be rejected unless its log message has a particular format – some non-empty text for each of the following headings (formatted exactly as below):

- \* changes to functionality and/or problem solved:
- \* subroutines affected:
- \* dependencies:
- \* changes to required input:
- \* status:

Some suggestions to make this process as efficient as possible:

- Keep a commit log template file and edit a copy of this for each commit by adding content; then run `svn commit -F <commit-log-tmp-file>`
- Keep a commit log template file and have Subversion automatically open it with your favourite editor when you commit, to write the message on the fly. You’ll need to edit the Subversion config file, which should be in the `.subversion/` directory on your local system. Make sure the line `[helpers]` is uncommented, and that the `editor-cmd =` line is uncommented and contains something appropriate. For instance, with the vim editor, try `editor-cmd = vim + "r commit-template-filename"`.

Some important points to note:

- A commit will fail with an appropriate message if the pristine BASE version of a file you're trying to change is out-of-date with respect to the HEAD of the repository. That is, since your last update of the working copy, someone has committed a change to the file that you have not yet brought into your working copy. You should run an update, dealing with the differences and/or any conflicts as usual, and then try the commit again.
- Immediately after a successful commit, it's usual to run `svn update` in order to set your whole working copy (BASE) to the new HEAD, including the revision numbers. You should not see any conflicts with this update, as the files in your working copy and in the new HEAD will be identical. You won't be able to see your commit message with `svn log` (see section 6) until after you've run `svn update`.

## 6 The log file

You can examine the commit messages from the repository, up to and including the latest update, for all files (including subdirectories) in the current working directory of the working copy with:

- `svn log`

This file quickly gets large, so you'll probably want to redirect the output. Note that revision numbers are incremented for every commit to a repository as a whole, so consecutive commits to a particular branch need not have consecutive revision numbers.

`svn log <filename>` gives all log messages from commits involving the file `<filename>` to the branch you're looking at. A URL to be examined instead of the local working copy can also be given, and a revision specifier or range may be supplied via the `-r n[:m]` syntax. The default range is `BASE:1` for working copies.

## 7 Branching

As discussed in Section 2, any non-trivial development work should take place not in trunk but in a separate branch. A branch is created in the repository by remotely copying one of the existing development streams (usually trunk) as follows:

- `svn copy url-old url-new`

where, for example, `url-old` is `http://ccpforge.cse.rl.ac.uk/svn/ukrmol-in/trunk` and `url-new` is `http://ccpforge.cse.rl.ac.uk/svn/ukrmol-in/dev-branches/new-branch-name`. Since this procedure acts on the repository directly it causes a near-instantaneous commit. This means that you

will have to provide a log message in some way (the command given exactly as above can act in the same way as `svn commit`, by opening your favourite editor for you to type the message).

Now that your branch exists in the repository you can get a working copy of it in a number of ways:

- `svn checkout` from an appropriate place in your local directory structure.
- `svn update` from an appropriate place in your local directory structure (within a directory that came from the repository at some point and from which the new branch would be a subdirectory. This would be `<local path>/ukrmol-in/dev-branches/` in the above example).
- `svn switch` – see the manual for details.

## 8 Merging (also known as Porting)

When a branch is created for the development of a new feature, the aim is for the completed changes to eventually be incorporated into the trunk as part of the mainstream evolution of the codebase. However, the trunk itself will probably have progressed since the point at which you copied it to make your branch. Just as it's important to run `svn update` regularly to pull other people's changes made on the same development stream (branch) into your working copy, you should also regularly *\*\*and carefully\*\** merge the concurrent changes on trunk into your own branch.

*Before merging, always make sure that your working copy of the destination branch (the one that will receive the changes) has no local modifications and is up-to-date with respect to the repository, i.e. your working copy and the HEAD of the branch are identical. Otherwise the process of merging gets unnecessarily complicated...*

The procedure for merging depends on which versions of `svn` you and the repository server are running: if both  $\geq 1.5$  then new functionality exists to help you; if either  $< 1.5$  then you have to keep track of some information 'by hand', as described below. For all versions of `svn`, though, a merge involves two distinct stages: the changes from the repository are made locally, to your working copy, and then those changes are committed to the branch in the repository.

An example, part 1: merging into your branch the set of changes committed to trunk between the point (say revision 30) at which you created your branch and the current HEAD of trunk (call it revision number 50).

With `svn` versions up to 1.5:

- From the local directory containing the working copy of your branch, run `svn merge -r 30:50 <URL-of-trunk>`
- Examine the changes applied locally with `svn diff` and test them thoroughly.

- When you're satisfied, commit the changes to your branch: `svn commit` with a helpful log message that records the revision range you merged, followed by `svn update` as usual.

Note that you need to tell Subversion explicitly the revision range you are merging. When the time comes to make another merge into your branch of the subsequent changes on trunk, i.e. those between revision 50 and the now-current HEAD (revision 60 say), you follow the same steps but give a different revision range, `-r 50:60`, in the merge command. In order to find out the revision number of the end-point for the previous merge (which is 50 in this example), you should consult the commit log: `svn log | grep -i merge` or similar. The developer should take care that Subversion is *not* asked to apply the same set of changes more than once to a branch.

With `svn` version 1.5 or greater, which will store the revision number at which you created your branch and the revision number ranges of prior merges:

- From the local directory containing the working copy of your branch, simply run `svn merge <URL-of-trunk>`
- Examine the changes applied locally with `svn diff` and test them thoroughly.
- When you're satisfied, commit the changes to your branch: `svn commit` with a helpful log message that includes the word 'merge', followed by `svn update` as usual.
- See 'The Final Word on Merge Tracking' section of the Subversion book (for example, <http://svnbook.red-bean.com/en/1.5/svn-book.html#svn.branchmerge.advanced.finalword>) for some additional recommendations.

When the time comes to make another merge into your branch of the subsequent changes on trunk, i.e. those between revision 50 and the now-current HEAD, you just run exactly the same set of commands, starting with `svn merge <URL-of-trunk>`. Subversion will automatically only bring into your branch the changes to trunk that you don't yet have.

An example, part 2: merging all the changes made on your branch into trunk when your development work is complete (and documented and tested!).

Common instructions:

- Make sure you have a working copy of trunk that has no local modifications and is up-to-date with respect to the HEAD of the repository.
- Update your branch with the latest changes on trunk, i.e. do a final merge from trunk into your branch, as above.

Then...

With `svn` versions up to 1.5:

- You need to identify the revision number at which your branch was created, as the lower limit of the merge range corresponds to the initial state of the branch. This is revision 31 in our example, which came from the trunk at revision 30. You can readily find this out with the following command: `svn log -v --stop-on-copy`, which stops the log messages at the point where the copy command was issued to create the branch.
- From the local directory containing the working copy of trunk, run `svn merge -r 31:HEAD <URL-of-your-branch>`  
With such a command you'll apparently be applying changes ported from trunk into your branch back into trunk – Subversion should behave sensibly and leave those parts of trunk as-is (and indeed as it should be).
- Examine the changes applied locally with `svn diff` and test them thoroughly. You may well have to resolve some conflicts (see section 9).
- When you're satisfied, commit the changes to trunk with a helpful log message that records the merge range.

If you go through this procedure again later, after more commits have been made to your branch, the merge range should be from the number corresponding to HEAD at the time of the previous such merge to the current HEAD.

With `svn` version 1.5 or greater, which will store the revision number at which you created your branch and the revision number ranges of prior merges:

- From the local directory containing the working copy of trunk, run `svn merge --reintegrate <URL-of-your-branch>`
- Examine the changes applied locally with `svn diff` and test them thoroughly.
- When you're satisfied, commit the changes to trunk with a helpful log message.
- Note that once you've done this `--reintegrate` merge from a branch to the trunk, that branch is no longer usable for further work. It cannot correctly pick up new trunk changes, and another `--reintegrate` merge cannot be applied. A new branch must be created for further development work.

Merging summary: the merge command has a `--dry-run` option that can provide a useful overview without actually applying any changes. Also remember that changes applied after a merge are local until they are committed: they can be reverted if necessary or tweaked to resolve any incompatibilities or conflicts, etc, before the actual commit. Merging can be confusing at first – take your time over it, don't panic, and consult the manual for much more information!

## 9 Conflicts

When a file becomes conflicted as part of an update or merge, there are a number of mechanisms for resolving the conflict before you can commit, some of which depend on the version of svn you're running. You will know when a conflict has arisen because it will be flagged in the output of the update/merge command (one of the files will be marked with a 'C' for 'conflict').

For svn versions 1.4 and below, three temporary files will also have been placed in your working copy. One of these (called <filename>.mine) will be a copy of the file from your working copy as it was immediately before the update / merge. The file itself in the working copy will have conflict markers (strings of less-than, equals and greater-than signs) added to delimit the 'sides' of the conflict. You can resolve the conflict by editing the file itself (and removing the conflict markers) or by copying one of the temporary files onto the working file. Then, run `svn resolved <filename>` to tell Subversion to remove the temporary files and allow a subsequent commit.

During an update, svn versions 1.5 and above allow interactive conflict resolution, where you're offered a range of options (documented inline via the 'h' option). One option is to postpone resolution, in which case the three temporary files described above for earlier versions are created and conflict markers are placed in the working copy of the file. You can resolve the conflict by editing the working file itself to make the appropriate changes (and remove the conflict markers) or by choosing one of the temporary files (no need to copy it onto the working file now). Then, run `svn resolve --accept <arg> <filename>`, where filename is simply the name of the file in conflict, to remove the temporary files and allow a subsequent commit. Note: the subcommand is resolve NOT resolved here. The argument given to `--accept` specifies the desired resolution:

- `base`: choose the file that was the BASE revision before you updated your working copy. That is, the file that you checked out before you made your latest edits.
- `working`: assuming that you've manually resolved the conflict, choose the version of the file as it currently stands in your working copy.
- `mine-full`: choose the file as it existed immediately before you ran the update.
- `theirs-full`: choose the file that was fetched from the repository when you ran the update.

*If a conflict occurs, please ensure that you resolve it to the satisfaction of all parties involved.*