Concurrent Versions System, A Method for Independent Cooperation

Dick Grune

Vrije Universiteit
de Boelelaan 1081
1081 HV Amsterdam

People working together on a set of files in real-world circumstances may often interfere with each other. To avoid such interference, the idea of an abstract data type can be used. The abstract data type introduced here consists of a single repository, containing versions of the files, together with a (small) number of access routines. Each participant has his own copy of a set of files as represented by the repository, and uses essentially two access routines, one to merge into his copy changes made to the repository by others, and one to merge changes in his copy into the repository. (There are several other access routines, for listing contents, examining differences, etc.) The abstraction is not perfect, but violations (conflicts) are generally detected and reported.

The implementation of the abstract data type "repository" starts from version control primitives that can handle a single file and then coordinates their actions. All access routines are programmed as UNIX† shell scripts and use the RCS programs as version control primitives. In programming the access routines, many more possibilities had to be taken into account than was intuitively reasonable. Examples are given.

Index terms: concurrency, version control systems, RCS, multiple file update, user interface

I. The Problem

In a medium-sized project, it often happens that a (relatively small) number of people work simultaneously on a single set of files, the "program" or the "project". Often these people have additional tasks, causing their working speeds to differ greatly. One person may be working a steady ten hours a day on the project, a second may have barely time to dabble in the project enough to keep current, while a third participant may be sent off on an urgent temporary assignment just before finishing a modification. It would be nice if each participant could be abstracted from the vicissitudes of the lives of the others.

The system described here provides this abstraction by keeping the "files of the project" in a repository. It gives each participant his or her own copy of them and offers a number of commands to update the copy, to commit changes to the repository, etc. It is akin to some distributed file systems with optimistic concurrency control (see, e.g., Mullender and Tanenbaum[1] or Svobodova[2]), in so far as these are capable of implementing concurrency over a group of files. Its main novelties are its ease of use, its relative simplicity, and the length of the concurrency time span it supports (effectively forever). It is implemented as a simple set of command files ("shell scripts") under UNIX.

II. A Simple Solution

If "the project" consists of one file only, and if we disregard efficiency, the problem can be solved with only two features: a version-recording system and a program for merging differences.

Both features are well-known and readily available; nevertheless a short description follows to establish terminology and to sketch the field.

The version system should accept new versions of the file and be able to (re)produce old versions − on request, according to name or number. It is immaterial, except for matters of storage efficiency, whether it does this by keeping all versions integrally or by using deltas (condensed differences between files). Likewise it is

---

† UNIX is a Registered Trademark of AT&T Bell Laboratories.

immaterial, except for run-time efficiency, whether it records the original file and applies positive deltas to produce newer versions, or records the newest version and applies negative deltas to obtain older versions. Some operating systems provide such a scheme directly; an example is VMS‡[3], which keeps entire copies. On other systems, a software package must be used: UNIX SCCS[4] uses positive deltas and UNIX RCS[5] uses negative deltas.

When two participants each obtain a copy of version *P*, and one edits his copy into version *Q* while the other edits his copy into *R* (Figure 1), we shall need a mechanism to combine these edit actions (mechanically) into a new version *S*. This is where the difference merger comes in. We can view the edit actions as operators Δ_1 and Δ_2 on *P*, and write

$$Q=\Delta_1(P), \qquad R=\Delta_2(P)$$

This gives us two ways of obtaining *S*:

$$S_{12}=\Delta_1(\Delta_2(P)), \qquad S_{21}=\Delta_2(\Delta_1(P))$$

If $S_{12}=S_{21}$, optimistic concurrency was successful (Figure 2); if $S_{12}\neq S_{21}$, it failed (Figure 3).

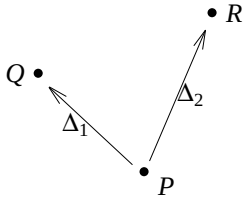

Figure 1

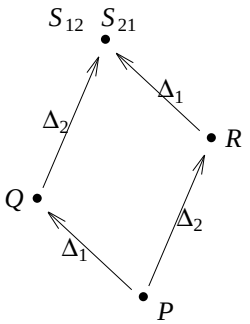

Figure 2

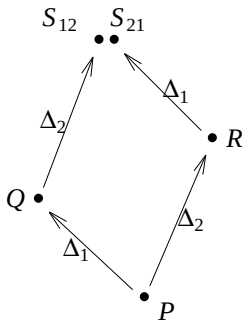

Figure 3

Experience with our Concurrent Versions System (CVS) has shown that optimistic-concurrency conflicts are rare, even if there are long lapses (say, of one month) in the updates. Those that do occur are easily mended; see the paragraph “Experience and Availability”.

Implementation of the merge actions is surprisingly simple (at a minimal loss in concurrency). Both Δs are constructed as lists of line-block replacement actions (it is the line orientation that partly accounts for the loss of concurrency). If all of the replacement actions affect different lines, the two lists can simply be merged while adjusting the line numbers, and the resulting list applied to *P* to obtain *S*. If, however, the lists collide somewhere, we have a concurrency conflict (and its location!).

It may be noted that this approach is weaker than the criteria set forth by Kung and Robinson[6] for optimistic concurrency control, in that we do not record that a line has been read by a user. Indeed one user can add, on a given line, a call to a function `F(a, b)` with two parameters, while another user changes the definition of `F` (and all the calls known to him) to have three parameters. Since the two sets of changes pertain to different lines, such a conflict would go unnoticed by CVS. However:

- the Kung & Robinson criteria can be defeated unwittingly by a user working from a listing,
- subsequent compilation of the “project” should throw up parameter mismatches,
- we have yet to see this happen in practice.

Also note that the Kung & Robinson criteria require all reading actions to be noticed and therefore either impose severe restrictions on the user or require extensive help from the operating system.

III. The One-File System in Practice

The repository holds all versions of the file; in our application the versions form a single sequence, without branches. In principle the repository may be a directory, an archive, a multi-file file or any other suitable object; it may even reside on a different machine. Our implementation uses a directory containing one or more RCS-files. RCS uses negative deltas, includes a difference merger and is available under UNIX; it automatically numbers its successive versions.

---

‡ VMS is a Trademark of Digital Equipment Corporation.

A call of “create version”

CV *repository-name*

in an empty user directory, will provide the new user with a personal copy of the file (newest version) *Q* and will record (in an auxiliary administration file) the number of the version used, *P*, as assigned by RCS. The user may now modify *Q* as he sees fit.

After a while the user may want his (possibly modified) file to be brought up to date. To this end he calls “update version”

UV

which, from the administration file, knows where to find the repository, what file is concerned and from which version it derives. Now for each newer version in the repository, UV calls the difference merger to move the user file *Q* one notch up the version ladder; see Figure 4 in which *Q′* and *Q″* are successively more updated versions of *Q*.

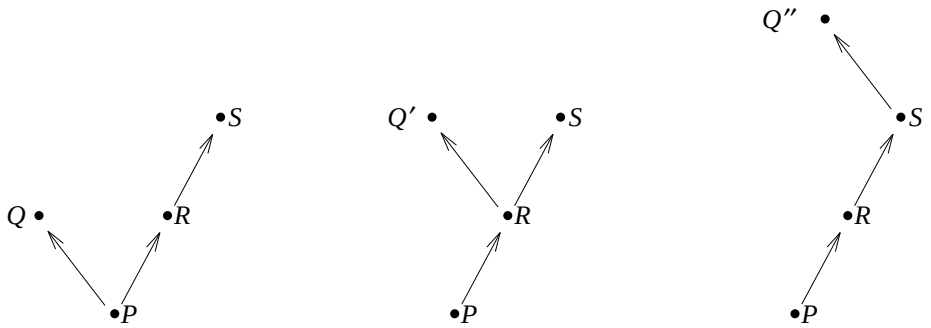

Figure 4

UV also updates the administration file to reflect the fact that *Q″* now derives from *S*. If the difference merger finds any conflicts (i.e., lines modified by the user *and* by the creator of *R* or *S*), these are reported to the user, to be corrected by hand. This correction may require consultation with the other participants, since at the bottom of the merge conflict there may well be a misunderstanding among the project members.

When the user is satisfied with his modified file *Q″*, he can commit it to the repository by calling

CM *′some text giving the reason why′*

(where the apostrophes are required by the UNIX shell syntax). This is allowed only if the file to be committed derives from the newest version; consequently the commit will always succeed, as depicted in Figure 5.

Figure 5

The user keeps his copy *Q″*; the newest version in the repository, *T*, will be equal to it; again the administration file is updated.

We are now back at the original situation after calling CV, i.e., the user has a copy, properly registered, of the newest version in the repository.

It should be noted that no change made by the user will be reflected in the repository until he calls CM, after which they will all appear simultaneously. (For possible synchronization problems with other participants, see below.)

IV. The Multi-File System

If restricted to one file only, the above commands would actually be of limited use; they would merely be packaged calls to RCS. It is only in the application to a−coherent−set of files that CVS finds its usefulness: it lets the user treat the whole set as a single object.

Extension of the one-file system to many files is conceptually simple: for each file, keep an entry in the administration file. Implementation, however, was far from simple, for several reasons:

- files may be added or removed concurrently;
- we need an optimization to avoid running the difference merger on every single file, even if only a fraction of them have been modified;
- we have to check data integrity, since the user file, the repository file, the administration entry, or any combination of these may have been lost; this interferes with the above;
- several users may attempt to commit their changes simultaneously.

We shall discuss these problems in turn.

A. Adding and removing files

To add a new file to the project, the user creates a file and then calls

`AE` *file-name*

Initially this just results in an entry in the administration file, marked “Added”. Only when this version is committed to the repository is an RCS-file made.

To remove a file from the project, the user removes the file from his own version and then calls

`RM` *file-name*

Again this just results in a “Removed” mark being set in the corresponding entry in the administration file. Upon the next call of `CM`, the RCS-file will be removed from the repository. (Actually, to be on the safe side, it is moved to a special part of the repository, the *Attic*.)

Since the actual adding and removing is delayed, we can allow the user to undo it. More particularly, the user can resurrect a file that was removed and RM-ed but not yet committed, by calling `AE` for that file. We have all the information to bring back exactly the same file that the user has removed.

B. The optimization

For the optimization to be effective, we need a way to check if a file was modified since the time it was derived from the repository, and we have to do better than to retrieve the original from the repository and do a compare. Fortunately, UNIX records the modification date (and time) of a file; we register the creation (modification) date of each file in the administration file as a time stamp, at the moment it is derived from the repository. Rather than just calling the difference merger, we first check the present time stamp of a file *Q* against the one recorded in the administration file. If they are equal the file has not been modified by the user, and Δ_1 was the identity operation; *Q* is thrown away and is replaced by a copy of the newest version. This is considerably faster than calling the difference merger.

Actually, this is not precisely good enough: a user might have modified a file and then changed it back to its original form. This would change the time stamp, but still the file should not be treated as modified, for two reasons:

- it would be confusing,
- the file could not be committed anyway, since RCS refuses to accept null updates.

This means that if the time stamps differ, an explicit comparison must be done; if the file turns out to be unmodified after all, the time stamp in the administration file is updated and the file is treated as unmodified. It should be noted that this involves local time only; time stamps are compared only to other time stamps obtained on the same machine.

C. Combining the information

For each file *F*, there are three sources of information:

- the administration entry, for user version number, time stamp and Added/Removed information;
- the RCS-file, for most recent version number and contents; and
- the user file itself, for time stamp and contents.

The entry may be absent, or indicate “Added”, “Removed” or “Normal”; the RCS-file may be absent, or

be the parent or be newer; the user file may be absent, or “Unmodified” or “Modified”. For a more precise definition of these terms, see Table 1. Together there are 4*3*3=36 combinations, ten of which cannot materialize: if there is no time stamp, we cannot distinguish between “Unmodified” and “Modified”, which eliminates six combinations; likewise, if there is no version number, we cannot distinguish between “Parent” and “Newer”, which eliminates four more combinations. The “Added” mark requires the user file to be present, the “Removed” mark requires the user file to be absent; this collapses eight combinations into two error combinations. The resulting twenty combinations are listed in Table 2.

Although some of these situations group together for some commands, they do so differently for different commands. This is illustrated in the listings of the actions for `AE` and `UV`, given in Tables 3 and 4; similar tables can be drawn up for `CM` and `RM`. In programming these commands, it has been tempting to let oneself be guided by common sense and intuition, but cases kept cropping up in which the command behaved oddly. Only when the exhaustive tables were drawn up and the programs coded after them, the errors disappeared.

D. Simultaneous commits

Two or more participants could, at roughly the same moment, decide to commit their versions; RCS guarantees the integrity of single files, but if the project consists of more than one file, mutual exclusion will be required. To this end, the repository has been protected with a simple multi-reader single-writer scheme, using the presence of temporary directories as locks. It should be noted that this is a very temporary locking, operative only as long as the actual commit goes on. It does, however, entail all the usual problems caused by a system crash during a commit.

V. The Use of the System

The system is normally used as follows. The user treats his files as private property, which in fact they are. Before leaving, he starts a delayed job, to run at say 5 AM, that does an update-version followed by a call of `make` (the UNIX utility for doing dependency-controlled compilation, see Feldman[7]), possibly followed by some simple tests. Next morning, he will find a freshly hatched and compiled version.

Use by a single person

Although CVS is normally used by a group of people, its use may also carry benefits for a single user. One is inherent in any version-recording system: the user can retrace his steps. Another is, that based on our information and time stamps, we can easily answer the user’s question: “What did I change since I last committed a version?” This gives the user the opportunity to examine his modifications just before the next commit, and so prevent, e.g., overlooked temporary modifications from being entered into the repository.

Another non-obvious application of CVS for the single user is the use of more than one version (= copy) by one person. This allows one and the same user to work on different aspects of the project without interfering with himself. It is useful, e.g., in the case where the user must make and test a quick correction to a part of the project that is under his responsibility but on which he is not working at the moment. In effect, the user works in split-personality mode.

VI. Distributed Use

If a version of RCS is used that can access files on a remote machine, the repository and the users can all be on different machines. The existence of local copies makes the system pretty indifferent to network partitionings. A temporarily unavailable repository machine causes no more than a slight inconvenience under normal operation: updating the local copy does not work and the user may not have the most recent version; this can be remedied as soon as the network recovers. As explained above, no global time is involved.

VII. Experience and Availability

The system has been in use for almost three years now, to control various projects by various people. It is used to maintain the source programs of the Amsterdam Compiler Kit[8] and of several other projects. It has shown to be able to sustain very long divergence times, as the following experience may illustrate.

One of our students was given a CVS copy of the C compiler files of the Amsterdam Compiler Kit, to which he added features for extensive type checking and heuristic logic checking, as part of his master’s thesis. The student did not use CVS, so no update version command was given during the six months of his thesis project. When finally an update version was done, it showed three conflicts in a total of 92 files, each of which was solved by hand in less than 10 minutes.

No CVS conflicts that were not easy to resolve have been brought to the attention of the author. Such difficult conflicts, though theoretically possible, seem to be very rare in practice.

The system consists of 25 shell-scripts, totalling 43 kbytes, and is available from the author, (dick@cs.vu.nl or ...!mcvax!vu44!dick) or as archive “cvs” from USENET mod.sources.

References

1. S. Mullender and A. S. Tanenbaum, “A Distributed File Service Based on Optimistic Concurrency Control”, in *Proceedings of the 10th Symposium on Operating Systems Principles*, (Orcas Island, Wash., Dec 1-4). ACM, New York, 1985, pp 51-62.
2. L. Svobodova, “File Servers for Network-Based Distributed Systems”, *ACM Computing Surveys*, Vol 16, #4, pp 353-398, Dec 1984.
3. *VAX Technical Summary*, Digital Equipment Corporation, Manyard, Massachusetts, 1980.
4. M. J. Rochkind, “The Source Code Control System”, *IEEE Transactions on Software Engineering*, SE-1, #4, Dec 1975.
5. W. Tichy, “Design, implementation, evaluation of a revision control system”, in *Proceedings of the 6th International Conference on Software Engineering* (Tokyo, Japan, Sept 13-16). ACM, New York, 1982, pp 58-67.
6. H. T. Kung and J. T. Robinson, “On Optimistic Methods for Concurrency Control”, *ACM Transactions on Database Systems*, Vol. 6, #2, June 1981, pp 213-226.
7. S. I. Feldman, “Make − A Program for Maintaining Computer Programs”, *Software − Practice & Experience*, 9, #4, pp 255-266, April 1979.
8. A. S. Tanenbaum, H. van Staveren, E. G. Keizer and J. W. Stevenson, “A Practical Toolkit for Making Portable Compilers”, *Communications of the ACM*, 26, #9, pp 654-660, Sept 1983.

Definitions of terms in tables 2 to 4		
Entry		
	−	The entry is absent
	Added	The file was added to the user copy but not yet committed
	Removed	The file was removed from the user copy but the removal has not yet been committed
	Normal	The file belongs to both the user copy and the repository
RCS file		
	Any	The state of the RCS file is immaterial
	−	The RCS file is absent
	Present	The RCS file exists
	Parent	The version numbers of the RCS file and the user file are the same
	Newer	The version number of the RCS file is higher than that of the user file
User file		
	−	The user file is absent
	Present	The user file exists
	Unmodified	The user file time stamp is equal to the entry time stamp
	Modified	The user file time stamp differs from the entry time stamp

Table 1

Possible states			
Entry	*RCS file*	*User file*	*Explanation*
–	–	–	Error: nothing is known about the file
–	–	Present	The file exists, but is unknown to RCS or the repository
–	Present	–	There is an RCS-file only
–	Present	Present	Warning: the CVS entry is missing
Added	Any	–	Error: the file to be added is missing
Added	–	Present	The file is to be added
Added	Present	Present	Conflict: the file to be added has been added simultaneously and independently by a second party
Removed	–	–	Warning: the file was removed simultaneously and independently by two parties
Removed	Parent	–	The file is to be removed
Removed	Newer	–	Conflict: the file is to be removed, but was modified by a second party
Removed	Any	Present	Error: the user file should be removed and is still there
Normal	–	–	Warning: spurious entry
Normal	–	Unmodified	There is no RCS-file
Normal	–	Modified	There is no RCS-file
Normal	Parent	–	Warning: the user file is missing
Normal	Parent	Unmodified	Normal situation, all quiet
Normal	Parent	Modified	Normal situation, the file is modified
Normal	Newer	–	Warning: the user file is missing
Normal	Newer	Unmodified	The repository is more recent
Normal	Newer	Modified	The copies have diverged

Table 2

Actions of AE (Add Entry)			
Entry	*RCS file*	*User file*	*Action*
–	–	–	Error: nothing is known about the file
–	–	Present	OK: build an entry for the user file
–	Present	Any	Error: the user file was added independently by a second party
Added	Any	Any	Error: the user file has already been entered
Removed	–	–	Error: CVS cannot resurrect the user file, the RCS file was removed by a second party
Removed	Present	–	OK: resurrect the user file
Removed	Any	Present	Error: the user file should be removed and is still there
Normal	Any	Any	Error: the user file already exists

Table 3

Actions of UV (Update Version)			
Entry	*RCS file*	*User file*	*Action*
–	–	–	Error: nothing is known about the file
–	–	Present	Error: the user should use AE to create an entry for this user file
–	Present	–	OK: check out a new version
–	Present	Present	Conflict: the user should move away this file; it is in the way
Added	Any	–	Error: the new-born user file has disappeared
Added	–	Present	OK: do nothing
Added	Present	Present	Conflict: a file of the same name has been created independently by a second user
Removed	–	–	Warning: this file was removed independently by a second party
Removed	Parent	–	OK: do nothing
Removed	Newer	–	Conflict: this removed file was modified independently by a second party
Removed	Any	Present	Error: the user file should be removed and is still there
Normal	–	–	Warning: this file is not pertinent
Normal	–	Unmodified	Notice: this file is no longer in the repository
Normal	–	Modified	Conflict: this modified file was removed independently by a second party
Normal	Parent	–	Warning: this file was lost
Normal	Parent	Unmodified	OK: do nothing
Normal	Parent	Modified	OK: (check if modification is genuine)
Normal	Newer	–	Warning: this file was lost
Normal	Newer	Unmodified	OK: check out a new version
Normal	Newer	Modified	OK: merge in a new version

Table 4