Professional Expertise Distilled

IBM Rational ClearCase 7.0: Master the Tools That Monitor, Analyze, and Manage Software Configurations

Take a deep dive into extending ClearCase 7.0 to ensure the consistency and reproducibility of your software configurations

Foreword by Lars Bendix, Ph. D., ETP, Lund University, Sweden

Marc Girod Tatiana Shpichko

[PACKT] PUBLISHING enterprise
professional expertise distilled

Table of Contents

7
Merging

As much as one may attempt to avoid conflicts, it is also necessary to solve them when they happen. This concern is the object of the present chapter.

Merging has become a pet peeve for many, to the point of being overused, and for purposes other than conflict resolution.

We'll review both practical aspects, and will guide about things to do and things to avoid.

Faithful to our focus, we shall ignore any UCM aspects and the graphical tools, with their additional and accidental complexities.

Somehow, this chapter might naturally feel of lesser importance than the preceding ones. This is of course by choice: we focused first on the essential, pushing forward the remaining explanations. Sharing our vision of the essential is part of our goal: to convince our reader to look at ClearCase *differently*! Our views are meant to be non-obvious at the start: they directly oppose the main stream thinking for which merging plays a central role. We believe, as we already tried to show, that this is unfortunate. In fact, a major part of our experience with merging comes from assisting users with concrete and complex problems that are seldom traced back to their real causes: processes and practices. Refusal to examine the cause of recurrent problems leads to treating the symptoms instead, especially with an extensive use of GUIs (naively seen as rescue buoys), or with scripts to force *copy merges* (see later); both of these have vicious circle effects and only make things worse.

We'll work out here the causes of some common misunderstandings. We believe it is fair to acknowledge a widely spread confusion: users don't feel confident with merging, and often resort to delegating it, hoping for (or requesting) miracles.

Although we strongly advise to avoid useless merges, merging is necessary to resolve conflicts and must thus be mastered.

It remains to say that the chapter is crossed by multiple themes that make the presentation of the material hard to order: they overlap, and will thus pop up several times at different points in the exposition. Let's mention the main "themes" and let you identify them in the sections:

- The tools (the practical aspect): `merge` and `findmerge`, but also `ln`, and anecdotal as it is, `rmmerge`
- The use and misuse of merging: conflict resolution versus mere version duplication
- The cumbersome undoing or rolling back
- The *Merge arrow*, or the tracing of contributions
- The nitty-gritty, that is, the type manager part, and the syntactic issues of data chunking

Patching and merging

One remembers the historical importance of *patch* in the birth of version control, and thus of *source code* management—the prehistory of SCM. This aspect of computing *deltas* (or *diffs*) and applying them on top of an existing text file is still at the heart of merging.

The differences, however marginal, lie on the one hand in using the identity of the element to determine a *base contributor* from the common version tree, and on the other hand in extending merging to directories and semi-binary files (lifting the strictest encoding limitations). The former is not a strict requirement (except for directories, maintained in the database and not as external *containers*), but covering the vast majority of the cases, builds upon the maintenance of *Merge arrows* to structure the version trees. In addition, bulk merging, using `findmerge`, only applies to versions of the same elements.

Patching text files

Reminiscent of the text file orientation of traditional languages, merging is offered by (several) *element type managers*, centered around the one for text files. While closer attention to element types will be postponed to *Chapter 9*, *Secondary Metadata*, we have to mention this delegation of functions to specific tools.

Let's first note that it is fully possible with files to use tools external to ClearCase to perform this function. In fact, this is exactly what we do ourselves, at least for verification, using the `ediff-buffers` function under *GNU emacs*, in any case of real merge involving more than three lines or more than one location in the file.

One has then to take care of the *Merge arrows*: see lower.

Looking more closely at the way the type managers are configured, that is, at the contents of the `lib/mgrs` directory under the installation root on every ClearCase client, one notices that on UNIX, the tools performing the various functions (such as `merge` for textual, and `xmerge` for graphical merges) are specified via symbolic links, whereas on Windows they are specified in a `map` file and linked there, either via paths or references into the Windows registry or the merge tools.

Either way, this allows sharing of some tools between various *managers* in a way which is in fact orthogonal to their inheritance hierarchy, and thus allows overriding it. What we can see is that the *text_file_delta* tools serve as basis for several of the other element types, and that in the case of the two functions we are concerned with in the current context, these resolve to: `cleardiff` and `xcleardiff` in UNIX and `cleardiff.exe` and `cleardiffmgr.exe` in Windows.

```
$ ll /opt/rational/clearcase/linux_x86/lib/mgrs/binary_delta
lrwxrwxrwx 1 root root 22 Nov 13 2008 compare -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 22 Nov 13 2008 merge -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xcompare -> ../../../bin/xcleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xmerge -> ../../../bin/xcleardiff

$ ll /opt/rational/clearcase/linux_x86/lib/mgrs/text_file_delta
lrwxrwxrwx 1 root root 22 Nov 13 2008 compare -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 22 Nov 13 2008 merge -> ../../../bin/cleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xcompare -> ../../../bin/xcleardiff
lrwxrwxrwx 1 root root 23 Nov 13 2008 xmerge -> ../../../bin/xcleardiff

$ cat /cygdrive/c/Program\ Files/Rational/ClearCase/lib/mgrs/map
text_file_delta  compare    ..\..\bin\cleardiff.exe
text_file_delta  xcompare ..\..\bin\cleardiffmrg.exe
text_file_delta  merge        ..\..\bin\cleardiff.exe
text_file_delta  xmerge      ..\..\bin\cleardiffmrg.exe

binary_delta  compare     ..\..\bin\cleardiff.exe
binary_delta  xcompare   ..\..\bin\cleardiffmrg.exe
binary_delta  merge         ..\..\bin\cleardiff.exe
binary_delta  xmerge       ..\..\bin\cleardiffmrg.exe
```

This gives us a hint as to how we might define an element type to use a custom merge tool. We'll leave this to those among our readers whom we couldn't convince that this is only a sidetrack.

Once we are already this far down (or up: matter of taste) technicalities, let's conjecture about the meaning of *binary_delta*, and of merging, in this context. Merging and *diffing* implies *chunking* the data into comparable items, which seems simple in the case of lines of text, the only problem there being deciding the exact syntax of separators: end-of-line codes. However, HTML and XML standards made it obvious that this stood only on convention: in fact, few of the traditional software languages made it mandatory for software texts to break on *lines*! Statements and definitions broke more often on semicolons, forms on parentheses... Yet, until the advent of the world-wide-web, systematically removing any whitespace was practiced only for code obfuscation purposes.

Not so anymore when the data had to be downloaded under the constraints of limited bandwidth. Some other ways to identify comparable patterns, of size compatible with this of reasonable buffers, had to be devised. A further step has since been made with the different variants of Unicode, bringing back issues of ordering of multibytes characters. This has now made its way into specialized element types in version 7.1 of ClearCase.

Managing contributions

Back now to the issue of managing the contributions to a given version, that is, to identifying the contributing versions. During the merge, diffs (or deltas) have to be computed between every contributor and a common reference: the **base contributor**—typically the closest common ancestor, possibly one of the contributors.

The `merge` tool will use the version tree to compute a suitable version for the role of *base*, aiming at producing as small diffs as possible. For that purpose, beyond direct ancestry, special kinds of hyperlinks (between versions) will be used: instances of the *Merge* type. They will be created at the end of every merge, for use at the start of any later one. They may also be created and removed explicitly.

Let's work out an example, as simple as possible. We are trying here to describe a mechanism, not to recommend a process.

We'll use a `foo` text file element, make changes in an `mg` branch (selected by the config spec: every access of the main branch thus has to be explicit), and merge them to the `main` one.

The initial data is a single line.

```
$ echo 111 > foo
$ ct mkelem -nc -ci foo
Created element "foo" (type "text_file").
Created branch "mg" from "foo" version "/main/0".
Checked in "foo" version "/main/mg/1".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/0") is different from ###
                 version selected by view before checkout ("/main/mg/1").
Checked out "foo" from version "/main/0".
$ ct merge -to foo -ver /main/mg/1
Trivial merge: "foo" is same as base "/vob/test/foo/merge/foo@@/main/0".
Copying "/vob/test/foo/merge/foo@@/main/mg/1" to output file.
Moved contributor "foo" to "foo.contrib".
Output of merge is in "foo".
Recorded merge of "foo".
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
```

```
Checked in "foo" version "/main/1".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/1
```

As we use the `-merge` option, `lsvtree` shows the *Merge* arrows.

We got here a first case of **trivial merge**: a merge is said trivial if the base contributor is the same as the merge target (the *to* contributor)—here, it is the version which was checked out, since no change was made to it yet. Trivial merges are automatic: the user is not prompted for a decision (unless he explicitly requires it with the `-qall` flag).

Let's now assume that for some reason, we want to roll back this change, that is, to *undo* the merge. The most natural option is to use a **subtractive merge** (not quite a merge at all in fact, but a related use of the same tool, `ct merge`, with `-del` option).

```
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/1") is different from ###
                 version selected by view before checkout ("/main/mg/1").
Checked out "foo" from version "/main/1".
$ ct merge -to foo -del -ver /main/1
Trivial merge: "foo" is same as base "/vob/test/foo/merge/foo@@/main/1".
Copying "/vob/test/foo/merge/foo@@/main/0" to output file.
Moved contributor "foo" to "foo.contrib.1".
Output of merge is in "foo".
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
Checked in "foo" version "/main/2".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/2
```

As we can see, this didn't result in a user noticeable change in the topology of the version tree, except for the creation of a new version: `/main/2`. Note that there are no Merge arrows for subtractive merges.

The base contributor was again selected as the closest common ancestor, which was also the contributor being deleted, as well as the *to contributor*; hence once again a *trivial* merge.

We also note the saving of view-private copies of the contributors, which we consider a minor nuisance.

Next, we'll make a further change to the version on the branch (maybe fixing the problem which resulted in the previous rollback):

```
$ ct ls foo
foo@@/main/mg/1    Rule: .../mg/LATEST
$ ct co -nc foo
Checked out "foo" from version "/main/mg/1".
$ echo 222 >> foo
$ ct ci -nc foo
Checked in "foo" version "/main/mg/2".
```

And we'll attempt to roll out (well, to merge) again. This should be understood as a common practice (note that we do not say a good one), yet its result is not *intuitive* (users typically notice at this stage that they weren't extremely clear about their own expectations):

```
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/2") is different from ###
                 version selected by view before checkout ("/main/mg/2").
Checked out "foo" from version "/main/2".
$ ct merge -to foo -ver /main/mg/2
********************************
<<< file 1: /vob/test/foo/merge/foo@@/main/mg/1
>>> file 2: /vob/test/foo/merge/foo@@/main/mg/2
>>> file 3: foo
********************************
-----[deleted 1 file 1]------|------[after 0 file 3]-------
111                          |-
                            -|
*** Automatic: Applying DELETION from file 3 [deleting base line 1]
============
============
----[after 1 file 1]---------|-------[inserted 2 file 2]----
                            -| 222
                             |-
Do you want the INSERTION made in file 2? [yes]
Applying INSERT from file 2 [line 2]
============
============
Moved contributor "foo" to "foo.contrib.2".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
222
```

For a start, this wasn't a trivial merge anymore: ClearCase had to prompt the user. We must understand that this kind of dialog is suboptimal:

- It stops the transaction (maybe multiple times)
- It is error-prone, and thus stressful to the user: no more predictable for her than for the tool
- The transcript will often get lost, which means that the merge result can only be inspected by others on the basis of less and different data than was presented to the user

This means that a procedure based on this kind of merging cannot be atomic and may introduce states that affect others.

Then, we were prompted for the *wrong* thing; the question concerned the last addition (line `222`, the *fix*), instead of the previous deletion. Why does this feel wrong? Because it differs from the *normal* behavior: what happened the first time, before the rollback. The common user does not expect this. What is different for the user is the deletion. The above sequence is, however, the correct behavior from the point of view of ClearCase, and we'll have to understand why. The result is anyway that the deletion was confirmed, against the intention; the line added to version `/main/mg/1` (`111`) might have had an undesirable side effect, which motivated the rollback, but it is still part of the fixed version `/main/mg/2` and the user expects it to be found in the version resulting from the merge.

Let's clean up, that is, abort this attempt by unchecking out its results and try again, taking this time the responsibility of naming ourselves the base contributor, picking first the new version in the branch, which results in skipping the history of the previous deletion.

```
$ ct unco -rm foo
Checkout cancelled for "foo".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/2") is different from ###
                 version selected by view before checkout ("/main/mg/2").
Checked out "foo" from version "/main/2".
$ ct merge -to foo -base foo@@/main/mg/2 -ver /main/mg/2
Trivial merge: "/vob/test/foo/merge/foo@@/main/mg/2" is same as base ####
                                                       "foo@@/main/mg/2".
Copying "foo" to output file.
Moved contributor "foo" to "foo.contrib.3".
Output of merge is in "foo".
$ cat foo

$
```

We are satisfied of reaching our goal of getting a trivial (therefore automatic) merge, but should be slightly surprised as this behavior doesn't match well with the ***documentation*** as we had read it; the *base* and *to* contributors are now different!

However, no visible change took place (the resulting file is still not correct, as it is empty now)! Let's pick the *to contributor* as *base*, thus reconciling the documentation of trivial merges:

```
$ ct merge -to foo -base foo@@/main/2 -ver /main/mg/2
Trivial merge: "foo" is same as base "foo@@/main/2".
Copying "/vob/test/foo/merge/foo@@/main/mg/2" to output file.
Moved contributor "foo" to "foo.contrib.4".
Output of merge is in "foo".
$ cat foo
111
222
```

The data is now the expected one. Let's thus check in and go forward.
Note, however, that in the last two merge examples, when we specified explicitly the base contributor ourselves, we did not get the `Recorded merge of "foo"` message, and hence, no Merge arrow was created automatically:

```
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/2
foo@@/main/2
foo@@/main/CHECKEDOUT view "marc"
```

This time, we'll make a change to the checked out version, prior to the merge (in parallel to a change to the version to be merged):

```
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
Checked in "foo" version "/main/3".
$ ct co -nc foo
Checked out "foo" from version "/main/mg/2".
$ echo 333 >> foo
$ ct ci -nc foo
Checked in "foo" version "/main/mg/3".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/3") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "foo" from version "/main/3".
$ echo 444 >> foo
$ cat foo
```

```
111
222
444
$ cat foo@@/main/mg/3
111
222
333
$ ct merge -to foo -ver /main/mg/3
********************************
<<< file 1: /vob/test/foo/merge/foo@@/main/mg/1
>>> file 2: /vob/test/foo/merge/foo@@/main/mg/3
>>> file 3: foo
********************************
-------[after 1 file 1]--------|-----[inserted 2-3 file 2]----
                              -| 222
                               | 333
                               |-
-------[after 1 file 1]--------|-----[inserted 2-3 file 3]----
                              -| 222
                               | 444
                               |-
Do you want the INSERTION made in file 2? [yes]
Applying INSERT from file 2 [lines 2-3]
Do you want the INSERTION made in file 3? [no] yes
Applying INSERT from file 3 [lines 2-3]
============
============
Moved contributor "foo" to "foo.contrib.5".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
111
222
333
222
444
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/3
 -> /main/CHECKEDOUT
foo@@/main/3
foo@@/main/CHECKEDOUT view "marc"
```

The result is that the base and to contributors are again different, hence the merge non-trivial.

Furthermore, the options we were prompted didn't once again leave us the choice of selecting anything satisfying (note though the automatic creation of the Merge arrow in the `lsvtree` command output above).

We'll record the first remark as advice: do not modify the checkedout version if you wish the merge to be trivial. Let's try and investigate the cause for the surprising prompts; thus clean up and retry without any change to the checked out version:

```
$ ct unco -rm foo
Checkout cancelled for "foo".
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/3") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "foo" from version "/main/3".
$ ct merge -to foo -ver /main/mg/3
********************************
<<< file 1: /vob/test/foo/merge/foo@@/main/mg/1
>>> file 2: /vob/test/foo/merge/foo@@/main/mg/3
>>> file 3: foo
********************************
-------[after 1 file 1]-------|-----[inserted 2-3 file 2]----
                             -| 222
                              | 333
                              |-
-------[after 1 file 1]-------|-----[inserted 2 file 3]------
                             -| 222
                              |-
Do you want the INSERTION made in file 2? [yes]
Applying INSERT from file 2 [lines 2-3]
Do you want the INSERTION made in file 3? [no]
============
============
Moved contributor "foo" to "foo.contrib.6".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
111
222
333
```

We now obtain the intended result in the data, but after being prompted: the merge was not trivial, and that is because the base contributor was selected to be the first version in the branch.

This is only possible because no *Merge arrow* was actually created from `/main/mg/2` to `/main/3` as a part of the step in which we forced the base contributor (contrary to our expectation, reinforced by the fact that, as we'll see later, such arrows *are* created in a similar context by `findmerge`)!

To verify this theory (that the cause was the absence of the arrow), let's clean up again, and create ourselves the expected and missing hyperlink and check that its presence suffices to drive the expected behavior:

```
$ ct unco -rm foo
Checkout cancelled for "foo".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/3
foo@@/main/3
$ ct merge -ndat -to foo@@/main/3 -ver /main/mg/2
Recorded merge of "foo".
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/2
 -> /main/3
foo@@/main/mg/3
foo@@/main/3
$ ct co -nc -bra /main foo
cleartool: Warning: Version checked out ("/main/3") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "foo" from version "/main/3".
$ ct merge -to foo -ver /main/mg/3
Trivial merge: "foo" is same as base ####################################
                                  "/vob/test/foo/merge/foo@@/main/mg/2".
Copying "/vob/test/foo/merge/foo@@/main/mg/3" to output file.
Moved contributor "foo" to "foo.contrib.7".
Output of merge is in "foo".
Recorded merge of "foo".
$ cat foo
111
222
333
$ ct lsvtree -merge foo
foo@@/main
foo@@/main/0
```

```
foo@@/main/mg
foo@@/main/mg/1
 -> /main/1
foo@@/main/mg/2
 -> /main/3
foo@@/main/mg/3
 -> /main/CHECKEDOUT
foo@@/main/3
foo@@/main/CHECKEDOUT view "marc"
$ ct ci -nc foo
cleartool: Warning: Version checked in is not selected by view.
Checked in "foo" version "/main/4".
```

We created the missing *Merge arrow* with the merge tool, using the `-ndata` flag. We could as well have used `mkhlink`, explicitly with the `Merge` type:

```
$ ct mkhlink Merge foo@@/main/mg/2 foo@@/main/3
Created hyperlink "Merge@232@/vob/test".
```

In the same way, to remove a *Merge arrow*, maybe in order to remove an element (as the presence of hyperlinks suffices to restrict the use of `rmelem` to the vob owner or the administrator), one may use either `rmmerge`, or `rmhlink` with a hyperlink name obtained with `describe -l`.

Note that `rmver` is also protected in a similar way from removing interesting versions, for example, the versions bearing hyperlinks. The way to bypass the protection is different, though—it is to use an `-xhl` flag (`-xla` for labels):

```
$ ct rmver foo@@/main/mg/3
cleartool: Error: Removal of "interesting" versions must be explicitly ##
                                                                 enabled.
Not removing these "interesting" versions of "foo":
 /main/mg/3 (has: hyperlinks)
cleartool: Error: No versions of "foo" to remove.
$ ct rmver -xhl -f foo@@/main/mg/3
Removing these versions of "foo":
 /main/mg/3 (has: hyperlinks)
Removed versions of "foo".
```

And by the way, `rmbranch` is protected in a different way: please refer to the section *Branches and branch types* from *Chapter 6*, *Primary Metadata*.

Despite the fact that we found ways to resolve the various contradictions and obtain the properties we wanted—triviality (automatic handling), expected outcome and tracing of the contributions—the complexity of the situations met may still be felt daunting. To recapitulate what is in our experience the most surprising to users: the merge behavior is driven in part by the presence or absence of Merge arrows, which depends on the prior history of merging.

In particular, some user communities will avoid subtractive merges and prefer destructive rollbacks, that is, removal of the versions resulting from the merges. We believe this is counter-productive: no less error-prone and removing the evidence on the basis of which the problems met may be analyzed.

In other words, radical as it may be, such a strategy is not radical enough; the problem is not subtractive merging, it is publishing by merging, instead of publishing in-place by moving labels.

Merging directories

The issue of merging directories is actually quite different from that of merging files.

Let's note that, under ClearCase, directories are not stored in containers (standard file objects found in vob *pools*) unlike file elements, so that their merging cannot be delegated to external tools, acting upon inodes accessed over NFS. It has to use ClearCase functions at the low level, to create and remove hard links.

In fact, with large directories, it is often more convenient and more manageable to use `rm` and `ln` explicitly than to use `merge`, and correct the results before checking in:

```
$ ct co -nc -bra /main .
cleartool: Warning: Version checked out ("/main/6") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "." from version "/main/6".
$ ct ln .@@/main/mg/3/foo .
Link created: "./foo".
$ ct ln .@@/main/mg/3/foo1 ./zoo
Link created: "./zoo".
$ ct rm bar
Removed "bar".
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/7".
```

This is especially true when trying to restore an entry removed in a previous version, or to ensure that parallel works will not result in the creation of *evil twins*, by selectively sharing some work before its delivery.

We have already dealt with the issues of recovering files by using `ln`, hard links and evil twins in *Chapter 4*, *Version Control*.

Let's now demonstrate how merging directories may produce hard links, and let's make it more obvious by making these hard links of the same element under different names.

In the example we start from, the file name exists only on an mg branch of the directory.

We'll first merge this "back" to the main branch, then rename the entry in the branch:

```
$ ct lsvtree .
.@@/main
.@@/main/0
.@@/main/mg
.@@/main/mg/1
$ ct co -nc -bra /main .
cleartool: Warning: Version checked out ("/main/0") is different from ###
                 version selected by view before checkout ("/main/mg/1").
Checked out "." from version "/main/0".
$ ct merge -to . -ver /main/mg/1
********************************
<<< directory 1: /vob/test/foo/merge@@/main/0
>>> directory 2: /vob/test/foo/merge@@/main/mg/1
>>> directory 3: .
********************************
-------[ directory 1 ]---------|-----[ added directory 2 ]----
                              -| foo --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 2
Recorded merge of ".".
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/1".
$ ct co -nc .
Checked out "." from version "/main/mg/1".
$ ct mv foo bar
Moved "foo" to "bar".
$ ct ci -nc .
Checked in "." version "/main/mg/2".
```

Now, we'll create a second branch of the directory (of a new fff type), and rename the entry there as well. We do this in order to confuse merge so that it doesn't remove the original name.

```
$ ct mkbrtype -nc fff
Created branch type "fff".
$ ct mkbranch -nc fff .@@/main/1
Created branch "fff" from "." version "/main/0".
Checked out "." from version "/main/fff/0".
$ ct mv foo zoo
Moved "foo" to "zoo".
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/fff/1".
```

Last, we merge the two branches at the same time into the `main` one, which we previously checked out.

The base contributor is selected to be `/main/0`, which results in keeping the original name `foo`, as well as adding the other two.

```
$ ct co -nc -bra /main .
cleartool: Warning: Version checked out ("/main/1") is different from ###
                 version selected by view before checkout ("/main/mg/3").
Checked out "." from version "/main/1".
$ ct merge -to . .@@/main/mg/2 .@@/main/fff/1
********************************
<<< directory 1: /vob/test/foo/merge@@/main/0
>>> directory 2: .@@/main/mg/2
>>> directory 3: .@@/main/fff/1
>>> directory 4: .
********************************
-------[ directory 1 ]-------|-----[ added directory 2 ]----
                            -| bar  --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 2
------[ directory 1 ]--------|-----[ added directory 4 ]----
                            -| foo  --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 4
-----[ directory 1 ]---------|-----[ added directory 3 ]-----
                            -| zoo  --08-01T12:04 marc
*** Automatic: Applying ADDITION from directory 3
Recorded merge of ".".
$ ls -la .
total 1
drwxrwxr-x 2 marc jgroup 69 Aug 1 20:45 .
drwxrwxr-x 4 marc jgroup 0 Jul 31 21:41 ..
dr-xr-xr-x 1 marc jgroup 0 Aug 1 12:04 bar
dr-xr-xr-x 1 marc jgroup 0 Aug 1 12:04 foo
dr-xr-xr-x 1 marc jgroup 0 Aug 1 12:04 zoo
```

We check that the three names are indeed hard links of the same element, by printing out their common *object id*:

```
$ ct des -fmt "%n %On\n" bar@@ foo@@ zoo@@
bar@@ 7cc55281.9d5d11df.926d.00:01:84:2b:ec:ee
foo@@ 7cc55281.9d5d11df.926d.00:01:84:2b:ec:ee
zoo@@ 7cc55281.9d5d11df.926d.00:01:84:2b:ec:ee
$ ct ci -nc .
cleartool: Warning: Version checked in is not selected by view.
Checked in "." version "/main/2".
$ ct lsvtree -merge .
.@@/main
.@@/main/0
.@@/main/mg
.@@/main/mg/1
```

```
 -> /main/1
.@@/main/mg/2
 -> /main/2
.@@/main/1
.@@/main/fff
.@@/main/fff/1
 -> /main/2
.@@/main/2
```

We showed here how merging directories is in fact handling hard links. We showed that the merge tool doesn't offer any significant additional safety against producing duplicates of the same element: *caveat emptor*. Let's examine this situation, and the reasons there might be to manage it.

It is the second time in this book we meet this case: we already mentioned it as a technique to reach certain goals in *Chapter 2*, *Presentation of ClearCase*, in the paragraph on config specs. We admitted that these goals were exceptional, but reckoned they might be legitimate, and we stated that offering a solution was better than forcing the users to all too simple and more dangerous workarounds.

Hard links open the door to selecting different versions of the same elements in the same view (via scope rules). This fights *the* basic strategy of SCM for identifying resources in two distinct steps: as a set and as a certain item in the set. Achieving this with hard links is a means to ensure the case remains manageable and will be reminded (for example at the time of applying labels: the application will be attempted once for every name, and while the first attempt will succeed, any latter will fail—unless using a `-replace` flag to silence the error).

But there is a case in which this may result in random looking errors, and it is when the hard linked elements are directories. Two preliminary remarks:

- The IBM documentation warns against creating hard links of directories in other contexts than directory merges; doing so is a red flag for the support.
- The procedure exclusively using merge that we showed above and which resulted in multiple names for the same element, did not depend on the element being a file; it works exactly the same with directories.

Now, the problem. As mentioned earlier, unlike files, directories are not mapped to cleartext containers, that is, to file objects outside the database. With files, the view merely identifies or produces a container, which it returns to the application, the interaction between the two being thus an atomic transaction. Not so with directories: the view, and thus the mvfs kernel module below it, services in turn all the requests as the application generates them. The problem is that concurrent applications selecting different versions of the same directory may interfere over time with each other. The effects are not easily predictable, but may at least be incorrect version selections or stale nfs handles.

Note that there has been a longstanding prevention in UNIX against hard linking directories. This was rooted, we believe, in recursion due to cycles while navigating directory trees, which led in the worst cases to file systems corruption. It is still easy to experiment such issues (i.e. endless recursion, not file system corruption) using the *find* tool down from the `/view` root, even if this may be a case of mount point recursion. The fact is directory hard links are pervasive in ClearCase MVFS (in the view extended space).

Rebase or home merge

After presenting multiple examples of *trivial merges* produced with the `merge` command, we must apologize to our reader for what could be seen as confusing and contradictory: we pushed rather far the extent of the acknowledgement that we would *describe a mechanism, not recommend a process*.

Let's make our point explicit: *any merges resulting in mere duplication of data are to be avoided whenever possible*. Note how this precisely condemns trivial and copy merges!

SCM attempts to factor commonalities and make *real* differences stand out. On the contrary, using both of two identical versions introduces *spurious* differences into the dependency tree of dependent artifacts, and results in invalidating derived objects for artificial reasons. This harmful noise makes it harder to detect meaningful differences.

There remains one legitimate, necessary use of merging. The issue is to resolve *real* conflicts between versions that typically arise in the context of parallel development when different users concurrently modify the same resource: at some point, one user notices that the baseline from which she started has now moved forward to integrate somebody else's contributions. This *purpose* is well covered by the concept of *rebase.* We feel however that speaking rather of **home merge** is *process agnostic*, hence more general; the idea is to modify one's working version with changes coming from somewhere else.

In any case, irrespective of how you name them, *rebase* operations can be carried out with the `merge` tool, as well as, for bulk merges (see below), with `findmerge`.

Let's dive back, for one more time, into the context of the (ill-advised as it is) tradition of publishing by merging *back* to *main* or *integration* branches. In this context, one attempts to achieve *trivial* merges for the *delivery* as a means to reduce the inherent instability of the process, which is to make the symptoms bearable instead of curing their causes.

The best way to ensure that a merge will be *trivial* is to have performed it already, to empty the remaining merge from its real contents. This is achieved by performing a preemptive merge in the reverse direction from the one actually intended. The object is to reach a situation satisfying the condition for trivial merges: the coincidence of the base and the *to contributors*. The notion of merging direction being inherently confusing, this first preemptive merge is termed *rebase*, and the next "merge" operation is consistently assimilated to its purpose: *delivery*.

Rebasing alone is not totally foolproof, as it takes a non-negligible time (especially if it concerns many files); there is always a window of opportunity for changes to take place in the integration branches between the "rebase" and the "delivery". This may be prevented with locking, and/or by wrapping up the whole procedure into a "higher level" and heavier operation (what UCM does).

A yet more desperate alternative to trivial merges is to force the delivery to use *copy* merges, that is, to ignore any unlikely but possible changes that might have happened since the last rebase. There are various ways to achieve this, the simplest involving an operating system copy instead of merge (and possibly using `merge -ndata` in addition, only to create the *Merge arrow*). The copy merge will also protect against surprises resulting from former subtractive merges.

A last note concerning automatic, hence *trivial* merges: fine-tuning the merge tool to resolve the differences correctly and perform the actual merges without, or with as little user intervention as possible, is in itself a useful task, as it enforces reproducibility and reduces the possibility for user errors. We saw how managing the base contributor could, in some cases, be a tool to affect this aspect of things.

Complex branching patterns

Until now we have unfortunately only scratched the surface of the real complexity met when trying with such technologies, to maintain cascading branching patterns, with several levels of integration and different release projects running in parallel. If these levels and projects have been assigned specific branch types, one will need to merge the same changes to several places in turn (including thus the possible rebasing).This alone would work as a red flag in the absence of all too common process anesthesia.

Just consider the task of identifying the base contributors in the contexts of the various patterns!

Rather than delving into such intricacies, we feel it is time to remind that they are purely artificial—the consequences of an adverse choice to stick to a tradition inherited from the early systems that had no support for branches—and to try to get back at delivery time to this historical simplicity.

Let us thus send you back to the advice spelled in *Chapter 5*, under the title: *Avoid depending on mastership*, and publish in-place, exclusively using labels.

Following this advice, one may share the same versions in the different contexts in which it makes sense (by applying different labels), and merging is left to the only cases from where it cannot be expelled: for resolving contradictions, where it need not anymore be called *rebasing*.

Rollback of in-place delivery

We asserted that in-place delivery would be reversible. Now is the time to show it in practice, on an example. First the trivial case: a new version of the file `foo` was edited in an `mg` branch, spawn from the baseline represented by the floating label type `AAA`, equivalent at this point to the fixed `AAA_2.37`. The situation prior to the delivery was thus seen with the `lsgenealogy` command from our ***ClearCase::Wrapper::MGi*** wrapper, invoked with a depth of two levels of ancestry:

```
$ ct lsgen -d 2 foo
foo@@/main/mg/5 (MG, MG_1.25)
 foo@@/main/mg/1
   foo@@/main/ts-012/7 (AAA, AAA_2.37, TS-005, TS_3.01)
```

After the delivery, including *archiving* (see Chapter 6) of the branch (`mg` -> `mg-003`) and label (`MG` -> `MG-013`) types, the situation changed to the following (only labels have been applied, and branch types renamed *away* in order to allow existing config specs to retain their semantics in presence of a modified environment):

```
$ ct lsgen -d 2 foo
foo@@/main/mg-003/5 (AAA, AAA_2.38, MG-013, MG_1.25)
 foo@@/main/mg-003/1
   foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
```

At this point, the rollback would be, as announced earlier, trivial, and would yield:

```
$ ct lsgen -d 2 foo
foo@@/main/mg-003/5 (AAA_2.38, MG-013, MG_1.25)
 foo@@/main/mg-003/1
   foo@@/main/ts-012/7 (AAA, AAA_2.39, AAA_2.37, TS-005, TS_3.01)
```

Let's not forget that this triviality contrasts with the weight of the procedures one might consider to use instead in the context of delivery by merging, and which would be in no way equivalent!

But let's consider now that instead of this rollback, there takes place a new phase of development (in the `ts` branch) before the flaw leading to the need to rollback is discovered:

```
$ ct lsgen -d 3 foo
foo@@/main/ts/1 (TS, TS_3.03)
 foo@@/main/mg-003/5 (AAA, AAA_2.38, MG-013, MG_1.25)
   foo@@/main/mg-003/1
     foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
```

Let this get delivered too:

```
$ ct lsgen -d 3 foo
foo@@/main/ts-013/1 (AAA, AAA_2.39, TS-006, TS_3.03)
 foo@@/main/mg-003/5 (AAA_2.38, MG-013, MG_1.25)
   foo@@/main/mg-003/1
     foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
```

And now, let's consider the challenge of rolling the previous change back!
The task decomposes in two steps:

- Rolling back to the stage before the faulty delivery
- Reapplying the changes that took place later, "on top of it"

We have already considered the first step; what happened later doesn't affect the pre-delivery stage version (labeled `AAA_2.37`).

The question is thus to merge the next changes (from the version carrying the label `AAA_2.39`), and to subtract from them the intermediate ones (`AAA_2.38`), but this time creating a unique new version and not a duplicate of an old one (`AAA_2.37`). The first merge is a normal ClearCase one; however, the subtractive merge is not. ClearCase subtractive merges can only affect those changes that took place in the same branch. We'll have to resort to the UNIX diff and patch tools that we mentioned in Chapter 2. We'll use them with a `-c` flag to generate and use context information, allowing to apply the negative differences to a different version of the file than the one on the basis of which they were computed.

```
$ ct co -nc foo@@/AAA_2.37
Created branch "mg" from "foo" version "/main/0".
Checked out "foo" from version "/main/mg/0".
$ ct merge -to foo -ver AAA
Trivial merge: "foo" is same as base ###################################
                                   "/vob/test/merge/foo@@/main/ts-012/7".
Copying "/vob/test/merge/foo@@/main/ts-013/1" to output file.
Moved contributor "foo" to "foo.contrib".
Output of merge is in "foo".
Recorded merge of "foo".
$ diff -c foo@@/AAA_2.38 foo@@/AAA_2.37 > foo.patch
$ patch -c foo <foo.patch
 Looks like a normal diff.
done
$ ct ci -nc foo
Checked in "foo" version "/main/mg/1".
$ ct mklabel MG foo
Created label "MG_1.26" on "foo" version "/main/mg/1".
Created label "MG" on "foo" version "/main/mg/1".
$ ct mklbtype -nc -inc AAA
Created label type "AAA_2.40".
$ ct mklabel -over MG AAA
Created label "AAA_2.40" on "./foo" version "/main/mg/1".
Moved label "AAA" on "./foo" from version ##############################
                                        "/main/ts-013/1" to "/main/mg/1".
$ ct lsgen -d 4 foo
foo@@/main/mg/1 (AAA, AAA_2.40, MG, MG_1.26)
[siblings: foo@@/main/mg-001/1]
foo@@/main/ts-012/7 (AAA_2.37, TS-005, TS_3.01)
foo@@/main/ts-013/1 (AAA_2.39, TS-006, TS_3.02)
 foo@@/main/mg-003/5 (AAA_2.38, MG-013, MG_1.25)
  foo@@/main/mg-003/1
    [alternative path: foo@@/main/ts-012/7]
```

Several points require comments. First, we used our wrapper for checking out, with explicit (in the config spec) support for implicitly branching off `/main/0`. This is what explains that the successor to `foo@@/AAA_2.37`, a.k.a `foo@@/main/ts-012/7`, is checked out from `foo@@/main/mg/0` instead of from `foo@@/main/ts-12/mg/0`. You'll remember there our strategy to avoid cascading forever.

Next, we show gradually deeper views of the genealogy, as we add new generations. This shows two parents (or contributors) to the same `foo@@/main/mg/1`, and presents them at the same level of indentation. We also notice two annotations to the genealogy output: information that version `/main/mg/1` has at least one *sibling*, not shown in this genealogy, but spawn from the same parent; and a reminder that version `/main/ts-012/7` is reachable by two distinct paths, and thus need not be repeated.

There is one important detail we ought not to hide: when we take the `AAA_2.37` label as the version to rollback, we make a *human*, interactive decision, not an automatic one. In this case, the choice is trivial, but if we were dealing with many files, nothing would guarantee the previous label would be the same for all. This is, however, only a missing functionality of our wrapper; in the next version, it will compute the correct version to rollback to, on an element basis.

This will allow the rollback procedure (the "first part", as mentioned earlier, and which we did not replay as an intermediate step) to be fully automatic.

The second part, reapplying the later changes, will however remain potentially interactive: it is a *real* merge, resolving differences, and producing a unique new version.

This example, which ought still to be concluded with an archiving, showed the clear superiority of our delivery model over the traditional one. If one still needs to convince oneself, one may compare this earlier case to the functionally simpler scenario we reviewed in the *Managing contributions* section. Do we need to list the advantages?

- **Atomicity**: No intermediate state, such as checking out
- **Reversibility**: The rollback itself is a toggle, rolling forward again would be instantaneous
- **Simplicity**: No version added
- **Fairness**, in a MultiSite context (using local labels): Delivery and rollback are decoupled from branch mastership concerns (consistent state of remote labels cannot be trusted anyway)
- **Speed** and hence **robustness**: No window of opportunity for errors

The further advantage we attempted to exploit is the practical availability of all the contributor versions for further fixing or development.

One may be tempted, and we'll not decide here, to modify the topology of the graph drawn by the Merge arrows (that is, to rewrite history) so that the version `AAA_2.38` which we rolled back would not be seen anymore as part of the genealogy of `AAA_2.40`. As explained previously, such a change could avoid later surprises with using the merge tool. We would of course retain both `2.37` and `2.39` as direct contributors, but replace the two Merge arrows, from `2.37` to `2.38` and from `2.38` to `2.39`, with a single and direct one, from `2.37` to `2.39`. We leave it to our reader to decide whether such bold historical revisionism would increase or decrease confusion.

Bulk merges

We've been analyzing merging in the simple case of one single element. Obviously, the more usual case concerns a whole change set, so we have to transpose what we came up with so far in the context of another tool: `findmerge`. As the name suggests, it combines the functions of `find` and of `merge`, performing in addition the necessary checking out in the middle. This applies to all the cases of merging we met, especially to the real merges, a.k.a rebases, with the possible exception of the *copy* merges (back to this below).

Because `findmerge` is expected to perform `checkout` operations, the *to contributors* are implicitly the ones selected by the view. The function has a lot of options, and we'll only cover the most useful ones. The most idiomatic (simplest) is probably to design the other set of contributors (*from*) with a label, using the `-fversion` option. One may also use another view and `-ftag`, but this slightly raises the risk of creating evil twins.

```
$ ct findmerge . -nc -fve FOO -merge
```

This will produce `.contrib` as well as a `findmerge.log.<date-time>` private files, and leave the modified elements checked out.

In case of non-trivial merges, it will prompt the user, which as we already stated, but even more so in the context of `findmerge`, which is often inconvenient.

There are two ways to improve on this:

- One is to add the `-gmerge` flag after `-merge`, which will start a graphical merge: clearly not to our taste, but maybe yours? Note though that such an option can be more harmful than useful (refer to the section named *Evil twins* later in the chapter).
- The other is to add the `-abort` flag after it: this will go on handling all the trivial cases, leaving you with only the interactive ones.

The `findmerge.log` file then proves to be of value: it contains an executable shell script to replay the remaining commands (the part already completed being commented away). You'll have to edit it (for example, remove the `-abort` flags), but at the very least, it gives you the list of files to process. Here is an example of the `findmerge.log` file:

```
$ cat findmerge.log.2010-08-06T14:28:12+01:00
#cleartool findmerge .@@/main/22 -d -fver /main/t1/6 -log /dev/ #########
                                      null -fbtag myview -merge -nc -abort
# Skipping "./bbb.txt".
# Skipping "./ccc.txt".
# Skipping "./mak".
# The following element is invisible in the "base" view: ################
                              /view/myview/vob/test/.@@/foo/t1/4/Makefile.
# Findmerge will select base contributor. The following is not a valid ##
           common ancestor: /view/myview/vob/test/.@@/main/t1/4/Makefile.
#cleartool findmerge ./tc/Makefile@@/main/0 -fver /m/t1/5 -log /dev/ ####
                                      null -fbtag myview -merge -nc -abort
# The following element is invisible in the "base" view: ################
                             /view/myview/vob/test/.@@/main/t1/4/pathconv.
# Findmerge will select base contributor. The following is not a valid ##
          common ancestor: /view/myview/vob/testi/.@@/main/t1/4/pathconv.
```

```
#cleartool findmerge ./tc/pathconv@@/main/0 -fver /main/t1/4 -log /######
                                 dev/null -fbtag myview -merge -nc -abort
# Skipping "./tmp/a.txt".
```

Another useful option is `-fbtag`: specifying the current view will force trivial merge by explicitly setting the *base* contributors to match the *to* ones.

Note that this doesn't guarantee copy merges, in the case of a history of subtractive ones.

To guarantee this, one might use the `-exec` option. One problem with this option (common to its use with the `find` command) is that it will spawn a new invocation of the specified command for every match. For this reason, it is not a good idea to use it to invoke **cleartool** commands as most of the load (and thus of the time) will be spent in initializing and finalizing the cleartool processes.

There also is a `-co` flag, which comes handy. We use it here in conjunction with `-exec` (despite our strict understanding of the man page) to achieve the *copy* merge (we have to deal with directories first, and to cope with file names possibly containing spaces):

```
$ ct findmerge . -nc -type d -fve FOO -fbtag marc -merge
$ ct findmerge . -nc -type f -fve FOO -fbtag marc -co \
 -exec 'cp "$CLEARCASE_FXPN" "$CLEARCASE_PN"'
Needs Merge "./f2" [(automatic) to /main/mg/1 from /main/1 ##############
                                                   (base also /main/mg/1)]
Checked out "./f2" from version "/main/mg/1".
Needs Merge "./f1" [(automatic) to /main/mg/3 from /main/5 ##############
                                                   (base also /main/mg/3)]
Checked out "./f1" from version "/main/mg/3".
Log has been written to "findmerge.log.2010-08-02T20:32:48+01".
```

`findmerge` command may also be used to test the effect of command without actually running them. This is obtained with the `-print` option, which may be usefully augmented with `-whynot`.

Another possible use of the `-print` option is to produce lists of versions to be merged, for arbitrary post-processing. In this case, the `-short` addition is probably useful, to restrict the output exclusively to lists of contributors (*to* first).

There may however still be a few corner cases, involving directory merges, in which the output obtained with such dry runs could diverge from the result of actually effectuating the merges.

Evil twins

We already mentioned evil twins in Chapter 4, so why mention them again when speaking of merging?

The answer is that one cannot (easily) merge evil twins. Merging, contrarily to patching, concerns versions of the same element, and evil twins are precisely different elements.

The fact is that evil twins are for this reason often detected while merging, and more specifically, while merging whole directory trees. It comes as a surprise to many users that `findmerge` will fail to find suitable contributors, a situation they may show you with a screen dump but cannot explain.

It must be said that it is our experience that evil twins are more frequently produced by Windows GUI users than by command line UNIX ones. The reason is that the GUI puts in the hands of a naive user a "power" she doesn't master, offering an illusion of help, but on the contrary contributing to adding to the complexity. As we already mentioned, interactive decisions based on perception are not traceable.

How then to merge evil twins? The best tool is once again ***synctree!*** Importing directory trees from one view to another, using the powerful `-vreuse` and related options to avoid adding to an already regrettable chaos.

Summary—wrapping up

We hope that we could give some evidence of the complexity of merging. We believe that this complexity is typically underestimated, and that designing development processes based on branch patterns that imply a lot of merging is asking for trouble: it spreads this complexity to realms in which it doesn't belong, making the resulting complexity artificial.

We believe that rolling back changes gives a convincing example: it is incomparably simpler and more reliable to move back a label than to perform a subtractive merge. The apparent problem with the former is that the version belonging to the baseline, the one bearing the label, is after the rollback not the latest on the branch. Why should *this* be a problem? We can only find bad and artificial reasons: "Doctor, when I hit my head on the wall, it hurts".

If one decides to identify the baseline with labels, one doesn't care anymore on what branch the baseline version of any element *is* sitting: merging "back" to integration branches cannot be justified.

We met here a new set of reasons to stick to the advice we gave first in Chapter 5, and which we came back to already in Chapter 6—to publish in-place.