



IPT Group, IT Division, CERN

SPIDER

CVS Getting Started

Version: 1.0
Issue: 7
Edition: English
Status: Reviewed
ID: SPIDER-CVSTUT-0001-DOC
Date: 25 May 1999
Other Ref: CERN-UCO/1999/206



European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

This document has been prepared using the Software Documentation Layout Templates that have been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics). For more information, go to <http://framemaker.cern.ch/>.



Abstract

This document describes a system used for version control called CVS (Concurrent Versioning System). Using CVS the various versions of files, the history and the differences between them can be managed. It also supports developers working simultaneously on the same file.

Since it is a “Getting Started” it describes only the basic commands and features of CVS. For a more detailed description of CVS see the links in section 10 .

The “Getting Started” is intended for developers who have no experience with CVS and would like to start working with it. Nevertheless some knowledge of UNIX (basic commands) is presumed.



Table of Contents

Abstract	iii
Table of Contentsv
1 Introduction7
1.1 Basic description of CVS7
1.2 History9
2 How to get started	10
2.1 Setting up the repository	10
2.1.1 Specifying the repository directory	10
2.1.2 Initializing the repository	10
2.1.3 Importing sources to your repository	11
3 A basic session under CVS	12
3.1 Checking files in and out	12
3.2 Defining a module	14
3.3 Adding and removing files	16
4 CVS used by many users	17
5 Viewing differences	22
6 Showing the history of files	23
6.1 CVS history	23
6.2 CVS log	24
7 Being informed on who else is working on a file	25
8 Tagging files	26
9 Branches	28
10 Useful Links	29
10.1 Where to find CVS	29
10.2 Documentation and third party utilities	30
11 Future Work	30



1 Introduction

CVS (Concurrent Versioning System) is a system used for version control. Using CVS the various versions of files, the history and the differences between them can be managed. It also supports developers working simultaneously on the same file.

This “Getting Started” describes only the basic commands and features of CVS. For a more detailed description of CVS see the links in section 10 .

The “Getting Started” is intended for developers who have no experience with CVS and would like to start working with it. Nevertheless some knowledge of UNIX (basic commands) is presumed.

1.1 Basic description of CVS

CVS (Concurrent Versioning System) is a version control system. Using CVS the different versions of files can be managed. CVS can be used by one user but also by many users working simultaneously on a file.

CVS used by one user

The benefit of CVS for a single user is that he can record the history of various versions of a file (or group of files). If you have ever developed software the following problem may not be unknown to you:

Real life example 1: In a rush to add a new rotation functionality for his brand new visualization package developer Jerry worked all the night since the new release should be delivered the next day. But finally, when trying to compile his source code he is unable to do so. He searches for hours to find the bug but cannot find it. He decides to forget about the new rotation functionality. “Delivering a working visualization package without the new functionality is better than delivering nothing!” he thinks. But where are the sources which he was able to compile just two days ago. Jerry has to admit that he has overwritten them. But luckily there is some backup around. Nevertheless the last backup was made a week ago, i.e. the newest features (i.e. the new geometries he liked so much!) are not included. So, finally Jerry had neither a rotation functionality nor any new geometry to deliver!

If he had used CVS it would have been no problem to get the older version which includes his geometries and Jerry’s nightly programming session would have ended happily.

CVS used by many users

When CVS is used by many people it shows its real strength: when several developers are working (i.e. editing) concurrently on the same file(s) they often over write the changes of the others. CVS can help to tackle this problem.

Real life example 2: Jerry asks his friend Tim to help him developing his visulatization package since he does not see any chance to finish it in time. Tim has lots of free time in the moment and he is happy to support Jerry. Jerry opens the file defining graphics elements

graphic.java for including his geometries. Jerry programs his new geometries in a nightly effort. Early in the morning Jerry saves the file and at home falls asleep - exhausted but happy as he is - immediately. This night (or day) he is dreaming of his nice new geometriess. An hour later Tim comes to work and opens the graphic.java file since he does not like the background colour. By saving the file he deletes - unfortunately but undeliberately- all the changes and thus all the geometries of which Jerry is dreaming in the same moment. Jerry and Tim are no friends any more since that sad day!

Unlike other systems which use *file locking* for solving this problem CVS uses so called *unreserved checkouts*.

File locking means that only one user can work simultaneously on one file. The file is stored in a shared directory to which all developers have access. The developers Tim and Jerry work directly on files contained in this directory (see Figure 1). To avoid the above mentioned problem of over writing, a file is locked when opened by one developer for editing. That means other developers can not open the file at the same time for editing it.

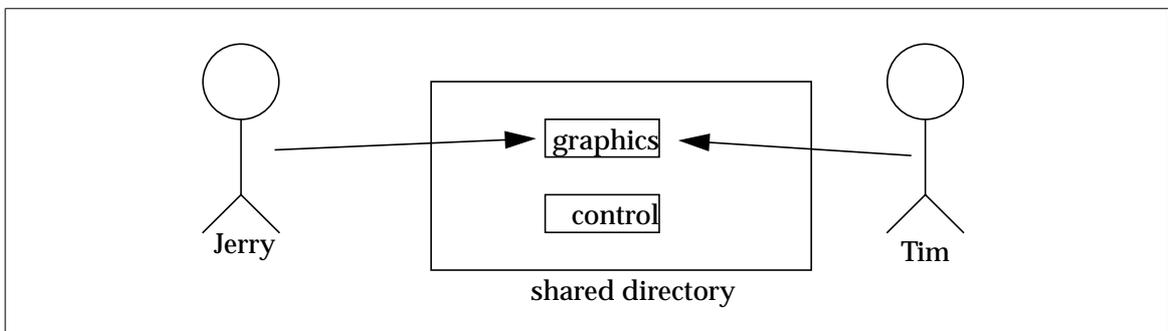


Figure 1 Jerry and Tim are working on the same file in the shared directory.

CVS by applying *unreserved checkouts* uses another approach: There is a central working area to which everybody has access which is called *repository*. But the repository is more than a simple shared directory: It is a kind of a database containing all versions of files put in the repository.

Actually, the different versions of one file are not saved as different files in the repository for reasons of disk consumption. CVS stores only the most recent version together with all differences (deltas) to older versions.

Everybody works on a local copy of the file instead of working directly on the files contained in the directory. The generation of a working file is done by copying the file stored in the

repository to some working area. This activity is called *checking out* (illustrated by the dashed line in Figure 2).

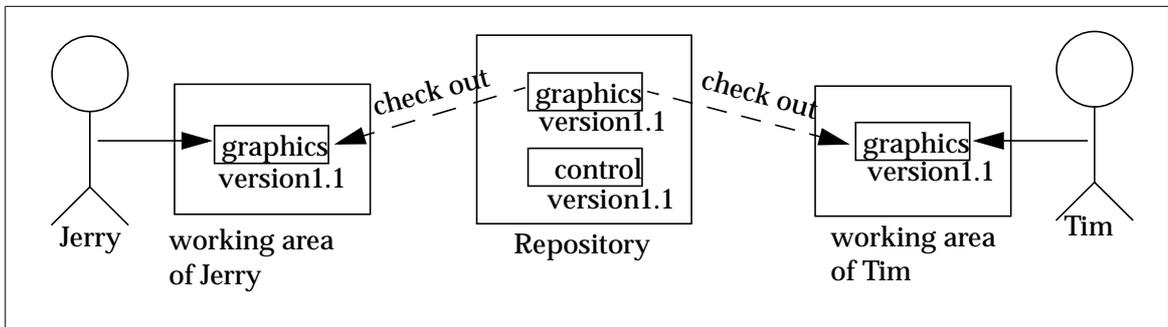


Figure 2 Jerry and Tim are working on their local files checked out of the repository to their own working areas.

After a developer (e.g. Jerry) has done some changes to the local copy, the file can be put back (or *committed*) to the repository¹. The commit does **not** overwrite the older version of the file but instead it generates automatically a new version of the file. Now Tim can *update* his local copy to integrate the changes done by Jerry. This integration process (or *merge*) is done by CVS if there is no conflict between Jerry's and Tim's changes. We will see later how CVS can help to resolve such a conflict.

When a developer is finished with his work and has committed it back to the repository he indicates that the file is not longer in use by *releasing* the local file(s). He does not need to keep the local copies since he can get it back from the repository by checking them out.

1.2 History

CVS was designed and coded by Brian Berliner in 1989.

Nevertheless some older work shall be mentioned here since CVS is based on it: Dick Grune wrote some shell scripts for concurrent versioning of files which he in posted to comp.sources.unix in 1986.

Another contributor to CVS is Jeff Polk who helped Brian Berliner with the design of the CVS module.

CVS is based on RCS (Revision Control System). The different versions of files are stored in the same way as with RCS. What CVS offers in addition to RCS is better support for being used by more than one user. Several users can work concurrently on the same file. RCS only allowed one user at a time to edit the same file.

1. This is what Jerry should have done more frequently in the first example.

2 How to get started

After all this trouble Jerry wants to use CVS to put his visualization project under configuration control. In this chapter you see how he can do this.

2.1 Setting up the repository

Before being able to work with CVS you have to set up the repository. This is done by

- Specifying the repository directory
- Initializing the repository and
- Importing sources to your repository

2.1.1 Specifying the repository directory

One has to tell CVS in which directory it can create the repository by setting the `CVSROOT` environment variable¹.

Since Jerry wanted the repository to be in the work directory `/work` and to be named `repository` he set `CVSROOT` to `/work/repository`.

If Jerry runs `csh` this is done with:

```
setenv CVSROOT /work/repository
```

Running `sh` this is done with:

```
CVSROOT=/work/repository  
export CVSROOT
```

2.1.2 Initializing the repository

This is done with the `cvs init` command:

```
~jerry> cvs init2
```

CVS creates the repository including a subdirectory `CVSROOT` containing administrative files. Figure 3 shows the resulting repository structure that is created by CVS.

1. Environment variables are used by UNIX to specify various system attributes.
2. Throughout the document all commands that have been typed by a user are in `Courier` font with the UNIX prompt preceding the command. The output generated by CVS is in `Courier` font.

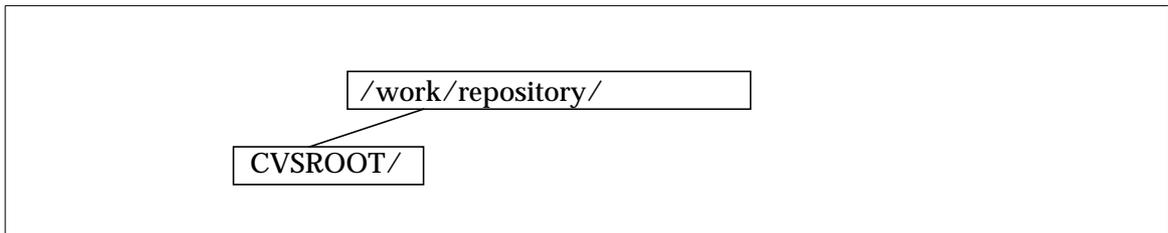


Figure 3 The initialized repository structure.

2.1.3 Importing sources to your repository

If one already has a directory structure which to put under CVS control this can be done by using the `import` command.

Suppose the files Jerry wants to put under CVS control are in directory `~/visualizationproject` which contains already two files `graphics.c` and `control.c` (see Figure 4). If he wants them to appear in the repository under directory `projects/visualization`, he can do this:

```
~jerry> cd ~/visualizationproject
~jerry/visualizationproject> cvs import -m "the visualization project"
projects/visualization vendor_release_1_4 release_3_4
```

This command imports the files of the current directory into a subdirectory `projects/visualization` in the repository.

The `-m` flag is used for a log message which is stored in the repository with the imported files. We will see later in section 6 how they can be used.

Don't worry about the last two flags for the moment¹.

The output of CVS after having issued this command is the following:

```
N projects/visualization/graphics.c
N projects/visualization/control.c

No conflicts created by this import
```

The `N` stands for **new**. CVS realized that `graphics.c` and `control.c` are new files being added to the `/work/repository` repository.

1. These are necessary for the import command and specify the vendor respectively release tag.

The resulting repository can be seen in Figure 4.

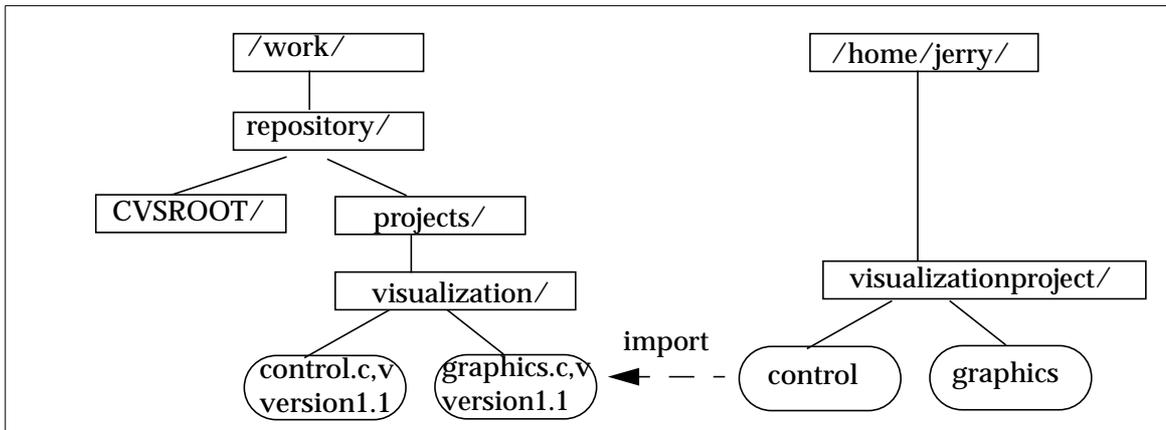


Figure 4 The files in the repository got a suffix `.v` which is generated automatically by CVS. These files contain the most recent version and all differences (deltas) to older versions.

CVS created a first version for each of the files. The initial version number is 1.1.

The local files under `~jerry/visualizationproject` can be deleted now since they are stored in the CVS repository.

Nevertheless do not forget to make a back up of the CVS repository. CVS does not relieve from the need for a back up since the loss of the file system will result in the loss of the CVS repository.

3 A basic session under CVS

3.1 Checking files in and out

Usually one single user (like Jerry) is working in the following way with CVS:

He changes to his working area:

```
~jerry> cd /home/jerry
```

He uses the `checkout` command to get a local copy of a directory residing in the repository

```
~jerry> cvs checkout projects/visualization  
cvs checkout: Updating projects/visualization  
U projects/visualization/control.c  
U projects/visualization/graphics.c
```

The `U` stands for updated. CVS is telling Jerry that new versions of `graphics.c` and `control.c` have been checked out of the repository under his working directory, ready to be used.

Now Jerry has a subdirectory `projects/visualization` in his working directory `/home/jerry` (see Figure 5).

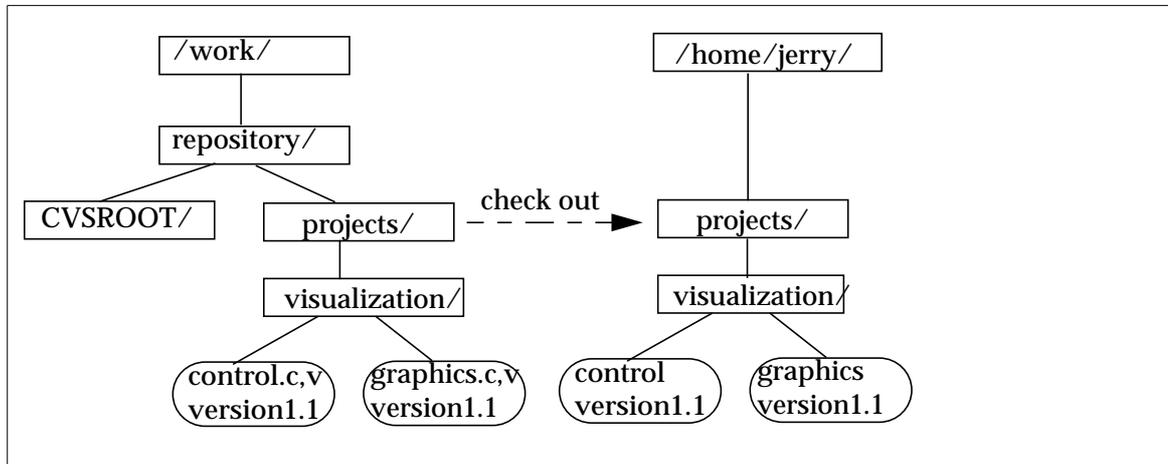


Figure 5 Checking out a directory.

He changes to subdirectory `projects/visualization` to work with the new checked out files.

```
~jerry> cd projects/visualization
```

and edits the file `graphics.c` using e.g. `vi` to finally add his new geometries.

```
~jerry/projects/visualization> vi graphics.c
```

When he thinks that his changes to the file are finished for the moment he puts the `graphics.c` file back to the repository by committing the current working directory. But first he changes to the directory where he checked out his project.

```
~jerry/projects/visualization> cd ../../
```

```
~jerry> cvs commit -m "new geometries added" projects
```

```
cvs commit: Examining projects
cvs commit: Examining projects/visualization
cvs commit: Committing projects/visualization
Checking in projects/visualization/graphics.c;
/work/repository/projects/visualization/graphics.c,v <-- graphics.c
new revision: 1.2; previous revision: 1.1
done
```

CVS recursively examines all subdirectories of `project` and commits to the repository only files which have been changed. In the output you see that CVS only reports on the `graphics.c` file which is the only file Jerry changed.

The `-m` flag, as for the input used, allows Jerry to write a comment on the reason for his change¹.

1. If the `-m` flag is not used `cvs` opens the preferred editor for Jerry to type a multi-line log text.

CVS automatically creates a new version of the changed file `graphics.c` file (see Figure 6). Do not conclude from the figure that the old version 1.1 of file `graphics.c` is deleted. The version number shows only the most recent version 1.2. The old version 1.1 is stored in the same file as described in section 1.1 .

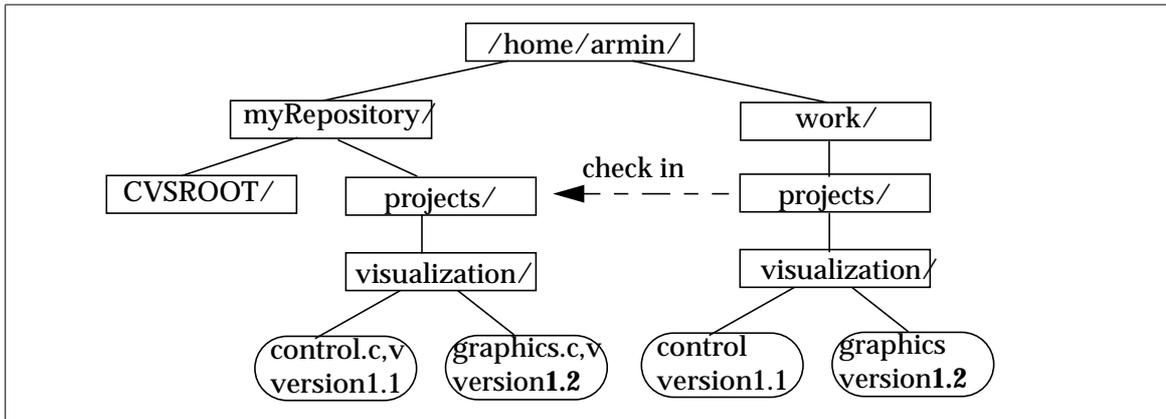


Figure 6 Checking in a directory.

Now the local copy in the working area can be released and removed:

```
~jerry> cvs release -d projects

You have [0] altered files in this repository.
Are you sure you want to release module `projects': y
```

The `-d` flag results in the removal of the `projects` subdirectory.

Thus the `cvs release -d` command is similar to the UNIX remove command (`rm -r`). The difference is that `release` checks before removing a file whether all changes have been committed back to the repository.

If Jerry had forgotten to commit, the `release` command would have resulted in the following message:

```
~jerry> cvs release -d projects

M visualization/graphics.c
You have [1] altered files in this repository.
Are you sure you want to release (and delete) module `projects': y
visualization/graphics.c has been modified; revert changes? y
```

3.2 Defining a module

To group together a set of related files and directories CVS offers *modules*. This can be convenient for e.g. checking out files and directories by its module name instead of its individual file and directory names.

Jerry wants to create a module for his visualization project. Therefore he has to edit the file `modules` which is in subdirectory `CVSROOT/` in the repository. He issues

```
~jerry> cvs checkout CVSROOT/modules
U CVSROOT/modules
~jerry> cd CVSROOT
~jerry/CVSROOT> vi modules
```

Jerry adds a new line that defines his module naming it `vis`:

```
vis projects/visualization
```

For the syntax of the `modules` file see the CVS reference manual which can be found in the links given in section 10 .

Now he commits the changes to the `modules` file:

```
~jerry/CVSROOT> cvs commit -m "added the module vis" modules
Checking in modules;
/work/repository/CVSROOT/modules,v <-- modules
new revision: 1.2; previous revision: 1.1
done
cvs commit: Rebuilding administrative file database
```

Finally he releases the `CVSROOT` directory:

```
~jerry/CVSROOT> cd ..
~jerry> cvs release -d CVSROOT
You have [0] altered files in this repository.
Are you sure you want to release (and delete) module `CVSROOT': y
```

Now Jerry can check out the visualization project by using its module name `vis` instead of its directory path `projects/visualization`:

```
~jerry> cvs checkout vis
cvs checkout: Updating vis
U vis/control.c
U vis/graphics.c
```

Notice that the working files of the visualization project are now in subdirectory `vis` not in subdirectory `projects/visualization`.

```
~jerry> cd vis
```

In this example Jerry used the module name as an alias. But there are more applications of modules. These are described in section *The modules file* in the Appendix *CVS Reference Manual for Administrative Files* of Cederquists "Version Management with CVS" document. This document can be found in the 1. link which is given in section 10.2 .

3.3 Adding and removing files

With `cv`s `add` files can be scheduled for being put under version control.

Jerry needs two new files `io.c` and `mem.c` in his visualization project. Therefore he creates the files using e.g. `vi` in his current working directory:

```
~jerry/vis> vi io.c
~jerry/vis> vi mem.c
```

Now he can add the files to the visualization project:

```
~jerry/vis> cvs add io.c mem.c
cvs add: scheduling file `io.c' for addition
cvs add: scheduling file `mem.c' for addition
cvs add: use 'cvs commit' to add these file permanently
```

Now the files have to be committed to actually check them into the repository. Other developers (like Tim) can not see the files until this step is performed.

```
~jerry/vis> cvs commit -m "files io.c and mem.c added"
RCS file: /work/repository/projects/visualization/io.c,v
done
Checking in io.c;
/work/repository/projects/visualization/io.c,v <-- io.c
initial revision: 1.1
done
RCS file: /work/repository/projects/visualization/mem.c,v
done
Checking in mem.c;
/work/repository/projects/visualization/mem.c,v <-- mem.c
initial revision: 1.1
done
```

To remove a file from a module `cv`s `remove` can be used. Files deleted with this command can be retrieved later if necessary from the repository.

Jerry wants to remove the `mem.c` file from his visualization module. To do this he first has to make sure that there are no uncommitted modifications to the file. Then he removes the file from his working directory:

```
~jerry/vis> rm mem.c
```

Now he can delete the file from his module `vis`:

```
~jerry/vis> cvs remove mem.c
cvs remove: scheduling `mem.c' for removal
cvs remove: use 'cvs commit' to remove this file permanently
~jerry/vis> cvs commit
```

4 CVS used by many users

When more than one users are working with CVS things get a little bit more complicated. This is due to the fact that other users can edit a file on which you are working currently. This problem has been addressed briefly in section 1.1 .

In this section you will see in detail how Jerry and Tim can use CVS to avoid overwriting their files when working simultaneously on them.

Figure 7 illustrates a scenario where Tim and Jerry are working on the same file. It shows vertically the sequence of actions (i.e. cvs commands or editing activities) performed by Jerry and Tim. The picture is divided horizontally into the working areas of the developers and the repository. Only the version number of file `graphics.c` is shown.

Step 1 Tim checks out the file `graphics.c` to add the new rotation functionality. He does this as described in section 3 by changing to his working area and using `cvs checkout`.

```
~tim> cd /home/tim
~tim> cvs checkout projects/visualization
cvs checkout: Updating projects/visualization
U projects/visualization/control.c
U projects/visualization/graphics.c
```

Step2 Jerry checks out the `graphics.c` file in the same way to his working area.

```
~jerry> cd /home/jerry
~jerry> cvs checkout projects/visualization
cvs checkout: Updating projects/visualization
U projects/visualization/control.c
U projects/visualization/graphics.c
```

Step3 Tim edits the file and includes some lines of code (e.g. to change the rotation functionality).

```
~tim> cd projects/visualization
~tim/projects/visualization> vi graphics.c
```

Step 4 When Tim is content with the new rotation functionality he commits the file back to the repository using `cvs commit`.

```
~tim> cd ../../
~tim> cvs commit -m "new geometries added"
cvs commit: Examining projects
cvs commit: Examining projects/visualization
cvs commit: Committing projects/visualization
Checking in projects/visualization/graphics.c;
/work/repository/projects/visualization/graphics.c,v <-- graphics.c
new revision: 1.2; previous revision: 1.1
```

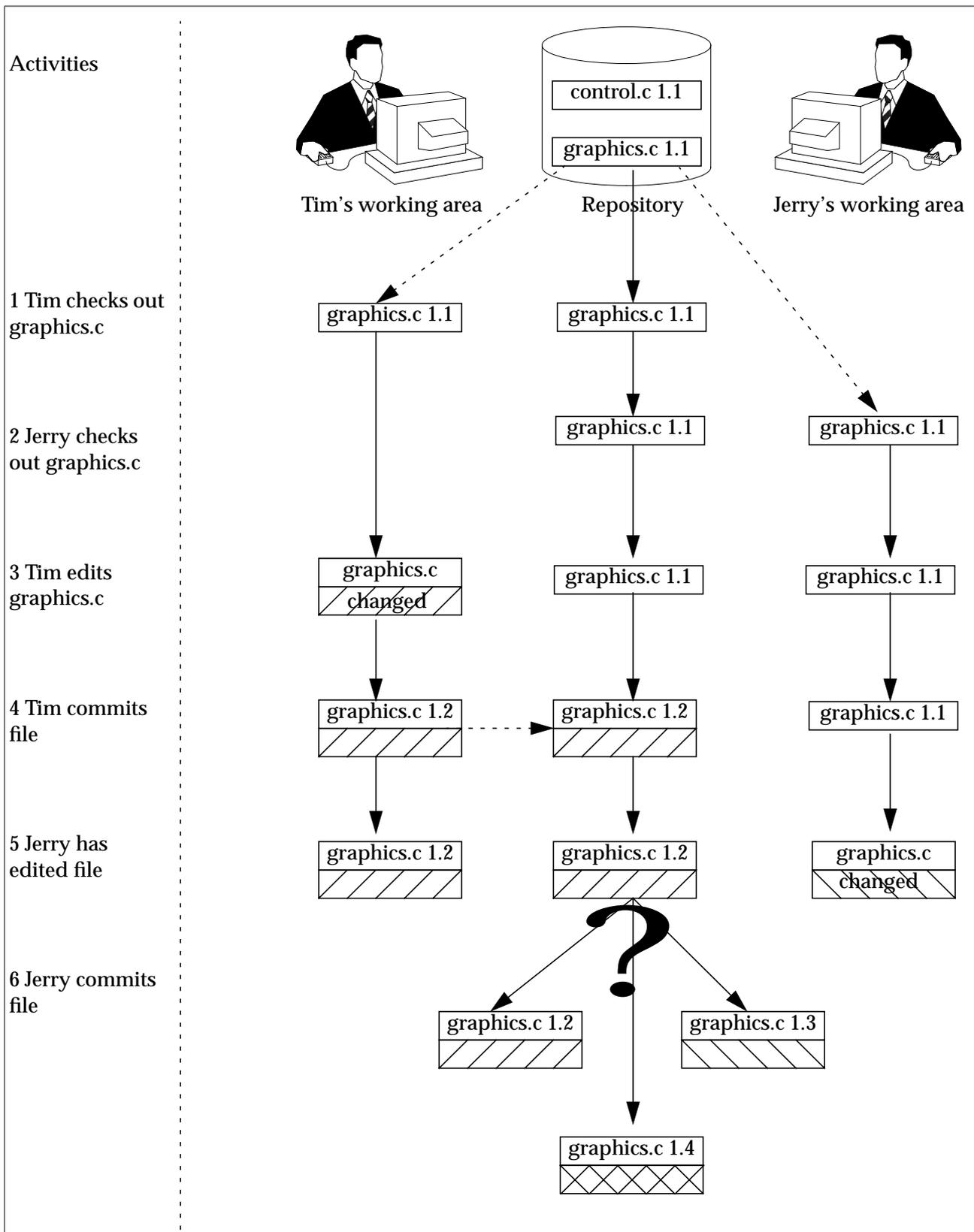


Figure 7 More than one user are working on the same file.
done

Step 5 In the meantime Jerry has edited the file. He has changed the background colour.

```
~jerry> cd projects/visualization  
~jerry/projects/visualization> vi graphics.c
```

Step 6 When Jerry commits the file back to the repository he has a problem. There is already a newer version in the repository (version 1.2 which contains the changes made by Tim) than the version on which Jerry's local file is changed. What should CVS do know? Generate a new version with Jerry's changes or merge the changes made by the two developers?

To make it short: CVS does not allow Jerry to commit the file since there exists a newer revision than the revision on which his modification is based.

The output message from CVS Jerry gets after having issued the `commit` command is the following:

```
~jerry/projects/visualization> cd ../../  
~jerry> cvs commit -m "background color changed"  
  
cvs commit: Examining projects  
cvs commit: Examining projects/visualization  
cvs commit: Up-to-date check failed for `projects/visualization/graphics.c'  
cvs [commit aborted]: correct above errors first!
```

The commit failed.

Step 7 What Jerry has to do before being able to commit the file is updating his local copy with the current version to include the modifications made by Tim:

```
~jerry> cvs update project
```

This update can have two effects:

1. Either CVS is able to merge automatically the changes (step 8a in Figure 8)

2. or, this merge has to be done manually because of a conflict (steps 8b and 9b in Figure 9):

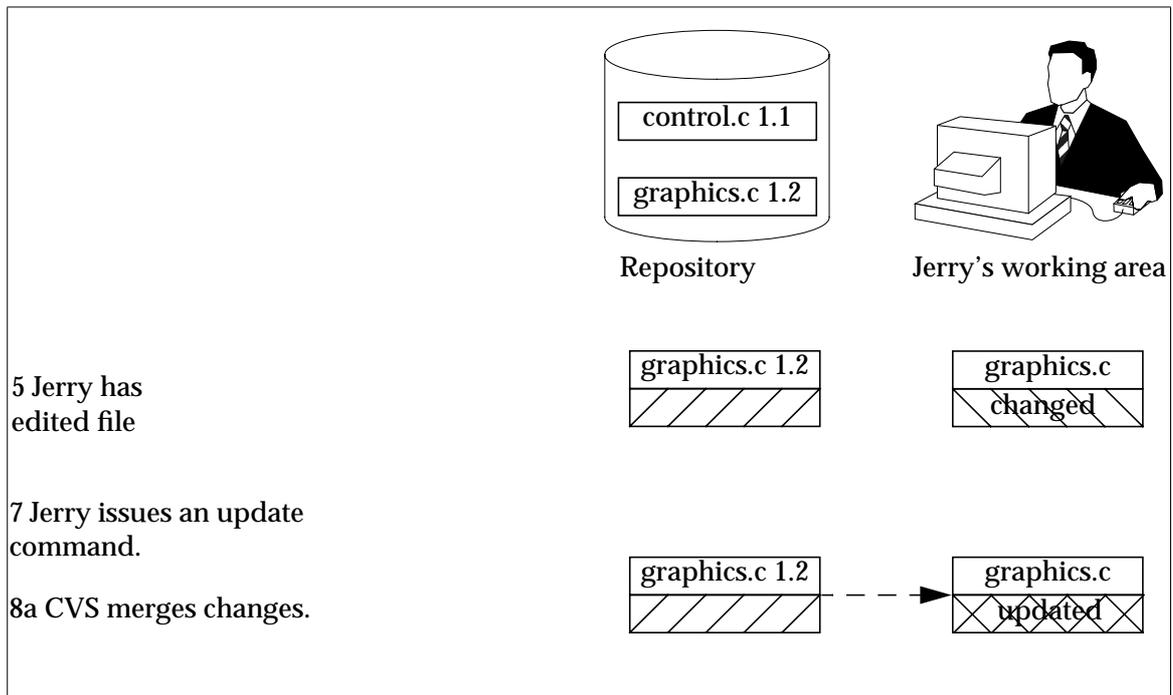


Figure 8 When more than one user have edited the same file an update may be necessary. Only Jerry is shown as Tim does not perform any other actions for the moment.

Step 8a CVS is able to merge all changes made by Tim to the `graphics.c` file into the working file of Jerry if the changes that Tim made are in a different part than the changes Jerry made.

```
~jerry> cvs update project  
  
RCS file: /work/repository/projects/visualization/graphics.c,v  
retrieving revision 1.1  
retrieving revision 1.2  
Merging differences between 1.1 and 1.2 into graphics.c  
M graphics.c
```

(Go to step 10).

Step 8b But if Jerry and Tim have changed the same lines of the file there is a conflict which cannot be solved by CVS. Thus CVS prints a warning and the resulting file in Jerry's working area contains both versions of the lines that overlap, delimited by special markers (see Figure 9).

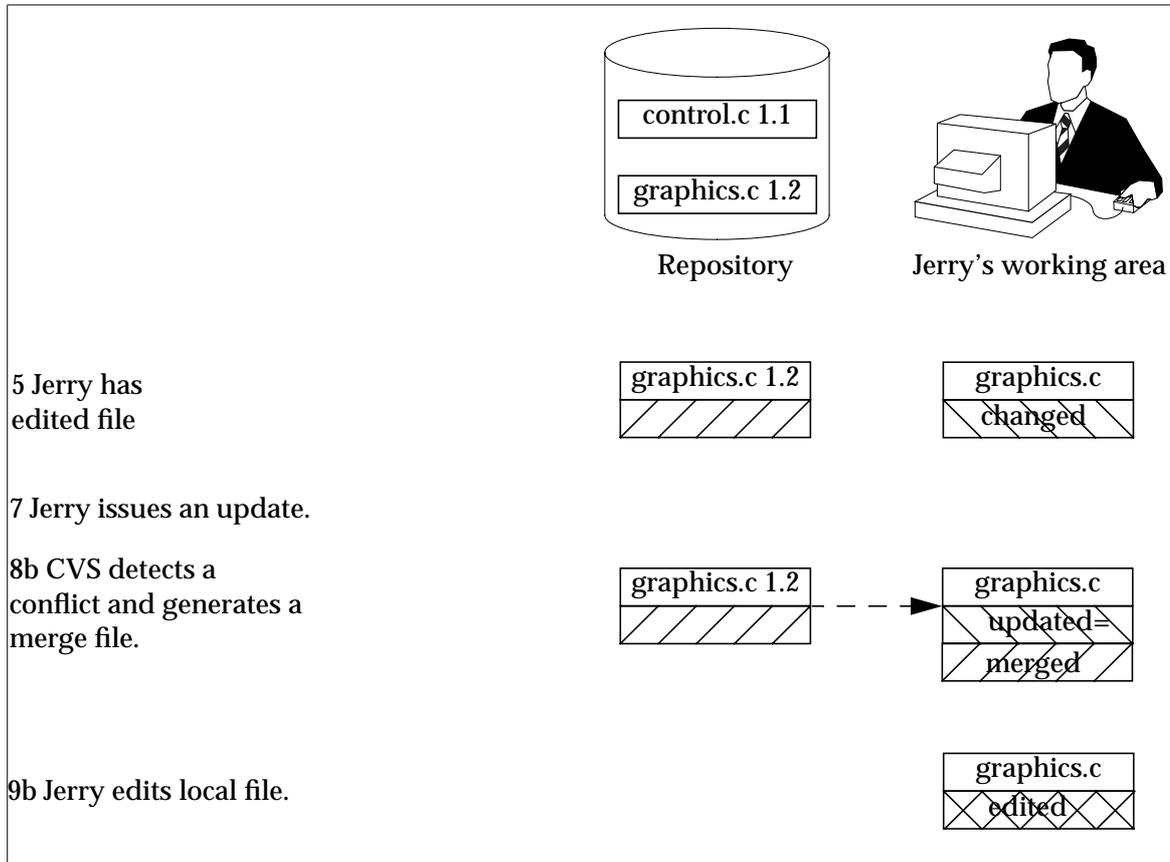


Figure 9 When more than one user are working on the same file a conflict can occur.

```
~jerry> cvs update project
```

This is the output when CVS detects a conflict:

```
RCS file: /work/repository/projects/visualization/graphics.c,v
retrieving revision 1.1
retrieving revision 1.2
Merging differences between 1.1 and 1.2 into graphics.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in graphics.c
C graphics.c
```

Step 9b Jerry has to resolve this *overlap* or *conflict* by editing the file again.

Here is the file resulting from the merge:

```
~jerry> more graphics.c

unchanged lines
.....
<<<<<<< graphics.c
# Jerry's added or changed lines
```

```
....  
=====  
# Tim's added or changed lines  
....  
>>>>>> 1.2  
unchanged lines  
....
```

The lines which differ and which were changed by Jerry are shown between the <<<<<< graphics.c delimiter and the ===== delimiter. This is the current state of the file in the working copy.

The lines which differ and which were changed by Tim are shown between the ===== delimiter and the >>>>>> 1.2 delimiter. This is the current state of the file in the repository (revision 1.2).

Jerry has to choose which of the 2 sections to keep, then he makes the changes and removes the other section.

Step 10 After having resolved the conflict (8b and 9b) or after an update without a conflict (8a) Jerry can commit his local copy to the repository. CVS generates a new version 1.3.

```
~jerry> cvs commit -m "background color changed"  
  
cvs commit: Examining projects  
cvs commit: Examining projects/visualization  
cvs commit: Committing projects/visualization  
Checking in projects/visualization/graphics.c;  
/work/repository/projects/visualization/graphics.c,v <-- graphics.c  
new revision: 1.3; previous revision: 1.2  
done
```

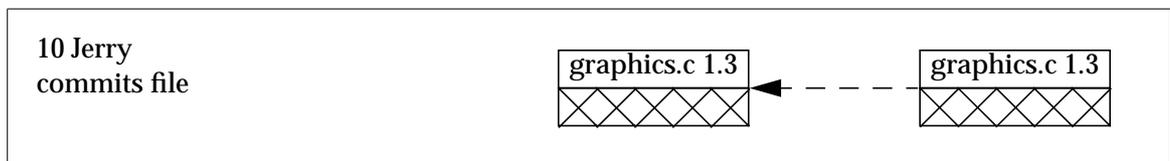


Figure 10 After the file has been updated or the conflict released it can be committed.

5 Viewing differences

Using `cvs diff` the differences between the local working copy and the file in the repository can be viewed.

Assume that Jerry after his commit, which resulted in the creation of revision 1.3 has made some changes to his local `graphics.c` file. Some hours later he does not know any more which changes he made. He can check this by using the `cvs diff` command:

```
~jerry> cvs diff projects/visualization/graphics.c
```

CVS uses the UNIX diff utility to compare the local copy of the file in Jerry's work directory with the most recent revision in the repository file (here 1.3) and outputs the result:

```
=====
RCS file: /work/repository/projects/visualization/graphics.c,v
retrieving revision 1.3
diff -r1.3 graphics.c
0a1
> # I added this line
> ....
```

In fact, the diff command allows to view the difference between any two revisions.

If you remember the scenario described in section 3 Tim had made a commit and created the version 1.2 of `graphics.c`. Since he did not perform a release yet, he still has in his working directory this version of `graphics.c`. Now Tim is notified that a new version 1.3 has been created by a commit issued by Jerry. Tim is interested in the changes made by Jerry. Thus he issues a `cv diff` command:

```
~tim> cvs diff -r 1.3 projects/visualization/graphics.c
```

This command compares the current working file of Tim (revision 1.2) with revision 1.3.

The output could be the following:

```
=====
RCS file: /work/repository/projects/visualization/graphics.c,v
retrieving revision 1.3
retrieving revision 1.2
diff -r1.3 -r1.2
3c3
< # Jerry's added or changed lines
<....
---
> # Tim's added or changed lines
>....
>
```

As you can see there is a difference in the same set of lines. This is what resulted in the conflict in step 8b in the scenario in section 3 .

6 Showing the history of files

6.1 CVS history

With the `cv history` command information about the CVS repository can be found. You can see who has issued which command when. The output of the history command can be adapted to the user-specific wishes showing only specific commands (like checkouts, commits etc.).

The history output for the example in section 4 (Figure 7) looks like the following:

```
~/tim/work/projects/visualization> cvs history -e  
  
O 04/13 08:48 +0000 tim projects/visualization =projects/visualization= ~/tim  
O 04/13 08:49 +0000 jerry projects/visualization =projects/visualization=  
~/jerry  
M 04/13 08:49 +0000 tim projects/visualization =projects/visualization= ~/tim  
G 04/13 08:53 +0000 jerry projects/visualization =projects/visualization=  
~/jerry  
M 04/13 08:53 +0000 jerry projects/visualization =projects/visualization=  
~/jerry
```

The single letters in the first line have the following meaning:

O	Checkout
M	A file was committed and modified
G	A merge was necessary but a collision was detected

6.2 CVS log

With the `cvs log` command the log messages entered for each commit can be looked through.

The log output for the example in section 4 could look like the following:

```
~/tim/work/projects/visualization> cvs log graph.c  
  
cvs log: Logging .  
  
RCS file: /work/repository/projects/visualization/graphics.c,v  
Working file: graphics.c  
head: 1.3  
branch:  
locks: strict  
access list:  
symbolic names:  
keyword substitution: kv  
total revisions: 3;      selected revisions: 3  
description:  
-----  
revision 1.3  
date: 1999/04/14 14:51:08;  author: jerry;  state: Exp;  lines: +1 -0  
background color changed  
-----  
revision 1.2  
date: 1999/04/14 12:51:08;  author: tim;  state: Exp;  lines: +1 -0  
new geometries added  
-----  
revision 1.1  
date: 1999/04/12 00:51:08;  author: jerry;  state: Exp;  lines: +1 -0  
initial revision  
-----
```

For a description of the log output see the links in section 10 .



7 Being informed on who else is working on a file

Sometimes it is interesting to know who else is working on a file. This information could be useful if developers would like to coordinate their work.

Jerry wants to know when Tim is working on the `graphics.c` file. Therefore he issues the following command after having checked out the visualization package files:

```
~/jerry/work/projects/visualization> cvs watch on graphics.c
```

Looking at the content of the local working directory (`projects/visualization`) he can see that the `graphics.c` file is now read only.

```
~/jerry/work/projects/visualization> ls -l
total 6
drwxr-xr-x  2 jerry  jp          2048 Apr 13 11:18 CVS/
-rw-r--r--  1 jerry  jp           0 Apr 12 11:56 control.c
-r--r--r--  1 jerry  jp          131 Apr 13 10:02 graphics.c
```

When Tim is checking out the `graphics.c` file it is also in read only mode. To edit the file he must first issue the `cvs edit` command:

```
~/tim/work/projects/visualization> cvs edit graphics.c
```

Now the `graphics.c` file is in read-write mode:

```
~/tim/work/projects/visualization> ls -l
total 6
drwxr-xr-x  3 tim  jp          2048 Apr 13 11:22 CVS/
-rw-r--r--  1 tim  jp           0 Apr 12 11:56 control.c
-rw-r--r--  1 tim  jp          131 Apr 13 10:02 graphics.c
```

One could object that you can do this with a simple UNIX `chmod` command. But the advantage of using `cvs watch on` is that Jerry can check whether Tim is working on `graphics.c` by issuing the following command:

```
~/jerry/work/projects/visualization> cvs editors graphics.c
```

He gets a list of all developers who have issued a `cvs edit` command on `graphics.c`.

```
graphics.c tim Tue Apr 13 09:22:42 1999 GMT ipts02
~/tim/work/projects/visualization
```

This information can also be sent automatically (e.g. by email) whenever somebody issues the `cvs edit` command. CVS provides a so called notification mechanism with the `cvs watch add` command.

The notification mechanism is specified in the `notify` administrative file.

For more details see section *Mechanisms to track who is editing files* in Cederquists "Version Management with CVS" document. This document can be found in the [1. link](#) which is given in section 10.2 .

8 Tagging files

After having added the new geometries and the new rotation functionality Jerry thinks he can deliver a new release of his visualization package. The new release is given the name `release-1-2`. The files which go into the new release are:

graphics 1.3

control 1.1

io 1.2

You can see this in Figure 11.

The revision numbers of the files have nothing to do with the release name.

To group together all the files which belong to one release CVS offers *release tags*. The files are tagged with a symbolic name. Afterwards, the files can be referenced by the tag. Thus, all the files which belong to one release can be referenced (e.g. checked out) easily.

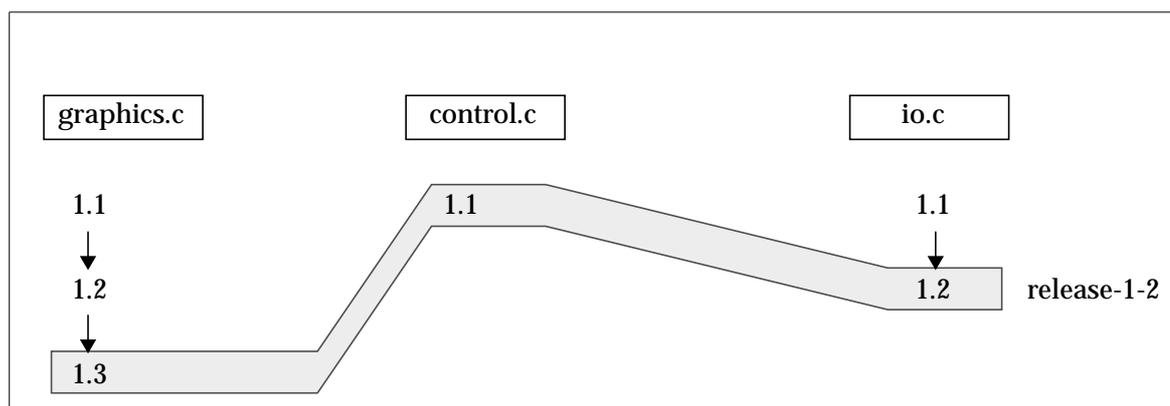


Figure 11 The repository contains the files `graphics.c`, `control.c` and `io.c` in different revisions. The most recent revision of each file is the revision on the bottom. `release-1-2` consists of the most recent revisions.

Assume that Jerry's working directory contains the above revisions. To tag all revisions of files which are currently in the working directory Jerry issues the following command

```
~/jerry/work/projects/visualization> cvs tag release-1-2 .
```

This command tells CVS that when we talk about `release-1-2` we mean `graphics.c 1.3`, `control.c 1.1` and `io.c 1.2`.

The output of CVS is the following:

```
cvs tag: Tagging .  
T control.c  
T graphics.c  
T io.c
```

Jerry and Tim continue working on the files and thus generate new revisions. A week later the repository contains the revisions of the files `graphics.c`, `control.c` and `io.c` as shown in figure Figure 12. The most recent revisions are shown again on the bottom and are `graphics.c` 1.4, `control.c` 1.3 and `io.c` 1.3. Nevertheless the release named `release-1-2` has not changed. It contains still `graphics.c` 1.3, `control.c` 1.1 and `io.c` 1.2.

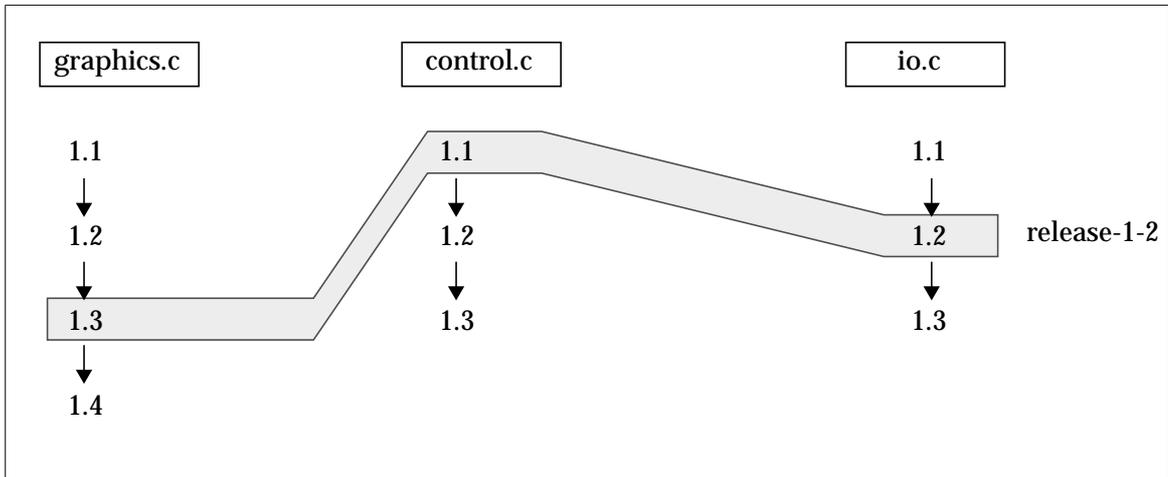


Figure 12 `release-1-2` consists of the revisions `graphics.c` 1.3, `control.c` 1.1 and `io.c` 1.2. The most recent revisions are `graphics.c` 1.4, `control.c` 1.3 and `io.c` 1.3.

Later, Jerry can check out these files by using the tag name:

```
~/jerry/work> cvs checkout -d visu-r12 -r release-1-2 projects/visualization
cvs checkout: Updating visu-r12
U visu-r12/control.c
U visu-r12/graphics.c
U visu-r12/io.c
```

The `-d` option is used to check out in a separate directory than the standard working directory. This allows to have different releases in different working directories.

After this command Jerry gets exactly the revisions of files which went into the release `release-1-2`.

This can be shown by using the `cvs status` command which shows the status of the files in the current directory:

```
~/jerry/work> cd visu-r12
~/jerry/work/visu-r12> cvs status
cvs status: Examining .
=====
File: control.c          Status: Up-to-date

Working revision:      1.1 Wed Apr 21 14:51:08 1999
Repository revision:  1.3 /work/repository/projects/visualization/control.c,v
Sticky Tag:           release-1-2 (revision: 1.1)
Sticky Date:         (none)
Sticky Options:      (none)
```

```
=====  
File: graphics.c      Status: Up-to-date  
  
Working revision:    1.3 Tue Apr 13 15:44:59 1999  
Repository revision: 1.4 /work/repository/projects/visualization/graphics.c,v  
Sticky Tag:          release-1-2 (revision: 1.3)  
Sticky Date:         (none)  
Sticky Options:      (none)  
=====  
File: io.c           Status: Up-to-date  
  
Working revision:    1.2 Tue Apr 13 15:44:59 1999  
Repository revision: 1.3 /work/repository/projects/visualization/io.c,v  
Sticky Tag:          release-1-2 (revision: 1.2)  
Sticky Date:         (none)  
Sticky Options:      (none)
```

It can be seen that there are newer revisions in the repository but by specifying the release flag the older revisions of each of the files are checked out. E.g. for `io.c` the version 1.3 exists in the repository but by specifying `release-1-2` as symbolic name when checking out Jerry got the version 1.2 for `io.c`.

9 Branches

Jerry and Tim continue to improve the `release-1-2` to create a new `release-1-3`. A week after the release date they get a complaint that there is an error in the release. They want to solve the bug immediately but the development for `release-1-3` is too advanced but not settled yet. So they need some mechanism to work independently on the bug fix and the improvements to `release-1-2`. For this aim CVS offers branches.

Jerry creates a branch out of `release-1-2`:

```
~/jerry/work/projects/visualization> cvs rtag -b -r release-1-2  
release-1-2-patch projects/visualization  
  
cvs rtag: Tagging projects/visualization
```

Figure 13 shows the resulting “main trunk” and “branch trunk” of the releases.

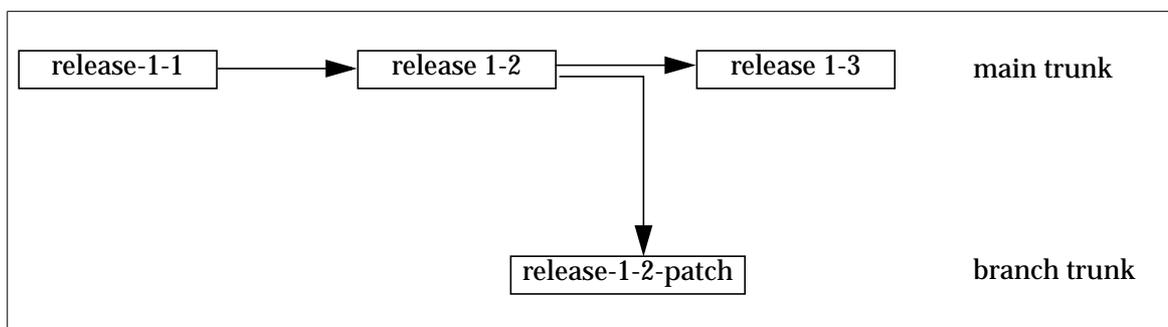


Figure 13 Creating a branch.

CVS created for the different versions of files of `release-1-2` new versions. The initial version numbers for the files in the branch `release-1-2-patch` are the version number of the file on which the new branch version is based added by 1.2. So `release-1-2-patch` consists of the files `graphics.c 1.3.1.2`, `control.c 1.1.1.2` and `io.c 1.2.1.2`. This can be seen in Figure 14.

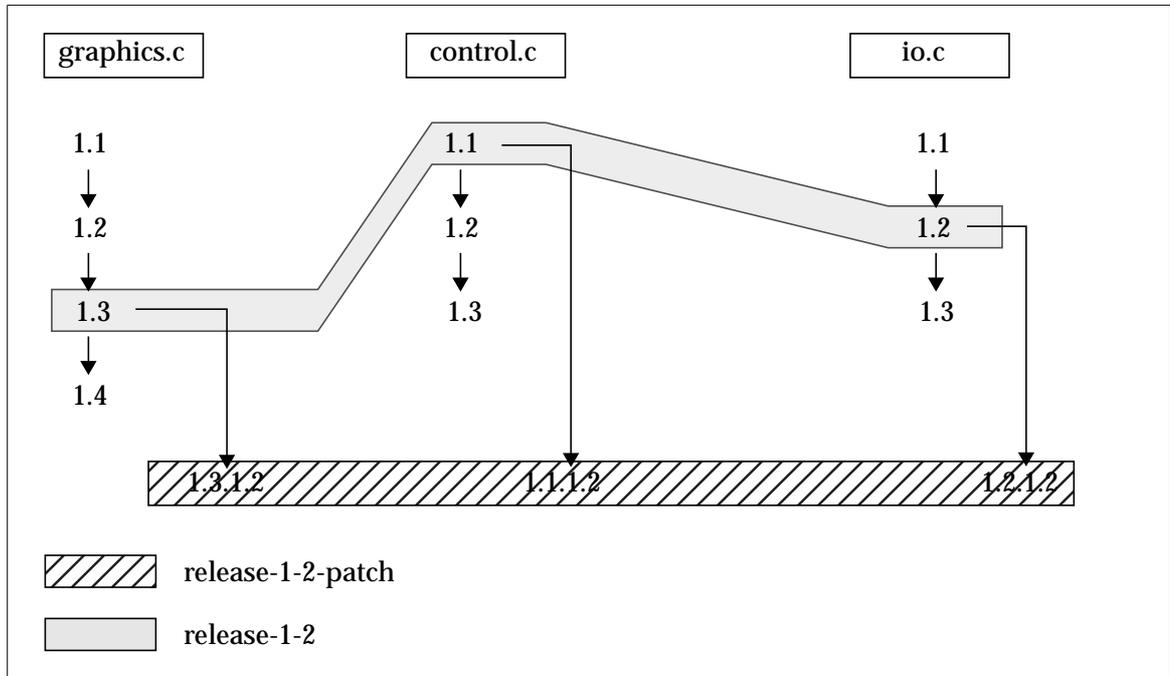


Figure 14 By creating a branch CVS created new versions. The figure shows to which release the different versions belong to.

Now they can work independently and in parallel on the main trunk and the branch trunk using the same mechanisms as described in 4 .

10 Useful Links

10.1 Where to find CVS

- At CERN it is available via ASIS on UNIX platforms
- <http://www.loria.fr/~molli/cvs-index.html>

This page is the so called *CVS bubbles* page. It contains the latest release of CVS as well as documentation on it. Additionally, it contains links to tools related to CVS, mailing lists and web pages on CVS.

10.2 Documentation and third party utilities

1. <http://www.loria.fr/cgi-bin/molli/wilma.cgi/doc>

This page contains different documentation on CVS ranging from FAQs, tutorials to detailed reference manuals on CVS (e.g. Cederquists “*Version Management with CVS*”).

2. <http://www.cyclic.com>

This page is the official page of cyclic, an organization providing support for CVS. It contains links to various CVS-related topics.

11 Future Work

This “Getting Started” does not cover all the commands and features of CVS since this would lead to far. It is not planned to include all of them in a future version, either, since there are existing documents describing the CVS commands in detail.

What we plan for a future version is to describe the usage of CVS on AFS (including e.g. security aspects) and how to use CVS on NICE/NT.