

Local Version Control (sccs, rcs)

Steven J Zeil

February 21, 2013



Outline

- 1 History
- 2 Exploration
- 3 Collaboration
- 4 Strengths and Weaknesses



Localized Version Control

The earliest version control systems in wide use were `sccs` and the open source `rcs`.

- We'll focus on `rcs`
- The “repository” of historical information is kept as a “special” subdirectory, named `RCS`
- Sharing of repositories is possible only via the operating system's underlying mechanism for sharing access to directories
 - permissions, linking



Basic rcs Operations

- **ci** Check In a file from the working directory into the repository
- **co** Check Out a file from the repository into the working directory
- **rcsdiff** Compare two versions of a file.
- **rcsmerge**




Outline I

- 1 History
- 2 Exploration
- 3 Collaboration
- 4 Strengths and Weaknesses



History I

- `mkdir RCS`
Creates an RCS repository for the files in the current directory (only)
 - The repository is currently empty
- `ci filename`
Checks files in to the repository
 - If the file is not in there yet, it is added
 - If it is in there, then this becomes the new/current revision
 - Each check in is assigned a new, ascending revision number

```
graph LR; 1.1 --> 1.2; 1.2 --> 1.3; 1.3 --> 1.4; 1.4 --> 1.5;
```

 - Somewhat surprisingly, deletes the file from the current directory
- `co -l filename`
Checks out the most recent version of that file from the repository, storing it in the working directory.

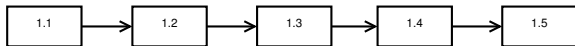


History II

- Adding a `-r v` option allows check out of a specific revision number



Revision Numbers



- Clearly there was an intent that revision numbers also serve as version numbers.
 - A special option allows you to force a change to the leading digit,
e.g., to move from version 1.12 to 2.0
- Problem is that each file's revision number changes independently
 - So your intended release "version 2.1" might use revision 2.1, revision 2.5 of `adt.cpp`, revision 2.3 of `main.cpp`, etc.
- Versions can be checked out by date instead:
"check out whatever version was current as of 12/13/2012"
 - Repeated over all files, would give a coherent view of the project status as of that date



Naming Revisions

- Revisions can be named:

```
ci -N "v1.2" -t "Public release 1.2" *.h *.cpp
```

and later checked out by name instead of by exact revision number

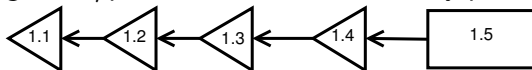
- Note also the option to add explanatory text at the time of the checkout
 - Later version managers would make this “mandatory”



Implementation

rcs is essentially a systematic way of creating and organizing patches.

- The repository always contains the current version of the file plus enough diffs/patches to move back to any prior revision.



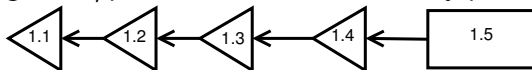
- The current version is always available immediately.
 - Diffs are used to go back in time



Implementation

rcs is essentially a systematic way of creating and organizing patches.

- The repository always contains the current version of the file plus enough diffs/patches to move back to any prior revision.



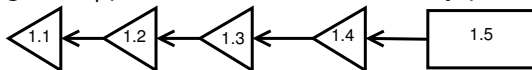
- The current version is always available immediately.
 - Diffs are used to go back in time
 - Originally considered an important point in supporting efficient access to the most commonly needed file.



Implementation

rcs is essentially a systematic way of creating and organizing patches.

- The repository always contains the current version of the file plus enough diffs/patches to move back to any prior revision.



- The current version is always available immediately.
 - Diffs are used to go back in time
 - Originally considered an important point in supporting efficient access to the most commonly needed file.
 - Now, probably not so important



Outline I

1 History

2 Exploration

3 Collaboration

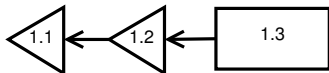
4 Strengths and Weaknesses



Exploring Alternatives

Suppose that we have worked through a few revisions and then get an idea that might not pay off.

We can start a *branch* to explore our idea while others continue work on the main trunk.



```
ci -r1.3.1 filename
```

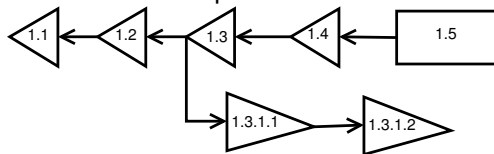
Checks in our current version of *filename* as a new branch of development, numbered 1.3.1.1

- 1.3.1.1 is the trunk version from which we branched out
- 1.3.1.1 is the branch number
- 1.3.1.1 is the revision number within the branch



Working in a Branch

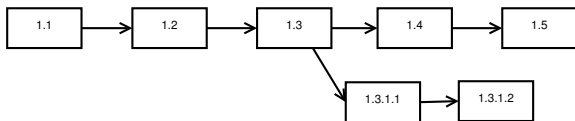
Subsequent check-ins of both the main trunk (1.3) and of our branch version will maintain separate revision numbers:



- Note that checking out the most recent version along a branch is not as efficient as checking out the most recent version on the trunk.



Merging a Branch



- If the idea in the branch does not pay off, the branch can simply be abandoned.
- You decide to adopt the changes in the branch, you can elect to *merge* it back into the trunk.
 - The **rcsmerge** command is used to conduct the merge,
 - Need to resolve any conflicts introduced by continued development along the trunk.
 - then the resulting combined file checked in with a trunk number

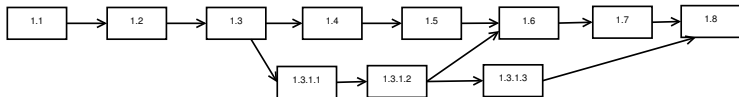


Multiple Merges

After a merge



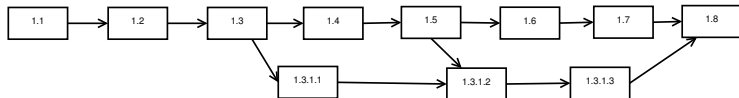
- We might opt to discontinue using the branch
- Or we might continue working long it, eventually generating more changes to be merged into the system



Combating Drift

Over time, a long-running branch can get so far out of sync with changes being made to the trunk that the final merge becomes difficult or even impossible.

- An effective strategy for combating this is to periodically merge the trunk into the branch
 - the reverse of the “normal” merge direction



Outline I

- 1 History
- 2 Exploration
- 3 Collaboration**
- 4 Strengths and Weaknesses



Collaboration

rcs supports collaboration by *locking* files

- Most checkouts like this

```
co filename
```

obtain a read-only copy of the file.

- *nix permissions 400
- Can be used to compile system, but cannot be changed
 - (Of course, you can always **chmod**, but that's cheating.)



Locks

- A checkout like this

```
co -l filename
```

requests a *locked* version of the file.

- Request fails if a locked version already exists somewhere.
- If successful, programmer receives a copy with write permission.
- Lock persists until the programmer checks in changes or explicitly releases the lock (which deletes the file from their directory, forcing them to check out an unlocked, read-only version again).



Outline I

1 History

2 Exploration

3 Collaboration

4 Strengths and Weaknesses



Strengths and Weaknesses

- rcs addresses history, exploration, & collaboration concerns
- but has weaknesses in each area



History

- rcs tracks files in a directory.
 - Each file is tracked separately.
- No support for deletion of file
 - Unless you *know* not to request a file, you will always get the last version before it was deleted.
- No support for creation of new files
 - If you request revisions associated with very old dates, you will get version 1.1 even if the file did not actually exist as of that date.
- No support for renaming files
 - Appears to be a deletion and a subsequent creation of a new, unrelated file
- Each directory is tracked separately
 - Poor support for multi-directory projects



Exploration Issues

- Branching and merging is often confusing.



Collaboration Issues

- Locks are frequently abused
 - e.g., people forget to release a lock, forcing team members to wait
 - People grab locks they don't really need.
- Cheating on locks is easy
 - People get in the habit of cheating to cope with lock abuse
 - And eventually start cheating with less and less provocation.

