CASEVision™/ClearCase User's Guide

Document Number 007-2369-001

CONTRIBUTORS

Written by John Posner
Illustrated by John Posner
Production by Gloria Ackley
Engineering contributions by Atria Software, Inc.
Cover design and illustration by Rob Aguilar, Rikk Carey, Dean Hodgkinson,
Erik Lindholm, and Kay Maitz

CASEVision™/ClearCase User’s Guide
Document Number 007-2369-001

Contents

Examples

Figures

Tables

Chapter 1

Overview of ClearCase Usage

This chapter presents an overview of day-to-day CASEVision™/ClearCase usage, from the perspective of an individual user. (We do not deal with administrative issues.)

Before reading this chapter, be sure to set up your user environment according to the instructions in the” Preparing to Use ClearCase” chapter in the *CASEVision™/ClearCase Tutorial* manual. For additional orientation, work through the ClearCase Tutorial manual and read (at least) the first chapter in the *CASEVision™/ClearCase Concepts Guide*.

Finding Your Niche

The following sections present a “minimalist” procedure for getting up and running in your organization’s ClearCase environment.

Finding Your ClearCase Host

ClearCase must be installed on a host before you can use it there. (Simply accessing ClearCase executables with network pathnames does not work. Your own host must have certain data structures, including the ClearCase *multiversion file system* — the MVFS.) There is no single command that will tell you which hosts in your network have already been installed — check with your system administrator.

If ClearCase is already installed on your workstation, then the administrative directory */usr/adm/atria* (or */var/adm/atria*) will exist. In addition, you should be able to execute the *clearlicense* utility program:

```
% ls -d /usr/adm/atria
/usr/adm/atria
% clearlicense

License server on host "newton".
Running since Thursday 01/27/94 21:15:54.

LICENSES:
     Max-Users  Expires       Password [status]
        20         none       aaa.bbb.ccc [Valid]

 Maximum active users allowed: 20
  .
  .
```

If the administrative directory exists, but your shell cannot find this program, there is a problem with your shell's startup script. Consult the "Preparing to Use ClearCase" chapter in the *CASEVision™/ClearCase Tutorial* manual.

If you find a host where you would *like* to run ClearCase, but it's not currently installed there, see your system administrator and/or consult the CASEVision™/ClearCase Release Notes for installation instructions.

Locating Your Main Tools

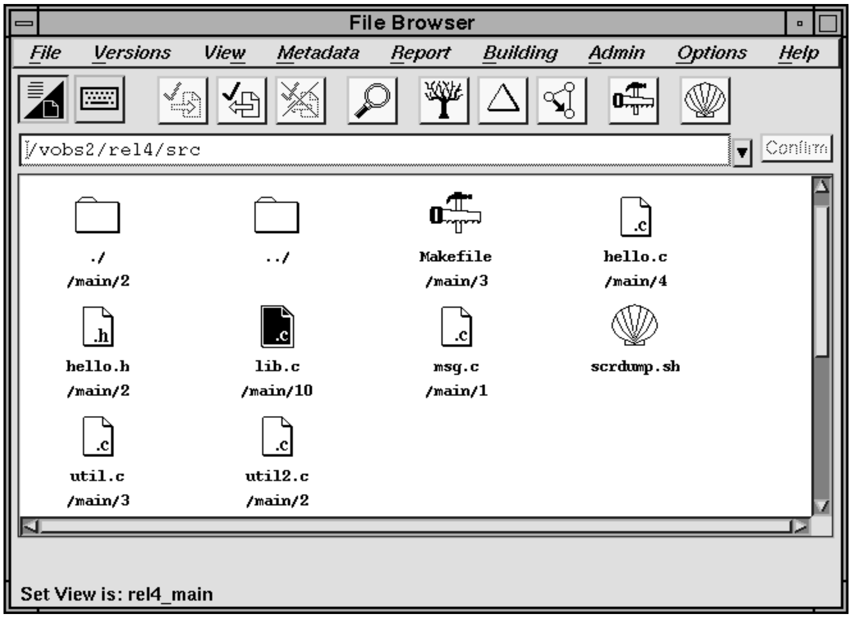
The tools in the ClearCase command line interface (CLI) and graphical user interface (GUI) are described in Chapter 2, "Using the ClearCase Command Line Interface," and Chapter 3, "Using the ClearCase Graphical User Interface.". For now, make sure that you can access the main CLI tool, *cleartool*, or the main GUI tool, *xclearcase*:

- Start a *cleartool* session, then enter a *quit* command to end the session.

  ```
  % cleartool
    cleartool> quit         (cleartool's interactive prompt)

  %
  ```

- Start an *xclearcase* session — the main panel appears, as in Figure 1-1:

```
% xclearcase
<prompt appears: Select view-tag — click the "Cancel"
button>
```



/vobs2/rel4/src

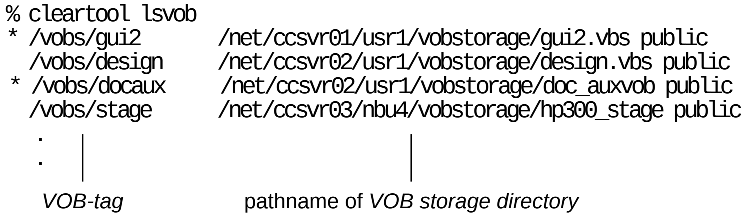
```
./          ../         Makefile    hello.c
/main/2                 /main/3     /main/4
hello.h     lib.c       msg.c       scrdump.sh
/main/2     /main/10    /main/1
util.c      util2.c
/main/3     /main/2
```

Set View is: rel4_main

Figure 1-1 *xclearcase* Main Panel

Locating Your Network's ClearCase Data Structures

All ClearCase data is stored in VOBs (versioned object bases, the "public" storage areas) and views (the "private" storage areas). They are all centrally registered, making it easy for you to determine their names. For example, the *cleartool* subcommand *lsvob* ("list VOB") shows the names of all of your networks VOBs (See Figure 1-2).



```
% cleartool lsvob
* /vobs/gui2       /net/ccsvr01/usr1/vobstorage/gui2.vbs public
  /vobs/design     /net/ccsvr02/usr1/vobstorage/design.vbs public
* /vobs/docaux     /net/ccsvr02/usr1/vobstorage/doc_auxvob public
  /vobs/stage      /net/ccsvr03/nbu4/vobstorage/hp300_stage public
   .
   .
```

VOB-tag
pathname of VOB storage directory

Figure 1-2 ”List VOB” command

Note: *xclearcase* has a “VOB Browser” for listing existing VOBs.

The asterisk (*) at the beginning of the line shown in Figure 1-2 indicates that the VOB is *active* on your host. The two pathnames for each VOB reflect that fact that it is activated by being *mounted* as a file system of type MVFS: the *VOB-tag* is the full pathname of the mount point on your host; the other pathname specifies the VOB’s actual location.

Figure 1-2 is typical: all VOBs are activated at (mounted on) locations in a single directory — here, */vobs*. This makes data structures that are actually distributed throughout the local area network all appear to be gathered together. Moreover, it is typical for some or all the VOBs to be linked together, effectively forming a single directory tree structure.

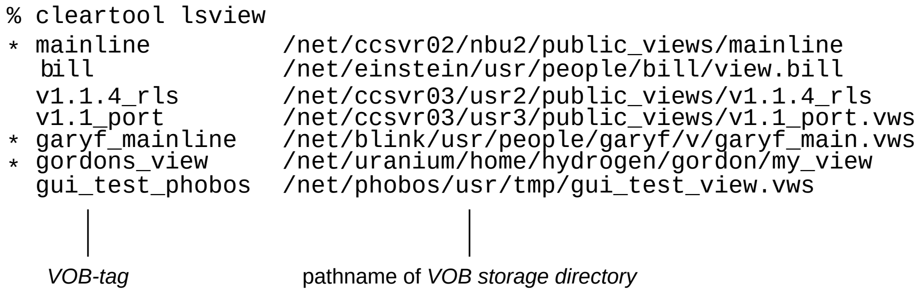
ClearCase has its own versions of the *mount* and *umount* commands, which allow non-*root* users to activate and deactivate *public* VOBs. For example, the *lsvob* listing in Figure 1-2 shows VOB-tag */vobs/design* to be public, but currently inactive. Any user can activate the VOB as follows:

```
% cleartool mount /vobs/design
```

Getting Yourself a View

Even if a VOB is active, you cannot access it directly. All user-level access to a VOB must go through a ClearCase *view*. (Certain administrative commands can process a VOB at its storage directory pathname.) Without a view, a VOB's mount point just appears to be an empty directory; but as seen through a view, a VOB appears to be an entire directory tree. Each file and directory in this tree in an *element*, which has a *version tree* containing all of its historical *versions* (Figure 1-4).

Just as you can list all of your network's VOBs, you can list all views: (See Figure 1-3.)



```
% cleartool lsview
* mainline          /net/ccsvr02/nbu2/public_views/mainline
   bill             /net/einstein/usr/people/bill/view.bill
  v1.1.4_rls        /net/ccsvr03/usr2/public_views/v1.1.4_rls
  v1.1_port         /net/ccsvr03/usr3/public_views/v1.1_port.vws
* garyf_mainline    /net/blink/usr/people/garyf/v/garyf_main.vws
* gordons_view      /net/uranium/home/hydrogen/gordon/my_view
  gui_test_phobos   /net/phobos/usr/tmp/gui_test_view.vws
```

Figure 1-3 "List Views" Commands

Note: *xclearcase* has a "View Browser" for working with existing views and creating new ones.

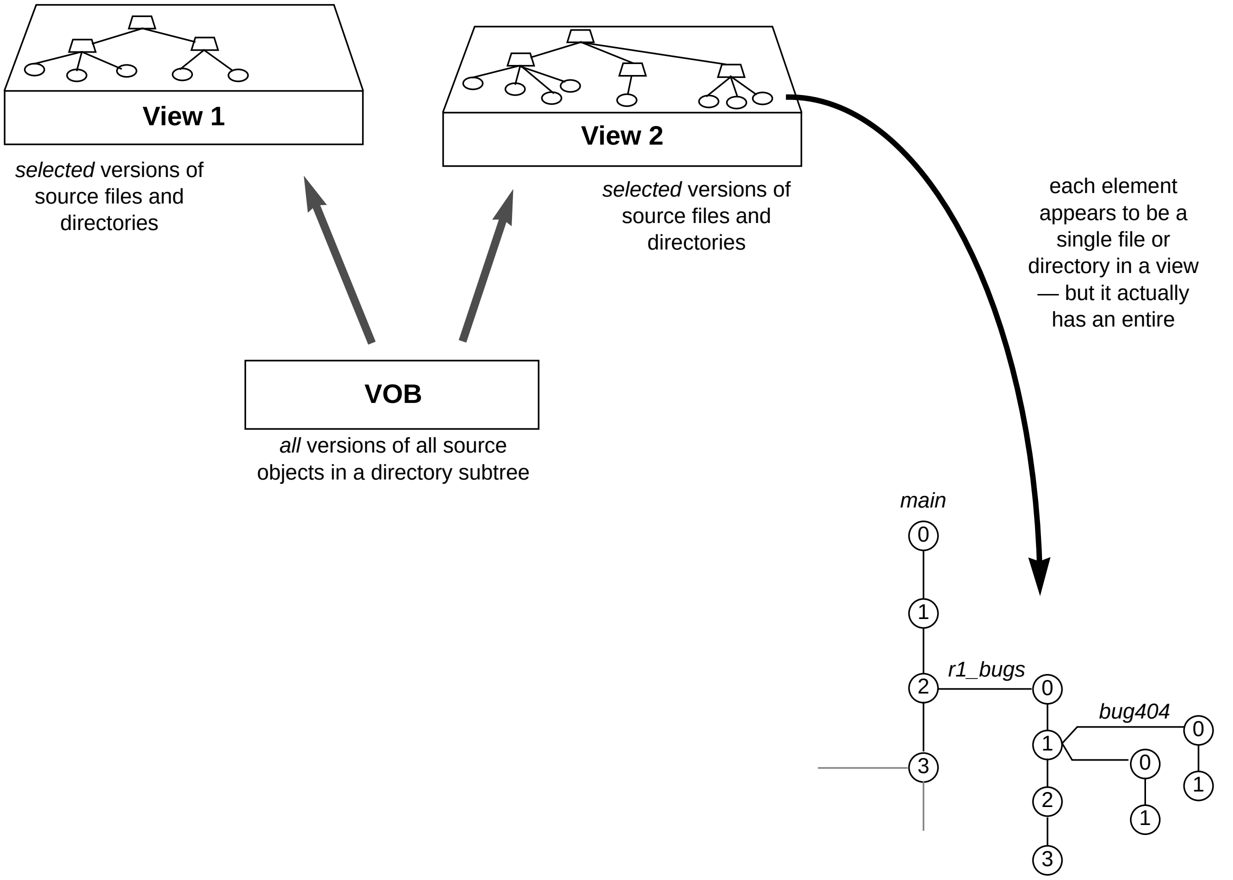

View 1
selected versions of source files and directories
View 2
selected versions of source files and directories
each element appears to be a single file or directory in a view — but it actually has an entire
VOB
all versions of all source objects in a directory subtree
main
0
1
2
3
r1_bugs
0
1
2
3
bug404
0
1
0
1

Figure 1-4 VOB as seen through view

Like a VOB, a view has a storage directory (its real location), but is accessed through a convenient *view-tag*. (A VOB-tag is a full pathname, because it is a mount point; but a view-tag is a simple name, because you can access it like a directory.) And like a VOB, a view must be explicitly *activated*; an active view is indicated by an asterisk in the *lsview* listing.

When a view is active on a host, it appears as a directory at a special location in the host's file system. In the ClearCase *viewroot* directory (usually */view*), the view-tags of all active views appear as subdirectories:

```
% ls -F /view
garyf_mainline/     gordons_view/     mainline/
```

We defer details on creating new views until Chapter 4, “Setting Up a View,” for now, let’s suppose that the existing view *gordons_view* is available for your use. The easiest way to use a view is to “set the view”. Setting a view creates a new shell process that you can use to work with any active VOB.

Going to a Development Directory

After setting a view, you can work with any VOB, much as if it were a standard directory tree:

- navigate with *cd*, *ls*, and so on
- view and edit files with *cat*, *more*, *vi*, *emacs*, and so on
- analyze files with *grep*, *sed*, *awk*, and so on

For example:

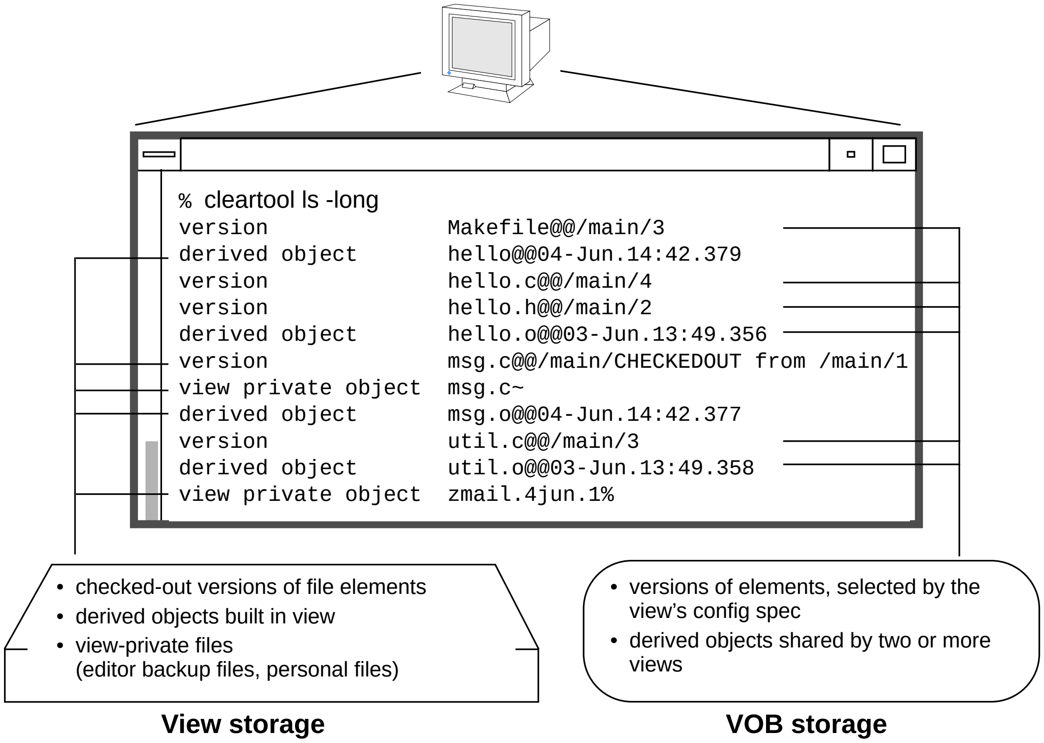
```
% cd /vobs/design              (go to the VOB-tag of any VOB
                                             —its mount point)
% ls -F                                     (what’s there?..)
% <no output>                          ..it appears to be empty)

% cleartool setview gordons_view                 (set a view)

% ls -F                                        (try again ..)
bin/   include/   lost+found/   (..the VOB’s contents appear)
src/   test/
  .
%
```

The View as Virtual Workspace

The reason you must use a view to work with VOBs stems from the two essential services provided by a view:

- **Version selection** — All of an element's versions are potentially accessible through the element's standard pathname. The view uses the rules in its *config spec to* select one of the versions. This is ClearCase's *transparency* feature — a view makes a VOB appears to be a standard directory tree to system software and third-party applications.
- **Private storage** — Each view has its own data storage area, enabling you (and other users of the same view) to perform development work without interfering with users working in other views — even those working with the same source elements and building the same libraries and executables.

For each VOB, a view presents a coherent *virtual workspace* — a directory tree in which you see both VOB-resident objects (the selected versions of elements) and view-resident objects (typically, the source files you're revising and the derived objects produced by your builds in that view). In general, the VOB-resident objects are read-only; the view-resident objects are writable and removable. Figure 1-5 illustrates the virtual workspace.



```
% cleartool ls -long
version              Makefile@@/main/3
derived object       hello@@04-Jun.14:42.379
version              hello.c@@/main/4
version              hello.h@@/main/2
derived object       hello.o@@03-Jun.13:49.356
version              msg.c@@/main/CHECKEDOUT from /main/1
view private object  msg.c~
derived object       msg.o@@04-Jun.14:42.377
version              util.c@@/main/3
derived object       util.o@@03-Jun.13:49.358
view private object  zmail.4jun.1%
```

• checked-out versions of file elements
• derived objects built in view
• view-private files (editor backup files, personal files)
• versions of elements, selected by the view's config spec
• derived objects shared by two or more views
View storage
VOB storage

Figure 1-5 View as a Virtual Workspace

When you (or your compiler) read a source file, you do not need to know whether you are accessing a version selected from VOB storage or a view-private file. Similarly, when you (or your linker) read an object module, you do not need to know whether you are accessing a shared (*winked-in*) binary from VOB storage or an unshared binary that was built in your view, and appears only in your view.

Standard and Extended Pathnames

Transparency enables you (and your *makefiles*, scripts, and other tools) to use standard pathnames to access ClearCase data. But you can also use *extended pathnames* — ClearCase extends the standard operating system file namespace both “upward” and “downward”.

View-Extended Pathnames

Typically most of your work involves just one view — you work in one or more processes that are “set” to that view. But you can also use other views that are active on your host, without having to “set” them. Instead, you can use a *view-extended pathname*. For example:

/vobs/design/src/msg.c
specifies the version of an element selected by your view

/view/bill/vobs/design/src/msg.c
specifies the version of the same element selected by view *bill*

/view/v1.1_port/vobs/design/src/msg.c
specifies the version of the same element selected by view *v1.1_port*

Conceptually, the *viewroot* directory is a “super-root” for your host’s file system. Through the super-root, you can access any active view; and through a view, you can access any active VOB (Figure 1-6). Thus, the view-extended pathname extended the file namespace upward.

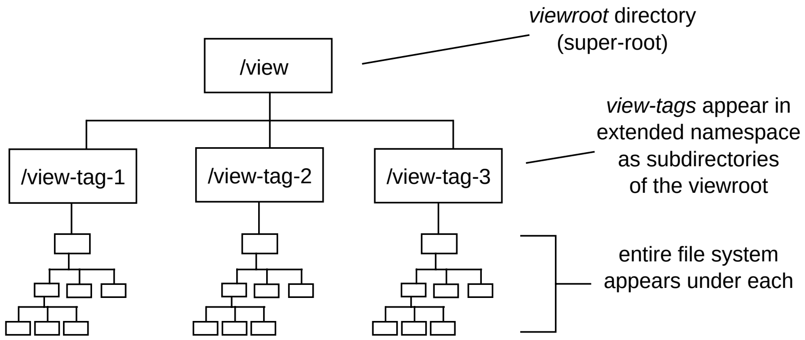


Figure 1-6 Viewroot Directory as Super-Root

VOB-Extended Pathnames

Each view selects only one object at a given pathname. Through your view, you will see a particular version of a source file — say *msg.c*. But the VOB also contains all other historical versions of the element, all of them potentially accessible through the file name *msg.c*. Similarly, your view sees a single derived object *msg.o* — the one produced when you compile *msg.c* using clearmake. But there may be other derived objects named *msg.o* in other views, built from different versions of *msg.c*, built using different header files, built using different command-line options, and so on.

ClearCase has a *VOB-extended pathname* scheme, which enables you to:

- access *any* version of an element, no matter which version is selected by your view. You can also reference other components of an element's version tree: its branches, and the element itself (Figure 1-7a).
- access *any* derived object, even if it is not the one produced by clearmake in your view (Figure 1-7b).

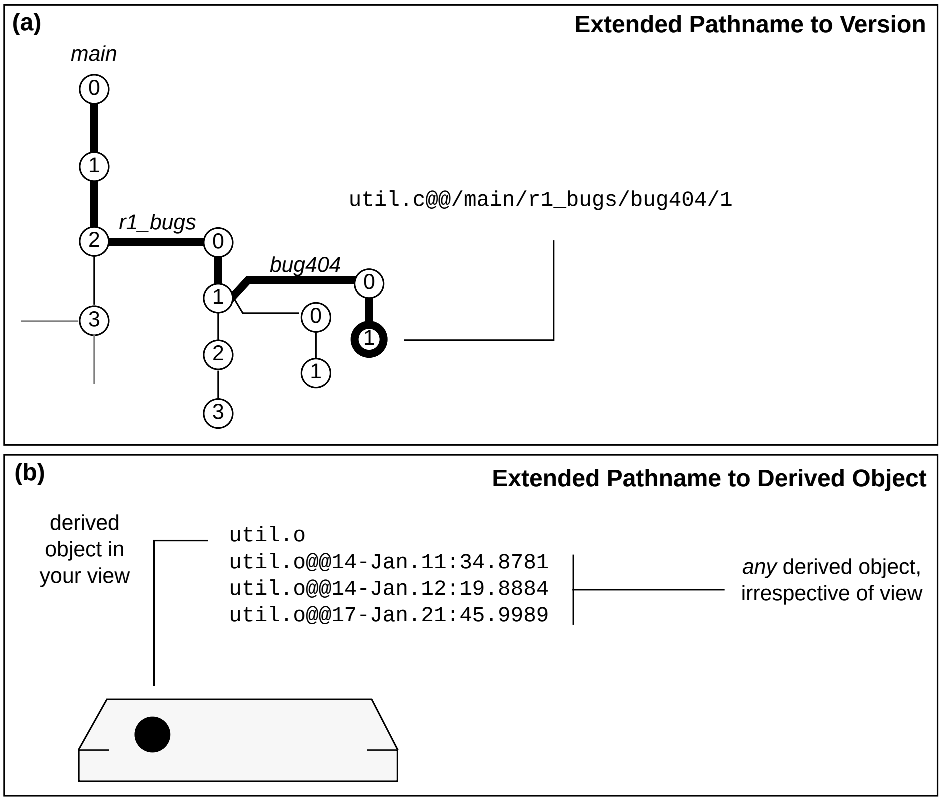

(a)
Extended Pathname to Version
main
0
1
2
3
r1_bugs
0
1
2
3
bug404
0
1
0
1
util.c@@/main/r1_bugs/bug404/1
(b)
Extended Pathname to Derived Object
derived object in your view
util.o
util.o@@14-Jan.11:34.8781
util.o@@14-Jan.12:19.8884
util.o@@17-Jan.21:45.9989
any derived object, irrespective of view

Figure 1-7 VOB-Extended Pathnames

An element’s version tree has the same hierarchical structure as a directory tree. This makes it natural to “embed” the entire version tree in the file system under the element’s pathname:

msg.c standard name of an element (your view accesses a particular version through ClearCase’s transparency feature)

msg.c@@ extended pathname to the element object

msg.c@@/main
extended pathname to the element’s *main* branch

msg.c@@/main/alpha_port
extended pathname to a subbranch of the *main* branch

msg.c@@/main/alpha_port/5
extended pathname to a version on the subbranch

Note: These are also called *version-extended pathnames*, because they indicate locations within an element's version tree.

The VOB catalogs all derived objects built at a given pathname. They have unique IDs, which incorporate timestamps:

msg.o@@24-Nov.21:11.8718
derived object built as *msg.o* on Nov 24 at 9:11 pm

msg.o@@25-Nov.07:31.8834
derived object built as *msg.o* on Nov 25 at 7:31 am

msg.o@@12-Jan.21:59.9501
derived object built as *msg.o* on Jan 12 at 9:59 pm

In both kinds of VOB-extended naming, think of the extended naming symbol (@@) as "turning off" transparency, allowing you to specify a particular object in the VOB database.

For a complete discussion of ClearCase's file-naming extensions, see the *pathnames_ccase* manual page.

Pathname Examples

This example in this section demonstrate how (and how well) ClearCase fits into a standard UNIX development environment. The examples involve both standard pathnames and extended pathnames, all processed by standard UNIX programs and by ClearCase commands:

- Display the version of file msg.c selected by your view:

  ```
  % cat msg.c
  ```

- Display the 5th version on the *main* branch of file *msg.c*:

  ```
  % cat msg.c@@/main/5
  ```

- Display the most recent version on the *main* branch of file *msg.c*:

  ```
  % cat msg.c@@/main/LATEST
  ```

- Compare your version of file *msg.c* with the version in a colleague's view:

  ```
  % diff msg.c /view/jjkim/vobs/design/src/msg.c
  ```

- Compare your version of file *msg.c* with a particular historical version:

  ```
  % diff msg.c msg.c@@/main/5
  ```

- Search for the string *tmpbfr_sz* in *all* the versions on the *main* branch of file *msg.c*:

  ```
  % grep 'tmpbfr_sz' msg.c@@/main/*
  ```

- Repeat the preceding search in another way — by going to the *main* branch of *msg.c,* then entering the *grep* command:

  ```
  % cd msg.c@@/main
  % grep 'tmpbfr_sz' *
  ```

- Search through an element's entire version tree for that same string:

  ```
  % find msg.c@@ -print -exec grep 'tmpbfr_sz' {} \;
  ```

The last two examples demonstrate that the embedding of version trees in the file namespace is "complete" — you can navigate elements, their branches, and their versions with standard operating system commands, just as if they were regular directories and files.

Modifying Elements - the Checkout/Checkin Model

This section describes the ways in which you "evolve" an element by adding new versions and branches to its version tree. Like many version-control systems, ClearCase uses a "checkout/checkin" model:

1. **Before you begin** — In the "steady state", an element is read-only — you can neither edit it nor remove it with standard operating system commands:

   ```
   % ls -l hello.c
   -r--r--r--   1 akp      user      168 May 13 19:30 hello.c
   ```

What you are seeing is one version of the file element — the version selected by your view, according to the rules in its config spec. Typically, it is the most recent version on some branch of the element's version tree.

2. **Checkout** — You issue a *checkout* command, naming the file. This produces an editable copy of the selected version

   ```
   % cleartool checkout -nc hello.c
   Checked out "hello.c" from version "/main/2".
   ```

   ```
   % ls -l hello.c
   -rw-rw-r--   1 sakai    user      168 May 19 19:31 hello.c
   ```

   The editable copy appears "in place", at the same pathname as the element — there is no need to copy the file to another location in order to work on it. In Step #1, your view selected an "old" version of the file, located in VOB storage; now it selects your *checked-out version*, located in view-private storage.

   Note: The listings above hint at this: the element was created by another user, *akp*, who owns all its "old" versions. But the checked-out version belongs to the user who performs the checkout — in this example, sakai.

3. **Edit** — You revise the contents of the file with any text editor.

   ```
   % vi hello.c
   ```

   ClearCase is integrated with the popular development-tool messaging systems, SoftBench and ToolTalk. You can use a text editor that works with either of these systems to work with your checked-out version.

4. **Checkin** — When the file is correct (or, at least, worth preserving), you issue a *checkin* command. This adds a new version to the version tree (the *successor* to the version that was checked out), and removes the editable copy of the file. You can specify a comment during the checkin process, to help document the changes you made.

   ```
   % cleartool checkin hello.c
   Comment for all listed objects:
   replaced message
   .
   Checked in "hello.c" version "/main/3".
   ```

   Canceling a Checkout Instead of Performing a Checkin — If you decide that you don't want to modify the file after all, you can cancel the checkout with an *uncheckout* command.

5. **After you end** — After you checkin a new version, the file reverts to its "steady-state" read-only status:

   ```
   % ls -l hello.c
   -r--r--r--   1 akp      user      233 May 19 19:44 hello.c
   ```

   The checked-in version is placed in VOB storage, and immediately becomes shared data, available to all users. In particular, *your* view now selects this newly-created version. (Since you no longer have the file checked out, your view reverts to selecting a VOB-resident version.)

Reserved and Unreserved Checkouts

In some version-control systems (for example, SCCS), only one user at a time can reserve the right to create the next version on a branch. In other systems, many users can compete to create the same new version. ClearCase supports both models by allowing two kinds of checkouts, *reserved* and *unreserved*:

- Only one *view* at a time can have a *reserved checkout* of a particular branch. A view with a reserved checkout has the exclusive, guaranteed right to extend the branch with a new version.

  After you perform the checkin, you no longer have any exclusive rights on that branch. Another user can now perform a reserved checkout to "grab" the right to create the next version on the branch.

- Many views can have *unreserved checkouts* of the same branch. Each view gets its own private copy of the most recent version on the branch; each copy can be edited independently of all the others.

  An unreserved checkout does not guarantee the right to create a successor version. If several views have unreserved checkouts of the same branch in different views, the first user to perform a checkin "wins" — other users must *merge* the checked-in changes into their own work before they can perform a *checkin*. (See "Scenario: Merging an Unreserved Checkout" on page 118.)

By default, the *checkout* command performs a reserved checkout; use *checkout –unreserved* to perform an unreserved checkout. The *reserve* and *unreserve* commands change the state of a checkout. Figure 1-8 illustrates checked-out versions created by reserved and unreserved checkouts, along with the effect of subsequent checkins.

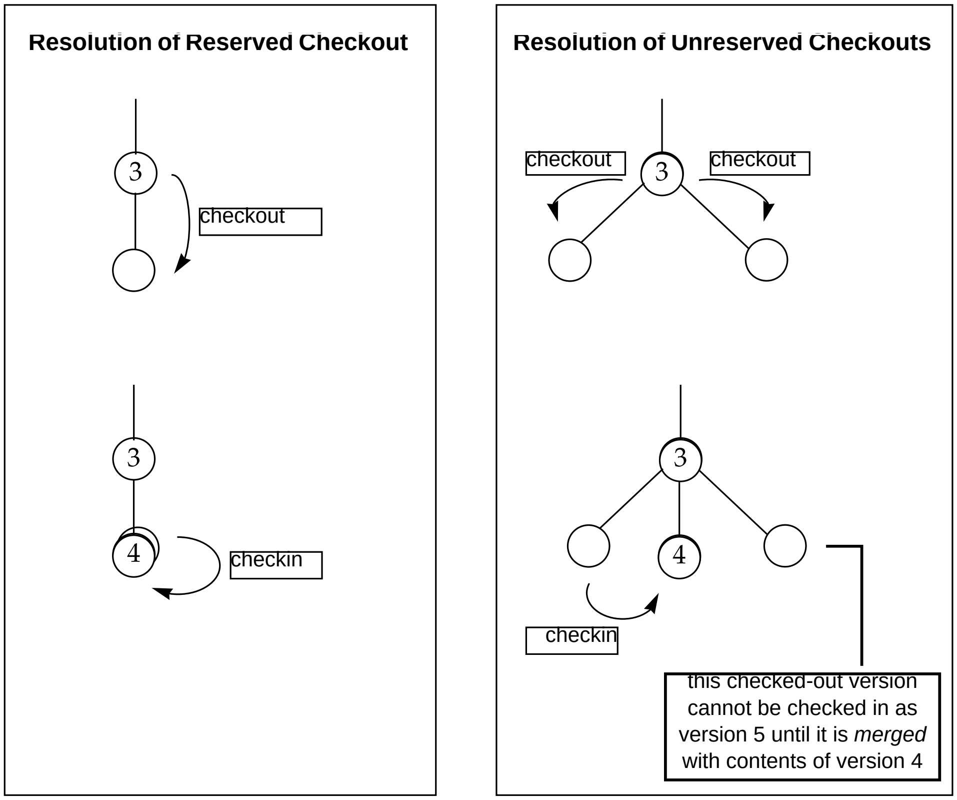

Resolution of Reserved Checkout
3
checkout
3
4
checkin
Resolution of Unreserved Checkouts
checkout
3
checkout
3
4
checkin
this checked-out version cannot be checked in as version 5 until it is *merged* with contents of version 4

Figure 1-8 Resolution of Reserved and Unreserved Checkouts

Tracking Checked-Out Versions

In a multiuser environment, it is very likely that a given element will have several checkouts at the same time:

- Several branches of the element may be under development; checkouts on different branches are mutually independent.
- As described in “Reserved and Unreserved Checkouts” on page 16 above, there can be multiple concurrent checkouts of a single version.

The *lscheckout* (“list checkouts”) command lists all the current checkouts of one or more elements:

```
% cleartool lscheckout -long sort.9

04-Mar-94.12:12:33     Allison K. Pak (akp.user@neon)
  checkout version "sort.9" from /main/37 (reserved)
  by view: "neon:/net/neon/home/akp/views/930825.vws"

26-Feb-94.08:59:02     Derek R. Philips (drp.user@saturn)
  checkout version "sort.9" from /main/gopher_port/8 (reserved)
  by view: saturn:/net/saturn/home/drp/mainvu.vws
  "incorporate david’s comments"
```

Note: In the example for *cleartool lscheckout* the same element is checked out on different branches in diferent views.

Checked-Out Versions - Ownership and Accessibility

As the *lscheckout* listing above indicates, your checked-out version belongs both to you and to your view:

- As the user who performed the *checkout*, you are the one who has permission to perform a corresponding *checkin* or *uncheckout*. (The *root* user also has permission, as does the owner of the element and the owner of the entire VOB.)
- You are the *owner* of the standard UNIX file in view-private storage that is the checked-out version. This file is created according to your current *umask*(1) setting, in the standard UNIX manner. Standard mechanisms also control whether other users, working in the same view, can read or write the checked-out version.
- A view can see only one object at a given pathname. Thus, ClearCase allows each view to have at most *one* checkout of a given element. If an element is to be checked out twice, on two different branches, then the checkouts must be performed in different views.

Users in other views do not see the checked-out version — they continue to see the version selected by their views’ config specs. They can use the *lscheckout* command to determine that a checkout has been performed and, if permissions allow, they can use a view-extended pathname to access the checked-out version.

Checkout and Checkin as Events

The *lscheckout* command determines all of an element's checkouts by examining *event records,* which are stored in the VOB database of that element. Each checkout command creates a *checkout version* event record; *lscheckout* lists some or all such event records.

Similarly, the *checkin* command writes a *create version* event record to the appropriate VOB database. In general, every ClearCase operation that modifies a VOB creates an event record in the VOB's database, capturing the "who, what, when, where, why" of the operation: login name of the user who entered the command, kind of operation, date-time stamp, hostname, user-supplied comment.

You can use the *lshistory* command to display some or all of the event records for one or more elements:

```
% cleartool lshistory util.c
25-May-92.15:45:19     Allison K. Pak (akp.user@neptune)
  create version "util.c@@/main/3" (REL3)
  "special form of username message for root user
   merge in fix to time string bugfix branch"
25-May-92.15:44:05     Derek R. Philips (drp.user@saturn)
  create version "util.c@@/main/rel2_bugfix/1"
  "fix bug: extra NL in time string"
25-May-92.15:43:03     Derek R. Philips (drp.user@saturn)
  create version "util.c@@/main/rel2_bugfix/0"
25-May-92.15:43:03     Derek R. Philips (drp.user@saturn)
  create branch "util.c@@/main/rel2_bugfix"
25-May-92.14:46:21     Allison K. Pak (akp.user@neptune)
  create version "util.c@@/main/2"
  "shorten HOME string"
```

The *chevent* command can modify the comment string stored in an event record:

```
% cleartool chevent -replace util.c@@/main/2
Comments for "util.c":
shorten HOME string, to comply with
AMOK guidelines
.
Modified event of version "util.c".
```

Building and Testing Software

One of ClearCase's principal design points is compatibility with your existing software-build procedures. If you use *makefiles* to organize your build procedures, you can use the clearmake build utility, either directly or using the *xclearcase* front-end. If your build procedures are implemented as shell scripts or other programs, you can use the *clearaudit* build utility.

In many cases, you may find that adapting your day-to-day build habits to ClearCase involves little more than switching from ...

```
% make target-name
... to ...
```

```
% clearmake target-name
```

Behind the scenes, ClearCase manages the results of builds:

- Newly-built files are cataloged as *derived objects* (DOs) in the appropriate VOB databases.
- *Configuration records* (CRs) are also stored in VOB databases, to record exactly how each derived object was built. Like an event record, a CR contains "who, what, when, where" information. It also contains a "bill of materials" that shows how the file was built: versions of source elements used in the build, build options, makefile macros, build script, and more.
- Based on a configuration-record analysis, clearmake may decide to *wink-in* an existing derived object (essentially, create a link to it) rather than executing a build script to create a new derived object.

All of this occurs automatically, though you can suppress features individually, using clearmake command-line options.

Using Build Management Structures

In some situations, you may find it useful to examine the structures ClearCase uses for build management. For example, you can use the *lsdo* command or the ClearCase variant of the *ls* command to see the unique identifiers with which derived objects are cataloged:

```
% cleartool ls hello.o
hello.o@@08-Mar.12:48.7261

% cleartool lsdo hello.o
08-Mar.12:48   akp        "hello.o@@08-Mar.12:48.7261"
07-Jan.11:40   sakai      "hello.o@@07-Jan.11:40.2143"
```

Note: *cleartool ls* is done in your view and *cleartool lsdo* is all DO's built at that pathname in any view.

The *catcr* command displays the contents of the configuration record that documents the building of one or more derived objects:

```
% cleartool catcr hello.o
Target hello.o built on host "neptune" by akp.user
Reference Time 19-May-92.19:30:12, this audit started
19-May-92.19:30:13
View was neptune:/home/akp/akp.vws
Initial working directory was neptune:/usr/hw/src
----------------------------
MFS objects:
----------------------------
/usr/hw/src/hello.c@@/main/4 <19-May-92.19:30:05>
/usr/hw/src/hello.h@@/main/2 <19-May-92.19:30:07>
/usr/hw/src/hello.o@@19-May.19:30.364
----------------------------
Build Script:
----------------------------
  cc -c hello.c
----------------------------
```

There is also a *diffcr* command, which you can use to compare two builds of the same target — that is, to compare the CRs of two derived objects built at the same pathname.

You can “grab” any existing derived object for use in your view, even if does not match your current configuration of source versions. That is, you can explicitly *wink-in* a derived object, even if clearmake wouldn’t.

The Chapter 10, “Building with clearmake; Some Basic Pointers,” section of this manual discusses build-related issues in greater detail.

Debugging and Testing Software

The best environment for debugging and testing a software build is the view in which you performed it. All source versions that went into a build are visible in the view; likewise, all object modules (.o files) are visible.

ClearCase does not include any specific debugging tools. However, integrations with a third-party tools, such as Centerline’s CodeCenter™, may be available for your platform.

Working in a Parallel Development Environment

ClearCase is designed for *parallel development*, wherein two or more projects can modify the same source files at the same time. The “standard” strategy for organizing the environment uses a baselevel-plus-changes model. Launching and pursuing a new development project involves the steps listed below. (You’ll notice that most of the work is administrative.)

1. An administrator defines a *baselevel* — a consistent set of source versions — by attaching the same version label to all the source versions — for example, *RLS_2.1*. The baselevel might be the set of versions that went into some product release; or it might just be a set of versions that yields a functional build of the software system.
2. The administrator designates a particular branch for use by the development project — for example, branch *eco78* for performing the fixes required for ECO #78 to Release 2.1
3. The administrator publishes a config spec for use by all developers working on the project. This config spec represents the project’s organization in terms of the chosen version label and branch (*RLS_2.1* and *eco78*).

4. You, the developer, create a view and configure it with the published config spec. (*Alternatives:* all developers share a single view with the proper configuration; the administrator creates the view(s), then tells developers to use them.)
5. Working in the properly-configured view, you start modifying an element simply by entering a *checkout* command; the checkout automatically takes place on the designated branch, *eco78*; if the branch does not already exist, it is created at the version labeled *RLS_2.1*. After you build and test with your changes, your *checkin* command creates a new version on the designated branch.

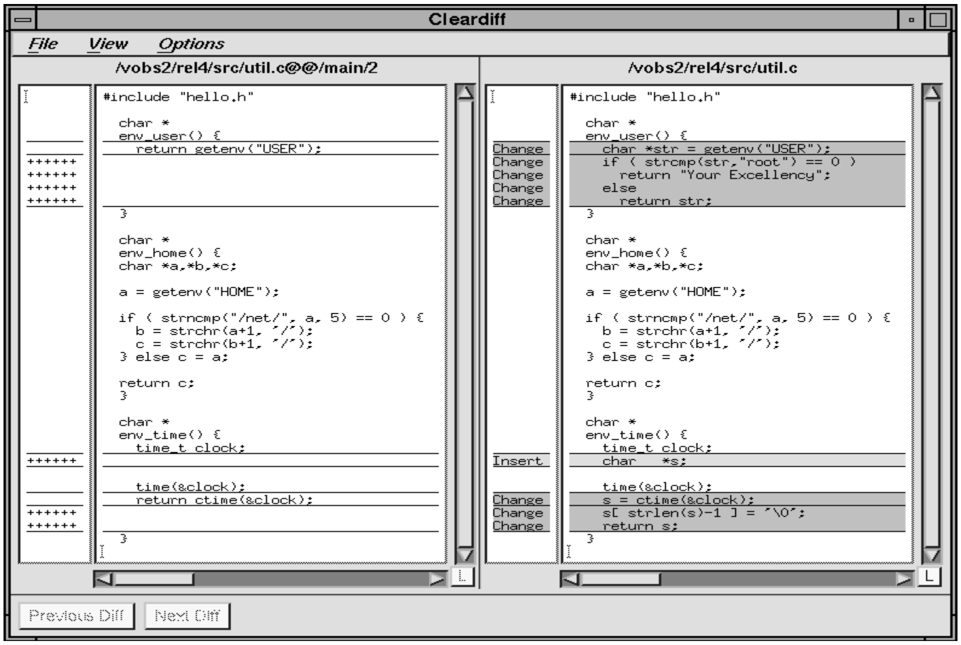
Thus, after some initial setup, "working on a branch" involves nothing special; you just work according to the basic *checkout-edit-checkin* scheme described in "Modifying Elements - the Checkout/Checkin Model" on page 14. Your view takes care of organizing your work according to the administrator-mandated structure.

Comparing Versions of Elements

As you modify source files, you'll often want to perform comparisons:

- What are all the changes I've made in my checked-out version?
- How does my checked-out version differ from a particular historical version, or from the version being used by one of my colleagues?

ClearCase includes powerful tools for comparing two or more versions of an element. For example, Figure 1-9 shows how you might display the changes you've made in a source file.



/vobs2/rel4/src/util.c@@/main/2

```
       #include "hello.h"

         char *
         env_user() {
           return getenv("USER");
++++++
++++++
++++++
++++++
         }

         char *
         env_home() {
         char *a,*b,*c;

         a = getenv("HOME");

         if ( strncmp("/net/", a, 5) == 0 ) {
           b = strchr(a+1, '/');
           c = strchr(b+1, '/');
         } else c = a;

         return c;
         }

         char *
         env_time() {
           time_t clock;
++++++

           time(&clock);
           return ctime(&clock);
++++++
++++++
         }
```

/vobs2/rel4/src/util.c

```
       #include "hello.h"

         char *
         env_user() {
Change     char *str = getenv("USER");
Change     if ( strcmp(str,"root") == 0 )
Change       return "Your Excellency";
Change     else
Change       return str;
         }

         char *
         env_home() {
         char *a,*b,*c;

         a = getenv("HOME");

         if ( strncmp("/net/", a, 5) == 0 ) {
           b = strchr(a+1, '/');
           c = strchr(b+1, '/');
         } else c = a;

         return c;
         }

         char *
         env_time() {
           time_t clock;
Insert     char    *s;

           time(&clock);
Change     s = ctime(&clock);
Change     s[ strlen(s)-1 ] = '\0';
Change     return s;
         }
```

Figure 1-9 Determining the Changes in a Checked-Out Version

To produce a similar display comparing your version of a file with a colleague's (say, the version that appears in view *gordons_view*), you might enter this command:

```
% cleartool xdiff base.h \
 /view/gordons_view/vobs/proj/include/base.h
```

In addition to simply comparing versions, ClearCase can produce a line-by-line analysis of a version. The *annotate* command indicates when each line of a file was added, and by whom ().

```
/vobs/atria/bin/clearapropos
---------------
11-Apr-94 jjp        /main/5 (MS_V1.BL5_BASE, MS_V1.BL5, V2.BL7, ...)
implement -glos-sary option
 .
 .
-------------------------------------
-------------------------------------
###   30-Mar-94 jjp      /main/4  | ###
###                     .         | ### clearapropos
###                     .         | ###
###                     .         |
###                     .         | # This script implements the cleartool apropos command ...
###   23-Feb-94 leblang  /main/1  | # the cleartool MAN page title lines for help topics.
###                     .         |
###   29-Mar-94 jjp      /main/2  | #------------------------------------------------------
###   30-Mar-94 jjp      /main/4  | # setup
###                     .         | #
###   29-Mar-94 jjp      /main/2  |
###                     .         | if [ "$ATRIA_DEBUG_SCRIPT" ] ; then set -x ; fi
###                     .         |
###                     .         | ARCH=`uname -s`-`uname -r`
###                     .         | case $ARCH in
###                     .         |    HP-UX-A.09.*) NAWK=/usr/bin/awk       ;;
###                     .         |    IRIX-5.*)     NAWK=/usr/bin/nawk      ;;
###                     .         |    SunOS-4.*)    NAWK=/bin/nawk          ;;
```

Figure 1-10 Annotated File Listing

Examining an Element's Version Tree

ClearCase includes both character-oriented and graphical tools for examining the version tree of an element. A graphical display, like that shown in , is "live" — for example, you can select one or more versions with the mouse, then click an icon to bring up a window that compares those versions.

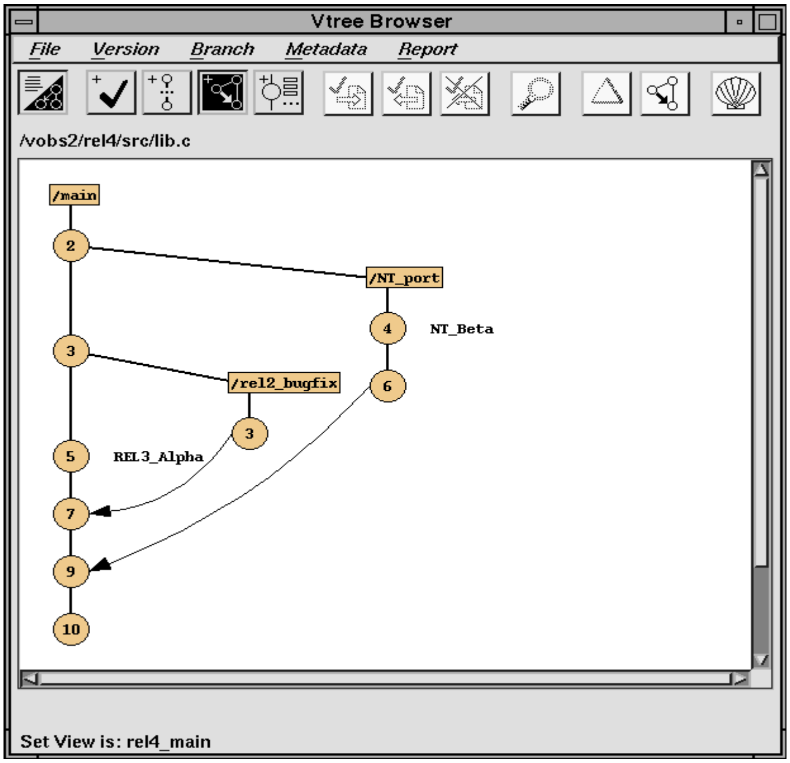


Figure 1-11 Graphical Version Tree Display

Merging Versions of Elements

Typically, an element's subbranches are thought of as "temporary". One branch — usually the *main* branch — is conceived as holding the element's "official" or "permanent" contents. In this scheme, work performed on each subbranch must eventually be merged back into the *main* branch. You may wish to wait until a project is finished before merging its branches back into the *main* branch. Or you may wish to perform frequent merges, in order to keep the contents of the branches from diverging too much.

A typical merge combines the most recent version on a subbranch with the most recent version on the *main* branch (Figure 1-12). In a real-world environment, an element is also likely to be involved in other projects, taking place on one or more additional branches.

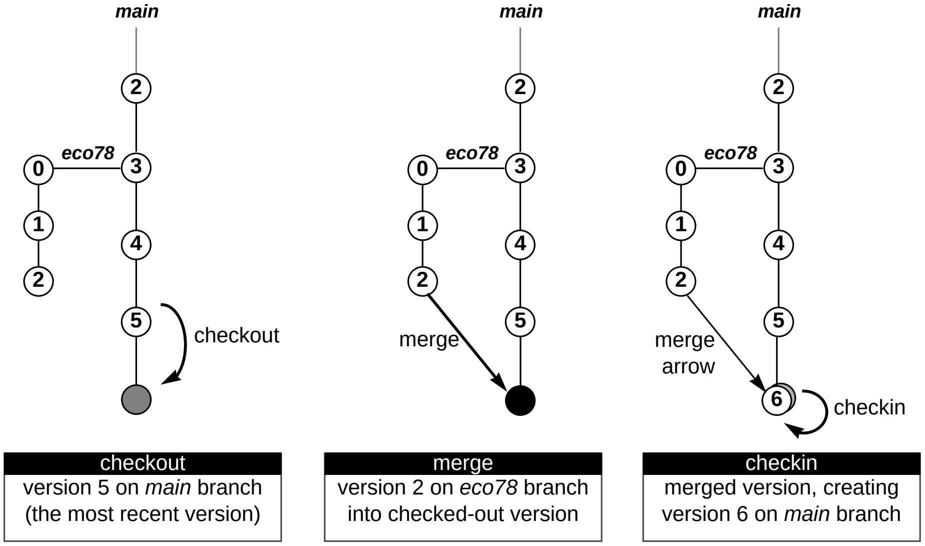

main
eco78
0
1
2
2
3
4
5
checkout
main
eco78
0
1
2
2
3
4
5
merge
main
eco78
0
1
2
2
3
4
5
6
merge arrow
checkin
checkout
version 5 on *main* branch (the most recent version)
merge
version 2 on *eco78* branch into checked-out version
checkin
merged version, creating version 6 on *main* branch

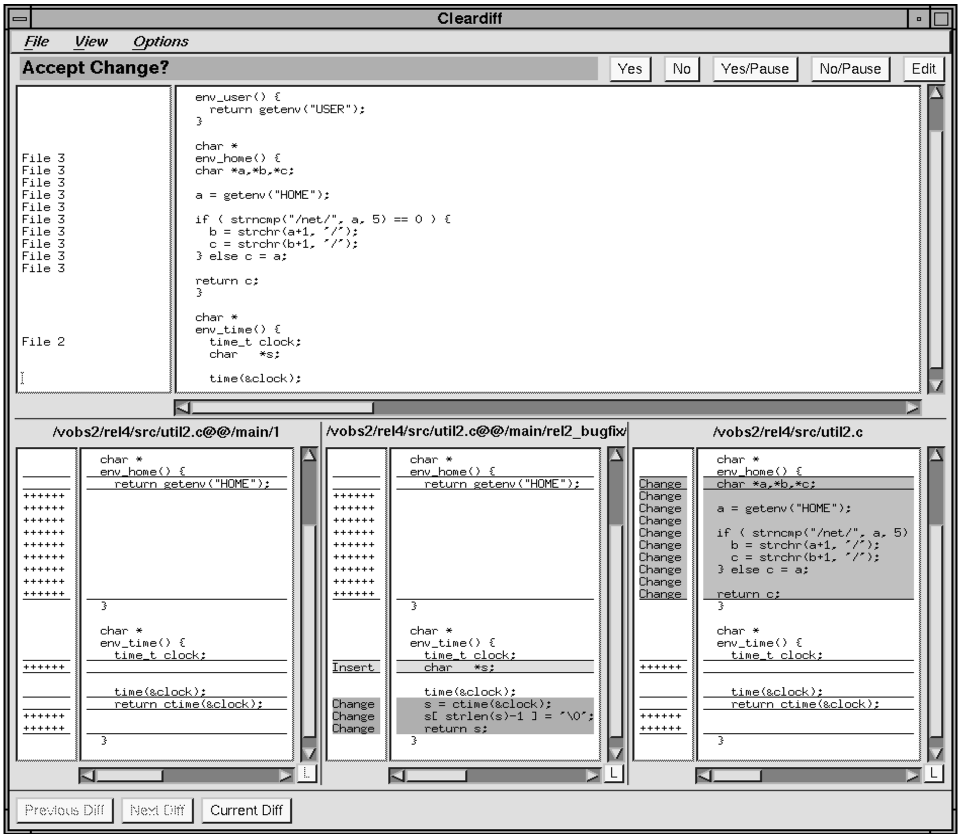
Figure 1-12 Merge of Subbranch Version into Main Branch

ClearCase makes merging as automatic as possible. A single *findmerge* command (“find, and then merge”) might merge all of a project’s changes, made on a subbranch, back into the *main* branch:

```
% cd /vobs/proj    (go to project top-level source directory)
        (merge all work on subbranch back into ’main’ branch)

% cleartool findmerge . -ftag eco_work_view -merge
```

For each merge it performs, ClearCase determines how each of the *contributor* versions have changed from the *base* version (in Figure 1-12, how contributor versions */main/5* and */main/eco78/2* have changed from the base version */main/3*). Often, all such changes are mutually distinct, and a merged version is created completely automatically. If there are any conflicts between the changes, you can resolve them using the graphical merge tool (Figure 1-13) or a character-oriented tool.

Whenever you merge two versions of an element, ClearCase annotates the element's version tree with a *merge arrow*. (Figure 1-13 shows a merge arrow connecting version */main/eco78/2* to version */main/6*.) These annotations make it possible to merge a project's changes a few files at a time, and to determine whether or not all required merges have been performed. Merge operations take into account any existing merge arrows involving the same branches; this makes frequent incremental merging both fast and simple.



```
                 env_user() {
                   return getenv("USER");
                 }

                 char *
File 3           env_home() {
File 3           char *a,*b,*c;
File 3
File 3           a = getenv("HOME");
File 3
File 3           if ( strncmp("/net/", a, 5) == 0 ) {
File 3             b = strchr(a+1, '/');
File 3             c = strchr(b+1, '/');
File 3           } else c = a;
File 3
                 return c;
                 }

                 char *
                 env_time() {
File 2             time_t clock;
                   char   *s;

                   time(&clock);
```

/vobs2/rel4/src/util2.c@@/main/1

```
          char *
          env_home() {
            return getenv("HOME");
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
          }

          char *
          env_time() {
            time_t clock;
++++++

            time(&clock);
            return ctime(&clock);
++++++
++++++
          }
```

/vobs2/rel4/src/util2.c@@/main/rel2_bugfix/

```
          char *
          env_home() {
            return getenv("HOME");
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
          }

          char *
          env_time() {
            time_t clock;
Insert      char   *s;

            time(&clock);
Change      s = ctime(&clock);
Change      s[ strlen(s)-1 ] = '\0';
Change      return s;
          }
```

/vobs2/rel4/src/util2.c

```
          char *
          env_home() {
Change    char *a,*b,*c;
Change
Change    a = getenv("HOME");
Change
Change    if ( strncmp("/net/", a, 5)
Change      b = strchr(a+1, '/');
Change      c = strchr(b+1, '/');
Change    } else c = a;
Change
Change    return c;
          }

          char *
          env_time() {
            time_t clock;
++++++

            time(&clock);
            return ctime(&clock);
++++++
++++++
          }
```

Figure 1-13 Graphical Merge Tool

Working with Meta-Data

In addition to storing *file system data* (source files, shared derived objects) in its storage pools, a VOB stores associated *meta-data* in its database. As you perform your development tasks, ClearCase automatically creates and stores a variety of meta-data. For example, preceding sections of this chapter have described how:

- ClearCase commands that modify a VOB (*checkout*, *checkin*, and others) write *event records* to document the change.
- The *clearmake* build utility creates *configuration records* to document software builds.
- Merge operations are documented by the creation of *merge arrows*.

In addition, meta-data annotations can be placed on objects explicitly. The most important example was discussed in "Working in a Parallel Development Environment" on page 22 — an administrator attaches version label annotations to a set of source file versions, in order to define a baselevel.

In many organizations, defining and attaching meta-data annotations is an administrator's or project leader's task; as an individual developer, you most often *use* existing meta-data annotations, rather than explicitly *creating* them.

Chapter 2

Using the ClearCase Command Line Interface

This chapter presents an overview of the principal programs in ClearCase's command-line interface: *cleartool* and *clearmake*.

Using *cleartool*

Note: Much of the information in this section is available on-line, in the *cleartool* manual page.

cleartool is the main CLI tool for interacting with your organization's data repository. *cleartool* has a rich set of subcommands, which create, modify, and manage the information in VOBs and views.

cleartool Subcommands

Table 2-1 lists all the *cleartool* subcommands, organized by function. The complete list can be quite daunting, because much of ClearCase's extensive feature set has been incorporated into this single tool. On a day-to-day basis, however, you'll probably use fewer than a dozen commands.

Table 2-1 *cleartool* Subcommands

***cleartool* Subcommands**	
Working with Views	
catcs	display configuration specification
edcs	edit configuration specification
ls	list VOB objects and view-private objects in a directory
lsprivate	list view-private objects
lsview	list view registry entries
mktag	create view-tag or VOB-tag
mkview	create and register a view
pwv	print working view
recoverview	recover a view database
reformatview	update the format of a view database
rmtag	remove a view-tag and unregister a view on the local host
rmview	remove a view storage area or remove view-related records from a VOB
setcs	set the configuration specification
setview	create a process that is set to a view
startview	start or connect to a view_server process
Working with Version Tree Structures	
checkin	create permanent new version of an element
checkout	create view-private, modifiable copy of a version
chtype	change the type of an element / rename a branch
describe	describe VOB object

Table 2-1 (continued) *cleartool* Subcommands

find	select objects from a directory hierarchy
ln	create VOB hard link or VOB symbolic link
lsvtree	list version tree of an element
mkbranch	create a new branch in the version tree of an element
mkbrtype	create a branch type object
mkdir	create a directory element
mkelem	create an element
mkeltype	create an element type object
mv	move or rename an element or VOB link
reserve	convert a checkout to reserved status
rmbranch	remove a branch from the version tree of an element
rmelem	remove an element from a VOB
rmname	remove the name of an element or VOB symbolic link from a directory
rmver	remove a version from the version tree of an element
uncheckout	cancel a checkout of an element
unreserve	change a checkout to unreserved status
xlsvtree	list version tree of an element graphically
Working with Derived Objects and Configuration Records	
catcr	display configuration record
diffcr	compare configuration records
lsdo	list derived objects
rmdo	remove a derived object from a VOB
winkin	wink-in a derived object

Table 2-1 (continued) *cleartool* Subcommands

Working with Meta-Data and Annotations and Type Objects	
lstype	list type objects
mkattr	attach attributes to VOB objects
mkattype	create an attribute type object
mkhlink	attach a hyperlink to a VOB object
mkhltype	create a hyperlink type object
mklabel	attach version labels to versions
mklbtype	create a version label type object
rmattr	remove an attribute from a VOB object
rmhlink	remove a hyperlink from a VOB object
rmlabel	remove a version label from a version
rmmerge	remove a merge arrow from versions
rmtype	remove a type object from a VOB
rntype	rename a type object
Working with Event Records	
chevent	modify comment string in existing event record(s)
lscheckout	list checkouts of an element
lshistory	list history
Working with the Contents of Versions	
annotate	annotate lines of text file with timestamps
diff	compare files or versions of an element
findmerge	determine what files require a merge

Table 2-1 (continued) *cleartool* Subcommands

merge	merge files or versions of an element
xdiff	compare files or versions of an element graphically
xmerge	merge files or versions of an element graphically
Administrative Commands	
chpool	change the storage pool to which an element is assigned
lock	lock a VOB object
lslock	list locks
lspool	list storage pools
lsview	list view registry entries
lsvob	list VOB registry entries
mkpool	create a VOB storage pool or modify its scrubbing parameters
mktrigger	attach a trigger to an element
mktrtype	create a trigger type object
mkvob	create and register a versioned object base
mount	activate a VOB
protect	change permissions or ownership of a VOB object
protectvob	change owner or groups of a VOB
reformatvob	update the format of a VOB database
register	create an entry in the VOB storage registry or view storage registry
rmpool	remove a storage pool from a VOB
rmtrigger	remove trigger from an element
rmvob	remove a VOB storage directory
rnpool	rename a VOB storage pool
space	report on VOB disk space usage

Table 2-1 (continued) *cleartool* Subcommands

umount	deactivate a VOB
unlock	unlock a VOB object
unregister	remove a VOB or view from the storage registry
Miscellaneous Commands	
cd	change current working directory
pwd	print working directory
help	help on cleartool command usage
man	display a ClearCase manual page
apropos	summary information on *cleartool* subcommands
quit	quit interactive cleartool session
shell	create a subprocess to run a shell or other specified program

For example, the following set of *cleartool* subcommands fulfills a typical developer's day-to-day needs:

mkview, edcs	to create a new view, and then adjust its configuration
setview	to start working in a view
checkout, *checkin*, *uncheckout*	to create new versions of source files (or change your mind)
mkelem	to create new version-controlled elements
lscheckout, *lshistory*, *lsvtree*	to determine what other work is currently taking place, and to determine what work has taken place in the past
diff, *merge*, *findmerge*	to work efficiently in a parallel development environment

Getting Help

When you do need to use a *cleartool* subcommand with which you're not familiar, you can take advantage of several on-line help facilities:

- **Syntax summary** — To display a syntax summary for an individual subcommand, use the *help* subcommand or the *-help* option:

  ```
  % cleartool help                (syntax of all subcommands)
  % cleartool help mklabel         (syntax of one subcommand)
  % cleartool mklabel -help        (syntax of one subcommand)
  ```

- **Manual pages** — *cleartool* has its own interface to the UNIX *man*(1) command. Enter `cleartool man command_name` to display the manual page for a subcommand.
- **Permuted index** — The file */usr/atria/doc/man/permuted_index* contains the same information as the permuted index printed in the CASEVision™/ClearCase Reference Pages.
- **Whatis' file** — File */usr/atria/doc/man/whatis* contains summary information from the manual pages. Use the *apropos* subcommand to extract information from this file.

cleartool Usage Overview

You can use *cleartool* in either single-command mode or interactive mode. To invoke a single *cleartool* subcommand from the shell, use this syntax:

```
% cleartool subcommand [ options-and-args ]
```

When entering a series of subcommands, you may find it more convenient to type "cleartool" without any arguments. This places you at the interactive-mode prompt:

```
cleartool>
```

You can then issue any number of subcommands (simply called "commands" from now on), ending with *quit* to return to the shell. *cleartool* commands can be continued onto additional lines with the backslash (\) character, as with UNIX shells.

Command options may appear in any order, but all options must precede any non-option arguments (typically, names of files, versions, branches, and so on). If an option is followed by an additional argument, such as *-branch /main/bugfix*, there must be white space between the option string and the argument. If the argument itself includes space characters, it must be quoted.

Command Abbreviations and Aliases

Many subcommand names and option words can be abbreviated. A command's syntax summary indicates all valid abbreviations. For example:

```
lsc·heckout (in printed manual pages)
lsc/heckout (in on-line manual pages)
```

This means that you can abbreviate the subcommand name to the minimal "lsc", or to any intermediate spelling: "lsch", "lsche", and so on.

A few *cleartool* commands have a built-in command alias. For example, *checkin*'s alias is *ci*; similarly, *checkout*'s alias is *co*. These commands are equivalent:

```
% cleartool checkin test.c
  and
% cleartool ci test.c
```

Command Options

cleartool commands use multiple-character options, such as `-all`, `-default`, and `-comment`. Long options can always be abbreviated; as with commands, the minimal abbreviation is always three characters or fewer, and any intermediate spelling is valid: you can abbreviate `-delete` to -del, -dele, or `-delet`.

Options that are commonly used in standard UNIX commands have single-letter abbreviations. For example, you can abbreviate `-directory` to `-d`. The others options in this category include `-all`, `-recurse`, `-long`, and -short.

Options rigorously distinguish between type objects and instances of those types. For example:

`-brtype` ... Specifies a particular *branch type* object.

`-branch` ...

Specifies a particular branch — that is, a particular *instance* of a branch type object, within the version tree of some element.

Pathnames in *cleartool* Commands

Many *cleartool* commands take one or more pathnames as arguments — typically, the name of a file or directory element, or a view-private file, or a derived object that you've built with *clearmake*. You can use either kind of standard UNIX pathname: full or relative. In many cases, you can also use a ClearCase *extended pathname*:

```
/vobs/proj/test.c                    (standard full pathname)
/view/akp/vobs/proj/test.c      (view-extended full pathname)
/vobs/proj/test.c@@/main/bugfix/4
                             (version-extended full pathname)

test.c                           (standard relative pathname)
test.c@@/RLS2.0          (version-extended relative pathname)
test.c@@/main/LATEST     (version-extended relative pathname)

../lib/libsort.a                 (standard relative pathname)
../lib/libsort.a@@/RLS4.2
                         (version-extended relative pathname)

hello.o                (standard pathname to derived object)
hello.o@@14-Mar.09:55.4388
                       (extended pathname to derived object)
```

For both full or relative pathnames:

- Your current view automatically resolves a standard pathname to a particular ClearCase object (this is called *transparency*):
  - The standard operating system pathname of an element implicitly references the version selected by your view.
  - The standard pathname of a derived object references the one built in your view. (Users in different views can build *makefile* targets independently; different derived objects produced by such builds appear at the same pathname in the respective views.)
- A *view-extended pathname* references the object that another view sees at a standard pathname.
- A *VOB-extended pathname* references an object using VOB database identifier. The most commonly-used is a *version-extended pathname*, which references a particular version of an element using its unique *version-ID* (for example, *test.c@@/main/bugfix/4)* or using a *version label* (for example, *test.c@@RLS2.0*). Other kinds of VOB-extended pathnames include:

```
hello.c@@               (extended pathname to element object)
hello.c@@/main/bugfix    (extended pathname to branch object)
hello.o@@14-Mar.09:55.4388
                       (extended pathname to derived object,)
                               (incorporating a unique DO-ID)

DesignFor@566          (extended pathname to hyperlink object)
Merge@268                (incorporating a unique hyperlink-ID)
SyncWith@4099
```

(Strictly speaking, the extended names for hyperlinks are not “pathnames”, since hyperlinks do not appear at all in the operating system’s file namespace. Syntactically, however, *cleartool* treats hyperlink names like other pathnames.)

For more information on ClearCase pathnames, see the *version_selector* and *pathnames_ccase* manual pages.

Command-Line Processing

In single-command mode, the *cleartool* command you enter is first processed by the UNIX shell. The shell expands file name patterns and environment variables, and it interprets quotes and other special characters. *cleartool* processes the resulting argument list directly, without any further interpretation.

In interactive mode, *cleartool* itself interprets the command line similarly, but not identically, to the UNIX shells. In particular, it does not expand environment variables and does not perform command substitution (`...`). For details, see the *cleartool* manual page.

Event Records and Comments

Each change to a VOB (checkin of a new version, attaching of a version label, and so on) is accompanied by the creation of an *event record* in the VOB database. Many *cleartool* commands allow you to annotate the event record(s) they create with a comment string. In some cases, your comment is appended to a ClearCase-generated comment. All commands that accept comment strings recognize the same options:

-c *comment-string*
Specifies a comment for all the event records created by the command.

-cq
The command prompts for a comment, which will be placed in the event records for all objects processed by this command.

-cqe
For each object it processes, this command prompts for a comment to be placed in the corresponding event record.

-nc
("no additional comment") For each object it processes, the command creates an event record with no user-supplied comment string.

Examining Event Records

cleartool includes several commands that display event records, optionally including the comment strings: *lshistory*, *lscheckout*, *lstype*, *lslock*, and *lspool*. See the *fmt_ccase* manual page for a description of the simple report-writing facility built into these commands.

The *chevent* command revises the comment string in an existing event record. See the *events_ccase* manual page for a detailed discussion of event records.

Customizing Comment Handling

Each command that accepts a comment string has *comment default*, which takes effect if you enter the command without any comment option. For example, the *checkin* command's comment default is -cqe, causing *cleartool* to prompt you to enter a comment for each element being checked in. The *ln* command's comment default is -nc: create the event record without a user-supplied comment.

You can customize *cleartool*'s comment-handling with a *user profile* file, *.clearcase_profile,* in your home directory. For example, you might establish -cqe as the comment default for the *ln* command. See the *user_profile* manual page for details.

Permissions Checking and Locks

All *cleartool* commands that modify ("write") a VOB are subjected to permissions checking. The following hierarchy is used, in a command-specific manner, to determine whether a command should proceed or be cancelled:

- the *root* user (superuser)
- the VOB owner (typically, the user who created the VOB storage area)
- the owner of the element involved in the command
- the creator of the type object (for modifications to objects of that type)

- the creator of a particular version or derived object
- members of an element's group or derived object's group (same UNIX group ID)

For example, the *root* user always has permission to use commands that modify a VOB. However, if you try to modify an element that you do not own, and are neither the VOB owner nor the root user, *cleartool* will not allow the operation to proceed.

ClearCase also provides for temporary access control through explicit locking of individual objects, with the *lock* command. When an object is locked, it cannot be modified by anyone (except, perhaps, for a list of explicitly-exempted users).

For details on permissions-checking and locks, see the *ct_permissions* manual page.

Exit Status

If you exit *cleartool* by entering a quit command in interactive mode, the exit status is 0. The exit status from single-command mode depends on whether the command succeeded (zero exit status) or generated an error message (nonzero exit status).

Error Logs

Some of the warning and error messages displayed by *cleartool* commands are also written to log files located in directory */usr/adm/atria/log*. You may sometimes find that a message has been written to a log on another host; this is a artifact of ClearCase's client-server architecture.

Using *clearmake*

clearmake is the ClearCase build utility, designed to be compatible with many different *make* variants. We recommend that, you read Chapter 5, *Building Software with ClearCase* in the *CASEVision™/ClearCase Concepts Guide*, before reading this section.

Invoking *clearmake*

You can invoke *clearmake* using the ClearCase CLI or GUI. The command-line interface is designed to be as similar as possible to other *make* variants. Single-letter command options have their familiar meanings. For example:

–n	no-execute mode
–f	specify name of *makefile*
–u	unconditional rebuild

clearmake recognizes additional options (also single-letter) that control its enhanced functionality: configuration lookup, creation of configuration records and derived objects, parallel and distributed building, and so on. For a complete description, see the *clearmake* manual page.

You can run *clearmake* as a background process or invoke it from a shell script, just like any other program. (In *clearmake* output, some names are emboldened, for clarity. On some architectures, running *clearmake* in the background suppresses this emboldening, but no characters are lost.)

A Simple *clearmake* Build Scenario

clearmake was designed to let developers in *makefile*-based build environments continue working in their accustomed manner. The following simple build scenario demonstrates how little adjustment is required to begin building with *clearmake*.

1. **Set a view** — Since working with ClearCase data requires a view context, it makes sense to set a view before starting a build.

   (Strictly speaking, this is not required: if your process has a *working directory view* context, but not a *set view* context, *clearmake* automatically sets the view by executing a *cleartool setview -exec clearmake* command. See “Setting a View” on page 80.)

2. **Go to a development directory** — Change to a directory within any VOB.

3. **Edit some source files —** Typically, you need to edit some sources before performing a build; accordingly, you *checkout* some file elements and revise the checked-out versions.

4. **Start a build —** You can use your existing *makefile*(s) to perform a ClearCase build. Just invoke *clearmake* instead of your standard *make* program. For example:

   ```
   % clearmake                     (build the default target)
   % clearmake cwd.o libproj.a
                       (build one or more particular targets)
   % clearmake -k monet CFLAGS=-g
              (use standard options and make-macro overrides)
   ```

   Note: We recommend that you avoid specifying make-macro overrides on the command line. See “Using a Build Options Specification (BOS) File” on page 156.

clearmake builds targets (or avoids building them) in a manner similar to, but more sophisticated than, other *make* variants. Figure 2-1 illustrates the results of a typical build:

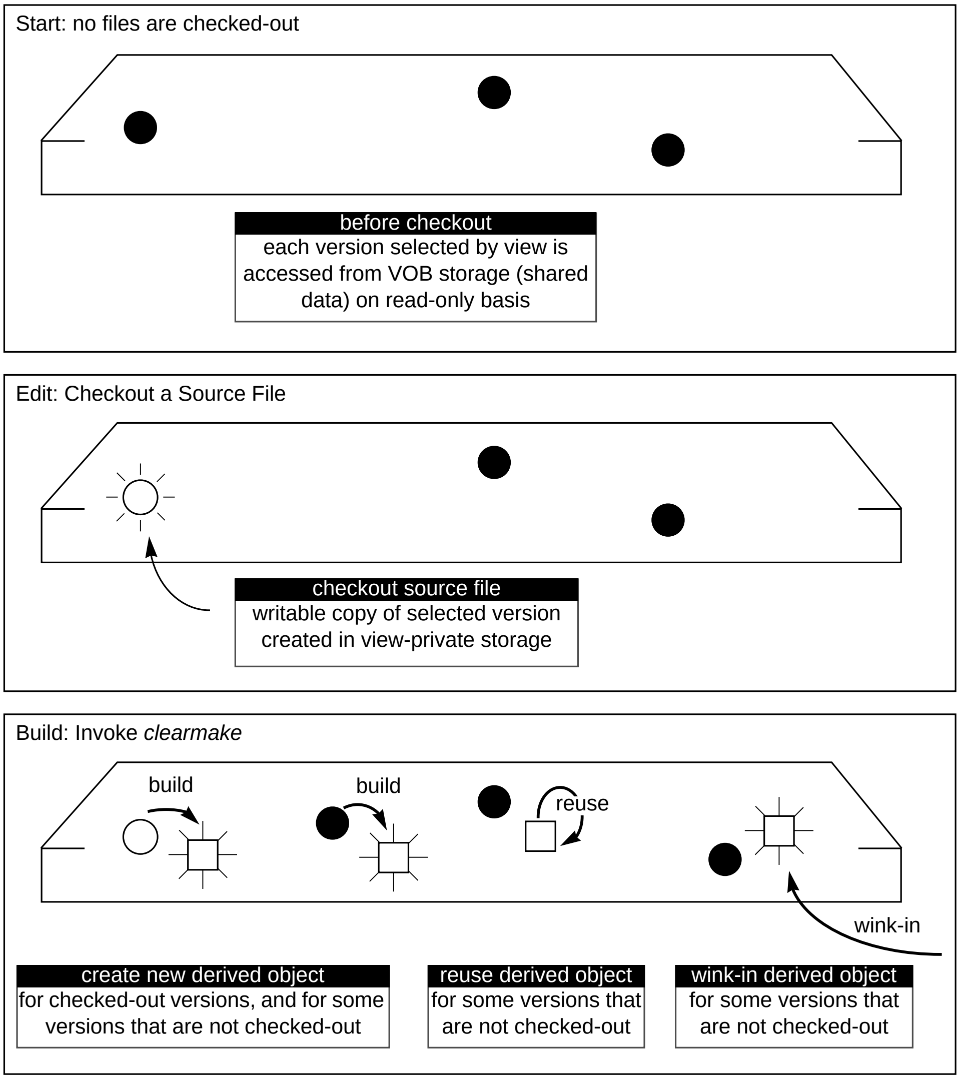

Start: no files are checked-out
before checkout
each version selected by view is accessed from VOB storage (shared data) on read-only basis
Edit: Checkout a Source File
checkout source file
writable copy of selected version created in view-private storage
Build: Invoke *clearmake*
build
build
reuse
wink-in
create new derived object
for checked-out versions, and for some versions that are not checked-out
reuse derived object
for some versions that are not checked-out
wink-in derived object
for some versions that are not checked-out

Figure 2-1 ‘clearmake’ Build Scenario *clearmake* builds a new derived object for each newly-checked-out source file, because no other build could possibly have used your checked-out version.

Note: *clearmake* does not attempt to verify that you have actually edited the file; the *checkout* itself triggers the rebuild. As you work, each text-editor "save file" followed by an invocation of *clearmake* will cause a rebuild of the updated file's dependents, in the standard *make* manner.

For source files that you have *not* checked out, *clearmake* may or may not actually build a new derived object:

- Sometimes, it reuses a derived object that already appears in your view, produced by a previous build.
- Sometimes, *clearmake* winks-in an existing derived object originally built in another view. (It's even possible that a winked-in DO was originally created in your view, but then deleted — for example, by a "make clean".)
- Sometimes, changes to other aspects of your build environment trigger a *clearmake* rebuild: revision to a header file; change to the build script, use of a make-macro override; change to an environment variable used in the build script.

More on Building with *clearmake*

This manual contains a great deal more information on using *clearmake* and related software build mechanisms starting with Chapter 10, "Building with clearmake; Some Basic Pointers".

Chapter 3

Using the ClearCase Graphical User Interface

This chapter contains basic background and usage fundamentals for the ClearCase graphical interface.

Starting *xclearcase*

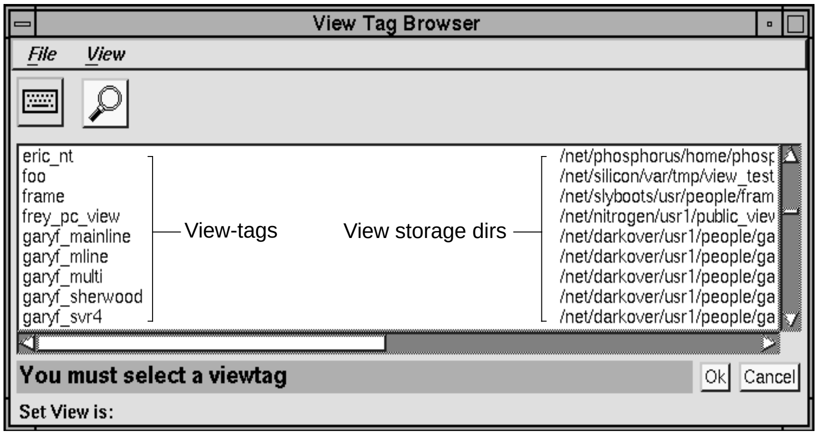
The *xclearcase* command line can include any of the numerous *X(1)* command options, but examples in this chapter are all derived from an interface invoked with:

```
% xclearcase &
```

You can start *xclearcase* with or without a view context. If you are not in a view, *xclearcase* first prompts you for a view-tag. Figure 3-1 shows a *view-tag browser.*

The view-tag browser lists all registered views. You can set the current view to any on the list, whether or not it is currently "active." (An *active view* already has an entry in the *viewroot* directory, */view.*)

Click *leftMouse* over a view-tag to select (highlight) it, and press the Ok button.



```
eric_nt              /net/phosphorus/home/phosp
foo                  /net/silicon/var/tmp/view_test
frame                /net/slyboots/usr/people/fram
frey_pc_view         /net/nitrogen/usr1/public_view
garyf_mainline       /net/darkover/usr1/people/ga
garyf_mline          /net/darkover/usr1/people/ga
garyf_multi          /net/darkover/usr1/people/ga
garyf_sherwood       /net/darkover/usr1/people/ga
garyf_svr4           /net/darkover/usr1/people/ga
```

You must select a viewtag

Set View is:

Figure 3-1 View-tag Browser at *xclearcase* Startup

File Browser

Figure 3-2 shows the *file browser*, which appears once you have an active view. Think of the file browser as your “home base.” A file browser displays the current directory name and, below it, the directory contents.

Like all *xclearcase* browsers, a file browser includes a variety of menus — *toolbar menu*, *pop-up menu*, and *pull-down menus* — to do the real work. Each menu item launches an operating system command script, which typically includes some ClearCase-specific enhancements.

The file browser in Figure 3-2 is displaying the contents of a directory under a VOB mount point. To change to a VOB directory, type the desired pathname into the directory input box, and press <**`Return`**>.

Figure 3-2 The *xclearcase* File Browser

Note: If the graphical interface has been customized at your site, your screen display may vary. See Chapter 18, “Customizing the Graphical Interface,” for more information.

With the main file browser displayed, you are ready to work. Notice that only some of the toolbar and pull-down menu items are enabled, while others are "grayed out." To familiarize yourself with the workings of a browser:

- Scan through the various pull-down menus.
- Display "pop-up help" for any *enabled* toolbar item by moving to it and clicking rightMouse. For pull-down menu items, click on the menu to "post" it, and press rightMouse on the desired item.
- Try selecting various files, and combinations of files, and watch how the set of enabled operations changes with your selections.
- Post the pop-up menu by clicking rightMouse in the browser.
- Use the Admin and Metadata menus to look at other browsers.

File Browser Toolbar

Here are brief descriptions for the default file browser toolbar items (see also each item's "pop-up help"):

Here are brief descriptions for the default file browser toolbar items (see also each item's "pop-up help"):

Toggle Graphic Mode — Toggle between iconic and textual display modes for directory listings. See Figure 3-3.

Figure 3-3 Toggle Graphic Mode

Toggle Keyboard Input Mode — Enable or disable the file browser's keyboard input box. See "Keyboard Input" on page 56 . See Figure 3-4.

Figure 3-4 Toggle Keyboard Input Mode

Checkin — Checkin the checked-out versions of the selected elements. See Figure 3-5.

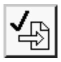


Figure 3-5 Checkin Versions

Checkout — For selected elements, checkout (reserved) the versions selected by your current view. See Figure 3-6.

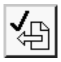


Figure 3-6 Checkout Versions

Uncheckout — Uncheckout one or more checked-out versions. See Figure 3-7.

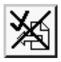


Figure 3-7 Uncheckout Versions

Describe — Describe each selected object (in a read-only *text output window*). See Figure 3-8.

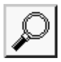


Figure 3-8 Describe Selected Object

Vtree — Start a vtree (version tree) browser on the selected element. See Figure 3-9.

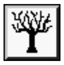


Figure 3-9 Vtree (Version Tree)

Diff — Diff the selected version against its predecessor version. See Figure 3-10.

Figure 3-10 Diff Versions

Merge — Merge from another version (which is prompted for) to the selected version. See Figure 3-11.

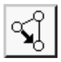


Figure 3-11 Merge Versions

Clearmake: default — Run *clearmake* on the default target (using *Makefile* or *makefile* in current directory). See Figure 3-12.

Figure 3-12 *clearmake* Default

Shell — Start up a shell process in a separate window. See Figure 3-13.

Figure 3-13 Shell Process

Basic Usage Model

The basic usage model involves a simple cycle:

1. Select data to operate on (one or more file elements, for example). Your selection enables some subset of the available toolbar buttons and menu commands.

2. Invoke an enabled menu item, either on the toolbar or from a pull-down menu.
3. If necessary, respond to interactive prompts (by the various kinds of browsers) for more information.

At any given time, some items are active, or *enabled*, while others are "grayed out" (or *insensitive*). Many operations are defined to remain insensitive until you select one or more data objects relevant to the operation. For example, the checkout button is not enabled until you *preselect* at least one unchecked-out element.

Menu Command Nesting

While *xclearcase* is displaying a browser prompt, you can start another menu command. When you have completed the "interrupt", the original prompt is still active, waiting for input. When nesting commands, you cannot cancel a browser started to collect input for a previous command. The command nesting level limit is ten.

Basic Pointer Actions and Keystrokes

Table 3-1 covers basic pointer actions and keystrokes for all file browsers. (In general, these actions apply to the other kinds of browsers, as well.).

Table 3-1 File Browser Pointer Actions and Keystrokes

Function	Pointer Action/Keystroke
Basic	
Select item	Click *leftMouse*
Select region	Drag leftMouse
Extend-select (discontiguous)	control-leftMouse
Extend-select (range) (for textual, not graphical, dir list)	shift-leftMouse

Table 3-1 (continued) File Browser Pointer Actions and Keystrokes

Function	Pointer Action/Keystroke
Display “pop-up help” for an *enabled* menu item	Toolbar: rightMouse on button Other menus: click leftMouse to post menu, then rightMouse over item
Change working directory	*doubleClick* on directory icon, or edit directory text input box
List directory history	Press button. See Figure 3-14
Display pop-up menu	*rightMouse* in browser
Exit *xclearcase*	Exit option on File menu
Menu Navigation	
Post (“pin up”) a pull-down menu; ...using mnemonic (underlined char)	Click leftMouse on menu; *AltKey-mnemonicChar*
Cycle through posted menu options	upArrow/downArrow
Post submenu	*mnemonicChar* *rightArrow*
Cycle through menus left-to-right	rightArrow/leftArrow
Invoke highlighted menu item	return spacebar

Figure 3-14 List directory history button

Keyboard Input

Each browser has an optional *keyboard input box* as shown in Figure 3-15, which lets you type in data selections directly. Some commands enable it automatically, but you can also enable it manually with the toolbar's *Keyboard input* button.

Figure 3-15 Keyboard Input Box

You can use the keyboard input box to type in one or more items. For browsers that accept pathnames, most commands allow wildcard patterns, including *, ?, and [] (but not {}). Any selection you make by pointing replaces the current contents of the input box. The items that appear in the keyboard input box constitute the current selection. Select a menu item to operate on them.

Note: The keyboard input box sidesteps many built-in protections against incorrect input to buttons and menu commands. When the keyboard input box is enabled, *all* menu items become active, whether or not they are applicable. Typed-in data is not validated until the command executes. (The *number* of typed-in data items is continually evaluated; if this number violates the conditions required to enable a menu item, the item becomes insensitive.)

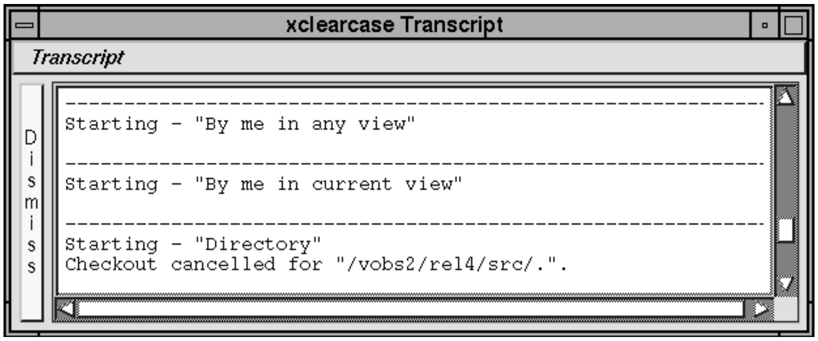
The *File* Menu

The File menu and the menu items in Table 3-2 are common to all browsers.

Table 3-2 File Menu Options

Option	Brief Description
Show transcript	Display the transcript pad. By default, the transcript pad pops up automatically only to display error and warning messages. It stays up until dismissed.
Update browsers	Manually update all browser displays.
New file browser	Start a new file browser.
Close window	Close current browser.

Figure 3-16 illustrates the transcript pad, a scrolling text window that functions as *xclearcase's* "standard output" and "standard error" devices.



```
--------------------------------------------------------------
Starting - "By me in any view"

--------------------------------------------------------------
Starting - "By me in current view"

--------------------------------------------------------------
Starting - "Directory"
Checkout cancelled for "/vobs2/rel4/src/.".
```

Figure 3-16 The Transcript Pad

As you work in the graphical interface, the transcript pad receives error, status, and warning messages, as well as command output from menu operations. By default, the transcript pad pops up automatically only in response to error and status messages. You can manually post the transcript pad at any time with the menu item *File* -> *Show transcript* icon.

Depending on how a menu operation is defined, its text output can appear in a variety of places, including:

- the *transcript pad*
- a *list browser*
- a read-only display window
- a text editor

Although a menu operation's primary output may be redirected from the transcript pad, a *Starting* - "*operationName*" message appears in the transcript pad for each operation you execute.

Transcript Menu

The menu options for the transcript pad are detailed in Table 3-3.

Table 3-3 Transcript Menu Options

Option	Brief Description
Clear transcript	Clear all text from the transcript pad.
Scroll to Bottom	When set, the transcript pad automatically scrolls to the bottom to display new output as it arrives. Unset this toggle button when you are examining a particular section of text and don't want to be interrupted by new output.

Browsers

The xclearcase interface includes numerous browsers. *File*, *VOB*, *viewtag*, *vtree*, *type*, *pool*, and *username* browsers let you query and select the data objects used by ClearCase. *List* and *string* browsers facilitate string data I/O. This section provides a brief introduction to each kind of browser.

Browser Basics

Browser interaction follows two distinct patterns:

- You explicitly start browsers (Metadata -> Label -> Label type... or Admin -> Vob..., for example) in order to view or *select* data objects. Bringing up a file browser with *xclearcase* also falls into this category. A browser started in this manner stays up until you close it
- Other browsers come and go automatically as you work. When executing the scripts attached to menu items, *xclearcase* frequently starts browsers to *prompt* for additional data, and terminates them after you Ok or Cancel the prompt. For example, a button labeled *Prepare vob report* might start numerous browsers (with interactive prompts) to collect information about the various data objects in a particular VOB.

At any one time, your screen display may include multiple instances of both “long-lived” and “transient” browsers. If *xclearcase* requires data from you, the same prompt may appear in multiple browsers, if more than one is capable of satisfying the prompt.

Browsers and Data Types

In general, each class of browser exists to handle a particular type of data. The following Table 3-4 shows the tight correspondence between browsers and *xclearcase* data types.

Table 3-4 Browsers and Data Types

Data type	Browsers that Display or Prompt for the Data Type
PNAME	File/Vtree
HYPERLINK	Vtree
LIST	List
ATTYPE/BRTYPE/ELTYPE/ HLTYPE/LBTYPE/TRTYPE	Attype/Brtype/Eltype/ Hltype/Lbtype/Trtype
POOL	Pool
STRING	String

Table 3-4 Browsers and Data Types

Data type	Browsers that Display or Prompt for the Data Type
USERNAME	Username
VIEWTAG	View-tag
VOBTAG	VOB-tag

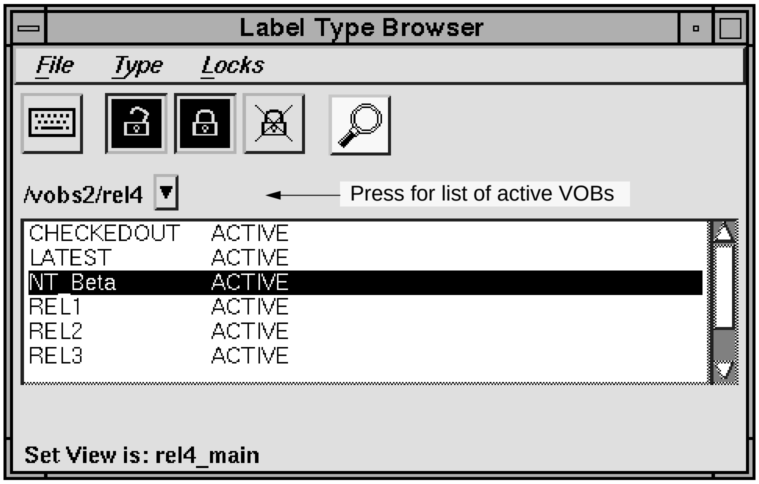
File Browsers

See the section "File Browser" on page 50.

Type Object Browsers

Each of the six type object browsers operates on the corresponding class of type object. You can start type object browsers explicitly from the file browser's Admin and Metatype menus, and with the Version -> Branch -> Branch type... menu item.

Figure 3-17 shows a label type browser.



```
CHECKEDOUT   ACTIVE
LATEST       ACTIVE
NT_Beta      ACTIVE
REL1         ACTIVE
REL2         ACTIVE
REL3         ACTIVE
```

Set View is: rel4_main

Figure 3-17 A Type Object Browser

The type object browsers share a common toolbar:

Toggle Keyboard Input Mode — Enable or disable the file browser′s keyboard input box. See Figure 3-18.

Figure 3-18 Toggle Keyboard Input

Toggle Unlocked Object Display — Enable or disable the display of unlocked type objects. See Figure 3-19.

Figure 3-19 Toggle Unlocked Object

Toggle Locked Object Display — Enable or disable the display of locked type objects. See Figure 3-20

Figure 3-20 Toggle Locked Object

Toggle Obsolete Object Display — Enable or disable the display of obsolete type objects. See Figure 3-21.

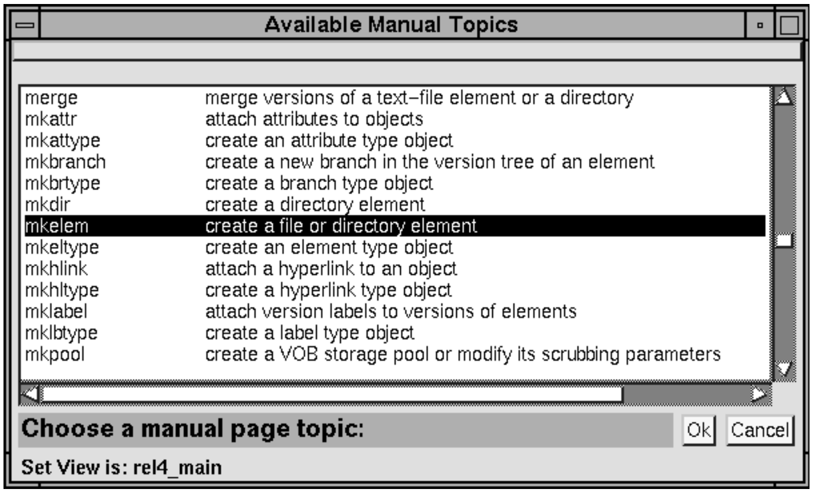
Figure 3-21 Toggle Obsolete Object

Describe — Describe the selected object (in a read-only text output window). See Figure 3-22.

Figure 3-22 Describe Selected Object

List Browsers

List browsers are not started directly. You encounter a list browser only when a menu command redirects output to one and prompts you to select data from it. Figure 3-23 shows a sample list browser — the one Help -> Manual page... uses to prompt you for a topic.



```
merge          merge versions of a text–file element or a directory
mkattr         attach attributes to objects
mkattype       create an attribute type object
mkbranch       create a new branch in the version tree of an element
mkbrtype       create a branch type object
mkdir          create a directory element
mkelem         create a file or directory element
mkeltype       create an element type object
mkhlink        attach a hyperlink to an object
mkhltype       create a hyperlink type object
mklabel        attach version labels to versions of elements
mklbtype       create a label type object
mkpool         create a VOB storage pool or modify its scrubbing parameters
```

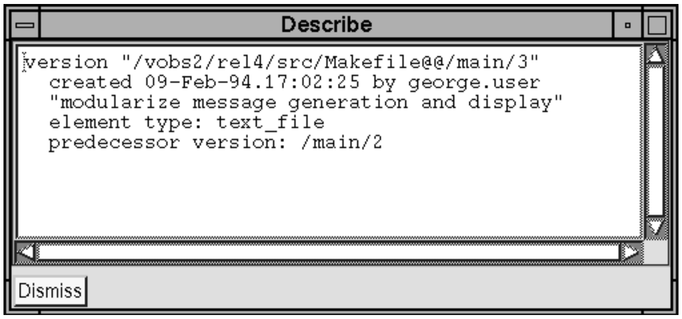
Set View is: rel4_main

Figure 3-23 A List Browser

A list browser prompts you to select one or more items (entire lines only, no partial lines or substrings). Press Ok to submit the selection, or Cancel to cancel the prompt (and, therefore, the entire command operation). You cannot edit the contents of a list browser.

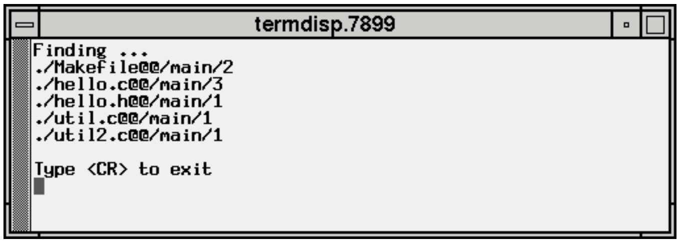
Text Output and Terminal Emulation Windows

For comparison with list browsers, Figure 3-24 shows a sample *text output window,* and Figure 3-25 shows a *terminal emulation window.* Neither prompts for, or accepts, user input; they are display-only devices.



```
version "/vobs2/rel4/src/Makefile@@/main/3"
  created 09-Feb-94.17:02:25 by george.user
  "modularize message generation and display"
  element type: text_file
  predecessor version: /main/2
```

Figure 3-24 A Text Output Window



```
Finding ...
./Makefile@@/main/2
./hello.c@@/main/3
./hello.h@@/main/1
./util.c@@/main/1
./util2.c@@/main/1

Type <CR> to exit
```

Figure 3-25 A Terminal Emulation Window

The text output window was generated by the Describe button and the terminal window by menu item Report -> Find query -> Whole VOB -> Versions with Label...

You can cancel output to a terminal emulation window with <`Ctrl-C`>.

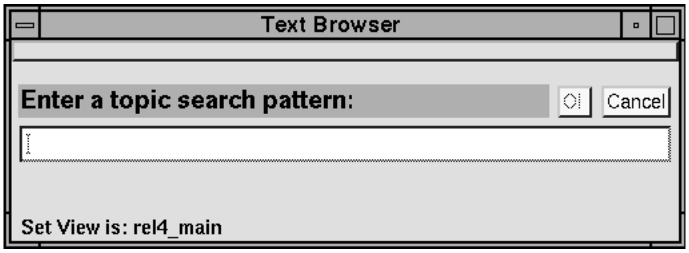
Pool Browsers

Admin -> Pool...

A pool browser lists the storage pools and their locations for any registered VOB. Click the down-arrow next to the text input box to display a list of currently registered VOBs.

String Browsers

String browsers exist only to prompt for text strings and, therefore, are more like simple dialog boxes than browsers Figure 3-26 shows the text string browser that results when you choose Help -> Apropos...



Set View is: rel4_main

Figure 3-26 A Text String Browser

A variety of menu command use string browsers to prompt for simple text string arguments (comments, for example) or for other data strings — any data that cannot be captured by the more specific data type browsers.

Username Browsers

Browse and select user's login names.

VOB-tag Browsers

Admin -> Vob...

Browse and select VOB-tags. The VOB browser lists the VOB-tags, or mount points, for all registered VOBs.

View-tag Browsers

Admin -> View... Contrast this with the Version -> Set... command, which starts a transient, prompting view-tag browser.

Browse and select view-tags. A viewtag browser lists all registered views. You can set the current view to any on the list, whether or not it is currently “active.” (An *active view* already has an entry in the *viewroot* directory, */view*.)

Vtree Browsers

You can start a vtree browser with the vtree toolbar button, or from the command line with either the *cleartool xlsvtree* or *xlsvtree* commands.

Use vtree browsers to scan version trees and to operate on file and directory versions, branch names, and merge arrows. (On a vtree browser, arrows show merge hyperlinks.) Figure 3-31 shows a sample vtree browser.

Vtree-specific toolbar items:

Toggle Checked-out Version Display — Enable or disable the display of checked-out versions. See Figure 3-27.

Figure 3-27 Toggle Checked-out Version

Toggle All Versions Display — Enable or disable the display of all versions in the element. If unset, only labeled versions, branch points, and merge endpoints are displayed. See Figure 3-28.

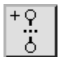


Figure 3-28 Toggle All Versions

Toggle Merge Arrow Display — Enable or disable the display of merge arrows. See Figure 3-29.

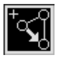


Figure 3-29 Toggle Merge Arrow

Toggle All Labels Display — Enable or disable the display of all labels on all versions. If unset, up to five labels are display for any one version (followed by "..." if there are more than five). See Figure 3-30.

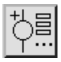


Figure 3-30 Toggle All Version Labels

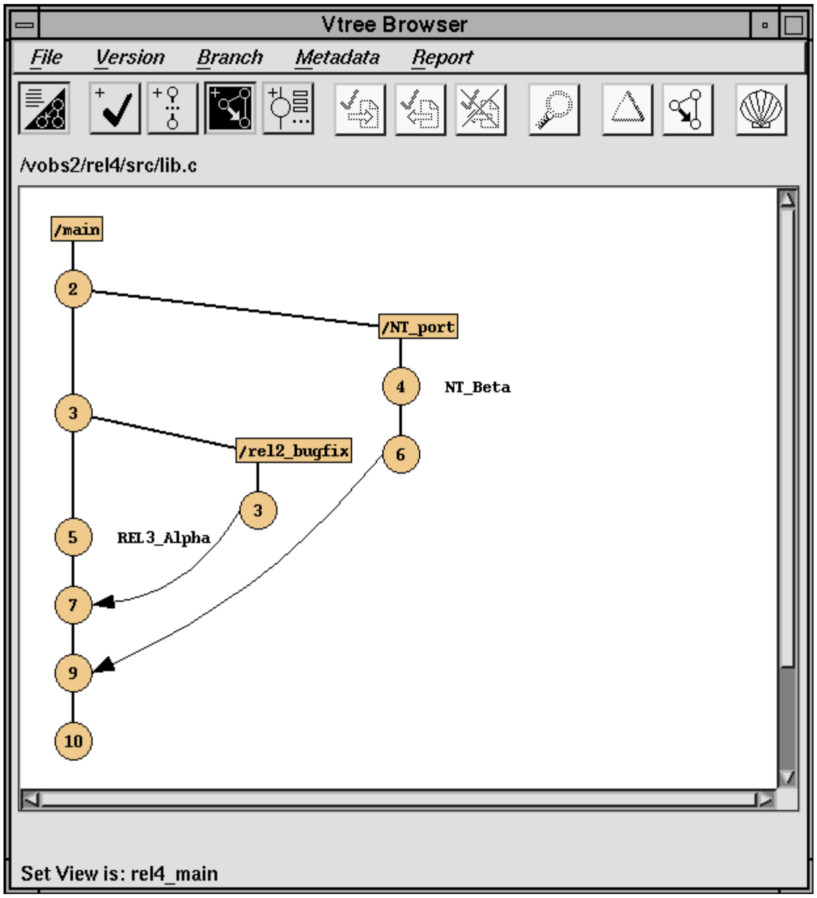


Figure 3-31 The Vtree Browser

Chapter 4

Setting Up a View

This chapter describes how to set up a new ClearCase *view* for a development project.

Note: Your organization may have policies or restrictions as to where you can create a view. For example, you might be required to use a particular disk that is part of a strictly-observed data-backup scheme. And in some organizations, individual users are not permitted to create their own views. Consult with your system administrator before actually creating any views.

Planning the View

Before creating a view, consider how, and by whom, it will be used:

- Should other users be able to read data in your view (perhaps the contents of a source file that you have checked out and edited)?
- Should other users be able to write data in your view (perhaps you will occasionally share the view with another user)?
- Will the view be used principally, or exclusively with a particular VOB or a small, localized set of VOBs?
- Will the view be used to export one or more VOBs to non-ClearCase hosts?

Keep in mind that others users working on the same project do not necessarily need to explicitly access your view in order to share your work. When you build software with *clearmake*, the resulting derived objects are automatically sharable. On the source level, a typical strategy is to have each project member use a separate, personal view; but *all* these views are configured with the same config spec. With this strategy, each user's changes to checked-out source files and directories will be visible only to that user. When the user checks in a version, the changes become visible to all other group members — all those using like-configured views.

Adjust Your 'umask'

Your *umask*(1) setting at the time you create a view affects how accessible it will be to others. For example:

- A umask of 002 is appropriate for a view that you will share with other users in the same group. Members of your group will be able to create and modify view-private data; those outside your group will be able to read view-private data, but won't be able to modify it. To completely exclude non-group members, set your umask to 007.
- A umask of 022 will produce a view in which only you can write data, but anyone can read data.
- A umask of 077 is appropriate for a completely private view. No other user will be able to read or write view-private data.

Change your umask in the standard way. For example:

```
% umask 022
```

Choose a Location

A view is implemented as a *view storage directory*, with an associated *view_server* process. Accordingly, ClearCase imposes these requirements on view creation:

- You can create a view storage directory only in locations where you have permission to create a standard directory.
- The *view_server* process runs on the host where the view storage directory physically resides; ClearCase must be installed on that host.

A typical location for a view is your home directory. At some sites, however, a user's home directory may fail to meet the second requirement — it may be located on a file-server host where ClearCase is not installed.

If the view will be used to access a particular VOB, placing the view on the same host as the VOB may provide a significant reduction in network traffic. In general, we don't advise placing additional loads on a VOB server host; so save this technique for special cases — for example, a shared view used for a final-integration-and-test task. Non-ClearCase access is a special case in which you *should* create a view on the same host as a VOB — see "Setting Up an Export View for Non-ClearCase Access," in the CASEVision™/ClearCase Administrator's Manual.

If you will be using the view on several hosts, make sure that the location at which you create the view can be accessed by all those hosts. For example, you use a view on several hosts at the same time when performing a distributed build. See Chapter 13, "Setting Up a Distributed Build."

Choose a Name

In selecting new view's *view-tag*, take into account the fact that it will be a unique, network-wide identifier for the view.[1] Thus, names like *myview*, *work*, or *tmpvu* are to be discouraged. You, and perhaps other users, may often need to type the view-tag in view-extended pathnames — for example, */view/**view-tag**/vobs/proj.* Thus, try not to select a name that is too long or too hard to type correctly. Following are some suggested names:

josef	personal view
akp_home	personal view, located in your home directory
akp_neptune	personal view, located on remote host *neptune*
RLS1.2_fix	shared view for use in a particular bugfixing task
monet_exp	view to be used to export a VOB named *monet*

In any case, keep in mind the restriction that a view-tag must take the form of a simple directory name.

Creating the View

Having adjusted your umask (if necessary), selected a location for the view storage directory, and selected a view-tag, you are ready to create the view.

GUI: Use the View Browser to Create a New View

In *xclearcase*, bring up the *View Browser*, which displays information on all existing views. Then, select the Create menu choice.

[1] Actually, a view can have different tags in different network regions; and it can have multiple tags within the same region. Taking advantage of this flexibility increases the likelihood of user confusion, though.

CLI: Enter a ‘mkview’ Command

Here’s how to use *cleartool* to create the same view as in the preceding section:

```
% cleartool mkview -tag gomez ~/views/gomez.vws
Created view.
Host-local path: einstein:/home/gomez/views/gomez.vws
Global path:     /net/einstein/home/gomez/views/gomez.vws
It has the following rights:
User : gomez    : rwx
Group: dvt      : rwx
Other:          : r-x
```

Verify the View’s Registry-Level Information

When you create a view, ClearCase stores information regarding its location in the network-wide *view storage registry*. It derives this information heuristically; in some networks, you must update this information to guarantee global accessibility of your new view.

As the example above shows, *mkview* displays its “guess” as to a globally-valid pathname to the view storage directory:

```
...
Global path:     /net/einstein/home/gomez/views/gomez.vws
...
```

If this pathname is not valid on all hosts in the network, you may be able to use the *register* command to substitute a globally-valid pathname in the storage registry entry. This topic is discussed in Chapter 3, “Using the ClearCase Graphical User Interface,” and Chapter 7, “Setting Up ClearCase Views,” , in the CASEVision™/ClearCase Administrator’s Manual.

Configuring the View

Before you start using your new view, you may need to revise its *config spec*, in order to select a particular configuration of source versions. Every view is created with the default config spec:

```
element * CHECKEDOUT
element * /main/LATEST
```

In many organizations, new development takes place on the *main* branch, while special projects take place on subbranches. The default config spec is appropriate for new development work in such a situation.

There are several ways to reconfigure a view with a non-default config spec:

- **Copy a project-specific file —** Your ClearCase administrator may publish the pathname of a file containing the correct config spec for your project. Use the *setcs* command to reconfigure your view; then use *catcs* to confirm the change. For example:

  ```
  % cleartool setview gomez
  % cleartool setcs /usr/local/lib/munchkin_proj
  % cleartool catcs
  .
  . <new config spec displayed>
  .
  ```

If the administrator subsequently revises the contents of that file, you'll need to enter the same *setcs* command again to update your view's configuration.

- **Include a project-specific file** — Instead a copying a file, you can incorporate its contents with an *include* statement:

  ```
  % cleartool setview gomez
  % cleartool edc
   .
   . use text editor to remove all existing lines, and then add this one:
    include /usr/local/lib/munchkin_proj
  ```

If the administrator subsequently revises the contents of that file, you can update your view's configuration by having the *view_server* reinterpret its current config spec:

```
% cleartool setcs -current
```

- **Compose your own config spec** — There is no single method for composing a set of rules that go into a config spec. The language described in the *config_spec* manual page has many features, and can be used to create many "special effects". Having stated that proviso, we present below a step-by-step procedure for writing a config spec that uses the standard ClearCase baselevel-plus-changes model. Be sure also to consult:
  - Chapter 5, "Defining View Configurations," which presents and explains a collection of config specs.
  - Chapter 6, "Working in a Parallel Development Environment,", which examines in more detail the creation of the "standard" config spec for a project that is to "work on a branch".

Composing Your Own Config Spec

A few simple questions are presented below, based on the assumption that your development proceeds according to a baselevel-plus-changes model. In a well-managed environment, the answers to the questions will be simple, too, and writing the correct config spec will be easy.

What versions should the project start with?

Describe the set of versions that make up the *baselevel* that constitutes the project's starting point. Typical answers are:

- "all the versions that went into Release 1.3"
- "the versions of source files that went into Release 1.3, but use the Release 1.2 version of the *libsort* library"
- "all the versions that went into last night's build of the *sortmerge* executable"

If the answer is "the most recent versions on the *main* branch", then you should probably just use the default config spec. We'll assume that you wish to work with a "more interesting" set of versions.

How can this set of versions be described in terms of ClearCase meta-data?

Often, the translation from English-language description to ClearCase meta-data is very simple: "the versions that went into Release 1.3 are all labeled RLS1.3" corresponds to this config spec rule:

```
element * RLS1.3
```

Similarly, "the versions that went into last night's *sortmerge* build are listed in its config spec" might correspond to:

```
element * -config sortmerge@@11-Mar.09:07.1559
```

Sometimes, the description is a bit more involved. It may help to think of simple sets of versions as being "layers" in a more complex configuration (See Figure 4-1).

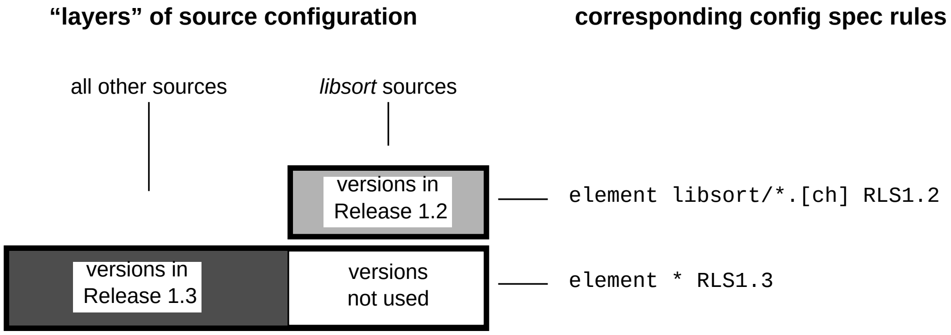


Figure 4-1 "Layers" in a Source Configuration

In this example, the *RLS1.2* rule should precede the *RLS1.3* rule, because the Release 1.2 versions of the *libsort* sources are to be selected in preference to (that is, are to be "layered on top of") the Release 1.3 versions.

Note: It is very important to describe the set(s) of versions in terms of *stable* meta-data — make sure that no one moves any of the *RLS1.3* version labels, and never define a baselevel in terms of the *LATEST* label, which automatically moves. Today's *LATEST* version of a source file may be compatible with the rest of your baselevel versions, but tomorrow's *LATEST* version of that file may be incompatible!

Will the project be modifying any source versions?

For most projects, the answer is "yes", in which case the config spec should begin with the standard *CHECKEDOUT* rule:

```
element * CHECKEDOUT
```

Some projects may not modify any sources — for example, a performance-testing or QA project.

On what branch will the project be working?

Devise a new branch name that describes your project. Often, the branch name is related to a version label that defines the baselevel. For example, a *rls1.3_fix* branch might be used to modify a baselevel defined with *RLS1.3* labels.

The config spec should include a rule that selects the most recent version on the branch:

```
element * .../rls1.3_fix/LATEST
```

Note the use of ellipsis ("..."), which allows the branch to be located anywhere within an element's version tree, not just as a subbranch of the *main* branch. This rule should precede the rule(s) that define the project's baselevel — versions created during the project are to be preferred to versions in the underlying baselevel.

At some point before beginning work on the project, you (or an administrator) must create a branch type with the chosen name. Be sure to enter a meaningful comment:

```
% cleartool mkbrtype rls1.3_fix
Comments for "rls1.3_fix":
Branch for project fixing bugs in Release 1.3 (version label RLS1.3)
.
Created branch type "rls1.3_fix".
```

Note: Label types and branch types share a single namespace. Observe the convention of spelling names of label types with capital letters, and names of branch types with lowercase letters.

Should branches be created automatically?

We encourage you to answer “yes”. Using the view to “work on a branch” is much simpler if you let the view do the branching. When you wish to modify any element, you simply use *checkout*; if no project-specific branch has been created at the baselevel version yet, a *mkbranch* (“make branch”) command is executed automatically.

Here’s how you would modify the rules in Figure 2-1 that define the baselevel, in order to turn on this *auto-make-branch* capability:

```
element libsort/*.[ch] RLS1.2 -mkbranch rls1.3_fix
element * RLS1.3 -mkbranch rls1.3_fix
```

If you’ve defined your baselevel by referencing one or more derived objects, you cannot use a *-mkbranch* clause; you must create branches explicitly, using *mkbranch*.

Will you be creating new elements?

If so, include this rule from the default config spec as the final rule in your config spec:

```
element * /main/LATEST
```

We suggest that you include this rule in the *auto-make-branch* scheme, too:

```
element * /main/LATEST -mkbranch rls1.3_fix
```

With this rule, creating a new element will:

- create a *main* branch, along with version 0 on the branch
- create subbranch *rls1.3_fix* at version */main/0*, along with version */main/rls1.3_fix/0*
- checkout version */main/rls1.3_fix/0*

If you will not be creating any new versions, your view may not need to select *main* branch versions of any elements. As a convenience, however, you may wish to include the standard */main/LATEST* rule, to enable the view to access data belonging to other projects, located in other VOBs.

Modify the View’s Config Spec

Having devised your own config spec (on paper), configure your view with it:

```
% cleartool edcs -tag gomez
  .
  .  <use text editor to revise default config spec — for example,>
    element * CHECKEDOUT
    element * .../rls1.3_fix/LATEST
    element libsort/*.[ch] RLS1.2 -mkbranch rls1.3_fix
    element * RLS1.3 -mkbranch rls1.3_fix
    element * /main/LATEST -mkbranch rls1.3_fix
```

Starting to Use the View

Now that your view is configured, you can start using it. (Actually, you can reconfigure a view at any time — before or after you start using it.) This is termed establishing a *view context* — a *set view* or a *working directory view*.

Setting a View

Typically, most of your work involves just one view. Moreover, you will probably want to use standard operating system pathnames to access version-controlled objects. For both these reasons, you will probably want to begin your ClearCase work by *setting a view*. This creates a process in which an element’s standard name automatically accesses a particular version of that element — the version selected by the view’s config spec, as discussed above.

This *set view* capability completes ClearCase’s *transparency* feature — the version-control mechanism disappears completely, allowing system software and third-party applications to process versions of elements as if they were ordinary files, using ordinary pathnames.

A process with a *set view* can spawn any number of subprocesses, to any nesting level, using standard UNIX commands and subject to standard UNIX restrictions. All these subprocesses are also set to the view.

Transparency also applies to derived objects, in a slightly different manner. The standard name of a DO can reference different files in different views. But a DO appears in a view by virtue of having been built there by *clearmake*, not through the config spec facility.

GUI: Select the View from the View Browser

Select View ->Set, which brings up the *View Browser*. Then, select a view from this browser.

CLI: Enter a *setview* Command

The *cleartool* subcommand *setview* creates a shell that is set to a specified view:

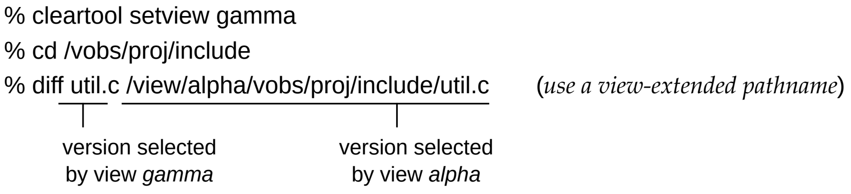
```
% cleartool setview gamma
%
```

Hint: Include the view-tag in your shell prompt. See *CASEVision™/ClearCase Tutorial*.

Working Directory View

When you are set to a particular view, you may still occasionally wish to access other views. For example, if you are set to view *gamma*, you can compare your version of *util.c* with the one selected by view *alpha*. Following are two ways to accomplish this.

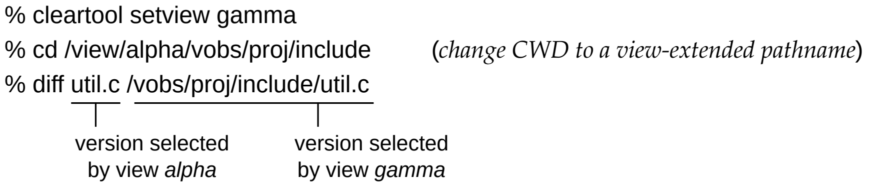
- While in a set view, use a *view-extended pathname* to "reach into" another view. See Figure 4-2.



```
% cleartool setview gamma
% cd /vobs/proj/include
% diff util.c /view/alpha/vobs/proj/include/util.c        (use a view-extended pathname)
```

Figure 4-2 "Reach Into" Another View from Set View

- "Temporarily" go to another view, then "reach back" to the set view. See Figure 4-3.



```
% cleartool setview gamma
% cd /view/alpha/vobs/proj/include        (change CWD to a view-extended pathname)
% diff util.c /vobs/proj/include/util.c
```

Figure 4-3 "Reach Back" to Set View from Another View

In the second method, you change your current working directory (CWD) to another view — that is, to a "remote" location in ClearCase's view-extended namespace. See Figure 1-6. This is termed "changing your *working directory view*", and is reported by the *pwv* command like this:

```
% cleartool pwv
Working directory view: alpha
Set view: gamma
```

Using a Working Directory View without a Set View

There may be some situations in which you find it necessary (or simply prefer) to use working directory view contexts, dispensing with the *set view* facility. For example, processes started by *init*(1M) at system startup time cannot be set to a view. Such a process can process VOB data only by referencing files with view-extended pathnames and/or by setting its current working directory to a view-extended pathname.

If you routinely work with several views, you may find it easier to keep yourself organized by explicitly specifying the view context in which each pathname is to be interpreted.

The following commands illustrate this mode of usage:

```
% cd                                        (go to home directory)

% cleartool pwv                                  (no view context)
Working directory view: ** NONE **
Set view: ** NONE **     (full pathname has no view context,)

% ls /usr/hw/src/util.c       (and so cannot access VOB data)
ls: /usr/hw/src/util.c: No such file or directory

% cd /view/akp/usr/hw/src      (go to view-extended pathname)

% cleartool pwv                                   (you now have a)
Working directory view: jj   (working directory view context)
Set view: ** NONE **

% ls util.c        (relative pathname works, because it uses)
util.c                  (your working directory view context)
```

Note: The standard full pathname is unable to access VOB data in this situation.

View Contexts: Summary

In deciding how to use views, bear in mind this capsule summary of the discussion in the preceding sections. When using ClearCase data, you must use a view — without a view context, a process or pathname cannot "see into" a VOB. A pathname can acquire a view context in several ways:

- A *set view* endows any pathname with a view context.
- A *working directory view* endows a relative pathname with a view context (perhaps overriding a set view context)
- A *view-extended pathname* specifies a particular view context, perhaps overriding a working directory view and/or set view context.

dSymbolic links (either UNIX-level links or VOB symbolic links) can cause unexpected behavior if you have not set a view. For example, suppose your file system includes this symbolic link:

```
% ls -l /vobs/aardvark
/vobs/aardvark -> /vobs/all_projects/aardvark
```

If your shell is not set to a view, you might attempt to visit the *aardvark* VOB with this command:

```
% cd /view/gamma/vobs/aardvark/src
```

But the component-by-component resolution of the pathname by the OS kernel effectively transforms this command to:

```
% cd /vobs/all_projects/aardvark/src
```

By specifying a full pathname, the symbolic link "pops you out" of the *gamma* working directory view context. And because your shell is not set to a view, the pathname will have no view context at all, and thus will fail.

The same analysis applies to view-extended pathname. For example, changing the command from *cd* to *cat* or *ls* in the above scenario would produce the same failure to access ClearCase data.

In consideration of this behavior, avoid creating UNIX-level or VOB-level symbolic links whose texts are full pathnames — use relative pathnames only. For example:

```
% ls -l /vobs/aardvark
/vobs/aardvark -> ../all_projects/aardvark
```

For more on this topic, see the *pathnames_ccase* manual page.

Chapter 5

Defining View Configurations

This chapter presents a series of config specs that accomplish useful configuration management goals. For specificity, we use the following development environment:

Developers use a VOB whose VOB-tag is */proj/monet*, which has this structure:

```
/proj/monet         (VOB-tag, VOB mount point)
   src/             (C language source files)
   include/         (C language header files)
   lib/             (project's libraries)
```

For the purposes of this chapter, suppose that the *lib* directory has this substructure:

```
lib/
libcalc.a           (checked-in "staged" version of library)
libcmd.a            (checked-in "staged" version of library)
libparse.a          (checked-in "staged" version of library)
libpub.a            (checked-in "staged" version of library)
libaux1.a           (checked-in "staged" version of library)
libaux2.a           (checked-in "staged" version of library)

libcalc/            (sources for 'calc' library)
libcmd/             (sources for 'cmd' library)
libparse/           (sources for 'parse' library)
libpub/             (sources for 'pub' library)
libaux1/            (sources for 'aux1' library)
libaux2/            (sources for 'aux2' library)
```

Sources for libraries are located in subdirectories of *lib.* After a library is built in its source directory, it can be "staged" to */proj/monet/lib* by checking it in as a *DO version*. The build scripts for the project's executable programs can instruct the link editor, *ld(1*), to use the libraries in this directory (the "library staging area") instead of a more standard location (for example, */usr/local/lib*).

The following version labels have been assigned to versions of *monet* elements:

Version Labels	**Description**
R1.0	First customer release
R2_BL1	Baselevel 1 prior to second customer release
R2_BL2	Baselevel 2 prior to second customer release
R2.0	Second customer release

These version labels have been assigned to versions on the main branch of each element. Most of the project's development tasks take place on the main branch. For some special tasks, however, development takes places on a subbranch::

Subbranches	**Description**
major	Used for work on the application's graphical user interface, certain computational algorithms, and other major enhancements
r1_fix	Used for fixing bugs in Release 1.0

The following sections present ClearCase config specs, explaining in detail how each one achieves a particular configuration management goal.

Dynamic 'Mainline' View

This config spec defines a dynamic configuration, which automatically "sees" changes made on the *main* branch of every element — throughout the entire source tree, by any developer. See Example 5-1.

Example 5-1 Spec #1

```
element * CHECKEDOUT                                            (1)
element * /main/LATEST                                          (2)
```

This is ClearCase's default config spec, to which each newly-created view is initialized. (The *mkview* command automatically uses the contents of file */usr/atria/default_config_spec*.)

A view with this config spec provides a private work area that "sees" your checked-out versions (Rule 1). By default, a *checkout* command processes the currently-selected branch — in this case, the *main* branch (Rule 2). As long as an element remains checked-out, you can change it without affecting anyone else's work. As soon as you perform a *checkin*, the changes become visible instantly to other users whose views select */main/LATEST* versions.

The view also "sees" all other elements (that is, all elements that you have not checked out), on a read-only basis. If another user checks in a new version on the *main* branch of such an element, the new *LATEST* version appears in this dynamic view, automatically and instantly.

The Standard Configuration Rules

The two configuration rules in the default config spec will reappear in many of this chapter's examples. The *CHECKEDOUT* rule enables modification of existing elements. You *can* perform *checkout* commands in a view that lacks this rule, but your *view_server* process will complain:

```
% cleartool checkout -nc cmd.c

cleartool: Warning: Unable to rename "cmd.c" to "cmd.c.keep":
Read-only filesystem.

cleartool: Warning: Checked out version, but could not copy
to "cmd.c": File exists.

cleartool: Warning: Copied checked out version to
"cmd.c.checkedout".

Checked out "cmd.c" from version "/main/7".
```

In this example, the view continues to select version 7 of element *cmd.c*, which is read-only. A read-write copy of this version, *cmd.c.checkedout*, is created in view-private storage. (This is not a recommended way of working!)

The */main/LATEST* rule selects the most recent version on the *main* branch to appear in the view. Often, this is the version that represents the state-of-the-art for that element.

In addition, a */main/LATEST* rule is required to enable creation of new elements in a view. More precisely, you *can* create a new element in the absence of such a rule, but your view will then be unable to “see” the element you just created. (Element creation involves creating a *main* branch, and an empty version, */main/0*).

Omitting the Standard Configuration Rules

It makes sense to omit one or both of the standard configuration rules only if a view is *not* going to be used to modify data. For example, you might configure a “historical” view, to be used only for browsing old data, not for creating new data. Similarly, you might configure a view in which to compile and test only, or to verify that sources have been properly labeled.

Frozen View, Defined by Version Labels

This config spec defines a “frozen” configuration. See Example 5-2.

Example 5-2 Spec #2

```
element * R1.0 -nocheckout                                 (1)
```

The view always selects the same set of versions — the ones that have been labeled *R1.0*. In this scenario, all these versions are on the *main* branch of their elements; but this config spec works even if the *R1.0* version is on a subbranch.

Note: This assumes the R1.0 label type is “one-per-element”, not “one-per-branch” — see the mkbrtype manual page

To reinforce “frozenness”, the -*nocheckout* qualifier prevents any element from being checked out in this view. (It also prevents creation of new elements, since this requires the parent directory element to be checked-out.) Thus, there is no need for the standard *CHECKEDOUT* configuration rule.

Note: This configuration is not *completely* frozen, since version labels can be moved and deleted. For example, using the command *mklabel -replace* to move *R1.0* from version 5 of an element to version 7 would automatically change which version appears in the view. Similarly, using *rmlabel* would suppress the specified element(s) from the view. (The ClearCase *ls* command lists them with a *[no version selected]* annotation.) If the label type is locked with the *lock* command, the configuration becomes truly frozen.

This configuration is not appropriate for development. It might be used to rebuild Release 1.0, thus verifying that all source elements have been labeled appropriately. It might also be used by a developer or maintenance engineer to browse the old release.

As noted above, elements that have no version labeled *R1.0* will be suppressed from the view. This might include recently-created elements, elements from which the *R1.0* label has been removed, and elements in other VOBs.

Frozen View, Defined by Time

This config spec defines a “frozen” configuration in a slightly different way than the preceding one. See Example 5-3.

Example 5-3 Spec #3

```
element * /main/LATEST -time 4-Sep.02:00 -nocheckout      (1)
```

This configuration is “more frozen” than the preceding one: for each element, it selects the version that was the most recent on the *main* branch on September 4 at 2am (presumably, a time when no development was taking place). Subsequent *checkout*/*checkin* activity cannot change which versions satisfy this criterion — only deletion commands such as *rmver* or *rmelem* can change the configuration. As with the preceding config spec, the *-nocheckout* qualifier prevents elements from being checked out or created.

This configuration might be used to “roll back the clock” to a point when a consistent set of versions existed. If modifications must be made to this source base, you must modify the config spec to “unfreeze” the configuration (see Example 5-5).

View That Allows an ‘Old’ Configuration to be Modified

This config spec allows modifications to be made to a configuration defined with version labels. See Example 5-4.

Example 5-4 Spec #4

```
element * CHECKEDOUT                                       (1)
element * .../r1_fix/LATEST                                (2)
element * R1.0 -mkbranch r1_fix                            (3)
```

The configuration initially selects the same set of versions as Spec #2 in Example 5-2. This set of versions constitutes a *baselevel* configuration, which can then be modified:

- Elements can be checked out (Rule 1).
- The *checkout* command automatically creates a branch named *r1_fix* at the initially selected version (the *auto-make-branch* clause in Rule 3).

A key aspect of this scheme is that the same branch name, *r1_fix*, is used in every modified element. The only administrative overhead is the creation of a single branch type, *r1_fix*, with the *mkbrtype* command.

This config spec is efficient: just two rules (Rules 2 and 3) configure the appropriate versions of all elements:

- For elements that have not been modified, it is the most recent version on the *main* branch (Rule 2).
- For elements that have been modified, it is the most recent version on the *r1_fix* subbranch (Rule 3).

Figure 5-1 illustrates the two kinds of elements. In this illustration, the *r1_fix* branch is a subbranch of the *main* branch. But Rule #2 handles the more general case, too: the "..." wildcard allows the *r1_fix* branch to occur anywhere in any element's version tree, and at different locations in different elements' version trees.

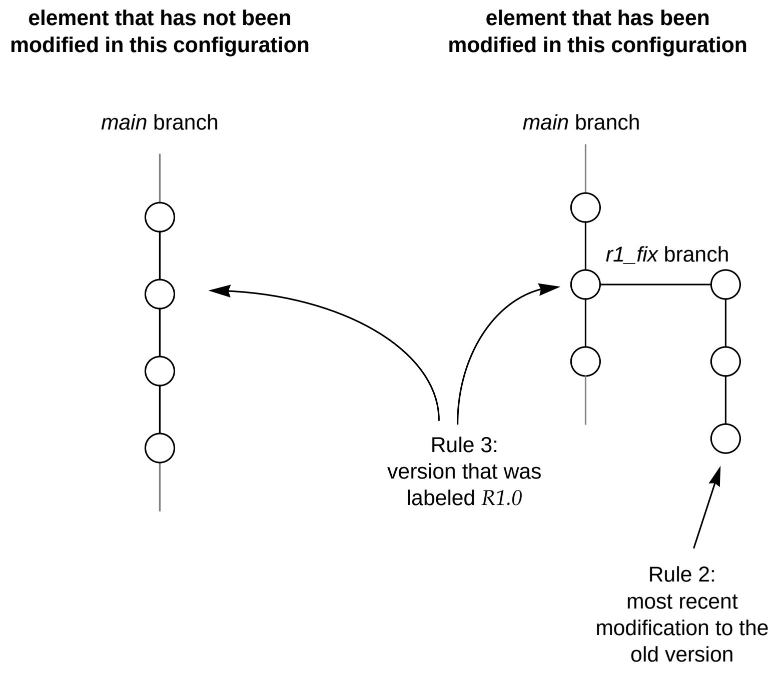

element that has not been modified in this configuration
element that has been modified in this configuration
main branch
main branch
r1_fix branch
Rule 3:
version that was labeled *R1.0*
Rule 2:
most recent modification to the old version

Figure 5-1 Making a Change to an Old Version

Where Is the '/main/LATEST' Rule?

This config spec lacks the standard */main/LATEST* rule. It is not useful for work with VOBs in which the version label *R1.0* does not exist. In addition, it is not useful in situations where new elements are created, as described in *Composing Your Own Config Spec* on page 75. If your organization forbids creation of new elements during maintenance of an old configuration, the lack of a */main/LATEST* rule is appropriate.

To allow creation of new elements during the modification process, add a fourth configuration rule:

```
element * CHECKEDOUT                              (1)
element * /main/r1_fix/LATEST                     (2)
element * R1.0 -mkbranch r1_fix              (3)
element * /main/LATEST -mkbranch r1_fix      (4)
```

When a new element is created with *mkelem*, the -mkbranch clause in Rule 4 causes the new element to be checked out on the *r1_fix* branch (which is automatically created). This conforms to the scheme of localizing all changes to *r1_fix* branches.

Variations on the Theme

This config spec as shown in Example 5-5 combines aspects of Spec #3 found in Example 5-3 and Spec #4 found in Example 5-4.

Example 5-5 Spec #5

```
element * CHECKEDOUT                                        (1)
element * /main/r1_fix/LATEST                               (2)
element * /main/LATEST -time 4-Sep:02:00 -mkbranch r1_fix (3)
```

This baselevel configuration is defined not with version labels like Rule 3 in Spec #4 (See Example 5-4), but with a -time rule as in Spec #3 (See Example 5-3).

View for New Development on a Branch

You can use this config spec for work that is to be isolated on branches named *major* as shown in Example 5-6.

Example 5-6 Spec #6

```
element * CHECKEDOUT                                        (1)
element * .../major/LATEST                                  (2)
element * BASELEVEL_X -mkbranch major                       (3)
element * /main/LATEST -mkbranch major                      (4)
```

The scheme is essentially similar to the one introduced above, in which all "fixup" work is performed on branches named *r1_fix*. Here, all work on a project (say, a command-line syntax overhaul) is isolated on branches named *major* (Rule 2).

Once again, *major* branches should be created at versions that constitute a consistent baselevel: a major release, a minor release, or just a set of versions that produces a working version of the application. In this config spec, the baselevel is defined by the version label *BASELEVEL_X*.

Variations on the Theme

Turning back the clock on a recent change — Sometimes, other developers checkin versions that become visible in your view, but which are incompatible with your own work. In such cases you can "turn back the clock" to a time before those changes were made. For example, Rule 2 in this variant shown in Example 5-7 turns back the clock on the branch to 4:00 PM on November 12.

Example 5-7 Spec #7

```
element * CHECKEDOUT                                     (1)
element * /main/major/LATEST -time 12-Nov.16:00          (2)
element * BASELEVEL_X -mkbranch major                    (3)
element * /main/LATEST -mkbranch major                   (4)
```

Note: Your own checkouts are unaffected by this rollback.

Config spec include files — ClearCase supports an "include file" facility that makes it easy to ensure that all members of the group are using the same config spec. For example, the configuration rules in Spec #7 as shown in Example 5-7 might be placed in file */public/config_specs/major.cspec*. Each developer then needs just a single-line config spec as shown in Example 5-8.

Example 5-8 Spec #8

```
include /public/config_specs/major.cspec                 (1)
```

If the project leader decides to modify this config spec (for example, to adopt the no-directory-branching policy), only the contents of */public/config_specs/major.cspec* need be changed. You can use this command to reconfigure your view with the modified spec:

```
% cleartool setcs -current
```

View That Implements Multiple-Level Branching

This config spec shown in Example 5-9 is a variant of Spec #6 (See Example 5-6); it implements and enforces consistent *multiple-level branching*:

Example 5-9 Spec #9

```
element * CHECKEDOUT             (1)
element * .../major/autumn/LATEST  (2)
element * .../major/LATEST -mkbranch autumn   (3)
element * BASELEVEL_X -mkbranch major      (4)
element * /main/LATEST -mkbranch major     (5)
```

A view configured with this config spec is appropriate in the following situation:

- As in Spec #6, all changes from the baselevel designated by the *BASELEVEL_X* version label must take place on a branch named *major*.
- Moreover, you are working on a special side-project, whose changes are to be made on a subbranch of *major*, named *autumn*.

(It is important for each modified element to have a *major* branch; it will be used to integrate all the changes made in your side-project and other sub-projects.) Figure 5-2 shows what happens in such a view when you *checkout* an element that has not been modified since the baselevel.

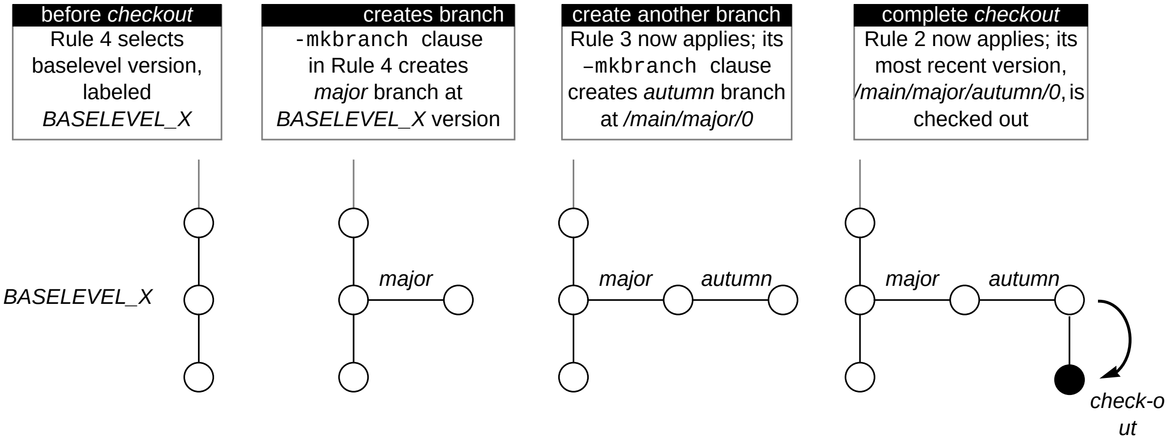

before *checkout*
Rule 4 selects baselevel version, labeled *BASELEVEL_X*
creates branch
-mkbranch clause in Rule 4 creates *major* branch at *BASELEVEL_X* version
create another branch
Rule 3 now applies; its –mkbranch clause creates *autumn* branch at */main/major/0*
complete *checkout*
Rule 2 now applies; its most recent version, */main/major/autumn/0*, is checked out
BASELEVEL_X
major
major
autumn
major
autumn
check-o ut

Figure 5-2 Multiple-Level Auto-Make-Branch

For more on multiple-level branching, see the *config_spec* and *checkout* manual page.

View That Selects Versions Using ‘External Criteria’

Suppose that some members of the development group working on the *major* branch (see Spec #6 in Example 5-6) are designated as the “QA team”. Individual developers are responsible for making sure that their modules pass a *lint*(1) check. The QA team builds and tests the application, using the most recent versions that have passed *lint*.

The QA team might work in a view with this config spec as shown in Example 5-10.

Example 5-10 Spec #10

```
element -file src/* /main/major/{lintOK=="Yes"}          (1)
element * /main/LATEST                                   (2)
```

This scheme calls for an attribute type, *lintOK*, to be created. Whenever a new version that passes *lint* is checked in on the *major* branch, an instance of *lintOK* with the value "`Yes`" is attached to that version. (This might be performed manually or with a ClearCase trigger.)

If an element in the */src* directory has been edited on the *major* branch, this view selects the branch’s most recent version that has been marked as passing *lint* (Rule 1). If no version has been so marked, or if no *major* branch has been created, the most recent version on the *main* branch is used (Rule 2).

Note: Rule 1 on this config spec does not provide a match if an element has a *major* branch, but no version on the branch has a *lintOK* attribute. This command can locate the no-such-attribute branches:

```
% cleartool find . -branch '{brtype(major) \
&& \! attype_sub(lintOK)}' -print
```

The backslash “\” character is required in the C shell only, to keep the exclamation point “!” from indicating a history substitution. The `attype_sub` primitive searches for attributes throughout an element — on its versions and branches, as well as on the element itself.

This scheme allows the QA team to track the progress of the rest of the group, without having to keep absolutely up-to-date. The development config spec always selects the most recent version on the *major* branch, but the QA config spec may select an intermediate version (Figure 5-3).

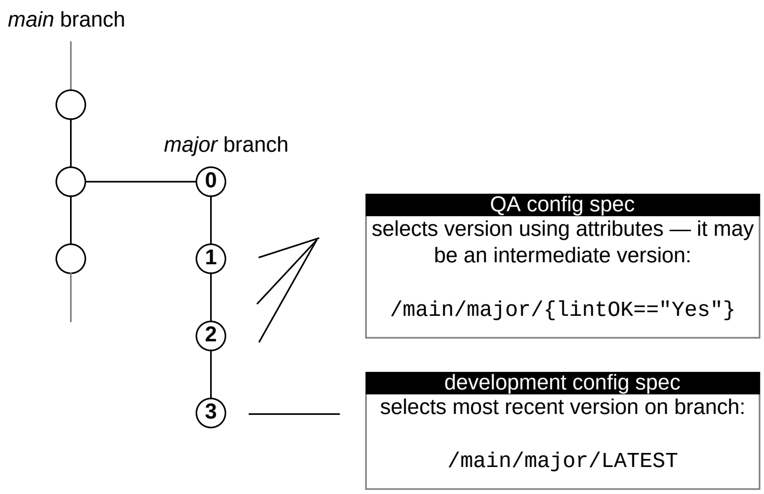

main branch
major branch
0
1
2
3
QA config spec
selects version using attributes — it may be an intermediate version:
/main/major/{lintOK=="Yes"}
development config spec
selects most recent version on branch:
/main/major/LATEST

Figure 5-3 Development Config Spec vs. QA Config Spec

Can This Configuration Be Used for Development?

It might be tempting to add a “CHECKEDOUT” rule to the above config spec, turning the “QA configuration” into a development configuration” shown in Example 5-11.

Example 5-11 Spec #11

```
element * CHECKEDOUT                                      (0)
element -file src/* /main/major/{lintOK=="Yes"}           (1)
element * /main/LATEST                                    (2)
```

More generally, it may seem desirable to use attributes, or other kinds of meta-data in addition to (or instead of) branches to control version- selection in a development view. But such schemes involve complications. Suppose that the config spec above selects version */main/major/2* of element *.../src/cmd.c* (Figure 5-4).

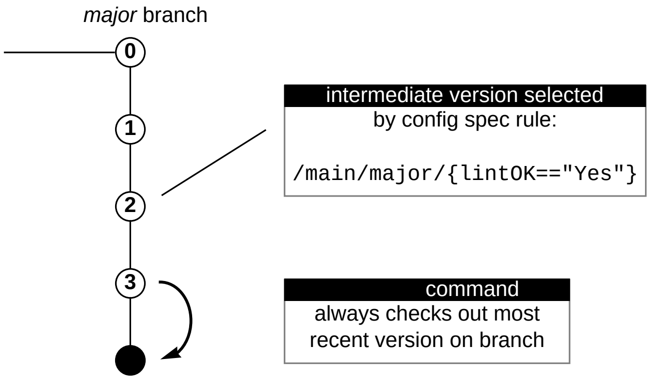


Figure 5-4 Checking Out a Branch of an Element

Performing a *checkout* in this view checks out version */main/major/*3, not version */main/major/*2:

```
cleartool: Warning: Version checked out is different from
version previously selected by view.
Checked out "cmd.c" from version "/main/major/3".
```

This reflects the ClearCase restriction that new versions can be created only at the end of a branch. While such operations are possible, they are potentially confusing to users. And in this situation, it is almost certainly not what the developer performing the checkout desired.

You can avoid the “wrong-version-checked-out” problem by modifying the config spec and creating another branching level at the attribute-selected version. The new config spec might be as shown in Example 5-12.

Example 5-12 Spec #12

```
element * CHECKEDOUT                                     (0)
element * /main/major/temp/LATEST                       (0a)
element -file src/* /main/major/{lintOK=="Yes"}
  -mkbranch temp                                         (1)
element * /main/LATEST                                   (2)
```

View That Shows Only One Developer’s Changes

This config spec shown in Example 5-13 makes it easy to peruse all of the changes a developer has made since a certain milestone.

Example 5-13 Spec #13

```
element * '/main/{created_by(jackson) && created_since(25-Apr)} '          (1)
element * /main/LATEST -time 25-Apr
```

Note: Rule 1 must be wholly contained on a single physical text line.

A particular date, April 25, is used as the milestone. The configuration is a “snapshot” of the main line of development at that date (Rule 2), overlaid with all changes that user *jackson* has made on the *main* branch since then (Rule 1).

Directory listings made by the ClearCase *ls* command distinguish *jackson*’s files from the others: each listing entry includes an annotation as to which configuration rule applies to the selected version.

This is a “perusal view”, not a development view. The selected set of files may not be consistent: some of *jackson*’s changes may rely on changes made by others, and those other changes are excluded from this view. Thus, this config spec lacks the standard “CHECKEDOUT” and “/main/LATEST” rules.

View That Restricts Changes to a Single Directory

This config spec shown in Example 5-14 is appropriate for a developer who is to be restricted to making changes in just one directory, */proj/monet/src*.

Example 5-14 Spec #14

```
element * CHECKEDOUT                                        (1)
element src/* /main/LATEST                                  (2)
element * /main/LATEST -nocheckout                          (3)
```

The most recent version of each element is selected (Rules 2 and 3), but Rule 3 prevents checkouts to all elements except those in the desired directory.

Note: Rule 2 matches elements in *any* directory named *src*, in any VOB. The pattern */proj/monet/src/** would restrict matching to just one VOB.

This config spec can easily be extended with additional rules that allow additional areas of the source tree to be modified.

View That Uses Results of a Nightly Build

Many development organizations use scripts to perform unattended software builds each night. Such "nightly builds" verify that the application is still buildable. In layered build environments, they can also provide up-to-date versions of lower-level software: libraries, utility programs, and so on.

Suppose that each night, a script:

- builds libraries in various subdirectories of */proj/monet/lib*
- checks them in as DO versions in the "library staging area", */proj/monet/lib*
- labels the versions *LAST_NIGHT*

You can use this config spec shown in Example 5-15 if you wish to use the libraries produced by the nightly builds.

Example 5-15 Spec #15

```
element * CHECKEDOUT                                        (1)
element lib/*.a LAST_NIGHT                                  (2)
element lib/*.a R2_BL2                                      (3)
element */main/LATEST                                       (4)
```

The *LAST_NIGHT* version of a library is selected whenever such a version exists (Rule 2). If a nightly build fails, the previous night's build will still have the *LAST_NIGHT* label, and will be selected. If no *LAST_NIGHT* version exists (the library is not currently under development), the stable version labeled *R2_BL2* is used instead (Rule 3).

For each library, selecting on the *LAST_NIGHT* label rather than simply taking the most recent version in the staging area allows new versions to be staged during the next day, without affecting developers who use this config spec.

Variations on the Theme

The scheme described above uses version labels to select particular versions of libraries. For more flexibility, the *LAST_NIGHT* version of some libraries might be selected, the *R2_BL2* version of others, and the most recent version of still others as shown in Example 5-16.

Example 5-16 Spec #16

```
element * CHECKEDOUT                                          (1)
element lib/libcmd.a LAST_NIGHT                              (2a)
element lib/libparse.a LAST_NIGHT                            (2b)
element lib/libcalc.a R2_BL2                                 (3a)
element lib/*.a /main/LATEST                                 (3b)
element * /main/LATEST                                        (4)
```

(Rule 3b is not required here, since Rule 4 would handle “all other libraries”. It is included for clarity only.)

Other kinds of meta-data could also be used to select library versions. For example, *lib_selector* attributes might take values such as "*experimental*", "`stable`", and "*released*". A config spec might mix-and-match library versions as shown in Example 5-17.

Example 5-17 Spec #17

```
element * CHECKEDOUT                                          (1)
element lib/libcmd.a {lib_selector=="experimental"}           (2)
element lib/libcalc.a {lib_selector=="experimental"}          (3)
element lib/libparse.a {lib_selector=="stable"}               (4)
element lib/*.a {lib_selector=="released"}                    (5)
element * /main/LATEST                                        (6)
```

Playing Mix-and-Match with Application Subsystems

This config spec shown in Example 5-18 extends the scheme used in Spec #15 through Spec #17 from programming libraries to the application's subsystems.

Example 5-18 Spec #18

```
element * CHECKEDOUT                                          (1)
element /proj/monet/lib/... R2_BL1                            (2)
element /proj/monet/include/... R2_BL2                        (3)
element /proj/monet/src/... /main/LATEST                      (4)
element * /main/LATEST                                        (5)
```

In this situation, a developer is making changes to the application's source files on the *main* branch (Rule 4). Builds of the application use the libraries in directory */lib* that were used to build Baselevel 1, and the header files in directory */include* that were used to build Baselevel 2.

Selecting Versions That Built a Particular Program

This config spec shown in Example 5-19 defines a "sparse view", which sees just enough files to rebuild a particular program or peruse its sources.

Example 5-19 Spec #19

```
element * -config /proj/monet/src/monet                       (1)
```

All elements that were *not* involved in the build of *monet* will be listed by ClearCase *ls* with a *[no version selected]* annotation.

This config spec selects the versions listed in the config rec of a particular derived object (and in the config recs of all its build dependencies). It can be a derived object that was built in the current view, or another view, or it can be a *DO version*.

In this config spec as shown in Example 5-20, *monet* is a derived object in the current view. You can reference a derived object in another view with an extended pathname that includes a *DO-ID*.

Example 5-20 Spec #20

```
element * -config /proj/monet/src/monet@@09-Feb.13:56.812
```

But typically, this kind of config spec is used to configure a view from a derived object that has been checked in as a DO version.

Configuring the Makefile

By default, a derived object's config rec does *not* list the version of the *makefile* that was used to build it. Instead, the config rec includes a copy of the build script itself. (Why? — when a new version of the *makefile* is created with a revision to one target's build script, the config recs of all other derived objects built with that *makefile* are not rendered out-of-date.)

But if the *monet* program is to be rebuilt in this view using clearmake (or even standard *make*), a version of the *makefile* must be selected somehow. You can have *clearmake* record the *makefile* version in the config rec by including the special clearmake macro $(MAKEFILE) in the target's dependency list:

```
monet: $(MAKEFILE) monet.o ...
       cc -o monet ...
```

clearmake always records the versions of explicit dependencies in the config rec.

Alternatively, you can configure the makefile at the source level: attach a version label to the *makefile* at build time, then use a config spec like Spec #2 or Spec #4 to configure a view for building.

Making a Fix in the Program

If a bug is discovered in the *monet* program, as rebuilt in a view configured with Spec #19 (See Example 5-19), it is easy to convert the view from a perusal/build configuration to a development configuration. As usual when making changes in "old" sources, the strategy is to:

- create a branch at each version to be modified
- use the same branch name (that is, create an instance of the same branch type) in every element

If the "fixup" branch type is *r1_fix*, then this modified config spec shown in Example 5-21 reconfigures the view for performing the fix.

Example 5-21 Spec #21

```
element * CHECKEDOUT                                      (1)
element * .../r1_fix/LATEST                               (2)
element * -config /proj/monet/src/monet -mkbranch r1_fix  (3)
element * /main/LATEST -mkbranch r1_fix                   (4)
```

Selecting Versions That Built a Set of Programs

It is easy to expand Spec #19 (See Example 5-21) so that it configures a view with the sources used to build a *set* of programs, rather than a single program as shown in Example 5-22.

Example 5-22 Spec #22

```
element * -config /proj/monet/src/monet                   (1)
element * -config /proj/monet/src/xmonet                  (2)
element * -config /proj/monet/src/monet_conf              (3)
```

There can be version conflicts in such configurations, however. For example, different versions of file *params.h* might have been used in the builds of *monet* and *xmonet*. In this situation, the version used in *monet* is configured, since its configuration rule came first. Similarly, there can be conflicts when using a single *-config* rule: if the specified derived object was created by actually building some targets, but simply using DO versions of other targets, multiple versions of some source files might be involved.

You can apply a transformation to this config spec similar to the one that transforms Spec #19 to Spec #21, in order to change the perusal/build configuration to a development configuration.

Chapter 6

Working in a Parallel Development Environment

This chapter describes techniques for working in an environment where elements are to be branched and merged.

Parallel Development Using Branches

ClearCase supports *parallel development*, in which an element evolves simultaneously along several branches, with new versions being added to each branch independently. Parallel development has many uses:

- It allows different projects — for example, different versions of a product — to use the same source tree(s) at the same time.
- It isolates the work of developers whose changes should not (yet) be dynamically shared with others.
- It isolates work that should never be shared with others — for example, a bugfix in a “retired” release.
- It prevents roadblocks — development need not cease during a software integration period; it can proceed on branches, to be reintegrated later.

A version tree can have any number of branches, organized in a hierarchy of arbitrary depth. A checkout can be performed on the *main* branch in one view, on the *bug404* branch in another view, on the *motif* branch in yet another view, and so on.

Branches are merely logical structures for organizing the versions of an element. The storage requirement for an element with 100 versions is the same whether the version tree has a single branch, or many branches organized into a deep hierarchy. (In either case, all the versions of an element of type *text_file* are stored as *deltas* in a single structured file.)

'Working on a Subbranch'

ClearCase keeps your work on a project organized by operating at two levels (and keeping the levels synchronized):

- **Version control** — At the individual-element level, new versions created for a particular project go onto a dedicated branch, as discussed above. This isolates the project's changes from other concurrent work.
- **Configuration management** — At the aggregate ("entire-set-of-sources") level, all the branches created in individual elements' version trees must form a well-defined, isolated group. The simplest way (and the ClearCase-standard way) to accomplish this grouping is to require that all the branches have the same name.

All like-named branches are created as instances of a single object, the *branch type*. When you modify a set of elements for a project, you create a branch in each element's version tree (or, more typically, ClearCase creates the branches for you). All the branches are instances of the same branch type, and have the same name. Thus, working on the project is often termed "working on a subbranch", an abbreviation for "working on a set of like-named subbranches".

Note: Each VOB must have its own branch type object.

Setting Up a View for Parallel Development

Chapter 4, "Setting Up a View," included a general discussion of setting up a view. This section presents a view-setup procedure for (perhaps) the most typical situation: launching a new project that will work "on a branch", starting from a well-defined *baselevel*. Suppose that:

- Your username is *sakai*.
- The baselevel is defined as "all versions labeled *PROJ_BASE*".
- Work for the project is to take place on branches names *koala*.

The following steps set up a view to work on this project, assuming that the *PROJ_BASE* version labels are all in place.

1. **Create the branch type —** By convention, names for branch types have lowercase letters only:

```
% cleartool mkbrtype koala
Comments for "koala":
branch type  for KOALA project, from baselevel defined by
label PROJ_BASE
.
Created branch type "koala".
```

2. **Create the view** — See Chapter 4, "Setting Up a View," for a discussion of selecting a view's storage location, its view-tag, and so on.

```
% cleartool mkview -tag sakai_koala ~/view/koala.vws
Host-local path: pluto:/home/sakai/views/koala.vws
Global path:     /net/pluto/home/sakai/views/koala.vws
It has the following rights:
User : sakai    : rwx
Group: user     : rwx
Other:          : r-x
```

3. **Configure the view for your project —** Establish the standard "working on a branch" config spec, using the particular version label and branch names for your project.

```
% cleartool edcs -tag sakai_koala
  .
  .  use text editor to revise current config spec:
     element * CHECKEDOUT
     element * .../koala/LATEST
     element * PROJ_BASE -mkbranch koala
     element * /main/LATEST -mkbranch koala
```

This view is now ready to be used for project-specific work.

Automatic Creation of Branches

The last two config spec rules listed in Step 3 take advantage of ClearCase's *auto-make-branch* facility. Using this feature guarantees that all modifications you make to elements in this view will be made on a *koala* branch. It also simplifies your work, by automatically creating such branches, as needed — that is, the first time you *checkout* an element in this view.

The *auto-make-branch* facility hides the difference between "working on the main branch" and "working on a subbranch" — *checkout* and *checkin* are the only commands you need to modify sources on branches, and you do not even need to remember the branch name! For most purposes, you can "forget" about the special way in which your work is organized, leaving that job to ClearCase.

Working in a Multiple-View Environment

This section presents some useful techniques for working in a group environment where each user works in a separate view. Before accessing a view — in particular, one whose storage directory is located on a remote host — you may first need to activate the view on your host. You can check the ClearCase *viewroot* directory to determine whether the view is active. For example, this command determines whether view *alpha* is active on your host:

```
% ls /view | grep alpha
```

An active view appears as a subdirectory entry in the viewroot directory, */view*.

Alternatively, enter an *lsview* command to determine whether a view is active on your host:

```
% cleartool lsview alpha
* alpha           /net/ccsvr01/shared_views/alpha.vws
```

Note: Asterisk (*) indicates view is active on the local host.

If a view is not active on your host, enter a *startview* command to activate it:

```
% cleartool startview alpha
```

Using a File in Another View

If you are set to a particular view, you may still occasionally wish to access data stored in (or merely visible in) other views. You can use a view-extended pathname access any object within another active view:

```
% grep 'order-dependent' /view/alpha/vobs/project/src/util.c
```

View-extended pathnames work no matter what your current view context: you can be set to another view, set to the *same* view, or not set to any view at all.

Comparing Your Version of an Element to Another View's

You may sometimes wish to compare a version selected by your view with the version of the same element selected by another view. If the element appears at the same pathname in both views, you can use the *command substitution* feature of UNIX shells:

```
% cleartool setview david
% cd /vobs/project/src
% cleardiff util.c /view/alpha/‘pwd‘/util.c
```

You could also use the standard *diff*(1) command, or access *cleardiff* with a *cleartool diff* command. Some variants of *diff* allow you to omit the redundant "util.c" at the end of the command.

Resolving Namespace Differences between Views

Sometime, the same element appears at *different* pathnames in different views. ClearCase can track directory-level changes, from simple renamings to wholesale reorganizations. In such situations, a colleague may direct your attention to a particular element, using a pathname that is not valid in your view. Given the "foreign" pathname to the object, you can use a *describe -cview* command to determine its pathname in your own view:

```
% cleartool describe -cview \
 /view/gordon/vobs/project/include/hello_base.h@@
file element "/vobs/project/src/hello.h@@"
  created 20-May-93.14:46:00 by rick.devt@saturn
  .
  .
```

You might then compare your version of the element with your colleague's version as follows:

```
% cleardiff hello.h \
 /view/gordon/vobs/project/include/hello_base.h
```

Merging Versions of an Element

In a parallel development environment, the "flip side" of branching is *merging*. In the simplest scenario, the changes made by a project "on a subbranch" are incorporated back into the *main* branch:

- If there has been no activity on the *main* branch of an element in the meantime, this involves a trivial copy operation.
- If an element's *main* branch has evolved, than an intelligent manual or (preferably) automated merge operation is required.

More generally, work from *any* branch can be merged into any other branch. ClearCase includes automated merge facilities for handling just about any scenario. Often, all of a project's work can be integrated with other work using a single command.

A merge involves computation of the "sum-of-pairwise-differences" of a set of files and/or versions (Figure 6-1).

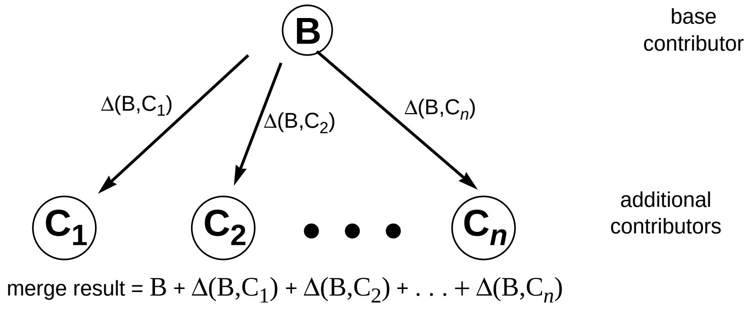
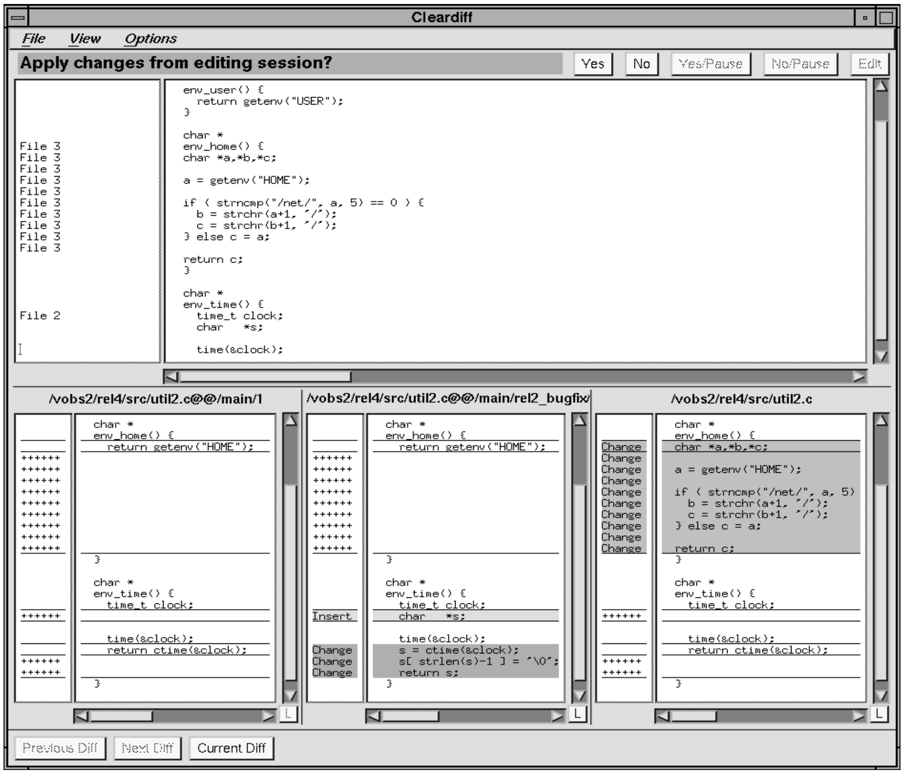


Figure 6-1 ClearCase Merger Algorithm

Critical to this algorithm is the appropriate selection of one file to be the *base contributor*. Usually, ClearCase selects the base automatically, and can even take into account previous merges, in order to simplify and speed its work. For special

cases, you can specify a particular base contributor to the *merge* command; this command also has options (*-insert* and *-delete*) to implement common special cases.

The *xcleardiff* utility makes it easy to merge versions of an element, even when you need to make "manual" adjustments. Merge output appears in an *edit panel*, in which you can make minor edits directly; for more significant changes, you can pause the *xcleardiff* session to invoke your favorite text editor at any time. The current contents of the edit panel are channeled to your editor; when you return to xcleardiff, the changes you've made "externally" appear the edit panel, appropriately annotated (Figure 6-2).



Apply changes from editing session?

```
                    env_user() {
                      return getenv("USER");
                    }

                    char *
File 3              env_home() {
File 3              char *a,*b,*c;
File 3
File 3              a = getenv("HOME");
File 3
File 3              if ( strncmp("/net/", a, 5) == 0 ) {
File 3                b = strchr(a+1, '/');
File 3                c = strchr(b+1, '/');
File 3              } else c = a;
File 3
                    return c;
                    }

                    char *
                    env_time() {
File 2                time_t clock;
                      char   *s;

                      time(&clock);
```

/vobs2/rel4/src/util2.c@@/main/1

```
            char *
            env_home() {
              return getenv("HOME");
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
            }

            char *
            env_time() {
              time_t clock;
++++++

              time(&clock);
              return ctime(&clock);
++++++
++++++
            }
```

/vobs2/rel4/src/util2.c@@/main/rel2_bugfix/

```
            char *
            env_home() {
              return getenv("HOME");
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
            }

            char *
            env_time() {
              time_t clock;
Insert        char   *s;

              time(&clock);
Change        s = ctime(&clock);
Change        s[ strlen(s)-1 ] = '\0';
Change        return s;
            }
```

/vobs2/rel4/src/util2.c

```
            char *
            env_home() {
Change      char *a,*b,*c;
Change
Change      a = getenv("HOME");
Change
Change      if ( strncmp("/net/", a, 5)
Change        b = strchr(a+1, '/');
Change        c = strchr(b+1, '/');
Change      } else c = a;
Change
Change      return c;
            }

            char *
            env_time() {
              time_t clock;
++++++

              time(&clock);
              return ctime(&clock);
++++++
++++++
            }
```

Figure 6-2 *xcleardiff* Graphical Merge Utility

The following sections present a series of "merge scenarios" — situations that call for information on one branch of an element to be incorporated into another branch. In each scenario, we show the version tree of an element that requires a merge, and indicate the appropriate command to perform the merge.

Scenario: Merging All the Changes Made on a Subbranch

This is the simplest case (Figure 6-3). Bugfixes for an element named *opt.c* are being made on branch *r1_fix*, which was created at the baselevel version *RLS1.0* (*/main/4*). Now, all the changes made on the subbranch are to be incorporated back into *main*, where a few new versions have been created in the meantime.

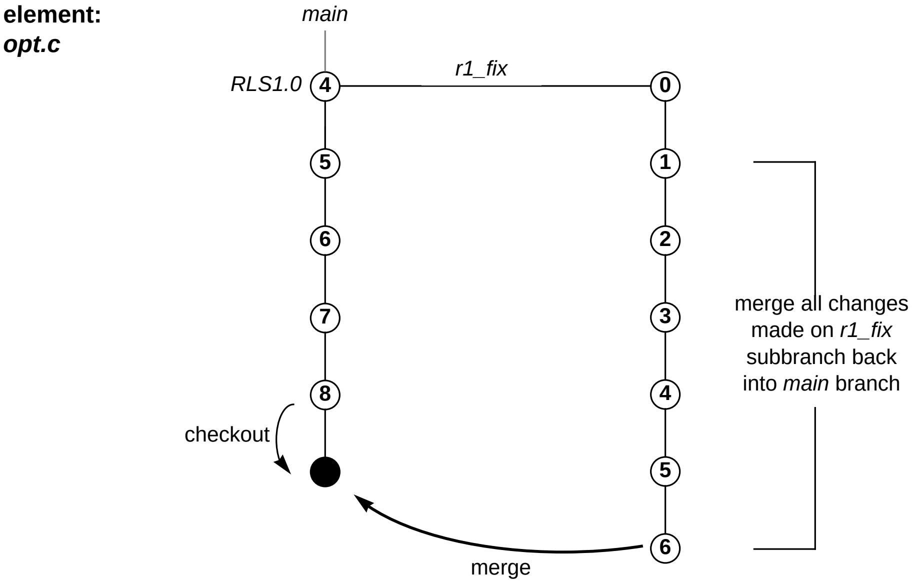


Figure 6-3 Version Tree of an Element Requiring a Merge

Set a view configured with the default config spec:

```
element * CHECKEDOUT
element * /main/LATEST
```

Go to the source directory and enter this command to perform the merge:

```
% cleartool findmerge opt.c -fversion .../r1_fix/LATEST
```

Scenario: Selective Merge from a Subbranch

This a variant of the preceding merge scenario. The project leader wants the changes in version */main/r1_fix/4* (and *only* that version — it's a particularly critical bugfix) to be incorporated into new development. In performing the merge, you specify which version(s) on the *r1_fix* branch to be included as shown in Figure 6-4.

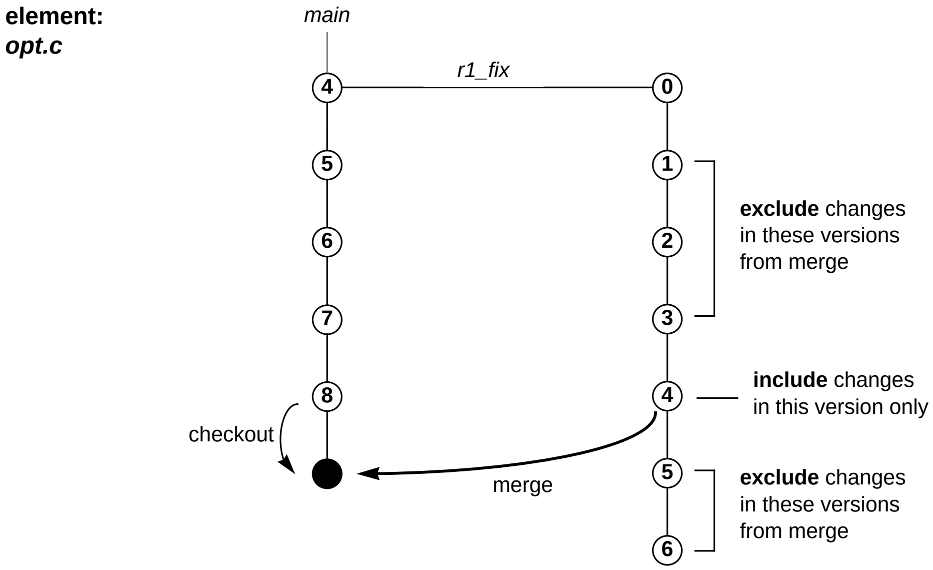


Figure 6-4 Selective Merge of a Version from a Subbranch

In a view configured with the default config spec, enter these commands to perform the selective merge:

```
% cleartool checkout opt.c
% cleartool merge -to opt.c -insert -version /main/r1_fix/4
```

You can also specify a range of consecutive versions to be merged. For example, this command merges selects only the changes in versions */main/r1_fix/2* through */main/r1_fix/4*:

```
% cleartool merge -to opt.c -insert -version /main/r1_fix/2 \
/main/r1_fix/4
```

Scenario: Removing the Contributions of Some Versions

The project leader has decided that a new feature, implemented in versions 14 through 16 on the *main* branch in Figure 6-5, will not be included in the product. You must perform a "subtractive merge" to remove the changes made in those versions.

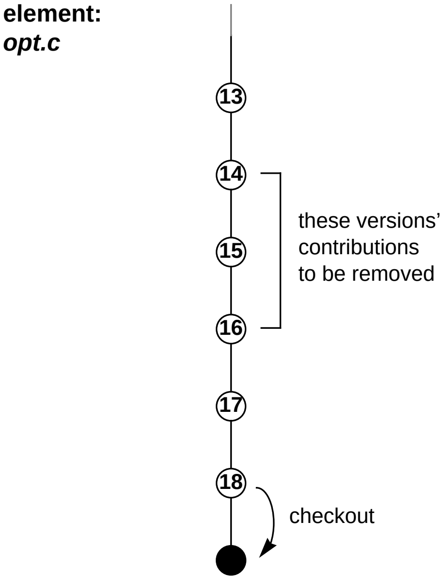


Figure 6-5 Subtractive Merge

Enter these commands to perform the subtractive merge:

```
% cleartool checkout opt.c
% cleartool merge -to opt.c -delete -version /main/14 /main/16
```

Scenario: Merging an Unreserved Checkout

ClearCase allows the same version to have several checkouts at the same, each in a different view. At most one of the checkouts is *reserved* — all the others (or perhaps every one of them) is *unreserved*. This mechanism allows several users to work on the same file at the same time, without having to use separate branches.

To prevent confusion and loss of data in such situations, ClearCase imposes a constraint: if the version from which you performed an unreserved checkout (version 7 in Figure 6-6) is not the most recent version on the branch, then you cannot simply checkin your work — this would obliterate the contributions of the versions created in the interim (versions 8 and 9 in Figure 6-6). Instead, you must first merge the most recent version into your checked-out version, then perform the *checkin*.

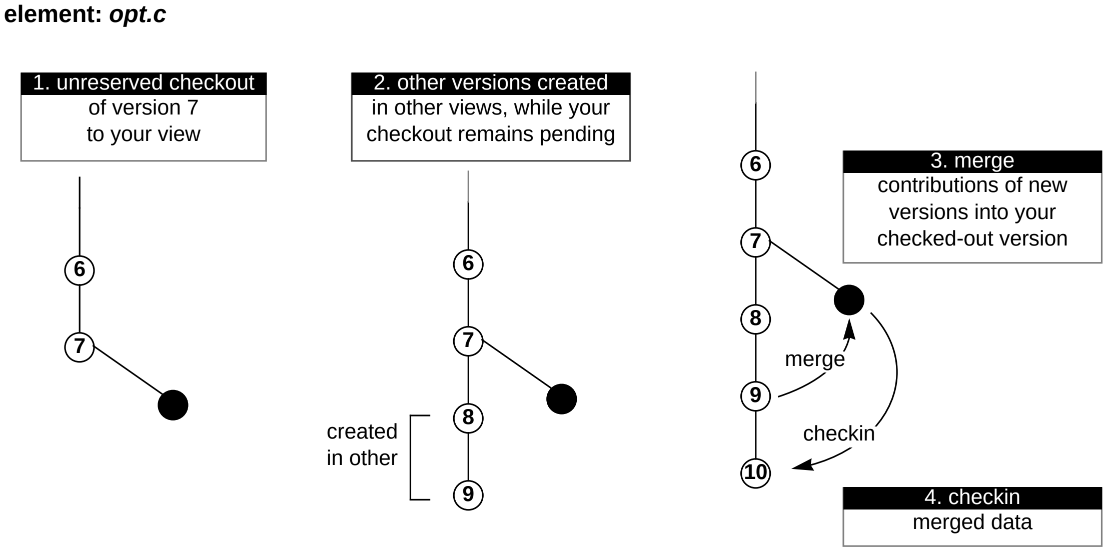

element: *opt.c*
1. unreserved checkout of version 7 to your view
6
7
2. other versions created in other views, while your checkout remains pending
6
7
8
9
created in other
6
7
8
9
10
3. merge contributions of new versions into your checked-out version
merge
checkin
4. checkin merged data

Figure 6-6 Merge of an Unreserved Checkout

Enter these commands to merge your unreserved checkout:

```
% cleartool merge -to opt.c -version /main/9
                                        (Step 3 in Figure 6-6)
% cleartool checkin opt.c
                                        (Step 4 in Figure 6-6)
```

Scenario: Merging All of a Project's Work

In the preceding scenarios, a merge was performed on a single element; now for a more realistic scenario. Suppose a team of developers has been working in isolation on a project for an extended period (weeks or months). Now, your job is to merge *all* the changes back into the *main* branch.

The *findmerge* command has the tools to handle most common cases easily. It can accommodate the following schemes for isolating the project's work.

All of Project's Work Isolated "On a Branch"

In the standard ClearCase approach to parallel development, all of a project's work takes place "on the same branch". More precisely, new versions of source files are all created on like-named branches of their respective elements (that is, on branches that are instances of the same *branch type*). This makes it possible for a single *findmerge* command to locate and incorporate all the changes. Suppose the common branch is named *gopher.* You can enter these commands in a "mainline" view, configured with the default config spec:

```
% cd root-of-source-tree
% cleartool findmerge . -fversion .../gopher/LATEST \
-merge -xmerge
```

The *-merge -xmerge* syntax causes the merge to take place automatically whenever possible, and to bring up the graphical merge utility if an element's merge requires user interaction (see Chapter 7, "Comparing and Merging Files Graphically with xcleardiff.") If the project has made changes in several VOBs, you can perform all the merges at once by specifying several pathnames, or by using the *-avobs* option to *findmerge*.

All of Project’s Work Isolated “In a View”

Some projects are organized so that all changes are performed in a single view (typically, a shared view). For such projects, use the *-ftag* option to *findmerge*. Suppose the project’s work has been performed in a view whose view-tag is *goph_vu*. These commands perform the merge:

```
% cd root-of-source-tree
% cleartool findmerge . -ftag goph_vu -merge -xmerge
```

Scenario: Merging a New Release of an Entire Source Tree

Here is another project-level merge scenario as shown in Figure 6-7. Your group has been using an externally-supplied source-code product, maintaining the sources in a VOB. The successive versions supplied by the vendor are checked into the *main* branch and labeled *VEND_R1* through *VEND_R3.* Your group’s fixes and enhancements are created on subbranch *enhance.* The views in which your group works are all configured to branch from the *VEND_R3* baselevel:

```
element * CHECKEDOUT
element * .../enhance/LATEST
element * VEND_R3 -mkbranch enhance
element * /main/LATEST -mkbranch enhance
```

- The version trees illustrated below shows three different likely cases:
- an element that your group started changing at Release 1 (*enhance* branch created at the version labeled *VEND_R1*)
- an element that your group started changing at Release 3
- an element that your group has never changed

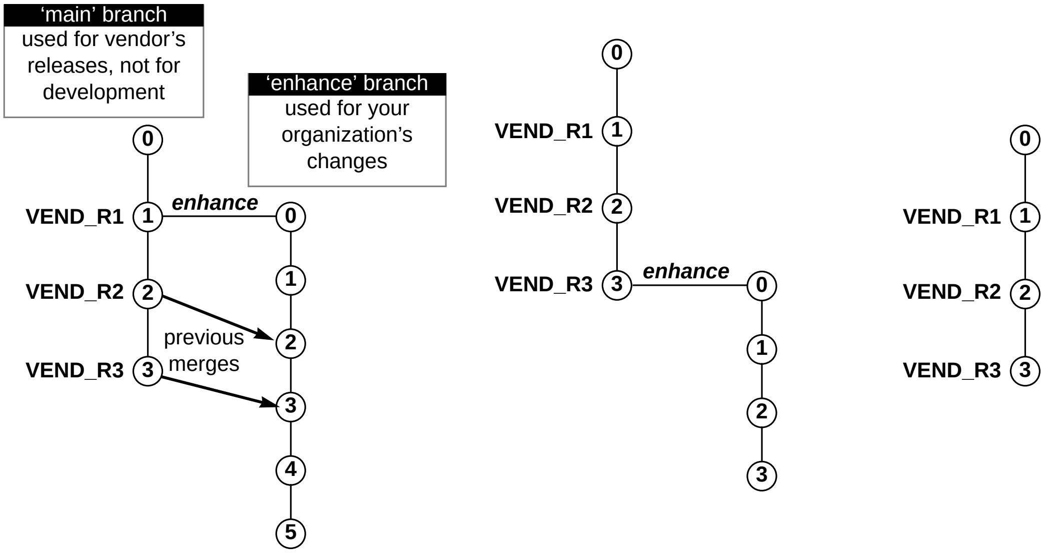


Figure 6-7 Merge a New Release of an Entire Source Tree

Now, a tape containing Release 4 arrives, and you need to integrate the new release with your group's changes. After you add the new release to the *main* branch and label the versions *VEND_R4*, it will be easy to specify the merge process: "for all elements, merge from the version labeled *VEND_R4* to the most recent version on the *enhance* branch; if an element has no *enhance* branch, don't do anything at all"

Here's a complete procedure for accomplishing the integration:

1. Load the vendor's "Release 4" tape into a standard UNIX directory tree:

   ```
   % cd /usr/tmp
   % tar -xv
   ```

   Suppose this creates directory tree *mathlib_4.0*.

2. As the VOB owner, run the UNIX-to-ClearCase conversion utility, *clearcvt_unix,* to create a script that will add new versions to the *main* branches of elements (and very likely, to create new elements, as well).

   ```
   % cd ./mathlib_4.0
   % clearcvt_unix
    . (lots of output)
    .
   ```

3. In a view configured with the default config spec, run the conversion script created by *clearcvt_unix,* creating Release 4 versions on the *main* branches on elements:

   ```
   % cleartool setview mainline
   % cd /vobs/proj/mathlib
   % /usr/tmp/mathlib_4.0/cvt_dir/cvt_script
   ```

4. Label the new versions:

   ```
   % cleartool mklbtype -c "Release 4 of MathLib \
           sources" VEND_R4
   Created label type "VEND_R4".
   % cleartool mklabel -recurse VEND_R4 /vobs/proj/mathlib
    . (lots of output)
    .
   ```

5. Go to a view that is configured with your group's config spec, selecting the versions on the *enhance* branch:

   ```
   % cleartool setview enh_vu
   ```

6. Merge from the *VEND_R4* configuration to your view:

   ```
   % cleartool findmerge /vobs/proj/mathlib -fver \
           VEND_R4 -merge -xmerge
   ```

   The *-merge -xmerge* syntax says "merge automatically if possible; but if not possible, bring up the graphical merge tool".

7. Verify the merges, and *checkin* the modified elements.

8. You have now established Release 4 as the new baselevel. Developers in your group can update their views' configurations as follows:

   ```
   element * CHECKEDOUT
   element * .../enhance/LATEST
   element * VEND_R4 -mkbranch enhance
   element * /main/LATEST -mkbranch enhance
   ```

   Note: VEND_R3 was changed to VEND_R4 in Step 8.

   Elements that have been active will continue to evolve on their *enhance* branches. Elements that are revised for the first time will have their *enhance* branches created automatically at the *VEND_R4* version.

Scenario: Merging Directory Versions

One of ClearCase's most powerful features is versioning of directories. Each version of a directory element catalogs a set of file elements, directory elements, and VOB symbolic links. In a parallel development environment, directory-level changes are just as frequent as file-level changes. And with ClearCase, merging the changes to another branch is just as easy.

Let's take a closer look at the externally-supplied source tree scenario from the preceding section: suppose you find that in the vendor's new release (the one you've labeled *VEND_R3*), several changes have been made in directory */vobs/proj/mathlib/src*:

1. file elements *Makefile*, *getcwd.c*, and *fork3.c* have been revised
2. file elements *readln.c* and *get.c* have been deleted
3. a new file element, *newpaths.c*, has been created

When you use *findmerge* to merge the changes made in the VEND_R3 release out to the *enhance* branch (Step #6 in "Scenario: Merging a New Release of an Entire Source Tree" on page 120), both the file-level changes (Step #1 in this section "Scenario: Merging Directory Versions"), and the directory-level changes (Step #2 and #3 in this section "Scenario: Merging Directory Versions"), are handled automatically.

The following *findmerge* excerpt in Example 6-1 shows the directory-level merge activity.

Example 6-1 Directory -level Merge Activity

```
********************************
<<< directory 1: /vobs/proj/mathlib/src@@/main/3
>>> directory 2: .@@/main/enhance/1
>>> directory 3: .
********************************
----[ removed directory 1 ]----|----[ directory 2 ]----
get.c  19-Dec-1991 drp         |-
*** Automatic: Applying REMOVE from directory 2
----[ directory 1 ]----|----[ added directory 2 ]----
                      -| newpaths.c  08-Mar.21:49 drp
*** Automatic: Applying ADDITION from directory 2
----[ removed directory 1 ]----|----[ directory 2 ]----
readln.c  19-Dec-1991 drp      |-
*** Automatic: Applying REMOVE from directory 2
Recorded merge of ".".
```

Using Your Own Merge Tools

If you wish, you can create a merged version of an element completely manually, or with any available analysis and editing tools. Check out the target version, revise it, and check it back in. Just before (or just after) the *checkin*, be sure to record your activity, by using the *merge* command with the *-ndata* (“no data”) option:

```
% cleartool checkout -nc nextwhat.c
Checkout comments for "nextwhat.c":
merge enhance branch
.
Checked out "nextwhat.c" from version "/main/1".
```

```
% <Invoke your tools to merge data into checked-out version>
```

```
% cleartool merge -to nextwhat.c -ndata -version \
 .../enhance/LATEST
Recorded merge of "nextwhat.c".
```

This form of the *merge* command does not change any file system data — it merely attaches a *merge arrow* (a hyperlink of type *Merge*) the specified versions. After you've made this annotation, your merge is indistinguishable from one performed only with ClearCase tools.

Chapter 7

Comparing and Merging Files Graphically with *xcleardiff*

The previous chapter discussed file comparison and merge operations. This chapter describes the graphical diff/merge utility, *xcleardiff*, in more detail. The main topics are:

- invoking *xcleardiff*
- comparing files
- merging files

For a discussion of the actual file comparison and merge algorithms, see the *diff* and *merge* manual pages.

Summary

xcleardiff is a graphical diff and merge utility for text files. (It implements the *xcompare* and *xmerge* methods for the predefined element types *text_file* and *compressed_text_file*.) *xcleardiff* can also compare, but not merge, directory versions.

On color display monitors, *xcleardiff* uses different colors to highlight *changes*, *insertions*, and *deletions* from one or more contributing files. During merge operations, input files are processed incrementally and, when necessary, interactively, to visibly construct a merged output file. You can edit the merged output as it is being built—either directly in the merged output display pane, or with an arbitrary text editor—to add, delete, or change code manually, or to add comments.

xcleardiff is implemented with a standard window system toolkit. See your window system documentation for a description of general mouse and keyboard conventions.

Invoking *xcleardiff*

You can invoke *xcleardiff* directly from the command line, specifying files or versions to compare or merge. However, because *xcleardiff* implements the *xcompare* and *xmerge* methods for the *text_file_delta* and *z_text_file_delta* type managers, the following *cleartool* subcommands, when applied to text files, also invoke *xcleardiff*:

- *xdiff*
- *xmerge*
- *findmerge* (with options *-xmerge* or *-okxmerge)*

The *xdiff*, *xmerge*, and *findmerge* commands include the advantage of some extra command options—optional ClearCase preprocessing—in the same way that *diff* and *merge* offer more flexibility than direct calls to the character-based *cleardiff* utility. *xdiff -pred*, *xmerge -insert*, and *findmerge -ftag* are all examples of commands that perform useful ClearCase processing before invoking *xcleardiff*.

You can also invoke *xcleardiff* using *xclearcase* buttons and menu options:

- file or vtree browser menubar: Versions -> Diff and Versions -> Merge
- file or vtree browser toolbar: Diff button (see Figure 7-1) and Merge button (see Figure 7-2).

Figure 7-1 Diff Button

Figure 7-2 Merge Button

Setting Your Color Scheme

The ClearCase GUI utilities support several predefined color schemes, which are collections of X resource settings. ClearCase schemes are stored in the directory */usr/atria/config/ui/Schemes*. (A special scheme, *Willis*, is designed for use with monochrome monitors.)

You can specify a scheme in your standard X resources file (typically *$home/.Xdefaults*) with a line like:

```
*scheme: Monet
```

See the *schemes* manual page for more details.

You can also use standard X Window System mechanisms to customize the *xcleardiff* display window. The X class name is *xcleardiff* or *Diff*. The following color-related resources are specific to graphical diff and merge operations:

```
xcleardiff*promptBrightColor
                    (highlight prompt and current diff (default: yellow))
xcleardiff*changeColor
              (highlight change relative to base contributor (default: blue))
xcleardiff*deleteColor
              (highlight deletion relative to base contributor
(default: red))
xcleardiff*insertColor
           (highlight insertion relative to base contributor (default: green))
```

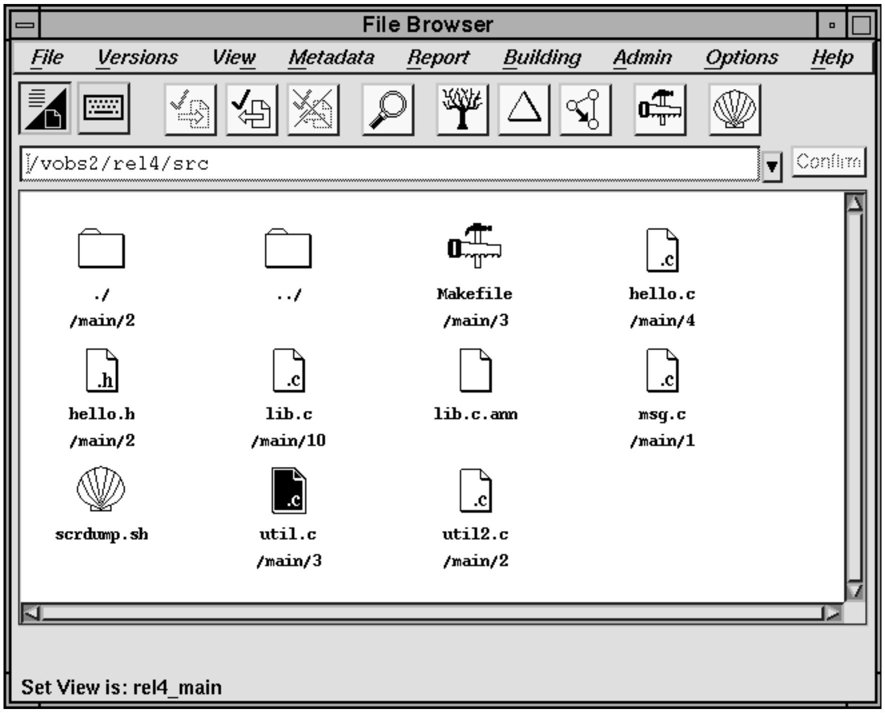
Comparing Files

As you study the examples in this chapter, keep in mind that there are usually several ways to accomplish the same thing.

Example 1: Compare with Predecessor

Let’s compare versions of a file element. The most common comparison operation is to *diff* the version selected by your view with its predecessor version.

1. Open the desired directory in a file browser as shown in Figure 7-3 and select the element, *util.c* for example.



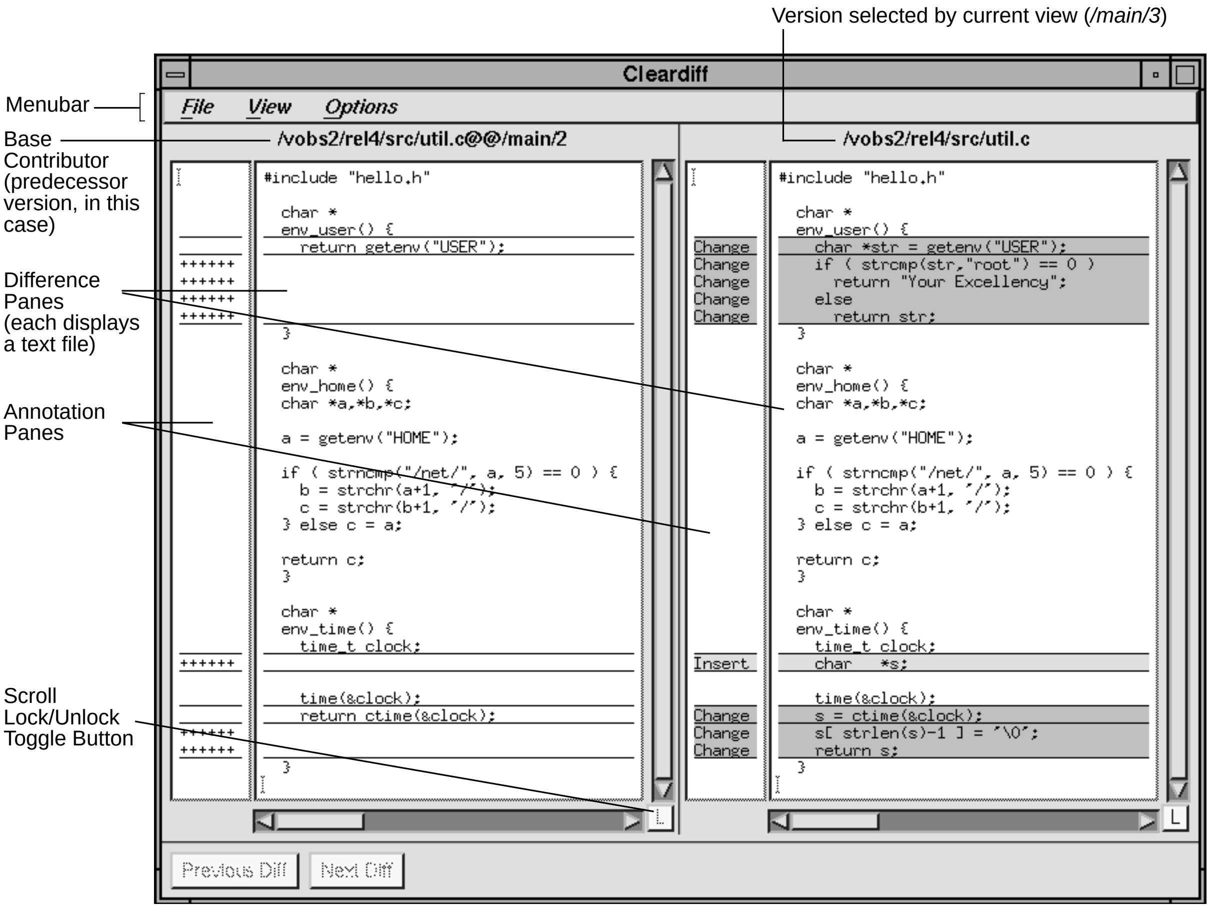
/vobs2/rel4/src

Set View is: rel4_main

Figure 7-3 File Browser with File Element *util.c* Selected

2. Press the Diff button (see Figure 7-1) .

   This button (as explained in its “pop-up help”, accessible with *rightMouse*) compares the version selected by the current view with its direct predecessor. Figure 7-4 shows the resulting output.



Version selected by current view (/main/3)
Menubar
Base Contributor (predecessor version, in this case)
Difference Panes (each displays a text file)
Annotation Panes
Scroll Lock/Unlock Toggle Button
Cleardiff
File
View
Options
/vobs2/rel4/src/util.c@@/main/2
/vobs2/rel4/src/util.c
++++++
Change
Insert
L
Previous Diff
Next Diff

```
#include "hello.h"

  char *
  env_user() {
    return getenv("USER");




  }

  char *
  env_home() {
  char *a,*b,*c;

  a = getenv("HOME");

  if ( strncmp("/net/", a, 5) == 0 ) {
    b = strchr(a+1, '/');
    c = strchr(b+1, '/');
  } else c = a;

  return c;
  }

  char *
  env_time() {
    time_t clock;


    time(&clock);
    return ctime(&clock);


  }
```

```
#include "hello.h"

  char *
  env_user() {
    char *str = getenv("USER");
    if ( strcmp(str,"root") == 0 )
      return "Your Excellency";
    else
      return str;
  }

  char *
  env_home() {
  char *a,*b,*c;

  a = getenv("HOME");

  if ( strncmp("/net/", a, 5) == 0 ) {
    b = strchr(a+1, '/');
    c = strchr(b+1, '/');
  } else c = a;

  return c;
  }

  char *
  env_time() {
    time_t clock;
    char   *s;

    time(&clock);
    s = ctime(&clock);
    s[ strlen(s)-1 ] = '\0';
    return s;
  }
```

Figure 7-4 Graphical File Comparison

The following command line and the Diff button are equivalent.

```
% cleartool xdiff -predecessor util.c
```

The menu item Versions -> Diff -> Selected vs. predecessor provides a third path to the same result.

Base Contributor File

The *base contributor file* is the first file specified (first on the command line or selected first in a browser) and is displayed on the left of the screen. All other contributors are compared against the base file.

Text Line Annotations

By default (that is, unless line numbering is enabled), the following annotations can appear in the annotation panes:

Insert Line occurs in this file, but not in base contributor file. (Insertions default to *green* on a color display monitor.)

Delete Line occurs in base contributor file, but not in this file. (Deletions default to *red* on a color display monitor.)

Change Line has changed from the base contributor file to this file. (Changes default to *blue* on a color display monitor.)

****** Empty lines, introduced to keep corresponding source lines aligned when files are displayed side-by-side.

The Options Menu

Show Line Numbers

Enable/disable line number annotations for all lines of all contributor files.

Stack Vertically

Enable/disable vertical stacking of the difference panes. Side-by-side is the default.

The View Menu

base-file

Enable/disable display of the base file. Typically, you want to see the base file during a diff operation, but probably do not need to see it during a merge.

contributor-1 ...

Enable/disable display of this contributor file.

Display Lock Icon

You can scroll difference panes independently or synchronously. Each difference pane has a lock/unlock toggle button. If vertical and/or horizontal scrollbars are active, all locked panes scroll together. An unlocked pane can be scrolled independently. Whenever you press Previous Diff or Next Diff, the difference panes all resynchronize with the base file. You cannot unlock the base file.

Example 2: Comparing Arbitrary Versions of an Element

The Versions -> Diff submenu (and the *xdiff* command) lets you compare any combination of versions (including three, or even more, simultaneously). Suppose you wanted to compare the version of *util.c* selected by your view with */main/1*, instead of its immediate predecessor:

1. Select *util.c* from a file browser.
2. Choose the menu item Versions -> Diff -> Selected vs. other...
   A vtree browser (Figure 7-5) comes up and prompts you for the (older) version to compare with the selected version of *util.c*.

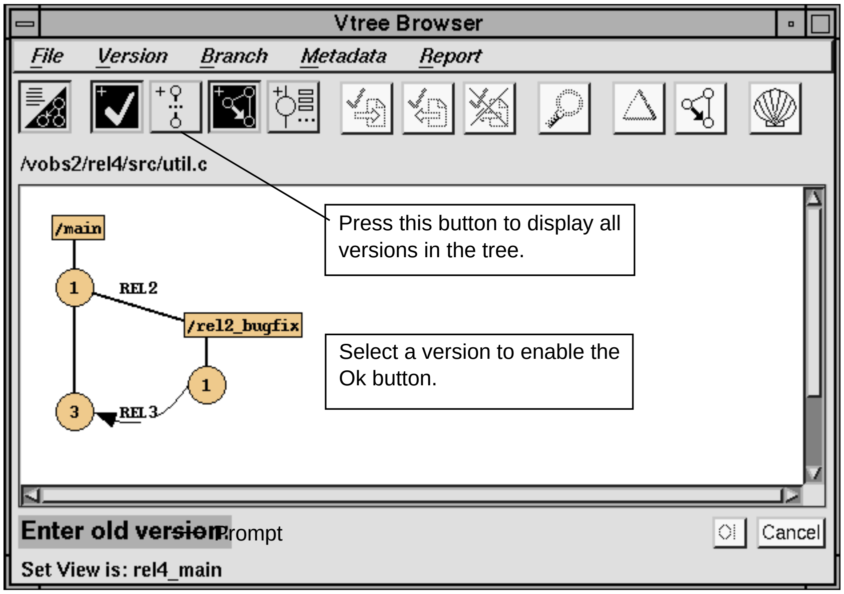


Figure 7-5 Vtree Browser Prompting for a Version to Compare

Click on version 1 to select it, and then press the Ok button. A *Cleardiff* window appears, like the one in Figure 7-4, but with different versions this time.

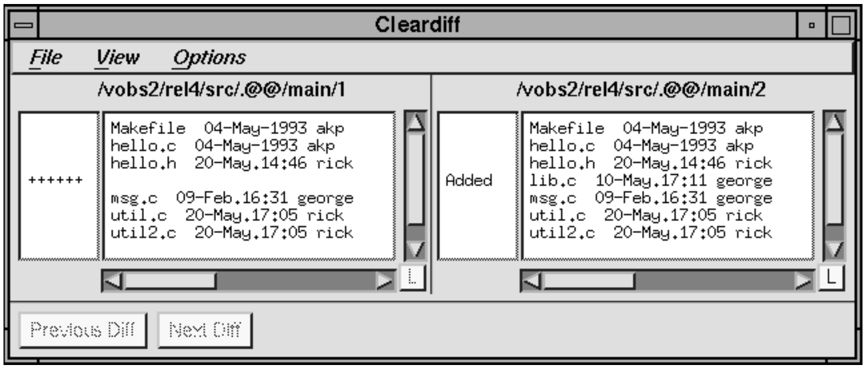
The command line equivalent for this merge operation is:

```
% cleartool xdiff util.c@@/main/1 util.c
```

Comparing Directories

You can compare directory versions as well as file versions (see Figure 7-6.) Use the following procedure to compare two versions of a directory element, neither of which is selected by your current view.

1. Select the directory icon from a file browser.
2. Choose Versions -> Diff -> Other vs. other...
3. In the prompting vtree, supply the “older” version and press Ok. Then supply the “newer” version and press Ok.



/vobs2/rel4/src/.@@/main/1

```
++++++

Makefile  04-May-1993 akp
hello.c  04-May-1993 akp
hello.h  20-May.14:46 rick

msg.c  09-Feb.16:31 george
util.c  20-May.17:05 rick
util2.c  20-May.17:05 rick
```

/vobs2/rel4/src/.@@/main/2

```
Added

Makefile  04-May-1993 akp
hello.c  04-May-1993 akp
hello.h  20-May.14:46 rick
lib.c  10-May.17:11 george
msg.c  09-Feb.16:31 george
util.c  20-May.17:05 rick
util2.c  20-May.17:05 rick
```

Figure 7-6 Comparing Directory Versions

The equivalent command line is:

```
% cd /vobs2/rell4/src; cleartool xdiff .@@/main/1 .@@/main/2
```

Merging Files

As we did for file comparison, let's start with a common operation.

Example: Merging from a Branch to a Checked-out Version

A sample element, *util2.c*, has the following version tree (see Figure 7-7.)

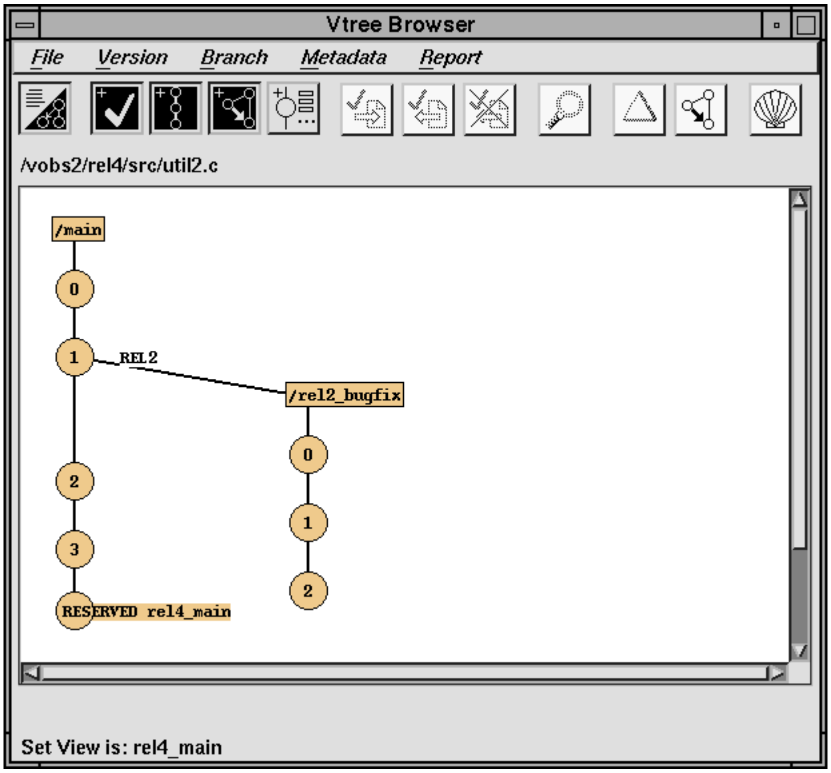
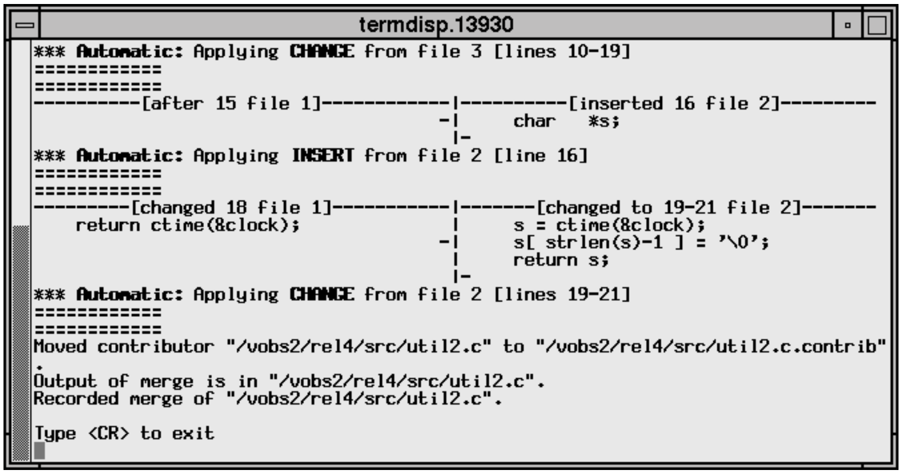


Figure 7-7 Sample Version Tree for *util2.c*

To merge from the version */main/rel2_bugfix/LATEST* to the checked-out version:

1. From a file browser, select the checked-out element (*util2.c*) and choose the menu item Versions -> Merge -> From ...branch/LATEST -> view...
2. A branch type browser prompts for a branch; select *rel2_bugfix* and press Ok.
3. At this point, a prompt asks whether to do a fully automated merge, if one is possible. If you answer "Yes" here, a terminal emulation window (see Figure 7-8) displays a character mode summary of the merge. If the merge can be completed automatically, press <**`Return`**> to exit the display window; the merge is complete.

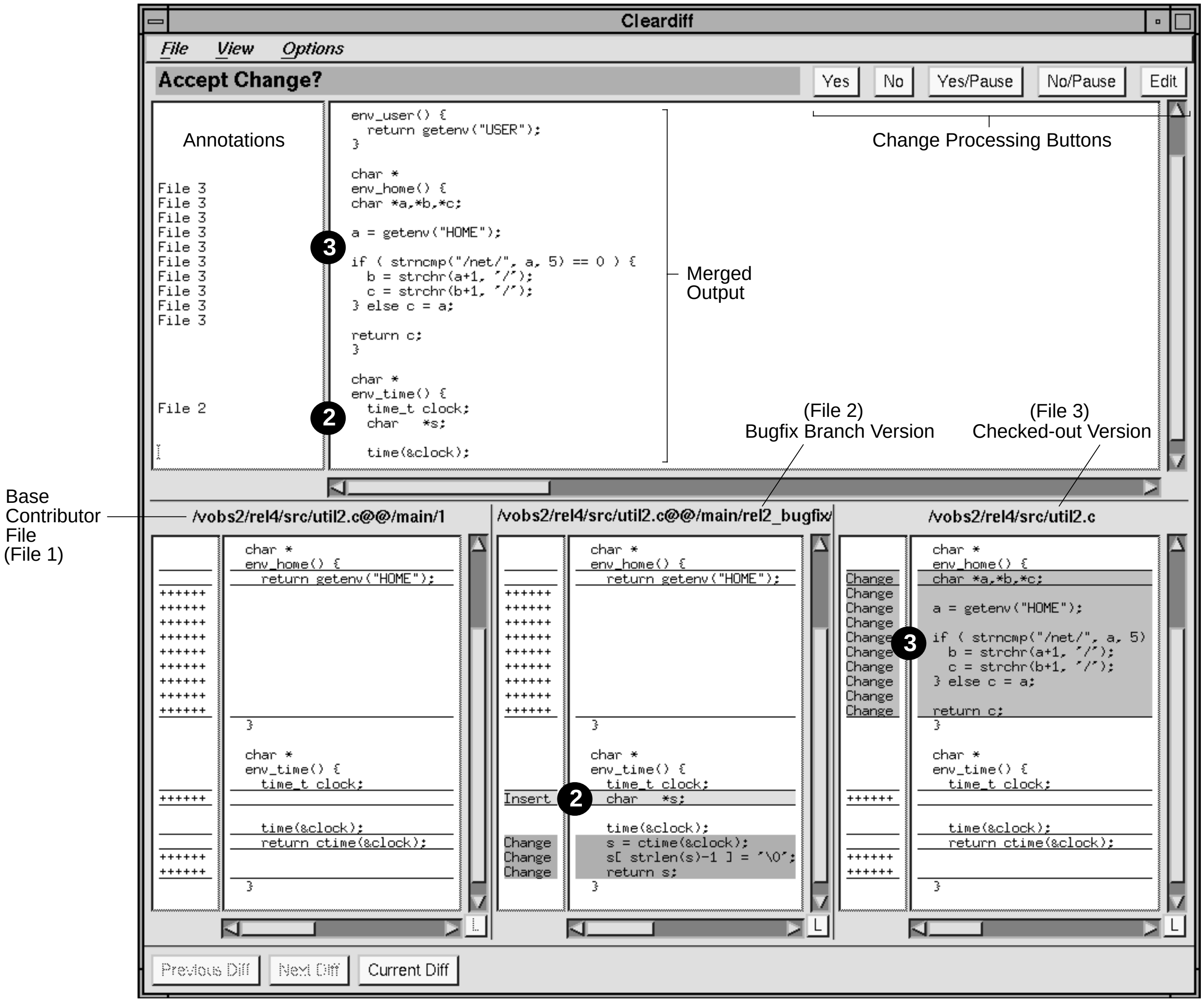


```
*** Automatic: Applying CHANGE from file 3 [lines 10-19]
============
============
----------[after 15 file 1]------------|----------[inserted 16 file 2]---------
                                      -|     char   *s;
                                       |-
*** Automatic: Applying INSERT from file 2 [line 16]
============
============
---------[changed 18 file 1]-----------|-------[changed to 19-21 file 2]-------
    return ctime(&clock);              |     s = ctime(&clock);
                                      -|     s[ strlen(s)-1 ] = '\0';
                                       |     return s;
                                       |-
*** Automatic: Applying CHANGE from file 2 [lines 19-21]
============
============
Moved contributor "/vobs2/rel4/src/util2.c" to "/vobs2/rel4/src/util2.c.contrib"
.
Output of merge is in "/vobs2/rel4/src/util2.c".
Recorded merge of "/vobs2/rel4/src/util2.c".

Type <CR> to exit
```

Figure 7-8 Automatic Merge Output

If a conflict occurs during the automatic merge (two or more contributor files differ from the base contributor file at the same location), or if you answer No to the "Automatic merge?" prompt, the merge operation takes an interactive, graphical form.

Figure 7-9 shows the graphical merge window. In this example, we have chosen not to attempt an automatic merge. Furthermore, we have already responded Yes to the first two "Accept changes?" prompts, accepting in the merged output file several modified lines from "file 3" (the checked-out version) and a one-line insertion from "file 2" (the *rel2_bugfix* version). Note that ClearCase calculates the base contributor file, *util2.c@@/main/1*, automatically (see the *merge* manual page for details) and includes it in the display. All other contributing files are compared against the base contributor to identify changes, insertions, and deletions.



```
                 env_user() {
                   return getenv("USER");
                 }

                 char *
File 3           env_home() {
File 3           char *a,*b,*c;
File 3
File 3           a = getenv("HOME");
File 3
File 3           if ( strncmp("/net/", a, 5) == 0 ) {
File 3             b = strchr(a+1, '/');
File 3             c = strchr(b+1, '/');
File 3           } else c = a;
File 3
                 return c;
                 }

                 char *
                 env_time() {
File 2             time_t clock;
                   char    *s;

                   time(&clock);
```

/vobs2/rel4/src/util2.c@@/main/1

```
          char *
          env_home() {
            return getenv("HOME");
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
          }

          char *
          env_time() {
            time_t clock;
++++++

            time(&clock);
            return ctime(&clock);
++++++
++++++
          }
```

/vobs2/rel4/src/util2.c@@/main/rel2_bugfix

```
          char *
          env_home() {
            return getenv("HOME");
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
++++++
          }

          char *
          env_time() {
            time_t clock;
Insert      char    *s;

            time(&clock);
Change      s = ctime(&clock);
Change      s[ strlen(s)-1 ] = '\0';
Change      return s;
          }
```

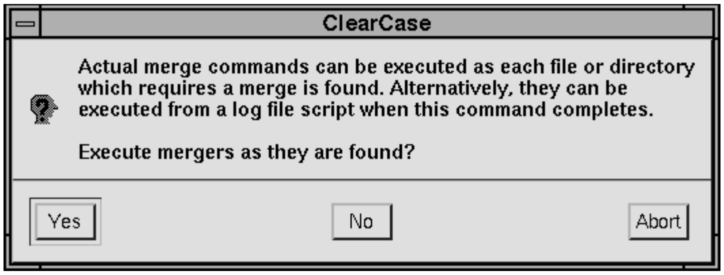
/vobs2/rel4/src/util2.c

```
          char *
          env_home() {
Change    char *a,*b,*c;
Change
Change    a = getenv("HOME");
Change
Change    if ( strncmp("/net/", a, 5)
Change      b = strchr(a+1, '/');
Change      c = strchr(b+1, '/');
Change    } else c = a;
Change
Change    return c;
          }

          char *
          env_time() {
            time_t clock;
++++++

            time(&clock);
            return ctime(&clock);
++++++
++++++
          }
```

Figure 7-9 Graphical Merge Display

Before moving on to describe the graphical merge display, note that we could perform the same merge — from */main/rel2_bugfix/latest* to */main/checkedout* — with any of the following procedures, and some others as well:

- From a file browser, select the checked-out element, press the Merge button (see Figure 7-2), and select *rel2_bugfix* when prompted by the branch type browser.
- From a vtree browser on element *util2.c*, select the menu item Version -> Merge -> From version -> branch... At the prompts, click on the "from" version and press Ok, then click on the "to" version and click Ok.
- From a file browser, select the checked-out element, and choose the menu item Versions -> Findmerge -> Selected items -> From ...branch/LATEST -> view... Select */main/rel2_bugfix* from the branch type browser. Answer "Yes" to the prompt. Alternatively, answer "No" to the prompt in Figure 7-10.



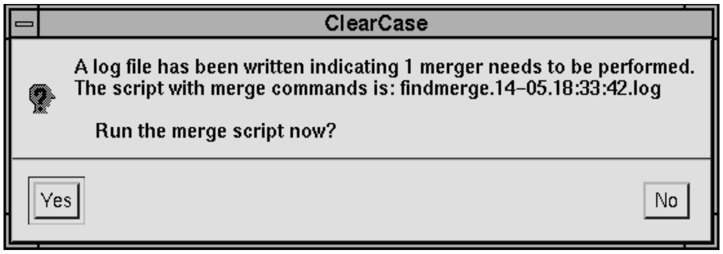
ClearCase

Actual merge commands can be executed as each file or directory which requires a merge is found. Alternatively, they can be executed from a log file script when this command completes.

Execute mergers as they are found?

Figure 7-10 Reply Prompt "No"

Answer Yes to the prompt in Figure 7-11 (after *findmerge* has written merge commands to a log file/command script)



A log file has been written indicating 1 merger needs to be performed.
The script with merge commands is: findmerge.14-05.18:33:42.log

Run the merge script now?

Figure 7-11 Reply Prompt “Yes”

Or, answer No to both prompts and run the *.log* script later.

- From the command line:

```
% cleartool xmerge -to util2.c \
 util2.c@@/main/rel2_bugfix/LATEST
```

- From the command line:

```
% cd src; cleartool findmerge . -fversion\
/main/rel2_bugfix/LATEST -merge -xmerge
```

The Graphical Merge Display Window

The graphical merge display shares many features with the file comparison display: difference panes, text line annotations, color usage, Next/Prev Diff buttons, and so on. Following are the significant additions.

Merged Output Pane (Editable)

Displays the merge output as it is being built. You can edit the merged text directly (using primitive editing keystrokes) or press the Edit button to invoke a text editor. The merged output can include the following annotations:

File # Identifies the file that contributed the line (no annotations for lines from the base contributor file).

Edtblk Marks a block of text modified in a text editor with the Edit button (described below).

Edited Marks each line that has been manually edited in the merged output pane itself.

Calculated Base Contributor File

Unless you invoke *xcleardiff* directly and without a -base argument,[1] a base contributor file is calculated automatically and displayed on the left of the screen, or on the top with vertical stacking in effect. The base contributor file is the common ancestor for all versions of the element being merged. See *merge* for a discussion of how ClearCase calculates the base contributor for various types of merge operations.

Merge-Related Menu Options

File -> Restart — Cancel and restart the entire merge operation.

Options -> Query on Conflicts — The default merge automation mode. If a difference section (insertion, deletion, or change) has exactly one contributor that differs from the base file, the change is accepted automatically and applied to the merged output. You are prompted to intervene manually only when the same text section in two or more contributors differs from the corresponding section in the base file.

Options -> Query on All — Turns off automatic change acceptance, and prompts you to take action on every change from every contributor.

Options -> Pause after Auto Decisions — After each automatic merge decision (see Query on Conflicts), *xcleardiff* pauses and prompts “Continue merge?”. This option has no effect in Query on All mode.

[1] You might do this if attempting to merge unrelated files, or files that are not elements.

Merge Processing Buttons

Yes

Accept the current prompt (“Accept change?”, “Save Merged output?”, and so on).

No

Refuse the current prompt. Specifically, reject the current change, insertion, or deletion section in a contributor file; preserve the base file’s version of the applicable text section.

Yes/Pause

Accept the change/insertion/deletion, but then pause merge processing; a “Continue merge?” prompt appears.

No/Pause

Reject the change/insertion/deletion, and then pause merge processing; a “Continue merge?” prompt appears.

Edit

(Active whenever processing is paused) Press Edit to edit the merged output file in a text editor (the system searches, in order, the *visual*, *editor*, and *wineditor* environment variables).While in the text editor, the Yes button is enabled as an “escape hatch” in case the editor process hangs

Note: The *Edited* annotation marks every line edited manually (directly in the output pane, not with a text editor), but the *Edtblk* annotation is somewhat less precise; it marks each text block in which one or more changes were made with the text editor.

Current Diff

(Bottom of display window) Any pause in automatic merge processing enables the *Previous Diff*, *Next Diff*, and *Current Diff* buttons. If you use *Previous Diff* or *Next Diff* to move around during a pause, *Current Diff* resynchronizes all difference panes with the base file and returns you to the pause point—a yellow highlighted insertion, change, or deletion that the system is waiting for you to accept or reject.

Example Revisited

Let’s return to the scenario in Figure 7-9 to edit the merged output, complete the merge, and check the results.

Edit the Merged Output

With merging paused at the “Accept change?” prompt:

1. Position the cursor above the change block from File 3.
2. Type in a comment. For example:

   ```
   /* Merged from rel2_bugfix branch:5/1/94 */
   ```

3. Use a text editor to supply a matching comment.

   Press the *Edit* button. When the text editor comes up, add a comment below the changed block. For example:

   ```
   /* End merged section */
   ```

4. Quit the editor and answer Yes to the prompt “Apply changes from editing session?”. The prompt returns to its state before you began editing:
   “Accept change?”
5. Compare the new *Edited* and *Edtblk* annotations on the merged output.

Complete the Merge

6. Answer Yes to the remaining prompts.

Check the Results of the Merge

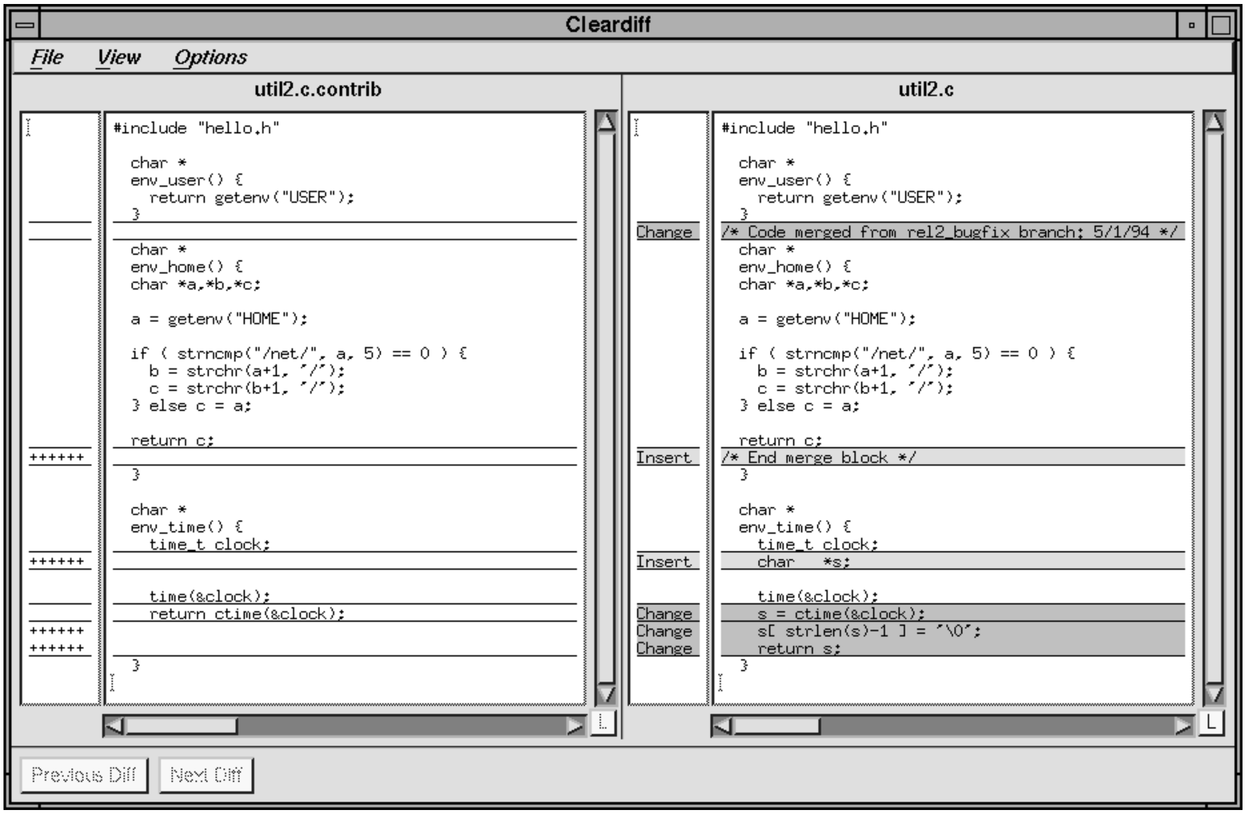
7. Finally, let’s check the merge results for consistency by comparing the new *util2.c@@/main/checkedout* — which was overwritten with the merged output — against *util2.c.contrib*, an automatically saved copy of the original checked-out version.

   ```
   % cleartool xdiff util2.c.contrib util2.c
   ```

   Figure 7-12 shows the comparison.

8. Checkin the merged file.

   ```
   % cleartool checkin -c "merged from rel2_bugfix" util2.c
   ```



util2.c.contrib

```
        #include "hello.h"

          char *
          env_user() {
            return getenv("USER");
          }

          char *
          env_home() {
          char *a,*b,*c;

          a = getenv("HOME");

          if ( strncmp("/net/", a, 5) == 0 ) {
            b = strchr(a+1, '/');
            c = strchr(b+1, '/');
          } else c = a;

          return c;
++++++
          }

          char *
          env_time() {
            time_t clock;
++++++

            time(&clock);
            return ctime(&clock);
++++++
++++++
          }
```

util2.c

```
        #include "hello.h"

          char *
          env_user() {
            return getenv("USER");
          }
Change  /* Code merged from rel2_bugfix branch: 5/1/94 */
          char *
          env_home() {
          char *a,*b,*c;

          a = getenv("HOME");

          if ( strncmp("/net/", a, 5) == 0 ) {
            b = strchr(a+1, '/');
            c = strchr(b+1, '/');
          } else c = a;

          return c;
Insert  /* End merge block */
          }

          char *
          env_time() {
            time_t clock;
Insert      char   *s;

            time(&clock);
Change      s = ctime(&clock);
Change      s[ strlen(s)-1 ] = '\0';
Change      return s;
          }
```

Figure 7-12 Verifying Merged Output

Chapter 8

Using the ClearCase/SoftBench Integration

This chapter describes ClearCase support for the SoftBench integrated software development environment. The ClearCase Encapsulation for SoftBench enables integration of ClearCase with all SoftBench tools. ClearCase services and broadcasts all the messages prescribed for configuration-management systems in the document “CASE Communique: Configuration Management Operation Specifications” from the “historical” standard.

ClearCase adds a menu to the SoftBench Development Manager, providing users with a familiar interface to ClearCase’s most important version-control and configuration-management functions. You can customize the SoftBench environment to add items to this menu, accessing more sophisticated features.

You can configure the SoftBench Builder to use the ClearCase build tool, *clearmake*. All other SoftBench tools (debugger, browser, static analyzer, and so on) work within ClearCase environments by using view-extended pathnames.

ClearCase can broadcast SoftBench messages whenever ClearCase performs a configuration-management operation, no matter how that operation was requested: from the SoftBench or ClearCase graphical user interface, from the ClearCase command line interface, from the ClearCase API, from other SoftBench tools, and so on. This flexibility accommodates a variety of working styles without sacrificing tool integration.

Architecture

SoftBench tools communicate with ClearCase through the SoftBench Broadcast Message Server (BMS), and two ClearCase processes:

- *clearencap_sb* — ClearCase Encapsulator for SoftBench
- *sb_nf_server* — ClearCase Notice Forwarder for SoftBench

When a SoftBench tool makes a configuration-management request, such as *checkout*, the BMS receives the message and passes it on to the ClearCase encapsulator, *clearencap_sb*. The BMS starts the encapsulator process if it is not already running. *clearencap_sb* then evaluates the message and invokes the appropriate ClearCase operation, such as a *checkout* command:

- If the operation succeeds, the individual ClearCase tool sends a message to the Notice Forwarder process, *sb_nf_server*, starting it if necessary. (For example, *cleartool* might send notice of a successful checkout.) *sb_nf_server* then informs the BMS that the operation succeeded.
- If the operation fails, the BMS receives the message directly from *clearencap_sb*.

In both cases, the BMS passes the final status message back to the SoftBench tool. This "alternate path" architecture for sending success and failure statuses to the BMS enables ClearCase events to generate SoftBench messages without using the encapsulator. For example, you can perform a *checkout* command in a non-SoftBench shell, and still have your SoftBench processes be notified that the checkout succeeded. See "One-Way Messaging" on page 151 for more information on this topic.

Configuring the Development Manager for ClearCase

To include the "ClearCase" menu item in the SoftBench Development Manager Main menu, place this line in your X Window System resources file (typically, *$HOME/.Xdefaults*):

```
Softdm*menuDirSelect_CM: ClearCase
```

Configuring HP VUE

If you are using HP VUE with SoftBench, you must add the *$ATRIAHOME/bin* directory to your *Vuelogin*userPath* resource. Edit the file */usr/lib/X11/vue/Vuelogin/Xconfig* to include a line like:

```
Vuelogin*userPath: /usr/bin/X11:/bin:/usr/bin:/etc: \
/usr/contrib/bin:/usr/atria/bin:/usr/lib:/usr/lib/acct
```

Using SoftBench

The sections below discuss various topics pertaining to day-to-day usage of the ClearCase Encapsulation for SoftBench.

Using the SoftBench Development Manager

The SoftBench Development Manager is a graphical browser and file manager. The ClearCase installation procedure adds a “ClearCase” submenu to the Development Manager menu:

Check Out ...

Sends a CM VERSION-CHECK-OUT message to the ClearCase Encapsulator. This pops up a window in which you specify a (multiline) *checkout* comment, then checks out the selected element.

Cancel Check Out ...

Sends a CM VERSION-CHECK-IN message with the CANCEL keyword to the ClearCase Encapsulator. This pops up a window, in which you can specify that the view-private copy is to be saved (*uncheckout -keep* option), or to be removed (*uncheckout -rm* option); then, the checkout of the selected element is cancelled.

Check In ...

Sends a CM VERSION-CHECK-IN message to the ClearCase Encapsulator. This pops up a window, in which you specify a (multiline) *checkin* comment, then checks in the selected element. Specifying no comment preserves the *checkout* comment, if any.

Cleartool List ...

Sends a CM VERSION-LIST-DIR message to the ClearCase Encapsulator. This pops up a window that lists the particular version of the selected element that appears in the view. The configuration rule that selects this version is also listed.

List Checkouts ...

Sends a CM VERSION-LIST-CHECKOUTS message to the ClearCase Encapsulator. This pops up a window that lists the selected element's checkout records (if any).

List History ...

Sends a CM VERSION-SHOW-HISTORY message to the ClearCase Encapsulator. This pops up a window showing the event history of the selected element. For directory elements, the history of the directory's contents is displayed, rather than the history of the directory element itself.

Display Version Tree ...

Sends a CM VERSION-SHOW-VTREE message to the ClearCase Encapsulator. This pops up an *xlsvtree* window showing a graphical version tree of the selected element.

Describe ...

Sends a CM VERSION-DESCRIBE message to the ClearCase Encapsulator. This pops up a window that lists information on the particular version of the selected element that appears in the view.

Compare Versions ...

Sends a CM VERSION-COMPARE-REVS message to the ClearCase Encapsulator. This opens an *xcleardiff* window that compares the *predecessor* version of the selected element with the version that appears in your view.

Merge Versions ...

Sends a CM VERSION-MERGE-REVS message to the ClearCase Encapsulator. This pops up an *xcleardiff* window, in which you perform the merger interactively.

Make Element ...

Sends a CM VERSION-INITIALIZE message to the ClearCase Encapsulator. This pops up a window, in which you specify an element-creation comment, then checks in the selected file as the first version of a new element.

Make Directory ...

Sends a CM VERSION-MKDIR message to the ClearCase Encapsulator. This pops up windows, in which you specify a directory name and an element-creation comment; then, a new directory element is created.

Make Branch ...

Sends a CM VERSION-MAKE-BRANCH message to the ClearCase Encapsulator. This pops up a window, in which you specify a branch name (the branch type must already exist) and a (multiple-line) branch-creation comment; then, a branch is created in the selected element and version 0 on the branch is checked out.

Make Label ...

Sends a CM VERSION-MAKE-LABEL message to the ClearCase Encapsulator. This pops up a window, in which you specify a version label (the label type must already exist); then, a version label is attached to the version of the selected element that appears in your view.

Make Attribute ...

Sends a CM VERSION-MAKE-ATTRIBUTE message to the ClearCase Encapsulator. This pops up windows, in which you specify an attribute name (the attribute type must already exist) and a value for the attribute; then, an attribute is attached to the version of the selected element that appears in your view.

Cat Configuration Record ...

Flattened Cat Configuration Record ...

Sends a CM DERIVED-CAT-CONFIG-REC message to the ClearCase Encapsulator. This pops up a window that lists the contents of the config rec for the selected derived object.

Compare Configuration Records ...

Flattened Compare Configuration Records ...
: Sends a CM DERIVED-DIFF-CONFIG-REC message to the ClearCase Encapsulator. This pops up a window, in which you specify the DO-ID of a derived object; then, the contents of that derived object's config rec are compared with those of the selected derived object.

Start View ...
: Sends a CM START-VIEW message to the ClearCase Encapsulator. This pops up a window, in which you specify a view-tag; then, the view with that view-tag is activated.

Edit Configuration Specification ...
: Sends a CM VERSION-SET-MASTER message to the ClearCase Encapsulator. This pops up a "text widget" window, in which you revise the current config spec.

Using Views

Do not start any SoftBench process from a shell that is set to a view. As you work, use view-extended pathnames to indicate the desired view context(s). For example:

- To do new development, you might use the "Set Context" selection on the "File" menu to change the current directory to */view/jones_vu/proj/libpub*.
- To do maintenance work, you might use the "Set Context" selection on the "File" menu to change the current directory to */view/r1fix_vu/proj/monet/include*.

Before referencing files in a particular view, make sure that the view is started, using "Start View". Attempting to start a view that is already started generates a harmless error message.

Caution: Do not attempt to communicate with SoftBench servers on other hosts using view-extended pathnames. This may cause NFS deadlocks.

Setting the Build Program

By default, the SoftBench *softbuild*(1) program (invoked with the "Build" or "Rebuild" selection on the "Action" menu) runs *make*(1). You can have it run *clearmake* by setting two X resources in your *.Xdefaults* file (typically, in your home directory):

```
*buildProgram    : clearmake
*knownBuildProgs : clearmake
```

As an alternative, you can make the change system-wide by setting these X resources in */usr/softbench/app-defaults/Softdm*.

One-Way Messaging

Success messages can be sent from ClearCase to SoftBench, even if the successful operation was not initiated by a SoftBench tool. If environment variable CLEARCASE_MSG_PROTO is set to *SoftBench*, ClearCase tools will generate success messages, whether or not the operation was initiated by a SoftBench tool. The Notice Forwarder sends the success message to the BMS, even if the encapsulator process, *clearencap_sb*, is not running.

Error Conditions

Message passing succeeds only if ClearCase and SoftBench processes have the same value for the DISPLAY environment variable. A ClearCase tool generates an error message if CLEARCASE_MSG_PROTO is set, but DISPLAY is not set or is incorrect.

The Notice Forwarder process logs errors, warnings, and other messages in file */usr/adm/atria/ti_server_log*.

ClearCase Encapsulation Summary

The ClearCase pull-down menu (described in "Using the SoftBench Development Manager" on page 147) provides access to a subset of the message-handling capabilities of *clearencap_sb*. For a complete listing, see the *softbench_ccase* manual page.

Customization

This version of *clearencap_sb* conforms to the *CASE Communique: Configuration Management* operation specification (the one marked "historical"). You can customize the SoftBench environment to access additional features of the ClearCase encapsulation. See the Hewlett-Packard *SoftBench Encapsulator: Programmer's Guide* for details.

Chapter 9

Using the ClearCase/ToolTalk Integration

This chapter describes ClearCase support for the ToolTalk^{TM} integrated software development environment. ClearCase can broadcast ToolTalk messages whenever ClearCase performs a configuration management operation, no matter how that operation was requested: from a ToolTalk tool, from the ClearCase graphical user interface, from the ClearCase command line interface, from the ClearCase API, and so on. This flexibility accommodates a variety of working styles without sacrificing tool integration.

Architecture

ToolTalk tools communicate with ClearCase through the ToolTalk Session Server, *ttsession*, and two ClearCase processes:

- *clearencap_tt* — ClearCase encapsulator for ToolTalk
- *tt_nf_server* — ClearCase notice forwarder for ToolTalk

After ToolTalk has been configured to work with ClearCase, certain ToolTalk commands automatically invoke ClearCase operations. When a ToolTalk tool makes a configuration management request, such as *CM-Checkin-File*, the ToolTalk Session Server receives the message and passes it on to *clearencap_tt*. (The Session Server starts an encapsulator process if one is not already running.) *clearencap_tt* evaluates the message and invokes the appropriate ClearCase tool, such as *cleartool checkout*.

- If the operation succeeds, the ClearCase tool returns a success exit status to *clearencap_tt*, which sends a success reply back to the Session Server.
- If the operation fails (non-zero exit status), the encapsulator returns a failure status to the Session Server.

In both cases, the Session Server passes the final status message back to the requesting ToolTalk tool.

Using Views

On SunOS systems, do not start any ToolTalk program from a shell that is set to a view. As you work, use view-extended pathnames to indicate the desired view context(s).

On IRIX systems, you can work as described in the preceding paragraph. Alternatively, you can start a ToolTalk program from within a view; the program will be able to communicate only with other programs that were also started from within that view. You can use standard (non-view-extended) pathnames with such programs.

Standalone Notice Forwarding

A ClearCase tool can send a success message even if the operation was not initiated by a ToolTalk tool:

- Make sure that the ClearCase tool and the Session Server both have the environment variable DISPLAY set to the same value.
- Run the ClearCase tool in an environment with CLEARCASE_MSG_PROTO set to *ToolTalk*.

(An error occurs in a ClearCase tool that has CLEARCASE_MSG_PROTO set correctly, but not DISPLAY.) In this environment, the Notice Forwarder generates a success message on each applicable ClearCase operation that succeeds.

The notice forwarder process logs errors, warnings, and other messages in file */usr/adm/atria/ti_server_log*.

ClearCase Encapsulation Summary

For a complete listing of the messages handled by *clearencap_tt*, see the *tooltalk_ccase* manual page.

Chapter 10

Building with *clearmake*; Some Basic Pointers

This chapter presents some simple pointers on making best use of *clearmake*. See the chapters that follow for more detailed discussions.

Accommodating *clearmake*'s Build Avoidance

When you first begin to build software systems with ClearCase, the fact that *clearmake* uses a different build-avoidance algorithm than other *make* variants may occasionally "surprise" you. This section describes several such situations, and presents simple techniques for handling them.

Increasing *clearmake*'s Verbosity Level

If you don't understand *clearmake*'s build-avoidance decisions, use the *-v* (somewhat verbose) or *-d* (*extremely verbose*) option. Equivalently, set environment variable CLEARCASE_BLD_VERBOSITY to 1 or 2, respectively.

Handling Temporary Changes in the Build Procedure

Typically, you do not edit a target’s build script in the *makefile* very often. But you may often change the *effective* build script by specifying overrides for *make macros*, either on the command line or in the UNIX environment. For example, suppose target *hello.o* is specified as follows in the makefile.

```
hello.o: hello.c hello.h
      rm -f hello.o
      cc -c $(CFLAGS) hello.c
```

When it executes this build script, *clearmake* enters the effective build script, after macro substitution, into the config rec. The command:

```
% clearmake hello.o CFLAGS="-g -O1"
```

```
... produces this configuration record entry:
```

```
--------------------------------
Build script:
--------------------------------
        cc -c -g -O1 hello.c
```

So would this command:

```
env CFLAGS="-g -O1" clearmake -e hello
```

The *clearmake* build-avoidance algorithm compares effective build scripts. If you subsequently issue the command *clearmake hello.o* without specifying *CFLAGS="-g -O1"*, *clearmake* will reject the existing derived object, which was built with those flags. The same mismatch would be caused by creating a CFLAGS environment variable (EV) with a different value, and invoking *clearmake* with the *–e* option.

Using a Build Options Specification (BOS) File

To manage “temporary overrides” specified with make macros and EVs, place macro definitions in *build options specification* (BOS) files. There are several mechanisms for having *clearmake* use a BOS file. For example, if your *makefile* is named *project.mk*, macro definitions are automatically read from *project.mk.options*. You can also keep a BOS file in your home directory, or specify one or more BOS files with *clearmake -A;* see the *clearmake.options* manual page for details.

Using a BOS file to specify make macro overrides relieves you of the burden of having to remember which options you specified last time on the command line or in the environment. If you have not modified the BOS file recently, derived objects in your view will not be disqualified for reuse on the basis of build script discrepancies. Some of the sections below describe other applications of BOS files.

Handling Targets Built in Multiple Ways

clearmake's inclination to compare build scripts may produce undesirable results if your build environment includes more than one way to build a particular target. For example, there might be a *test_prog_3* target in two directories:

```
... in its source directory, util_src:
```

```
test_prog_3: ...
      cc -o test_prog_3 ...
```

```
... and in another nearby directory, app_src:
```

```
../util_src/test_prog_3: ...
      cd ../util_src ; cc -o test_prog_3
```

Derived objects built with these scripts are *potentially* equivalent, because they are built at the same file name (*test_prog_3*) in the same VOB directory (*util_src*). But by default, a build in the *app_src* directory will never reuse or wink-in a DO built in the *util_src* directory, because build-script comparison fails.

You can suppress build-script comparison for this target using a *clearmake* special build target, either in the *makefile* or in an associated BOS file:

```
.NO_CMP_SCRIPT: ../util_src/test_prog_3
```

For a one-time suspension of build-script comparison, you can use *clearmake -O*.

Using a Recursive Invocation of *clearmake*

It is easy to eliminate the different-build-scripts problem described in the preceding section. Figure 10-1 shows the recursive invocation of *clearmake*.

```
../util_src/test_prog_3: ..
                cd ../util_src ; $(MAKE) test_prog_3
```

invoke *clearmake* recursively

Figure 10-1 Recursive Invocation of *clearmake*

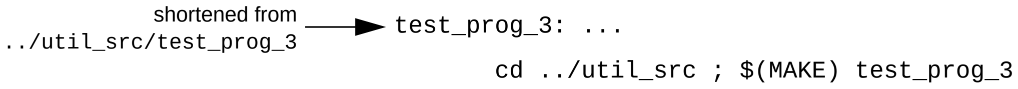
Now, target *test_prog_3* is built the same way in both directories. You can turn build-script comparison on again, by removing the .NO_CMP_SCRIP special target.

Changes of this kind may cause unexpected encounters with another *clearmake* build-avoidance feature, however. Whenever *clearmake* executes the build script of a lower-level target, it always rebuilds all higher-level targets that depend on it. (Many make variants are not as insistent as *clearmake* in this regard.) In this example, the execution of the build script for target `../util_src/test_prog_3` may or may not actually build a new *test_prog_3*; but because *clearmake* has executed the build script, it will *always* rebuild a higher-level target that depends on `../util_src/test_prog_3`:

```
TESTPROGS = ../util_src/test_prog_3 ../util_src/test_prog_7
TESTDATA  = test_input.1 test_input.2 test_input.3

test_output: $(TESTPROGS) $(TESTDATA)
rm test_output
../util_src/test_prog_3 test_input.1 >> test_output
../util_src/test_prog_3 test_input.2 >> test_output
../util_src/test_prog_3 test_input.3 >> test_output
```

Optimizing Wink-In by Avoiding Pseudo-Targets

Like other make variants, *clearmake* always executes the build script for a *pseudo-target*, a target that does not name a file system object built by the script. For example, in the preceding section, you might be tempted to use a pseudo-target in the *app_src* directory's *makefile*. Figure 10-2 provides an example of :



shortened from
../util_src/test_prog_3

```
test_prog_3: ...
        cd ../util_src ; $(MAKE) test_prog_3
```

Figure 10-2 Build Script Example

A build of any higher-level target that has *test_prog_3* as a build dependency will always build a new *test_prog_3*, which in turn triggers a rebuild of the higher-level target. If the rebuild of *test_prog_3* was not really necessary, then the rebuild of the higher-level target may not have been necessary, either. Such unnecessary rebuilds decrease the extent to which you can take advantage of ClearCase's derived object sharing capability.

Accommodating *clearmake*'s Different Name

The fact that the ClearCase build utility has a unique name, "clearmake", may conflict with existing build procedures that implement recursive builds. Most *make* variants automatically define the make macro $(*MAKE*) to be the name of the build program, as it was typed on the command line:

```
% make hello.o            .. sets MAKE to "make"
% clearmake hello.o       .. sets MAKE to "clearmake"
% my_make hello.o         .. sets MAKE to "my_make"
```

This enables recursive builds to use $(*MAKE*) to invoke the same build program at each level. The preceding section includes one such example; here is another one:

```
SUBDIRS = lib util src
all:
for DIR in $(SUBDIRS) ; do ( cd $$DIR ; $(MAKE) all ) ; done
```

Executing this build script with *clearmake* will recursively invoke *clearmake all* in each subdirectory.

Avoid “breaking” this mechanism by explicitly setting a particular build program in the makefile, or in a BOS file:

```
MAKE = make
```

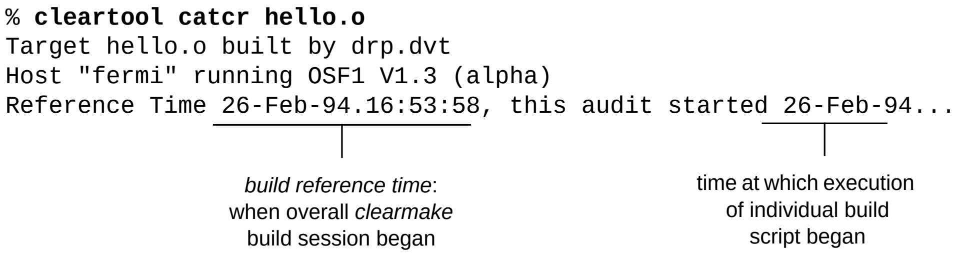
With the above setting, executing the build script above with *clearmake* would recursively invoke *make* all (instead of *clearmake all*) in each subdirectory. Presumably, this would be an error.

Continuing to Work During a Build / Reference Time

clearmake takes into account the fact that software builds are not instantaneous. As your build progresses, other developers continue to work on their files, and may check in new versions of elements that your build uses. If your build takes an hour to complete, you would not want build scripts executed early in the build to use version 6 of a header file, and scripts executed later to use version 7 or 8.

To prevent such inconsistencies from occurring, any versions that were checked in after the moment that *clearmake* was invoked are automatically “locked out”. The moment that the *clearmake* build session begins is termed the *build reference time*.

The same reference time is reported in each configuration record produced during the build session, even if the session lasts hours (or days!) as shown in Figure 10-3.



```
% cleartool catcr hello.o
Target hello.o built by drp.dvt
Host "fermi" running OSF1 V1.3 (alpha)
Reference Time 26-Feb-94.16:53:58, this audit started 26-Feb-94...
```

Figure 10-3 Build Reference Time Report

When determining whether an object was created before or after the build reference time, *clearmake* automatically adjusts for clock *skew*, the inevitable small differences among the system clocks on different hosts. For more on build sessions, see "Build Sessions, Subsessions, and Hierarchical Builds" on page 185.

Caution: A build's coordinated reference time applies to elements only, providing isolation from "after-the-last-minute" changes to them. You are not protected from changes to view-private objects and non-MVFS objects. For example, if you begin a build and then change a checked-out file used in the build, a failure may result. Thus, don't keep working on the same project in a view where a build is in progress.

Using Config Spec 'Time Rules' to Increase Your View's Isolation

Using the reference time facility described in the preceding section, *clearmake* automatically blocks out potentially incompatible source-level changes that take place after your build begins. But sometimes, the incompatible change has *already* taken place. ClearCase allows you to "roll back the clock" in order to block out recently-created versions.

A typical ClearCase team-development strategy is to have each user in the team work in a separate view, but to have all the views use the same config spec. In this way, the entire team works "on the same branch". As long as a source file remains checked-out, its changes are isolated to a single view; but as soon as the user checks in a new version, the entire team sees the new version on the dedicated branch.

This “incremental integration” strategy is often very effective — but suppose that another user’s recently-checked-in version has caused your builds to start failing. Through an exchange of electronic mail, you determine that the “killer checkin” was to header file *project_base.h,* at 11:18 AM today. You, and other team members, can reconfigure your views to roll back just that one element to a “safe” version:

```
element project_base.h .../onyx_port/LATEST -time 5-Mar.11:00
```

If many interdependent files have been revised, you might roll back the clock for *all* checked-in elements:

```
element * .../onyx_port/LATEST -time 5-Mar.11:00
```

For a complete description of time rules, see the *config_spec* manual page.

Overprecise Use of Time Rules

Your view’s *view_server* process interprets time rules with respect to the *create version* event record written by the *checkin* command. The checkin is read from the system clock on the host where the VOB resides. If that clock is seriously out-of-sync with the clock on your host (where the *view_server* runs), your attempt to roll back the clock may fail. Thus, don’t strive for extreme precision with time rules: select a time that is well before the actual cutoff time (for example, a full hour before, or in the middle of the night).

Inappropriate Use of Time Rules

Do not use time rules to “freeze” a view to the current time just before starting a build. Just allow *clearmake*’s reference time facility to perform this service automatically. Here’s an inappropriate use scenario:

1. You checkin version 12 of *util.c* at what is 7:05 PM on your host. You do not know that clock skew on the VOB host causes the time “7:23 PM” to be entered in the *create version* event record.
2. In an effort to “freeze” your view, you change your config spec to include this rule:

   ```
   element * /main/LATEST -time 19:05
   ```

3. You issue a *clearmake* command right away (at 7:06 PM) to build a program that uses *util.c*. When selecting a version of this element to use in the build, your *view_server* consults the event history of *util.c* and rejects version 12, because the “7:23PM” time stamp is too late for the *-time* configuration rule.

Problems with ‘Forced Builds’

clearmake has a *-u* option (“unconditional”), which forces rebuilds. Using this option reduces the efficiency of derived object sharing, however. If you force-build

a target in a situation where *clearmake* would have *winked-in* an existing DO, you create a new DO with same configuration as an existing one. In such situations, a user who expects a build to share a particular existing DO may get another, identically-configured DO instead. This may result in user confusion and wasted disk space.

clearmake tries to minimize the problems by selecting the *oldest* DO in such situations. With this strategy, most builds will tend to “stabilize” on old objects, rather than picking up newly-built equivalent candidates.

We suggest that you use a flag-file to force a rebuild, rather than using *clearmake –u*. (See “Explicitly-Declared Source Dependencies” on page 182.)

Wink-in, Permissions on Derived Objects, and clearcase_bld_umask

UNIX-level permissions on derived objects (DOs) affect the extent to which they are sharable:

- When you perform a build, *clearmake* winks-in a derived object to your view only if you have ‘read’ permission on the DO.
- *clearmake* will wink-in a DO to which you do not have ‘write’ permission. But “permission denied” errors may occur during a subsequent build, when a compiler (or other build script command) attempts to overwrite such a DO. (Removing the target before rebuilding it is an easy, reliable workaround; but you may wish to avoid revising your *makefiles* in this way.)

If you and other members of your group wish to share DOs, make sure that they are created with a mode that grants both 'read' and 'write' access to group members. To accomplish this, you (and other group members) can use either of the following alternatives:

- Set your umask value to 2 in your shell startup file.
- Leave a more restrictive umask value in your shell startup file. (The value 22 is commonly used, which denies 'write' rights to group members.) Instead, set environment variable CLEARCASE_BLD_UMASK to 2. The value of this
- EV temporarily replaces your current umask value during each invocation of *clearmake*.

  Note: If you wish, you can set CLEARCASE_BLD_UMASK as a make macro, instead of as an environment variable.

Don't worry about the "safety" of your derived objects — using a CLEARCASE_BLD_UMASK that grants 'write' rights to group members does not mean that they can overwrite and destroy a derived object that you still are using. If your DO has been winked into another view, and a user in that view actually rebuilds the corresponding *makefile* target, *clearmake* first "breaks the link" to your DO, and then provides an actual file in that view for the build script to overwrite.

Chapter 11

Derived Objects and Configuration Records

This chapter is for users who wish to learn more about how *clearmake* accomplishes its work, and to manipulate ClearCase's build management data structures.

Extended Naming Scheme for Derived Objects

In a parallel-development environment, it is likely that many DOs with the same pathname will exist at the same time. For example, suppose that source file *msg.c* is being developed on three branches concurrently, in three different views. ClearCase builds performed in those three views produce object modules named *msg.o*. Each of these is a DO, and each has the same standard pathname, say */vobs/proj/src/msg.o*.

In addition, each DO can be accessed with ClearCase extended names:

- Within each view, a standard UNIX pathname accesses the DO built in that view. This is another example of ClearCase's *transparency* feature.

  ```
  msg.o (the derived object in the current view)
  ```

- You can use a view-extended pathname to access a DO in any view:

  ```
  /view/drp/vobs/proj/src/msg.o
                               (the derived object in view ‘drp’)
  ```

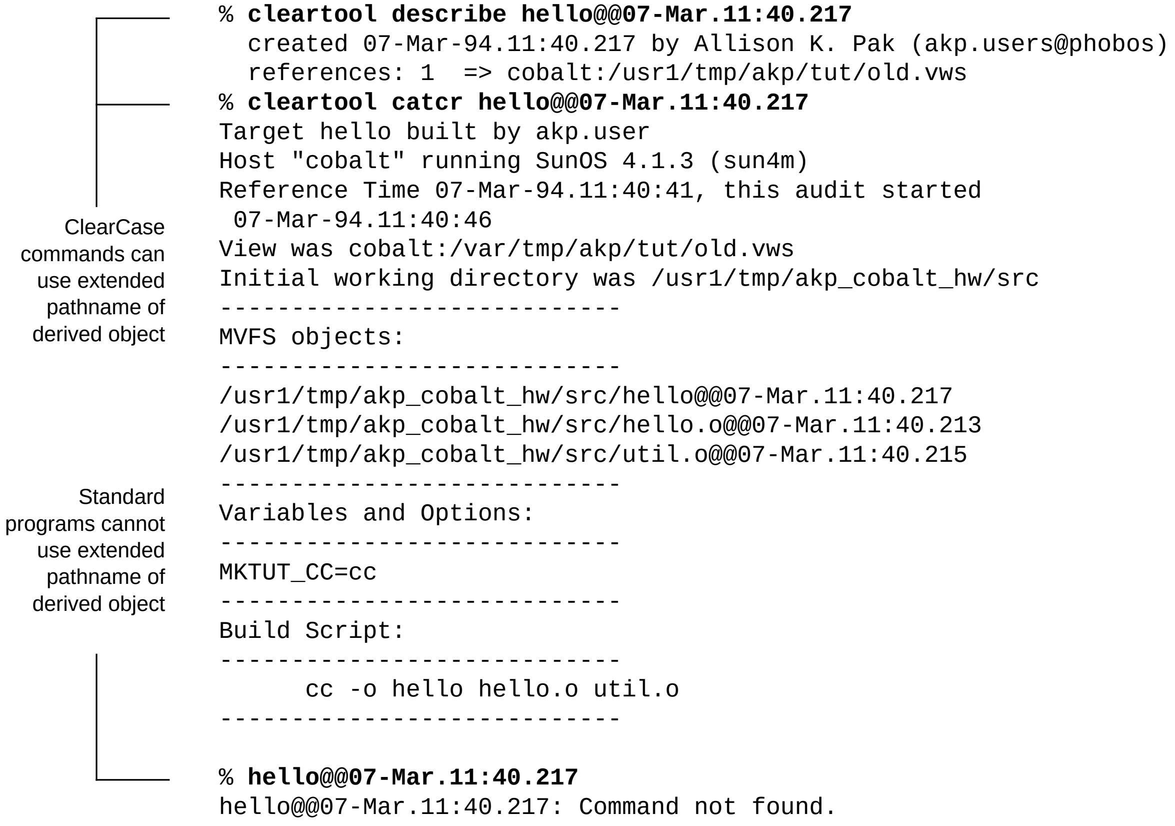
  ```
  /view/R2_integ/vobs/proj/src/msg.o
                         (the derived object in view ‘R2_integ’)
  ```

- Each derived object is cataloged in the VOB database with a unique identifier, it *DO-ID*, which references it independently of views. The *lsdo* (“list derived objects”) command can list all DOs created at a specified pathname, regardless of which views (if any) can “see” them:

  ```
  % cleartool lsdo hello
  07-May.16:09   akp   "msg.o@@07-May.16:09.623" on neptune
  06-May.12:47   akp   "msg.o@@06-May.12:47.539" on neptune
  01-May.21:49   akp   "msg.o@@01-May.21:49.282" on neptune
  03-Apr.21:40   akp   "msg.o@@01-May.21:40.226" on neptune
  ```

  Together, a DO’s standard name (*msg.o*) and its DO-ID (*07-May.16:09.623*) constitute a *VOB-extended pathname* to that particular derived object. (The extended naming symbol is host-specific; most organizations use the default value, `@@`.)

Standard software must access a DO through a view, using a standard pathname or view-extended pathname. You can use such names with debuggers, profilers, *rm*, *tar*, and so on. Only ClearCase programs can reference a DO using a VOB-extended pathname, and only the DO’s meta-data is accessible in this way (see Figure 11-1):



```
% cleartool describe hello@@07-Mar.11:40.217
  created 07-Mar-94.11:40.217 by Allison K. Pak (akp.users@phobos)
  references: 1  => cobalt:/usr1/tmp/akp/tut/old.vws
% cleartool catcr hello@@07-Mar.11:40.217
Target hello built by akp.user
Host "cobalt" running SunOS 4.1.3 (sun4m)
Reference Time 07-Mar-94.11:40:41, this audit started
 07-Mar-94.11:40:46
View was cobalt:/var/tmp/akp/tut/old.vws
Initial working directory was /usr1/tmp/akp_cobalt_hw/src
----------------------------
MVFS objects:
----------------------------
/usr1/tmp/akp_cobalt_hw/src/hello@@07-Mar.11:40.217
/usr1/tmp/akp_cobalt_hw/src/hello.o@@07-Mar.11:40.213
/usr1/tmp/akp_cobalt_hw/src/util.o@@07-Mar.11:40.215
----------------------------
Variables and Options:
----------------------------
MKTUT_CC=cc
----------------------------
Build Script:
----------------------------
      cc -o hello hello.o util.o
----------------------------

% hello@@07-Mar.11:40.217
hello@@07-Mar.11:40.217: Command not found.
```

ClearCase commands can use extended pathname of derived object

Standard programs cannot use extended pathname of derived object

Figure 11-1 Using an Extended Pathname of Derived Object

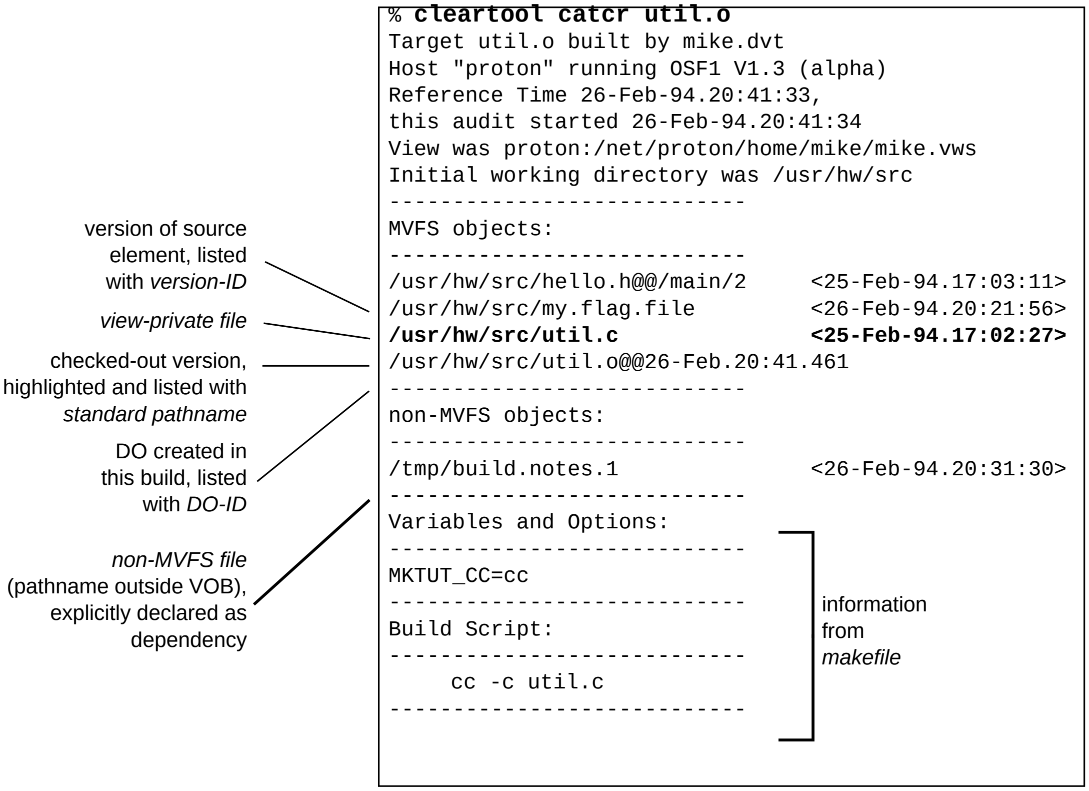
Exception: you can use a VOB-extended pathname with the *winkin* command, to copy the file system data of any DO into your view. See “Wink-In without Configuration Lookup / The ‘winkin’ Command” on page 174.

More on CRs and Configuration Lookup

The following sections discuss how you can use the configuration records with which ClearCase keeps track of builds.

Listing CRs

The *catcr* command displays the configuration record (CR) of a specified derived object (DO). Figure 11-2 shows a CR, with annotations to indicate the various kinds of information in the listing.



```
% cleartool catcr util.o
Target util.o built by mike.dvt
Host "proton" running OSF1 V1.3 (alpha)
Reference Time 26-Feb-94.20:41:33,
this audit started 26-Feb-94.20:41:34
View was proton:/net/proton/home/mike/mike.vws
Initial working directory was /usr/hw/src
----------------------------
MVFS objects:
----------------------------
/usr/hw/src/hello.h@@/main/2     <25-Feb-94.17:03:11>
/usr/hw/src/my.flag.file         <26-Feb-94.20:21:56>
/usr/hw/src/util.c               <25-Feb-94.17:02:27>
/usr/hw/src/util.o@@26-Feb.20:41.461
----------------------------
non-MVFS objects:
----------------------------
/tmp/build.notes.1               <26-Feb-94.20:31:30>
----------------------------
Variables and Options:
----------------------------
MKTUT_CC=cc
----------------------------
Build Script:
----------------------------
     cc -c util.c
----------------------------
```

version of source element, listed with *version-ID*
view-private file
checked-out version, highlighted and listed with *standard pathname*
DO created in this build, listed with *DO-ID*
non-MVFS file (pathname outside VOB), explicitly declared as dependency
information from *makefile*

Figure 11-2 Kinds of Information in a Configuration Record

Some notes on Figure 11-2:

- **Directory versions —** By default, *catcr* does not list versions of the VOB directories involved in a build. To list this information, use the *-long* option:

  ```
  % cleartool catcr -long util.o
   directory version    /usr/hw/.@@/main/1       <25-Feb-94.16:59:31>
   directory version    /usr/hw/src@@/main/3     <26-Feb-94.20:53:07>
  ```

- **Declared dependencies —** One of ClearCase's principal features is the automatic detection of source dependencies on MVFS files: versions of elements and objects in view-private storage. In addition, a CR includes non-MVFS objects that are explicitly declared as dependencies in the *makefile*. Figure 11-2 shows one such declared dependency, on file *build.notes.1*, located in the non-VOB directory */tmp*.

- **Listing of checked-out versions —** Checked-out versions of file elements are highlighted (for example, with boldface or reverse-video). Checked-out versions of directory elements are listed like this:

  ```
   directory version /usr/hw/src@@/main/CHECKEDOUT    <26-Feb...
  ```

  When the elements are subsequently checked in, a listing of the same configuration record shows the updated information. For example,

  ```
  /usr/hw/src/util.c                   <25-Feb-94.17:02:27>
  ```

  ... might become ...

  ```
  /usr/hw/src/util.c@@/main/4        <25-Feb-94.17:02:27>
  ```

  The actual configuration record contains a ClearCase-internal identifier for each MVFS object. After checkin converts the checked-out version object to a “real” version, catcr changes the way it lists that object.

Comparing CRs

When deciding whether or not to rebuild a target, *clearmake* compares one or more CRs of existing DOs with your view’s current build configuration. You can use the *diffcr* command to compare two existing CRs. This command is a valuable diagnostic tool in a parallel development environment.

For example, suppose that a new build of a program dumps core; you have no idea what changed from the preceding build, which ran correctly. A comparison of the CRs might show that the culprit is a change to one header file:

```
% cleartool diffcr -flat hello@@07-Mar.11:40.217 hello@@08-Mar.12:48.265
----------------------------
MVFS objects:
----------------------------
  .
  .
< First seen in target "hello.o"
<    2 /usr/hw/src/hello.h@@/main/5       <20-Mar-94.14:46:00>
> First seen in target "hello.o"
>    2 /usr/hw/src/hello.h                <28-Mar-94.12:48:12>
  .
  .
```

To get the build working again, you could fix the header file; alternatively, you could isolate yourself from this change by reconfiguring your view to select the old version of the header file. (See “Using Config Spec ‘Time Rules’ to Increase Your View’s Isolation” on page 161.)

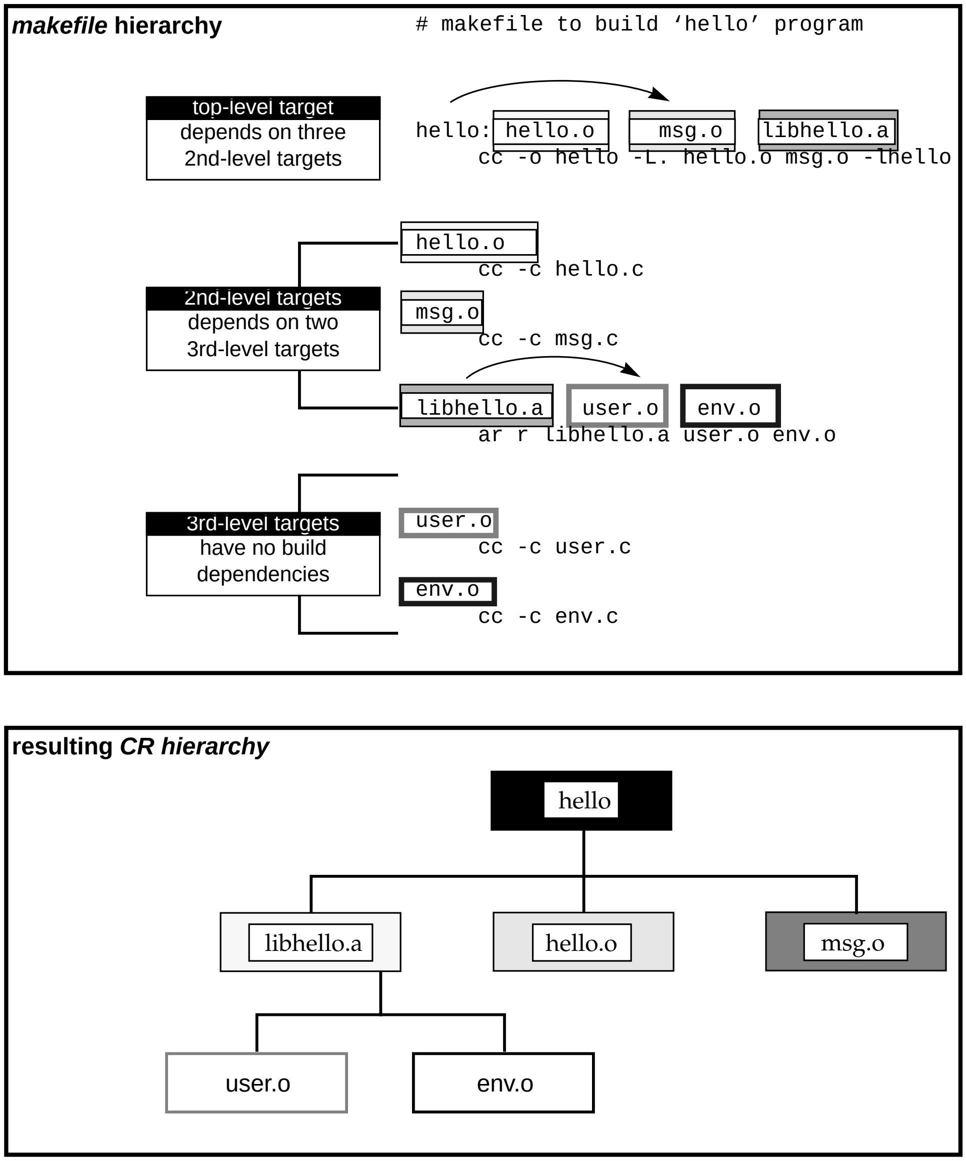
CR Hierarchies

Makefile-based builds of large software systems are almost always hierarchical: you explicitly request a build of a *goal target*; as necessary, this recursively invokes builds of one or more levels of *subtargets*. Thus, a single invocation of *clearmake* can involve the execution of many build scripts. Each execution of a build script is recorded in a separate CR; the result is a *CR hierarchy* that mirrors the structure of the *makefile* (Figure 11-3).

The *catcr* and *diffcr* commands have options for handling CR hierarchies:

- By default, these commands compare individual CRs.
- With the -*recurse* option, they process the entire CR hierarchy of each specified derived object, keeping the individual CRs separate.
- With the -*flat* option, they combine (“flatten”) the CR hierarchy of each specified derived object.

Some ClearCase features automatically process entire CR hierarchies. For example, when the *mklabel* command attaches version labels to all versions used to build a particular derived object (*mklabel -config*), it uses the entire CR hierarchy of the specified DO. Similarly, ClearCase maintenance procedures are careful not to “scrub” the CR associated with a deleted DO if it is a member of the CR hierarchy of a higher-level DO.



```
# makefile to build ‘hello’ program

hello: hello.o  msg.o  libhello.a
     cc -o hello -L. hello.o msg.o -lhello

hello.o
     cc -c hello.c

msg.o
     cc -c msg.c

libhello.a  user.o  env.o
     ar r libhello.a user.o env.o

user.o
     cc -c user.c

env.o
     cc -c env.c
```

Figure 11-3 Configuration Record Hierarchy

An individual "parent-child" link in a CR hierarchy is established in either of these ways:

- **In a target/dependencies line —** For example, the following target/dependencies line declares derived objects *hello.o*, *msg.o*, and *libhello.a* to be build dependencies of derived object *hello*:

  ```
  hello: hello.o msg.o libhello.a
  ...
  ```

  Accordingly, the CR for *hello* is the parent of the CRs for the *.o* files and the *.a* file.

- **In a build script —** For example, in the following build script, derived object *libhello.a* in another directory is referenced in the build script for derived object *hello*:

  ```
  hello: $(OBJS)
         cd ../lib ; $(MAKE) libhello.a    (1)
         cc -o hello $(OBJS) ../lib/libhello.a    (2)
  ```

  Accordingly, the CR for *hello* is the parent of the CR for *libhello.a.* Note that the recursive invocation of *clearmake* in the first line of this build script produces a separate CR hierarchy, which is not necessarily linked to the CR for *hello*. It is the reading of *../lib/libhello.a* in the second line of the build script that creates the link between the CRs of *../lib/libhello.a* and *hello*.

Why is Configuration Lookup Necessary?

The *configuration lookup* algorithm used by *clearmake* guarantees that your builds in a parallel-development environment will be both correct (never reuse an object that is not appropriate) and optimal (always reuse an existing object that *is* appropriate).

At the time you enter a *clearmake* command, quite a few wink-in candidates may exist. You cannot simply select the candidate most recently built, because it might have been built in another view, using a completely different set of source versions. Even a single-view scenario demands configuration lookup, given the dynamic nature of views:

1. Suppose that you build program *hello* in a view that is configured to select the most recent version of *hello.c* to which the attribute *QAed* has been attached with the value "*Yes*". This turns out to be version 12 on the *main* branch.
2. A user discovers a bug in *hello* that the QA department did not catch. As a result, the QA manager removes the attribute from version 12. Now, version 9 is the most recent version with the attribute, so your view is dynamically reconfigured to access that version.
3. You enter a *clearmake hello* command. Since the version of *hello.c* in the view (*/main/9*) does not match the version in the config rec of your current instance of *hello*, *clearmake* rebuilds the program.

A "standard" *make* program would have been fooled by the recent time-modified stamp in this situation. The program *hello* is not out-of-date with respect to version 12 of *hello.c*, so it is certainly not out-of-date with respect to the even-older version 9. Thus, the standard *make* algorithm would have declared *hello* up-to-date, and declined to rebuild it.

Wink-In without Configuration Lookup / The 'winkin' Command

clearmake is careful never to wink-in a DO that does not match your view's build configuration. But you can "manually" wink-in any DO to your view, using the *winkin* command:

```
% cleartool lsdo hello
08-Mar.12:48   akp        "hello@@08-Mar.12:48.265"
07-Mar.11:40   george     "hello@@07-Mar.11:40.217"

% cleartool winkin hello@@07-Mar.11:40.217
Winked in derived object "hello"

% hello
Greetings, sakai-san!
Your home directory is /net/neptune/home/sakai.
It is now Tue Mar  8 12:58:30 1994.
```

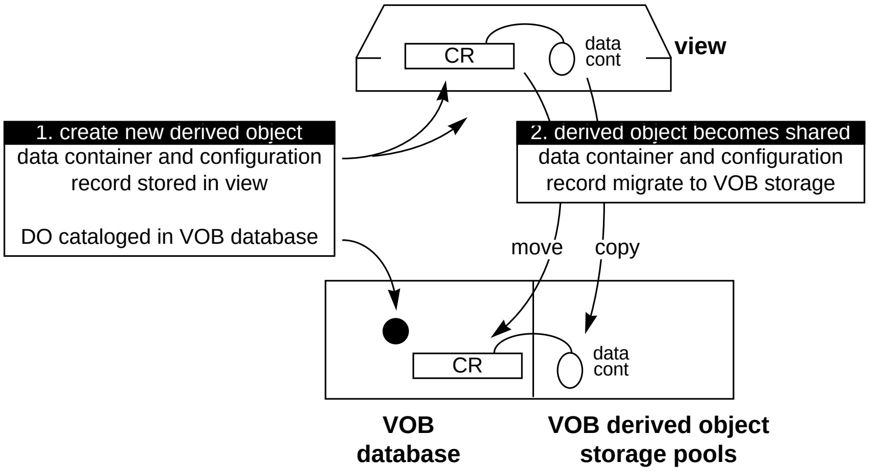
You might wink-in an DO with the "wrong" configuration in order to run it, to perform a byte-by-byte comparison with another DO, or to perform any other operation that requires access to the DO's file system data.

Management of DOs and CRs

This section describes the ways in which ClearCase manages derived objects and configuration records. This user-level discussion provides background information for your day-to-day development tasks. We also mention some administrative-level facilities, and provide cross-references to the *CASEVision/ClearCase Administration Guide*.

Storage of DOs and CRs

You probably think of derived objects (DOs) simply as build targets — the files produced by your compiler (or other translation program). But actually, a DO consists of several parts (Figure 11-4):



CR
data cont
view
1. create new derived object
data container and configuration record stored in view
DO cataloged in VOB database
2. derived object becomes shared
data container and configuration record migrate to VOB storage
move
copy
CR
data cont
VOB database
VOB derived object storage pools

Figure 11-4 Derived Object and its Configuration Record

- **VOB database object** — Each DO is cataloged in the VOB database, where it is identified both by an extended name that includes both its standard pathname (for example, */usr/hw/src/hello.c*) and a unique *DO-ID* (for example, *23-Feb.08:41.391*).

- **Data container** — The “data” portion of a derived object is stored in a standard file within a ClearCase storage area. This file is called a *data container* — it contains the DO’s *file system data* (as opposed to its *meta-data* in the VOB database).
- **Configuration record** — Actually, a CR is *associated with* a DO; it not an intrinsic part of the DO itself. More precisely, a CR is associated with the entire set of *sibling* DOs created by a particular invocation of a particular build script.

During the lifetime of a DO, both its data container and its CR can migrate between storage areas. When you first create a new DO in your view, its data container and CR are both stored in your view. (The actual pathname of the data container is probably of little concern to you; if you’re curious — or you’re troubleshooting — you can use the *mvfsstorage* utility to determine its “real” pathname. See Chapter 15,” Determining a Data Container’s Location”, in the *CASEVision/ClearCase Administration Guide*.)

When you are about to perform a large build, keep in mind the fact that derived objects are initially created in view storage. In the “worst-case” scenario, the disk partition containing your view’s private storage area must be able to accommodate building a new DO for every build target. (Conversely, building a large software system might cost your view very little disk space — if someone else recently built it using a similar configuration of sources.)

Promotion of DOs

The first time that the DO is shared with another view — during a *clearmake* build or with an explicit *winkin* command — the DO is *promoted* from view storage to VOB storage:

- The derived object in the VOB database remains where it is — this object never migrates.
- The data container is copied (by the *promote_server* utility) to one of the VOB’s derived object storage pools.
- The CR is moved from view storage to the VOB’s database.

Promotion allows any number of views to share derived objects without having to communicate directly with each other. For example, a view *alpha* can be “unaware” of the other views, *beta* and *gamma*, with which it shares a derived object. The hosts on which the view storage directories are located need not have network access to each other’s disk storage. Information on which views are currently sharing a DO is maintained in the VOB database, and is accessible with the *lsdo* command:

```
% cleartool lsdo -l hello.o
10-Mar-94.15:25:52     Allison K. Pak (akp.user@copper)
  create derived object "hello.o%10-Mar.15:25.213"
  references: 2 (shared)
  => copper:/var/tmp/akp/tut/old.vws
  => copper:/var/tmp/akp/tut/fix.vws
```

DO Reference Counts

When a new derived object is created, *clearmake* sets the DO’s *reference count* to 1, indicating that it is visible in one view. Thereafter, each wink-in of the DO to an additional view increments the reference count. You can also create additional UNIX-level hard links to an existing DO, each of which also increments the reference count.

Such additional hard links are *sometimes* subject to wink-in:

- If the additional hard link was created along with the original DO, in the same build script, then a wink-in of the DO during a subsequent *clearmake* build also causes a wink-in of the additional hard link.
- Additional hard links that you’ve created “manually” are not winked-in during subsequent builds.

Decrementing the Reference Count

When a program running in any of these views overwrites or deletes a shared derived object, the “link” is broken and the reference count is decremented. That is, the program deletes the view’s *reference* to the DO; the DO itself remains in VOB storage, safe and sound.

This occurs most often when a compiler overwrites an old build target. You can also explicitly remove the derived object with a standard *rm*(1) command, or a *make clean* invocation.

Zero Reference Counts

It is quite common for a derived object’s reference count to become zero. Suppose you build program *hello* and rebuild it a few minutes later. The second *hello* overwrites the first *hello*, decrementing its reference count. Since the reference count probably was 1 (no other view has winked it in), it now becomes 0. Similarly, the reference counts of “old” DOs — even those widely shared — eventually decrease to zero as development proceeds and new DOs replace them.

The *lsdo* command ignores such DOs by default, but you can use the *-zero* option to list them:

```
% cleartool lsdo -zero -long hello.o
  .
  .
08-Mar-94.12:47:54     Allison K. Pak (akp.user@cobalt)
  create derived object "hello.o@@08-Mar.12:47.259"
  references: 0
  .
  .
```

A derived object that is listed with a *references: 0* annotation does not currently appear in any view. But some or all of its information may still be available:

- If the DO was ever promoted to VOB storage (ever shared), then its data container is still in the VOB storage pool, and its CR is still in the VOB database. You can use *catcr* and *diffcr* to work with the CR; you can get to its file system data by performing a *clearmake* build in an appropriately-configured view, or by using the *winkin* command.
- If the DO was never promoted, then its CR may be gone forever:

  ```
  % cleartool catcr msg.o@@26-Feb.20:27.453
  Config record data no longer available for
  "msg.o@@26-Feb.20:27.453"
  ```

Explicit Removal of DOs

Before a DO becomes shared, ClearCase makes no special effort to preserve it. You can delete it with standard software that removes or overwrites files. Likewise, a subsequent *clearmake* build in the same view may overwrite data containers in that view. Such activity destroys the data container in view-private storage; typically, it also destroys the associated CR, which is also in view storage.

After a DO becomes shared, however, it can be deleted only with special ClearCase commands. You can delete individual DOs with the *rmdo* command, but be careful — it will cause errors in processes that are currently attempting to use the deleted DO.

Derived Object Scrubbing

A zero reference count means that the derived object has been deleted, overwritten, or rebuilt in every view that ever used it. This situation calls for *scrubbing*: automatic deletion of DO-related information from the VOB. This can include the removal of the derived object from the VOB database, removal of its data container from a VOB storage pool (if the DO had ever been shared), and in some cases removal of its associated CR, as well. For more on scrubbing, see Chapter 10, “Periodic Maintenance of the Data Repository”, in the *CASEVision/ClearCase Administration Guide*.

Chapter 12

Makefile Optimization

This chapter presents guidelines for *makefile* construction, which will help you to make best use of clearmake.

Pathnames in Build Scripts

ClearCase extends the standard UNIX file system, allowing you to use both view-extended and version-extended pathnames:

```
/view/gamma/usr/hw/src/msg.c         (view-extended pathname)
msg.c@@/main/REL3                 (version-extended pathname)
```

You can use such pathnames with <ProgramName>cleartool and with standard operating system programs. But such names do not work in *makefile* build scripts — use standard pathnames only.

Declaring Source Dependencies in Makefiles

To implement build avoidance based on files' time-modified stamps, standard *make* variants require you to declare all the source file *dependencies* of each build

target. For example, object module *hello.o* might depend on source files *hello.c* and *hello.h* in the same directory:

```
hello.o: hello.c hello.h
      rm -f hello.o
      cc -c hello.o
```

Typically, these source files depend on project-specific header files through #include directives, perhaps nested within one another. The standard UNIX files do not change very often, but it is a common programmer's lament that "it didn't compile because someone changed the project's header files without telling me".

To alleviate this problem, some organizations include *every* header file dependency in their *makefiles*. They rely on utility programs (for example, *makedepend*) to read the source files and determine the dependencies.

clearmake does not require that source-file dependencies be declared in *makefiles* (but see the next section). The first time a derived object is built, its build script is always executed — thus, the dependency declarations are irrelevant. On *rebuilds* of a derived object, its configuration record provides a complete list of source-file dependencies, including those on header files.

You can leave source-file dependency declarations in your existing *makefiles*, but you need not update them as you revise the *makefiles*. And you need not place source-file dependencies in new *makefiles* to be used with *clearmake*.

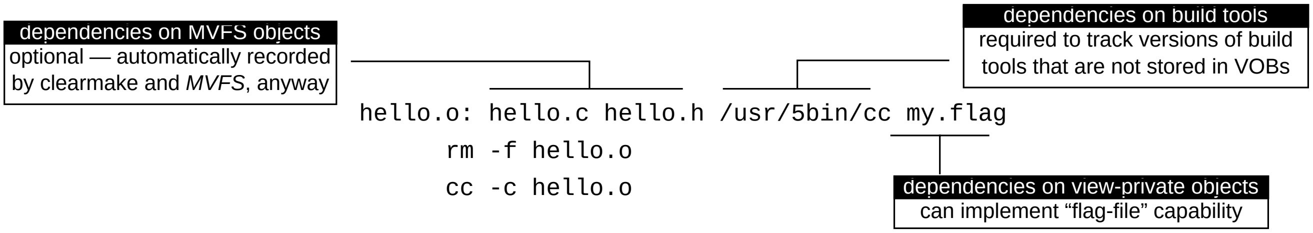
Note: Even though dependency declarations are not required, you may want to include them in your *makefiles*, anyway. The principal reason for doing so is portability — you may need to provide your sources to another group (or another company) that is not using ClearCase.

Explicitly-Declared Source Dependencies

ClearCase's automatic *build auditing* facility tracks only the MVFS objects actually used in the building of a target. Sometimes, however, you may wish to track other objects:

- the version of a compiler, which is not stored in a VOB
- the version of the operating system kernel, which is not referenced at all during the build
- the state of a *flag-file*, possibly a non-MVFS file, used to force rebuilds

You can force such objects to be recorded in a build's CR as shown in Figure 12-1 by declaring them as dependencies of the *makefile* target:



```
hello.o: hello.c hello.h /usr/5bin/cc my.flag
      rm -f hello.o
      cc -c hello.o
```

dependencies on MVFS objects
optional — automatically recorded by clearmake and *MVFS*, anyway

dependencies on build tools
required to track versions of build tools that are not stored in VOBs

dependencies on view-private objects
can implement "flag-file" capability

Figure 12-1 Explicitly-Declared Source Dependencies

We suggest that you use view-private files as *flag files*, rather than using non-MVFS files (such as */tmp/flag*). In a distributed build, a view-private flag file is guaranteed to be the same object on all hosts; there is no such guarantee for a non-MVFS file.

As an alternative to declaring your C compiler as a build dependency, you might place it (and other tools) in a "tools VOB". The versions of such tools will automatically be recorded, eliminating the need for explicit dependency declarations. Additional issues in the auditing of build tools are discussed in the next section.

Explicit Dependencies on 'Searched-For' Sources

There are situations in which clearmake's configuration lookup algorithm qualifies a derived object, even though an actual target rebuild would produce a different result. Configuration lookup requires that for each object listed in an existing CR, the current view must select the same version of *that* object. It does not take into account the possibility that a target rebuild might use a *different* object altogether.

For files that are accessed by explicit pathnames, this situation cannot occur. But it *can* occur if a file is accessed at build time by a search through multiple directories. For example, the following build script uses a search to locate a library file, *libprojutil.a*:

```
hello:
      cc -o hello -L /usr/project/lib -L /usr/local/lib \
            main.o util.o -lprojutil
```

The command *clearmake hello* might qualify an existing derived object built with */usr/local/lib/libprojutil.a*, even though performing a target rebuild would now use */usr/project/lib/libprojutil.a* instead.

clearmake addresses this problem in the same way as some standard *make* implementations:

- You must declare the searched-for source object as an explicit dependency in the *makefile*:

```
hello: libprojutil.a
...
```

You must use the VPATH macro to specify the set of directories to be searched:

```
VPATH = /usr/project/lib:/usr/local/lib
```

Given this makefile, clearmake will use the VPATH (if any) when it performs configuration lookup on *libprojutil.a*. If a candidate derived object was built with */usr/local/lib/projutil.a*, but would be built with */usr/project/lib/projutil.a* in the current view, the candidate is rejected.

Note: The VPATH macro is not used for *all* source dependencies listed in the config rec. It is used only for explicitly-declared dependencies of the target.

Build Tool Dependencies. You can use this mechanism to implement dependencies on build tools. For example, you might track the version of the C compiler used in a build as follows:

```
msg.o: msg.c $(CC)
       $(CC) -c msg.c
```

With this *makefile*, either your VPATH must include the directories on your search path (if the *$(CC)* value is simply “cc”), or you must use a full pathname as the *$(CC)* value.

Build-Order Dependencies

In addition to source dependencies, *makefiles* also contain build-order dependencies. For example:

```
hello: hello.o libhello.a
      ...
libhello.a: hello_env.o hello_time.o
      ...
```

These dependencies are buildable objects, and thus are termed *subtargets*. Executable *hello* must be built after its subtargets, object module *hello.o* and library *libhello.a*, and the library must be built after its subtargets, object modules *hello_env.o* and *hello_time.o*.

ClearCase does not automatically detect build-order dependencies; you must include such dependencies in *makefiles* used with clearmake, just as with other *make* variants.

Build Sessions, Subsessions, and Hierarchical Builds

Note: Throughout this section, references to "clearmake" should more precisely be "clearmake or clearaudit". See the *clearaudit* manual page for more on non-*makefile*-based building of software.

The following terms are useful in describing the details of ClearCase build auditing:

- A "top-level" invocation of *clearmake* starts a *build session*. The time at which the build session begins becomes the *build reference time* for the entire build session, as described on "Continuing to Work During a Build / Reference Time" on page 160.
- During a build session, one or more *target rebuilds* typically take place.
- Each target rebuild involves the execution of one or more *build scripts*. (A double-colon target can have multiple build scripts.)
- During each target rebuild, clearmake conducts a *build audit*.

Subsessions

A build session can have any number of *subsessions*, all of which inherit the reference time of the build session. A subsession corresponds to a “nested build” or “recursive make”, started when a clearmake process is invoked in the process family of a higher-level clearmake. Examples of clearmake invocations that start subsessions include:

- including a clearmake command in a *makefile* build script executed by clearmake
- entering a clearmake command in an interactive process started by *clearaudit*

A subsession begins while a higher-level session is still conducting build audits. The subsession conducts its *own* build audit(s), independent of the audits of the higher-level session — that is, the audits are not nested or related in any way, other than that they share the same build reference time.

Versions of Elements Created During a Build Session

Any version created during a build session and selected by a *LATEST* config spec rule will not be visible in that build session. For example, a build might checkin a derived object it has created; subsequent commands in the same build session will not “see” the checked-in version, unless it is selected by a config spec rule that does not involve the version label *LATEST*.

Coordinating Reference Times of Several Builds

Different build sessions have different reference times. The “best” way to have a series of builds share the same reference time is to structure them as a single, hierarchical build.

An alternative approach is to run all the builds within the same *clearaudit* session. For example, you might write a shell script, *multi_make*, that includes several invocations of clearmake (along with other commands). Running the script as follows ensures that all the clearmake builds will be subsessions that share the same reference time:

```
% clearaudit -c multi_make
```

Objects Written at More than One Level

Undesirable results occur when the same file is written at two or more session levels (for example, a top-level build session and a subsession): the build audit for the higher-level session does not contain complete information about the file system operations that affected the file. For example:

```
% clearaudit -c "clearmake shuffle > logfile"
```

The file *logfile* may be written both:

- during the *clearaudit* build session, by the shell program invoked from *clearaudit*
- during the clearmake subsession, when the *clearaudit* build session is suspended

In this case, *clearaudit* issues this error message:

```
Unable to create derived object "logfile"
```

To work around this limitation, “postprocess” the derived object at the higher level with a copy command:

```
% clearaudit -c "clearmake shuffle > log.tmp"
% cp log.tmp logfile
% rm log.tmp
```

No Automatic Creation of Configuration Record Hierarchy

CRs created during a build session and its subsessions are not automatically linked into a single configuration record hierarchy. For more information, see “CR Hierarchies” on page 171.

Incremental Updating of Derived Objects

The design of ClearCase's *build auditing* capability makes it ideal for use with tools that build derived objects "from scratch". Since such newly-created objects have no "history", ClearCase can learn everything it needs to know at build time. But this reliance on build-time file-system-level auditing causes ClearCase to record incomplete information for incrementally-updated objects, which *do* have a history.

From ClearCase's perspective, *incremental updating* means that an object is partially updated during the builds of multiple *makefile* targets, instead of being completely generated by the build of one target. clearmake does not incrementally update an existing CR when it builds a target. Instead:

- Each time a build script incrementally updates an object's file system data, clearmake writes a completely new CR, which describes only the most recent update, not the entire build history.
- The new CR does not match the desired build configuration for any of the other targets that incrementally update the object.

This results in a situation that is both unstable and incorrect: all incremental-update targets will be rebuilt each time that clearmake is invoked; when it's done, the DO will have the correct file system data, but its CR will not accurately describe the DO's configuration.

Example: Building an Archive

A common incremental-update scenario in "traditional" UNIX environments is the building of an archive (programming library) by *ar*(1).

A traditional *make* program treats an archive as a compound object; it can examine the time-modified stamps of the individual components (object modules) in the archive; and it can update the archive by replacing one or more individual object modules. Here is a simple *makefile* in which a special syntax enables multiple targets to incrementally update a single archive, *libvg.a*:

```
libvg.a:: libvg.a(base.o)
libvg.a:: libvg.a(in.o)
libvg.a:: libvg.a(out.o)
```

If you edit one of the library's sources (for example, *out.c*); a "traditional" *make* program uses the special syntax and a *.c.a* built-in rule to update the library as follows:

- It looks inside the archive *libvg.a*, and determines that it includes an *out.o* that is older than its source file.
- It compiles a new *out.o* from *out.c*.
- It uses *ar* to incrementally update *libvg.a*, replacing the old instance of object module *out.o* with the newly-built instance.

clearmake does not implement this algorithm, and includes no support for treating an archive as a compound object. ClearCase build-avoidance is based solely on meta-data (CRs), not on any analysis at the file-system-data level. clearmake interprets the above *makefile* as follows

- It considers all the *libvg.a*(...) dependencies to be multiple instances of the same "double-colon" build target.
- Accordingly, whenever *one* of those "double-colon" targets requires rebuilding, it rebuilds them all, using the standard *.c.a* built-in rule. This effectively rebuilds the entire archive *libvg.a* from scratch.

Thus, clearmake accepts the standard incremental-update syntax, but interprets it in a way that produces a non-incremental build procedure.

Remedies for the Incremental-Update Problem

Some *makefile* restructuring can ameliorate the situation described above. Often, a restructured build procedure can take advantage of *wink-in* to compensate for the loss of incremental updating. For example, you might revise the procedure for building the archive *libvg.a* (discussed in the preceding section) to dispense with the special *ar*-informed syntax:

```
libvega.a: base.o in.o out.o
        ar rv libvega.a base.o in.o out.o
base.o:
        cc -c base.c

in.o:
        cc -c in.c

out.o:
        cc -c out.c
```

Object modules built by this *makefile* are standard, sharable derived objects; typically, as they libraries sources stabilize over time, most builds of target *libvega.a* will reuse or wink-in most of the object modules.

Avoid the following restructuring; it will cause a complete rebuild of the archive each time any object module is updated:

```
base.o: base.c
        cc -c base.c
        ar rv libvg.a base.o
  .
  . and so on
```

Additional Incremental-Update Situations

You may encounter incremental updating in other situations, as well. For example, C++ compilers that support parameterized types (*templates)* often update *type map* files incrementally as different targets are built. ClearCase includes special *makefile* rules that store per-target type map files.

Ada compilers often update certain common files in Ada libraries incrementally, as different compilation units are built. There are no current clearmake workarounds to implement per-target CRs for Ada libraries. To produce a CR for an Ada library, you can perform a complete rebuild of the library from its sources in a single *clearaudit* session.

Build Auditing and Background Processes

The ClearCase build programs — clearmake, *clearaudit*, and *abe* — all use the same procedure to produce configuration records:

1. Sends a request to the host’s multiversion file system (MVFS), initiating build auditing.
2. Invoke one or more child processes (typically, shell processes), in which *makefile* build scripts or other commands are executed.
3. Turn off MVFS building auditing.
4. If all the subprocesses have indicated success by returning a zero exit status, and at least one MVFS file has been created, compute and store one or more configuration records.

Any subprocesses of the child processes invoked in Step #2 inherit the same MVFS build audit. (Recursive invocations of ClearCase build programs conduct their own, independent audits — see <Emphasis>Build Sessions, Subsessions, and Hierarchical Builds on page 185.)

A problem can occur if a build script (or other audited command) invokes a background subprocess, and exits without waiting for it to complete. The build program has no knowledge of the background process; it may proceed to Steps #3 and #4 before the background process has finished its work. In such situations, ClearCase cannot guarantee what portion, if any, of the actions of background commands will be reflected in the resulting CR — it depends on system scheduling and timing behavior. Thus, you should strictly avoid using background processes in audited build scripts.

Chapter 13

Setting Up a Distributed Build

This chapter describes the process of setting up and running builds that use several hosts in the local area network. Included are descriptions of both the “client-side” and the “server-side” control mechanisms.

Overview of Distributed Building

ClearCase can perform *distributed builds*, in which multiple hosts around the local area network execute the build scripts associated with *makefile* targets. This feature can provide a significant performance improvement — instead of using a single processor to perform the work, one build script at a time, you can have 5, 10, or 20 processors work in parallel. With large software systems, this performance improvement can make a critical difference — for example, enabling the entire application to be built each night.

You start a distributed build in much the same way as a single-host build: by entering a *clearmake* command. A command-line option or environment variable setting causes the build to “go distributed”.

A distributed build is controlled by specifications on all the hosts involved:

- **Client-side controls** — The host where you enter the *clearmake* command is the “build client” or “build controller”. On this host, you specify such information as the set of hosts to be used for building and the number of build scripts to be executed concurrently.
- **Server-side controls** — Each “build server” host used in a distributed build can have an access-control file. A build client must meet the access-control requirements in order to use the host as a build server.

The Audited Build Executor (abe)

To dispatch a build script, *clearmake* uses the standard UNIX “remote shell” facility to start an Audited Build Executor (*abe*) process, set to the current view, on a build host. A build script runs under *abe* control much as if it were being executed by *clearmake* — the typical result is the creation of a set of derived objects and an associated configuration record. *abe* collects terminal output produced by the build script, and sends it back to the build controller, where it appears in your *clearmake* window.

Figure 13-1 illustrates the ClearCase distributed build architecture. The following section presents a simple step-by-step procedure for setting up both the client and server sides of a distributed build.

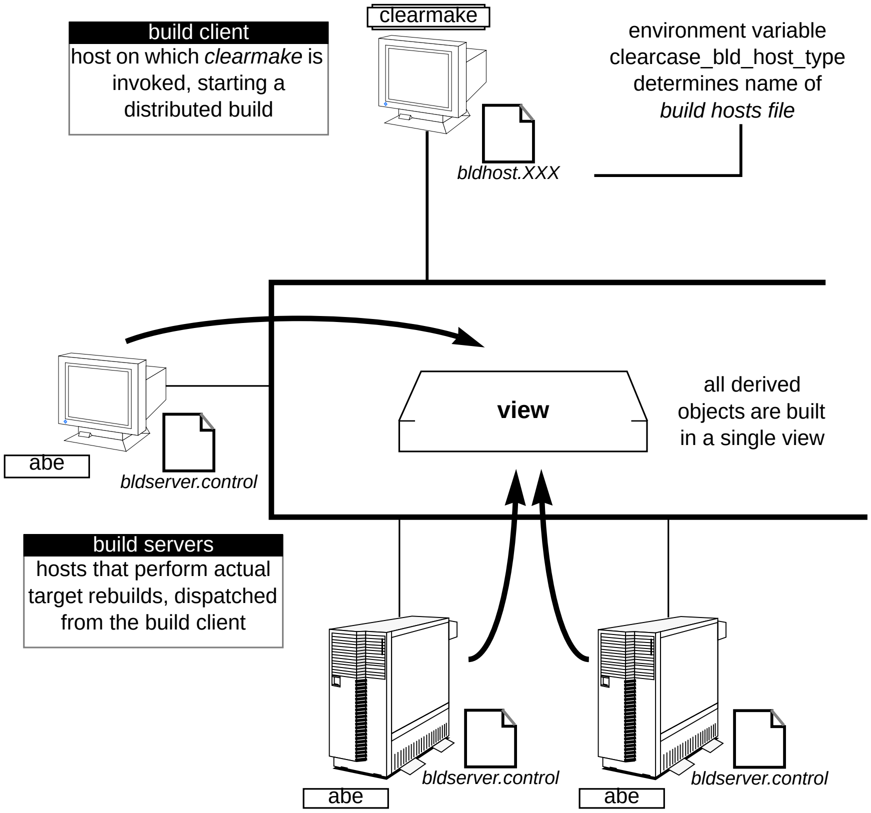

clearmake
build client
host on which *clearmake* is invoked, starting a distributed build
environment variable clearcase_bld_host_type determines name of *build hosts file*
bldhost.XXX
view
all derived objects are built in a single view
abe
bldserver.control
build servers
hosts that perform actual target rebuilds, dispatched from the build client
abe
bldserver.control
abe
bldserver.control

Figure 13-1 ClearCase Distributed Build Architecture

Client-Side Setup

No special setup is required for the client host itself where you enter the *clearmake* command. Rather, you must set up one or more *build hosts* files in your home directory. Each such file must have a name that begins with ".bldhost". Choose a file name suffix for each build hosts file that describes its intended use. For example:

.bldhost.sun5	list of hosts used to build SunOS 5 binaries
.bldhost.day	list of hosts used to perform distributed builds during the work day
.bldhost.night	list of hosts used to perform overnight distributed builds

Depending on your build environment, you may or may not need multiple build hosts files. In a heterogeneous network, for example, architecture-specific builds may or may not need to be performed on hosts of that architecture. (You may have cross-compilers, which eliminates this restriction.)

When you start a distributed build, *clearmake* selects a particular build hosts file using an environment variable — even if you have only one such file. See "Starting a Distributed Build" on page 198.

You might set up two build hosts files, for daytime and nighttime use, as follows:

1. **Create a build hosts file for daytime use —** For daytime builds, you might use the list of hosts that your system administrator has provided in */usr/local/lib*, along with your own host. To minimize the disruption to other work, you might specify that each host is to be used only if it is not heavily loaded: at least 75% idle.

   ```
   % cat > $HOME/.bldhost.day
   -idle 75
   neptune
   #include /usr/local/lib/day_builds
   <ctrl-d>
   ```

2. **Create a build hosts file for overnight use —** For overnight builds, you might use another list of hosts provided by the system administrator.

   ```
   % cat > $HOME/.bldhost.night
   #include /usr/local/lib/night_builds
   <ctrl-d>
   ```

   Since this file does not include a *-idle* specification, *clearmake* will default to using a host only if it at least 50% idle.

For details on build hosts files, see the *bldhosts* manual page.

Server-Side Setup

Each build server host can have a *bldserver.control* file, which controls its usage for distributed builds. This file can impose such restrictions as limiting *who* can use the host for distributed builds, and specifying *when* it can be used for this purpose. If a build server host has no such file, it will accept all distributed build requests. The *bldserver.control* manual page describes the details of this mechanism.

Here's how you might set up a build server host that is used both for your group's daytime builds and its overnight builds:

1. **Create a *bldserver.control* file —** Each line of the *bldserver.control* file defines a situation in which it will accept distributed build requests.

   ```
   % cat > /usr/adm/atria/config/bldserver.control
   -time 08:30,19:30 -idle 60                          (1)
   -time 19:30,05:30                                   (2)
   -user bldmeister                                    (3)
   <ctrl-d>
   ```

   Line 1 specifies that during the interval between 8:30am and 7:30pm, this host will honor a distributed request when it is at least 60% idle. Line 2 specifies that during the interval between 7:30pm and 5:30am, this host will honor any distributed request, no matter how busy it is. Line 3 specifies that a distributed build request from a *clearmake* invoked by user *bldmeister* will always be honored.

2. **Protect the *bldserver.control* file —** Make sure that your access-control settings can't be deleted or altered:

```
% chmod 444 /usr/adm/atria/config/bldserver.control
```

Handling of the Idleness Threshold

Note that the *idleness threshold* can be specified with *-idle* settings on both the client and server sides. If there is a conflict, the overall principle is that the build server host is the "master of its own fate". Examples:

- A *clearmake* process is searching for hosts that are at least 50% idle (the default). A build server that would appear to qualify because it is 70% idle will *not* be used if its *bldserver.control* file includes an *-idle* 75 specification.
- A *bldserver.control* file on a build server host permits access, because *–idle* 60 is specified on a host that is currently 75% idle. But *clearmake* does not dispatch a build script to this host, because the build hosts file specifies an even higher threshold, *-idle* 80.

Starting a Distributed Build

To start a distributed build, you must set an environment variable, then invoke *clearmake* with the appropriate option:

1. **Set the *clearcase_bld_conc* variable —** The value of this variable determines the name of the build hosts file in your home directory:

clearcase_bld_conc value	**Name of build hosts file**
sun5	.bldhost.sun5
SUN5	.bldhost.SUN5
day	.bldhost.day
night	.bldhost.night

2. **Invoke *clearmake* with distributed building enabled** — You can use a command-line options or an environment variable to enable distributed building. You can start a build that uses up to five hosts concurrently in either of these ways:

```
% clearmake -J 5 my_target           (command-line option)
% setenv CLEARCASE_BLD_CONC 7       (environment variable)
% clearmake my_target
```

Setting clearcase_bld_conc in a Shell Startup Script

In some distributed build environments, you may find it convenient to have your shell startup script set *clearcase_bld_conc* automatically. For example, your group may be supporting an application on several architectures. Building the application for a particular architecture should be as simple as ...

- ... logging in to a host of that architecture
- ... setting a view and go to the appropriate directory
- ... entering a *clearmake -J* command to start a distributed build

You can implement such a scheme as follows:

1. **Use architecture-specific build hosts files** — Each build hosts file should have a suffix that names a target architecture: *.bldhosts.hpux9*, *.bldhosts.sunos5*, and so on. Typically, each of these files would list hosts of just one architecture — for example, all SunOS 5 hosts in *.bldhosts.sunos5*.
2. **Set *clearcase_bld_conc* according to the local host's architecture —** Include a routine in your shell startup file that determines the hardware/software architecture of the local host, and sets *clearcase_bld_conc* to one of the suffix strings: *hpux9*, *sunos5*, and so on. Here is a code fragment from C shell startup script:

```
set ARCHSTRING = "`uname -s ; uname -r`"
switch ("$ARCHSTRING")
      case "SunOS 5*":
            setenv CLEARCASE_BLD_CONC sunos5
            breaksw
      case "HP-UX 9*":
            setenv CLEARCASE_BLD_CONC hpux9
            breaksw
  ...
```

Chapter 14

Building Software for Multiple Architectures

This chapter addresses the challenge of using a single source tree to develop an application for a variety of hardware/software platforms. We discuss various approaches, contrasting their advantages and disadvantages. An extended example incorporates some of the approaches.

Issues in Multiple-Architecture Development

The following issues arise in an environment where developers are creating and maintaining several architecture-specific *variants* of an application:

- **Different source code is required for different variants** — Different UNIX-based operating systems may use different functions to implement the same task (for example, *strchr*(3) vs. *index*(3)). Likewise, it may be necessary to include different header files for different variants (for example, *string.h* vs. *strings.h*).
- **Different build procedures are required for different variants** — The build procedures for different platforms vary. The differences might involve such particulars as compiler locations, compiler options, and libraries.
- **Builds for different variants must be kept separate** — Since there is a single source tree, care must be taken to ensure that object modules and executables for one architecture do not become confused with those for other architectures. For example, the link editor must not try to create an IRIX–5 executable using an object module that was built for SunOS–4.

The following sections discuss and compare approaches to these issues. There are additional issues to be addressed in situations where ClearCase itself does not run on one of the target platforms. See Chapter 15, “Setting Up a Build on a Non-ClearCase Host,” for a discussion of one such issue.

Handling Source Code Differences

We recommend that you use the same files (that is, the same versions of file elements) in all builds, for all platforms. You can usually achieve this goal using the standard UNIX approach: conditional compilation using the C preprocessor, *cpp*(1). If header file *string.h* is to be used for the architecture whose *cpp* symbol is ARCH_A, and header file *strings.h* is to be used for architecture ARCH_B, use this code:

```
#ifdef ARCH_A
#include <string.h>
#else
#ifdef ARCH_B
#include <strings.h>
#endif /* ARCH_B */
#endif /* ARCH_A */
```

If a file element is not amenable to conditional compilation (for example, a bitmap image), the traditional solution is to put architecture-specific code in different elements altogether (for example, *panel.image.sparc* vs. *panel.image.mc68k*). This approach requires that build scripts be made architecture-specific, too.

With ClearCase, you also have the alternative of splitting the element into branches. The ARCH_A variant might be developed on the element's */main/arch_a* branch; edits and builds for that variant would be performed in a view configured with this rule:

```
element * /main/arch_a/LATEST
```

Other variants would be developed on similarly-named branches, each using a different view, configured with a rule like the one above. In such a situation, the element's *main* branch might not be used at all.

We recommend that you use this branching strategy sparingly, because of these disadvantages:

- Each time platform-independent code is changed on one of the branches, you may need to merge the change to all the other branches.
- Developers must remember to set their views' config specs in an architecture-specific manner. In each view, only one variant of the application can be built.

Handling Build Procedure Differences

Ideally, a single file (that is, a single version of a file element) will drive all architecture-specific builds. One way to accomplish this is to revise *makefiles* as follows:

- regularize build scripts
- replace architecture-specific constructs (for example, */bin/cc)* with make-macros (for example, a *$(CC)* macro)
- use *clearmake*'s include directive to incorporate architecture-specific settings of the make-macros

For example, suppose that source file *main*.c is compiled in different ways for the SunOS–4 and IRIX–5 variants:

```
main.o                                                  (SunOS–4)
      /usr/5bin/cc -c -fsingle main.c

main.o:                                                 (IRIX–5)
      /usr/bin/cc -c main.c
```

To "merge" these two build scripts, abstract the compiler pathname and the compiler options into make-macros, CC and *CFLAGS*. Then, place an architecture-specific include at the beginning of the makefile:

```
include /usr/project/make_macros/$(BLD_ARCH)_macros
 .
 .
main.o:
       $(CC) -c $(CFLAGS) main.c
```

The files in the *make_macros* directory would have these contents:

```
CC      = /usr/5bin/cc   /usr/project/make_macros/sun4_macros
CFLAGS  = -fsingle

CC      = /usr/bin/cc   /usr/project/make_macros/irix5_macros
CFLAGS  =
```

The make-macro BLD_ARCH acts as a selector between these two files. The value of this macro might be placed in an environment variable by a shell startup script:

```
setenv BLD_ARCH ‘uname -s‘
```

Alternatively, developers might specify the value at build time:

```
clearmake main BLD_ARCH="IRIX5"
```

Alternative Approach, Using ‘imake’

The *imake* utility, distributed with many UNIX variants and available free-of-charge from MIT, provides an alternative to the scheme described in the preceding section. The *imake* methodology also involves architecture-specific make-macros, but in a different way. *imake* generates an architecture-specific *makefile* by running *cpp* on an architecture-independent template file, typically named *imakefile*.

A typical *imakefile* contains a series of *cpp* macros, each of which expands to a build target line and its corresponding multiline build script. Typically, the expansion itself is architecture-independent:

```
MakeObjectFromSrc(main)                (macro in ‘imakefile’)

                               (expansion in actual makefile)
main.o: $(SRC)/main.c
       $(CC) -c $(CFLAGS) $(SRC)/main.c
```

imake places architecture-specific make-macro settings at the top of the generated makefile. For example:

```
SRC    = ..
CC     = /usr/5bin/cc
CFLAGS = -fsingle
RM     = rm -f
```

An idiosyncrasy of *imake* usage is that *makefiles* are derived objects, not source files. The architecture-independent template file (*imakefile*) is the source file, and should maintained as a ClearCase element.

Segregating the Derived Objects of Different Variants

It is essential to keep derived objects (object modules, executables) built for different architectures separate. This section describes two approaches, though others are possible.

Approach 1: Use Architecture-Specific Subdirectories

Each variant of an application can be built in its own subdirectory of the source directory. For example, if executable *monet*'s source files are located in directory */usr/monet/src*, then the variants might be built in subdirectories */usr/monet/src/sun4*, */usr/monet/src/irix5*, and so on. It is simplest to have the *makefile* create view-private subdirectories for this purpose. But if you wish to use different derived object storage pools for the different variants, you must create the sudirectories as elements (*mkdir* command) and then adjust their storage pool assignments (*chpool* command).

Since the derived objects for the different variants are built at different pathnames (for example, */usr/monet/src/sun4/main.o*), they are guaranteed to be segregated by variant, and *clearmake* will never wink-in an object built for another architecture.

This approach has several advantages:

- All variants of the application can be built in a single view.
- It eliminates the burden of having to consider whether wink-in should be suppressed for some or all targets.
- Since the derived objects for different variants have different pathnames, it is easier to organize multiple-architecture releases.

But this approach may have the disadvantage of requiring build script changes: the binaries for a build are no longer in the source directory, but in a subdirectory. Note, however, that the build script in <Emphasis>Alternative Approach, Using 'imake' on page 204 is structured for just this situation:

```
main.o: $(SRC)/main.c
  .
  .
```

Approach 2: Use Different Views

Perform builds for different platforms in different views (*sun4_bld_vu*, *irix_bld_vu*, and so on). A group of developers working on the same variant can share a view, or each can work in his or her own architecture-specific view.

In most cases, the build script that creates a derived object varies from variant to variant, as discussed in "Handling Build Procedure Differences" on page 203. If so, *clearmake* automatically prevents wink-in of derived objects built for another architecture. If this is not the case, force the build script to be architecture-specific by including a well-chosen message or comment. For example, if BLD_ARCH is used as described the "Handling Build Procedure Differences" on page 203 section, you might include this message:

```
@echo "Building $@ for $(BLD_ARCH)"
```

This approach has the disadvantage that when an element is checked out, the developer can build only one variant of the application. Since the checked-out version is visible only in one view, builds of other variants (which take place in other views) do not "see" the checked-out version. The developer must checkin the element before building other variants.

Another disadvantage of this approach is a “combinatoric explosion” — if seven developers all wish to maintain their own views in which to build four variants, 28 views are required.

Multiple-Architecture Example, Using ‘imake’

The remainder of this chapter presents an example of multiple-architecture development. This example uses *imake* to support building in architecture-specific subdirectories.

Scenario

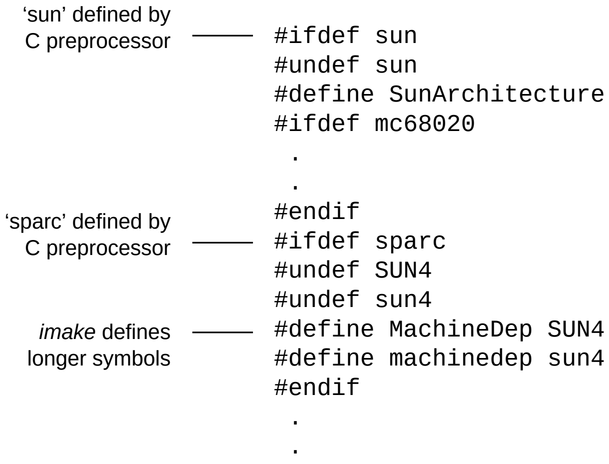
We will show how to set up multiple-architecture development in the */proj/monet/src* directory. A developer will be able to perform a build for a particular architecture as follows:

1. She logs into a machine of the desired architecture — for example, a workstation running SunOS 4.1.3.
2. In her regular view, she goes to the source directory, */proj/monet/src*, and enters a command to have *imake* generate a Makefile.
3. She enters the command *clearmake Makefiles* to have *imake* create the appropriate *Makefile* in the architecture-specific subdirectory *sun4*. Note that the *Makefile* is a derived object, not a source file. Thus, there is no need to create an element from this file.
4. She goes to the *sun4* subdirectory, and then builds software for that architecture, using *clearmake*.

The sections below describe the way in which *imake* is involved in each of these steps.

Defining Architecture-Specific CPP Macros

Step #1 places the developer in an environment where the C preprocessor, *cpp*, defines one or more architecture-specific symbols. On an SunOS–4 host, *cpp* defines the symbols *sun* and *sparc*. This, in turn, causes *imake* to generate many architecture-specific ("machine-dependent") *cpp* macros (See Figure 14-1):



```
#ifdef sun
#undef sun
#define SunArchitecture
#ifdef mc68020
 .
 .
#endif
#ifdef sparc
#undef SUN4
#undef sun4
#define MachineDep SUN4
#define machinedep sun4
#endif
 .
 .
```

Figure 14-1 Defining Architecture-Specific CPP Macros

Additional Sun-specific *cpp* macros are read in from the auxiliary file *sun.cf*.

Creating Makefiles in the Source and Build Directories

A file named *Imakefile* in the source directory is the *imake* input file. This file drives the creation of *Makefiles* both in the source directory itself, and in the architecture-specific subdirectories where software is actually built:

```
#ifndef InMachineDepSubdir
 .
 <code to generate Makefile in source directory>
 .
#else
 .
 <code to generate Makefile in an architecture-specific
subdirectory>
 .
#endif
```

The *Imakefile* code used in the source directory simply defines a symbol to record the fact that builds will not take place in this directory:

```
#define IHaveMachineDepSubdirs
```

The resulting *Makefile* generated by *imake* includes a *Makefiles* target that populates an architecture-specific subdirectory with its own *Makefile*:

```
Makefiles::
    @echo "Making Makefiles in $(CURRENT_DIR)/$$CPU"
    -@if [ ! -d $$CPU ]; then \
        mkdir $$CPU; \
        chmod g+w $$CPU; \
    else exit 0; fi
    @$(IMAKE_CMD) -s $$CPU/Makefile \
    -DInMachineDepSubdir \
    -DTOPDIR=$(TOP) -DCURDIR=$(CURRENT_DIR)/$$CPU
```

Note: *CPU* environment variable determines name of architecture-specific subdirectory

The command *clearmake Makefiles* invokes *imake* once again, using the same *Imakefile* for input. This time, however, the symbol InMachineDepSubdir is defined, causing the actual build code to be generated.

The *Imakefile* in */proj/monet/src* contains these macros:

```
OBJS = cmd.o main.o opt.o prs.o
LOCAL_LIBRARIES = ../../lib/libpub/libpub.a

MakeObjectFromSrc(cmd)
MakeObjectFromSrc(main)
MakeObjectFromSrc(opt)
MakeObjectFromSrc(prs)

ComplexProgramTarget(monet)
```

The resulting *Makefile* generated in the build directory, */proj/monet/src/sun4*, includes this build script:

```
$(AOUT): $(OBJS) $(LOCAL_LIBRARIES)
    @echo "linking $@"
    -@if [ ! -w $@ ]; then $(RM) $@; else exit 0; fi
    $(CC) -o $@ $(OBJS) $(LOCAL_LIBRARIES) \
        $(LDFLAGS) $(EXTRA_LOAD_FLAGS)
```

Chapter 15

Setting Up a Build on a Non-ClearCase Host

This chapter describes a technique that creates configuration records for a build that involves ClearCase data, but is performed on a non-ClearCase host. *Non-ClearCase access* (exporting a VOB through a view) makes the data available to that host; a remote shell is invoked to perform the build on that host.

Scenario

Suppose you wish to build library *libpub.a* for an architecture that is not currently supported by ClearCase, using a host of that architecture named *titan*. The VOB storage area for the library's sources is located at */vobstore/libpub.vbs* on host *sol*. This VOB is also mounted on *sol*, at */proj/libpub*.

Setting Up an Export View

A ClearCase *export view* allows limited access to one or more VOBs using standard NFS export facilities. Each NFS export provides remote access to one VOB through a particular view:

- The VOB itself cannot be modified through the export view: versions cannot be checked out or checked in; CRs cannot be created.
- Builds can be performed in the VOB through the export view; building creates view-private objects, however, not derived objects.

Several VOBs can be exported through the same view, with separate NFS exports. The *exports_ccase* manual page provides a detailed discussion of this subject.

Note: Export views are only to be used for non-ClearCase access to VOBs. To make a view accessible on a remote host, just use the *startview* or *setview* command on that host. An export view *can* be mounted on a ClearCase host — but never try to mount it on the *viewroot* directory, */view*.

The following steps enable the non-ClearCase host to access the *libpub* VOB. As discussed in *"Setting Up an Export View for Non-ClearCase Access"*, of the *CASEVision™/ClearCase Administration Guide*, we will avoid a "multihop" situation by co-locating the VOB storage area, the VOB mount point, and the view storage area on the same host.

1. Create a view through which the *libpub* VOB will be exported. This view must reside the same host as the VOB storage area (host *sol*).

   ```
   % cleartool mkview -tag libpub_expvu /public/export.vws
   Comments for "/public/export.vws":
   export view for libpub VOB
   .
   Created view "/public/export.vws".
   ```

2. Export the mount point of the VOB (*/proj/libpub*) through the export view (*/view/libpub_expvu*). The exact procedure varies from system to system — see *exports_ccase* for details. For example, on a SunOS-4 host:

   - Edit the ClearCase-specific exports table:

     ```
     % su
     Password: <enter root password>
     # vi /etc/exports.mvfs

      <add export entry>

     /view/libpub_expvu/proj/libpub -access:titan
     ```

   - Invoke *export_mvfs* to actually perform the export:

     ```
     # /usr/etc/export_mvfs /view/libpub_expvu/proj/libpub
     ```

Mounting the VOB through the Export View

On the non-ClearCase host, a standard NFS mount is performed on the exported pathname. For example, */view/libpub_expvu/proj/libpub* should be mounted at */proj/libpub* (the same location at which the VOB is mounted on ClearCase hosts).

Revising the Build Script

Build script revisions are required to produce an audited build on a non-ClearCase host. Thus, it makes sense to build in an architecture-specific subdirectory, with a customized *Makefile*. (See Chapter 14, "Building Software for Multiple Architectures," for more on this subject.)

To enable creation of a CR that lists all of the build's input files and output files, the build script executed by <ProgramName>clearmake must:

- declare all input files as explicit dependencies — since the MVFS does not run on the non-ClearCase host, there is no automatic detection of source dependencies
- invoke a remote shell to perform the actual build on the non-ClearCase host
- if the build performed by the remote shell succeeded, perform a *touch*(1) of all output files from the ClearCase host — this turns the view-private files created by the remote shell command into derived objects.

A simple build script might be transformed as illustrated in Example 15-1.

Example 15-1 Build Script for Non-ClearCase Build

```
Native build:

   OBJS = data.o errmsg.o getcwd.o lineseq.o

   data.o:          (no source dependencies need be declared)
       cc -c data.c
    .
    .                (other object modules produced
similarly)
    .

   libpub.a: $(OBJS)
      ar -rc $@ $(OBJS)

Non-ClearCase build:

   OBJS = data.o errmsg.o getcwd.o lineseq.o

   data.o: data.c libpub.h
                       (source dependencies must be declared)
      rm -f $@
      rsh titan 'cd /proj/libpub ; cc -c data.c'
      if [ -w $@ ]; then \
      touch $@ ; \
      fi
    .
    .               (other object modules produced similarly)
    .

   libpub.a: $(OBJS)
      rm -f $@
      rsh titan 'cd /proj/libpub ; ar -rc $@ $(OBJS)'
      if [ -w $@ ]; then
            \touch $@ ; \
      fi
```

The “remote shell” command (*rsh* in the Example 15-1) varies from system to system

The remote shell program typically exits with status 0, even if the compilation failed. Thus, you must use some other technique for checking the success of the build, after the remote shell returns. In the example above, the build scripts assume that the remote build has been successful if the target file exists and is writable.

Performing an Audited Build in the Export View

The following steps perform the desired build:

1. A developer registers and sets the export view on her own workstation, which is a ClearCase host:

   ```
   % cleartool mktag /public/export.vws libpub_expvu
   % cleartool setview libpub_expvu
   ```

2. The developer builds in the normal way, on her own host:

   ```
   % cd /proj/libpub
   % clearmake
   ```

The script listed above specifies a particular non-ClearCase host, *titan*, on which remote shells are to be executed. If there is more than one non-ClearCase host on which builds are to be performed, you must generalize this script.

Note: Since the remote hostname is part of the build script, wink-in of derived objects built on the various hosts will fail, unless you make further modifications (for example, using *clearmake -O* to disable build-script checking).

Chapter 16

Adding a Timestamp to an Executable

This chapter describes simple techniques for incorporating a “version string” and/or “timestamp” into a C-language compiled executable. This allows anyone (for example, a customer) to determine the exact version of a program by entering a simple shell command. The techniques described below support:

- Using the standard UNIX *what*(1) command to determine the version of an executable:

  ```
  % what monet
  monet R2.0 Baselevel 1
  Thu Feb 11 17:33:23 EST 1993
  ```

- Adding a “what version?” command-line option to the executable itself:

  ```
  % monet -Ver
  monet R2.0 Baselevel 1 (Thu Feb 11 17:33:23 EST 1993)
  ```

Once the particular version of the program is determined, you can use ClearCase commands to find a local copy, examine its config rec, and if appropriate, reconstruct the source configuration with which it was built. (Presumably, the local copy is a derived object that has been checked in as a version of an element.)

You can identify the appropriate derived object by attaching a ClearCase attribute with the version string to the checked-in executable, or you could simply rely on the timestamp and your ability to “what” the checked-in executable to find it.

Creating a ‘what’ String

The *what* program searches for a null-terminated string that starts with a special four-character sequence:

```
@(#)
```

To include a “what string” in a C-language executable, define a global character-string variable. For example, these source statements would produce the two-line *what* listing above:

```
char *version_string = "@(#)monet R2.0 Baselevel 1";
char *version_time   = "@(#)Thu Feb 11 17:33:23 EST 1993;
```

As an alternative, you can generate the timestamp dynamically when the *monet* program is linked, using this procedure:[1]

1. Create a new source file, *version_info.c*, which contains the statements that define the “what” strings. But instead of hard-coding a date string, use a *cpp*(1) macro, DATE:

   `In` *`version_info.c`*`:`

   ```
   char *version_string = "@(#)monet R2.0 Baselevel 1";
   char *version_time = DATE;
   ```

2. Revise your makefile so that before linking the executable, it compiles *version_info.c*. Use shell command substitution to dynamically incorporate the current time into the value for the DATE macro:

   ```
   SHELL = /bin/sh
   OTHER_OBJS = main.o cmd_line.o (and so on)

   monet: version_info.c $(OTHER_OBJS)
          cc -c -DDATE="\"@(#)‘date‘\"" version_info.c
          cc -o monet version_info.o $(OTHER_OBJS)
   ```

   A rebuild of *monet* will also be triggered if the *version_string* variable is edited manually in *version_info.c*.

   Note: If you use *clearmake* to build *monet*, you need not declare *version_info.c* as an explicit dependency.

[1] The *version_string* could be generated dynamically, too (for example, with environment variables). But it is more likely that the project leader would manually edit this string’s value before major builds.

Implementing a ‘-Ver’ Option

You need not depend on the *what* command to extract version information from your executable. Instead, you can have the program itself output the information stored in the *version_string* and *version_time* variables. Just revise the source module that does command-line processing to support a “what version” option (for example, *-Ver*):

```
#include <stdio.h>

main(argc,argv)

    int argc;
    char **argv;
{
/*
 * implement -Ver option
 */
    if (argc > 1 && strcmp(argv[1],"-Ver") == 0) {
        extern char *version_string;
        extern char *version_time;
        /*
         * Print version info, skipping the "@(#)" characters
         */
        printf ("%s (%s)\n",
                    &version_string[4], &version_time[4]);
        exit(0);
    }
}
```

Chapter 17

Compatibility between *clearmake* and Other *make* Variants

The *clearmake* program has been designed for compatibility with existing *make* programs, minimizing the work necessary to switch to *clearmake*. There are many independently-evolving variants of *make*, however, which provide different sets of extended features. *clearmake* does *not* support all the features of all the variants, and absolute compatibility is not guaranteed.

If your *makefiles* use only the common extensions, they will probably work with *clearmake* as-is. If you must use features that *clearmake* does not support, consider using another *make* program in a *clearaudit* shell. This alternative provides build auditing (configuration records), but does not provide build avoidance (wink-in).

‘clearmake’ Compatibility With Standard ‘make’

In its default mode, *clearmake* is designed to be compatible with System V Release 3 *make*(1) — “standard *make*”. Standard *make* represents a “least common denominator” for *make* functionality; most other variants of *make* are at least upward-compatible with standard *make*.

We suggest that you limit yourself to standard *make* functionality, since this allows maximum portability among hardware/software platforms, and among different variants of *make* on any single platform. It is also the most complete and most fully-tested of the *clearmake* modes.

In its default mode, *clearmake* supports most standard *make* description file syntax and command line options.

Standard ‘make’ Description File Features Not Supported

clearmake does not support standard *make* inference rules for SCCS files — special-case suffix rules using the tilde (~) character.

Standard ‘make’ Command Line Options Not Supported

The following standard *make* options are not supported by *clearmake*:

–t	“touch” option. This is not supported because *clearmake* configuration lookup differs significantly from standard *make*’s build-avoidance algorithm, which is based on file modification times.
–q	question option
–f –	reading a description file from *stdin*

‘clearmake’ Compatibility Modes

clearmake allows you to specify one *make-compatibility mode* with a command-line option. Complete compatibility is not guaranteed — only the features listed in the sections below are supported.

clearmake supports these make-compatibility modes:

-C *sgismake*	emulate IRIX 4.0.1 *smake* (on SGI hosts)
-C *sgipmake*	emulate IRIX 4.0.1 *pmake* (on SGI hosts)
-C *sun*	emulate SunOS 4.1.x *make* and SunOS 5.1 (Solaris) *make*
-C *gnu*	emulate Gnu *make*

Except where noted, descriptions of the features listed below can be found in the manual pages for the relevant *make* variant on the appropriate platform.

Supported SGI ‘smake’ Features

The following features are enabled when you specify -C *sgismake*:

- all extended macro-assignment operators:

```
?= assign if undefined
:= expand RHS immediately
+= append to macro
!= assign result of shell command
```

- all extended macro-expansion operators:

```
$(VAR:T)
$(VAR:S/pattern/replace/)
$(VAR:H)
$(VAR:R)
$(VAR:Mpattern)
$(VAR:E)
$(VAR:Npattern)
```

- most makefile conditional directives:

```
#if           (expressions may contain ‘defined’ operator
               and ‘make’ operator)
#ifdef, #ifndef
#ifmake, #ifnmake
#else
#elif
#elifmake, #elifnmake
#elifdef, #elifndef
#endif
```

- makefile inclusion with search rules similar to those of *cpp*(1):

#include <*file*>

look for file in */usr/include/make*

#include "*file*"

look for file in current directory, then in directories specified with -I command-line options, then in */usr/include/make*

- command line option –*I*, for use with #include statements
- aliases for internal make macros:

```
$(.TARGET)   alias for $@
$(.PREFIX)   alias for $*

$(.OODATE)   alias for $?
$(.IMPSRC)   alias for $<
$(.ALLSRC)   alias for $>
```

Note: $> is not supported by standard *make*(1).

- *smake*-specific builtins file: */usr/include/make/system.mk*
- inference rules with non-existent intermediates
- search paths for dependencies (.PATH and .PATH.*suffix*)
- deferring build script commands ("..." in build script)
- .NULL target: specifies suffix to use when target has no filename suffix
- .NOTPARALLEL target: disables parallel building
- .MAKE target: specifies that a target corresponds to a sub-make; that target's build script is be invoked even when -n is used

Supported SGI 'pmake' Features

When you specify *-C sgipmake*, all the SGI *smake* features listed above are enabled, along with the following:

- if no target description file is specified on the command line, search for *Makefile* before searching for *makefile*
- undefined macros in build scripts are left unexpanded
- undefined macros outside build scripts cause a fatal error
- one shell per build script (with *-C sgismake*, each command in the build script is executed in a separate shell)

Supported Sun ‘make’ Features

The following features are enabled when you specify -*C sun*:

- all extended macro-expansion operators:

  ```
  +=   append to macro
  :sh= assign result of shell command
  ```

- pattern-replacement macro expansions:

  ```
  $(macro:op%os=np%ns)
  ```

- shell-execution macro expansions:

  ```
  $(macro:sh)
  ```

- conditional (target-dependent) macro definitions:

  ```
  tgt-list := macro = value
  tgt-list := macro += value
  ```

  Target names must be explicit — patterns with % cannot be specified.

- special-purpose macros:

  ```
  HOST_ARCH
  TARGET_ARCH
  HOST_MACH
  TARGET_MACH
  ```

- sun-specific builtins file:
  - *./default.mk* or */usr/include/make/default.mk* (SunOS 4.1.x)
  - *./make.rules* or */usr/share/lib/make/make.rules* (SunOS 5.1)

- Sun pattern-matching rules:

  ```
  tp%ts : dp%ds
  ```

VPATH: Searches for Both Targets and Dependencies

When you specify *-C sun*, *clearmake* uses the VPATH search list (if there is one) to look for the target if both these conditions are true:

- the target's name is not an absolute pathname
- there is no existing file corresponding to the target's name

For each directory in the value of VPATH, the directory path is concatenated with the target's name, and if there is an existing file at the resulting path, then that file is evaluated.

This feature works whether or not *clearmake* uses configuration lookup (that is, either with or without the -T or -F option). If it does use configuration lookup, *clearmake* "prefers" to use a DO in the current view:

1. As always, clearmake tries to reuse the candidate DO (if any) in the current view, built at the target's name.
2. If such a candidate does not exist or does not qualify for reuse, clearmake searches for a candidate in the current view, built in directories on the VPATH.
3. If candidate with an appropriate name exists in a VPATH directory but is rejected by the configuration lookup algorithm, clearmake proceeds to look in the VOB database for other candidates that were built in that same VPATH directory.
4. If no VPATH directory has any candidate with an appropriate name, clearmake proceeds to search the VOB database for other candidates in the directory corresponding to the target's name.

Note: In all these cases, all the DOs on which clearmake performs configuration lookup were built in a single directory — it traverses multiple VPATH directories only in deciding where to begin performing configuration lookup.

VPATH Substitutions in Build Scripts

The names of targets and dependencies in build scripts are replaced by their VPATH-elaborated counterparts. If a file is found using the VPATH, then all white-space-delimited occurrences of the file’s name in a build script are replaced with the pathname at which the file was found. For example:

```
VPATH = tgtdir:depdir

bar.o : bar.c
        cc -c bar.c -o bar.o
```

If *bar.c* is found in directory *depdir*, and *bar.o* is found in directory *tgtdir*, and the target must be rebuilt, then this build script will be executed:

```
cc -c depdir/bar.c -o tgtdir/bar.o
```

Supported ‘Gnu make’ Features

clearmake provides partial compatibility with *Gnu make*. Some *Gnu make* features are supported directly by *clearmake*. Others are supported by preprocessing the makefile(s) using */usr/atria/bin/Gmake*, a program derived from *Gnu make*.

Note: Machine-readable sources to *Gmake* are available. For more information, call Atria Customer Support.

The *Gmake* preprocessor “elaborates” the makefile(s). This involves evaluation of conditional expressions, static pattern rules, and macro references (except those in build scripts). The elaborated makefile is stored as a temporary file, used as input to *clearmake*, and (usually) deleted when *clearmake* no longer needs it.

You can save an elaborated makefile (for example, to examine it) in either of these ways:

- Perform the build with a *clearmake -C gnu -d* command. *clearmake* will preserve the elaborated makefile and display its pathname.

- Invoke *Gmake* directly, specifying -E option and redirecting standard output. For example:

  ```
  /usr/atria/bin/Gmake -E > Makefile.elab
  ```

The following features are enabled when you specify -C gnu:

- “export” statement
- extended macro-assignment operator:

  := expand RHS immediately
- macro assignment ‘override’ keyword
- pattern-replacement macro expansions:

  ```
  $(macro:op%os=np%ns)
  ```
- all other extended-macro expansion operators (unless the expansion occurs in a build script)
- function calls, except those that occur in build scripts and those that occur on the RHS of an = macro assignment. (You can use a function call on the RHS of a := macro assignment.)
- conditional expressions
- static pattern rules
- pattern-matching rules

BOS Files and ‘Gnu Make’ Compatibility

When you use *-C gnu*, *clearmake* does not automatically use build options specification (BOS) files associated with the makefiles it reads. (This is due to the fact that the makefiles are read by the Gmake preprocessor, not by *clearmake* itself.) Use *clearmake*’s *-A* option or environment variable CLEARCASE_BLD_OPTIONS_SPECS to specify the BOS files when using *Gnu make* compatibility.

Compatibility Limitations

Different systems have different names for their “built-in makefiles” — for example, *system.mk* versus *default.mk*. Using -*C -sgismake* on a non-IRIX system, or -*C sun* on a non-SunOS system, may cause errors. You can disable use of built-in rules with *clearmake -r*.

With -*C sun*, *clearmake* uses the SunOS *arch*(1) and *mach*(1) commands to set the values of special macros (for example, HOST_ARCH and HOST_MACH). This generates error messages on systems that do not support these commands. You can safely ignore such messages if your build scripts do not use the special macros. Some alternatives:

- Comment out the lines in /usr/atria/etc/sunvars.mk that define the .CLEARMAKE_ARCH and .CLEARMAKE_MACH macros.
- Write shell scripts to implement the arch and mach commands.

Chapter 18

Customizing the Graphical Interface

This chapter contains both the guide and reference information necessary to customize the graphical user interface.

Introduction

Group Files and Item Definitions

Customizing the graphical interface involves editing *group files*. A group file (*.grp*) defines a set of named operations — a *menu*. A group file's *scope* declaration determines exactly where the menu appears in the interface — file browser toolbar, pulldown menu, vtree browser popup menu, and so on.

Each group file entry, or *item definition*, defines a single menu item. These menu items typically have names like checkout or merge, but there are no inherent restrictions; labels and operations can be entirely unrelated to ClearCase. You can customize any menu item to perform any operation expressible as a shell script.

The "meat" of an item definition is a function call, command line, or shell script (several of these can be combined in one item definition). An item definition can also specify multi-state icon bitmaps, a menu mnemonic, whether to execute in the background, and so on.

Editing the Predefined Group Files

The default, predefined group files are installed in */usr/atria/config/ui/grp*. You should not edit these files in place. It is also not advised to copy and edit them elsewhere — they amount to "source code"; future ClearCase enhancements and bug fixes will leave you behind.

To add your own menus, create new group files in the *.grp* subdirectory of your home directory (*$home/.grp:/usr/atria/config/ui/grp* is the the default group file search path). Alternatively, specify a group file search path with the environment variable grp_path, and do your customization work in any of its component directories. Copy to your work area the predefined group file template */usr/atria/config/ui/grp/user.grp.template*, and use it as a template for your own work. (This template is *scoped* to the Fast menu, so you can use your additions as pulldown menus or add them to the toolbar as "push buttons". See "Scope" on page 237 for details.)

Upcoming sections describe group file contents in detail. Look ahead to "Customization Procedures" on page 289 for more recommendations regarding the customization process.

Note: If you are responsible for implementing or maintaining the graphical interface for a group of people, make your group file customizations in one or more globally accessible directories and see that each user's grp_path environment variable gets set accordingly. Also, note that you can use grp_path to control which menus are visible to which users.

How *xclearcase* Processes Group Files at Startup Time

The default search path for group files is $home/*.grp:/usr/atria/config/ui/grp.* You can override this search path with the environment variable grp_path. The grp_path can include any combination of directories and files.

When you start *xclearcase*, it traverses the group file search path, collects all group files, and caches them in ASCII-sorted order. (This determines the order in which pull-down menus appear on the menubar.) If a group file has syntax errors, messages appear in the transcript pad, and all or part of the group file is ignored. If *xclearcase* encounters more than one group file with the same leaf name, it uses the first one found (permitting you to "eclipse" a group file with one of your own).

Using the cached group files, *xclearcase* builds the menubar, toolbar, and popup menus for various browser classes.

Group File Syntax

First, let's look at a group file. Figure 18-1 shows a file browser and its menus, which are defined by group files. The Basic pulldown menu is defined by the group file *basic.grp*, which appears in Figure 18-2.

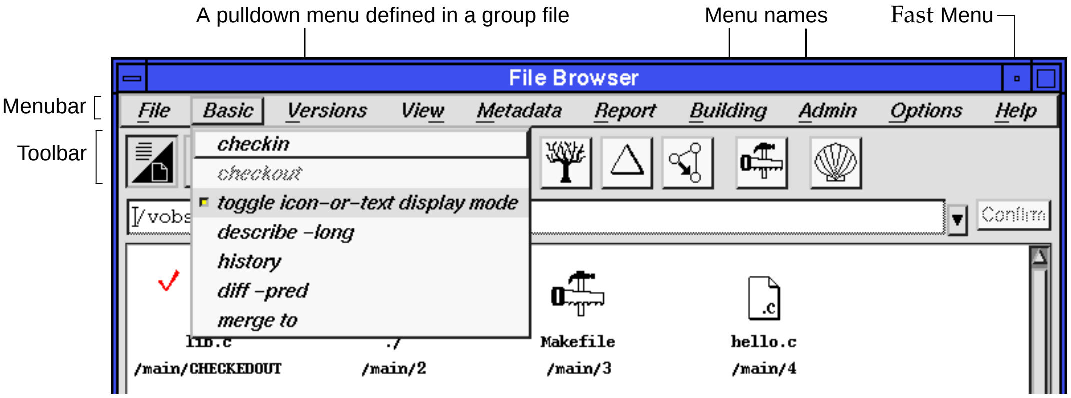
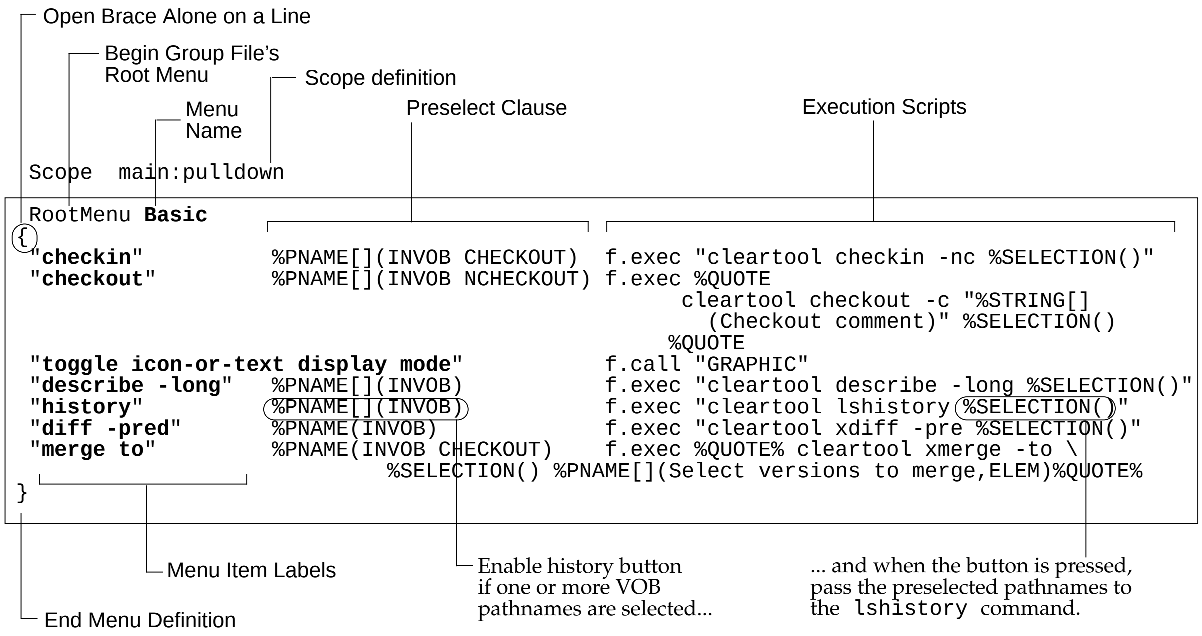


Figure 18-1 A File Browser and Some Menus


Open Brace Alone on a Line
Begin Group File's Root Menu
Scope definition
Menu Name
Preselect Clause
Execution Scripts
Menu Item Labels
End Menu Definition
Enable history button if one or more VOB pathnames are selected...
... and when the button is pressed, pass the preselected pathnames to the lshistory command.

```
 Scope  main:pulldown

 RootMenu Basic
{
 "checkin"          %PNAME[](INVOB CHECKOUT)  f.exec "cleartool checkin -nc %SELECTION()"
 "checkout"         %PNAME[](INVOB NCHECKOUT) f.exec %QUOTE
                                                    cleartool checkout -c "%STRING[]
                                                       (Checkout comment)" %SELECTION()
                                                   %QUOTE
 "toggle icon-or-text display mode"           f.call "GRAPHIC"
 "describe -long"   %PNAME[](INVOB)           f.exec "cleartool describe -long %SELECTION()"
 "history"          %PNAME[](INVOB)           f.exec "cleartool lshistory %SELECTION()"
 "diff -pred"       %PNAME(INVOB)             f.exec "cleartool xdiff -pre %SELECTION()"
 "merge to"         %PNAME(INVOB CHECKOUT)    f.exec %QUOTE% cleartool xmerge -to \
                             %SELECTION() %PNAME[](Select versions to merge,ELEM)%QUOTE%
}
```

Figure 18-2 A Sample Group File, *basic.grp*

Although the *basic.grp* file and Basic menu are fictitious, they illustrate the structure and function of group files.

Syntax Summary

You may have noticed similarities between group file syntax and that of X window manager startup files like *.mwmrc* or *.twmrc*. As in a *.mwmrc* file, the ! and # characters can begin comment lines, and an unquoted, un-escaped # in any column comments the rest of the line. The backslash (\) escape character doubles as the line continuation character.

The file begins with a *Scope* definition. (Scope is described below.) Each group file defines exactly one primary menu and names it with a *RootMenu* declaration. The *RootMenu* declaration is followed immediately by a line containing only an open brace character ({). Submenus, if there are any, are named with Menu declarations.

Each menu item definition is a *single logical text line* that includes, in order:

- a label
- optional bitmap files (for toolbar items)
- an optional menu mnemonic
- an optional preselect clause
- an optional ampersand (&), to run a menu item's command operation in the background
- one or more *f-dot* functions — usually an execution string
- a "popup" help text string

The menu declaration ends with a closed brace character (}).

You may also have observed that the execution scripts, or *f.exec* strings, in *basic.grp* contain unfamiliar uppercase keywords, prefixed with the % character. These are ClearCase *macros*, inserted to perform specialized tasks, like prompting the user for more data or redirecting output to a browser. An *input macro*, like %pname, can serve two functions: (1) to prompt for interactive input, and (2) as a preselect clause, which specifies the conditions necessary to activate the menu item.

Figure 18-3 restates these rules.

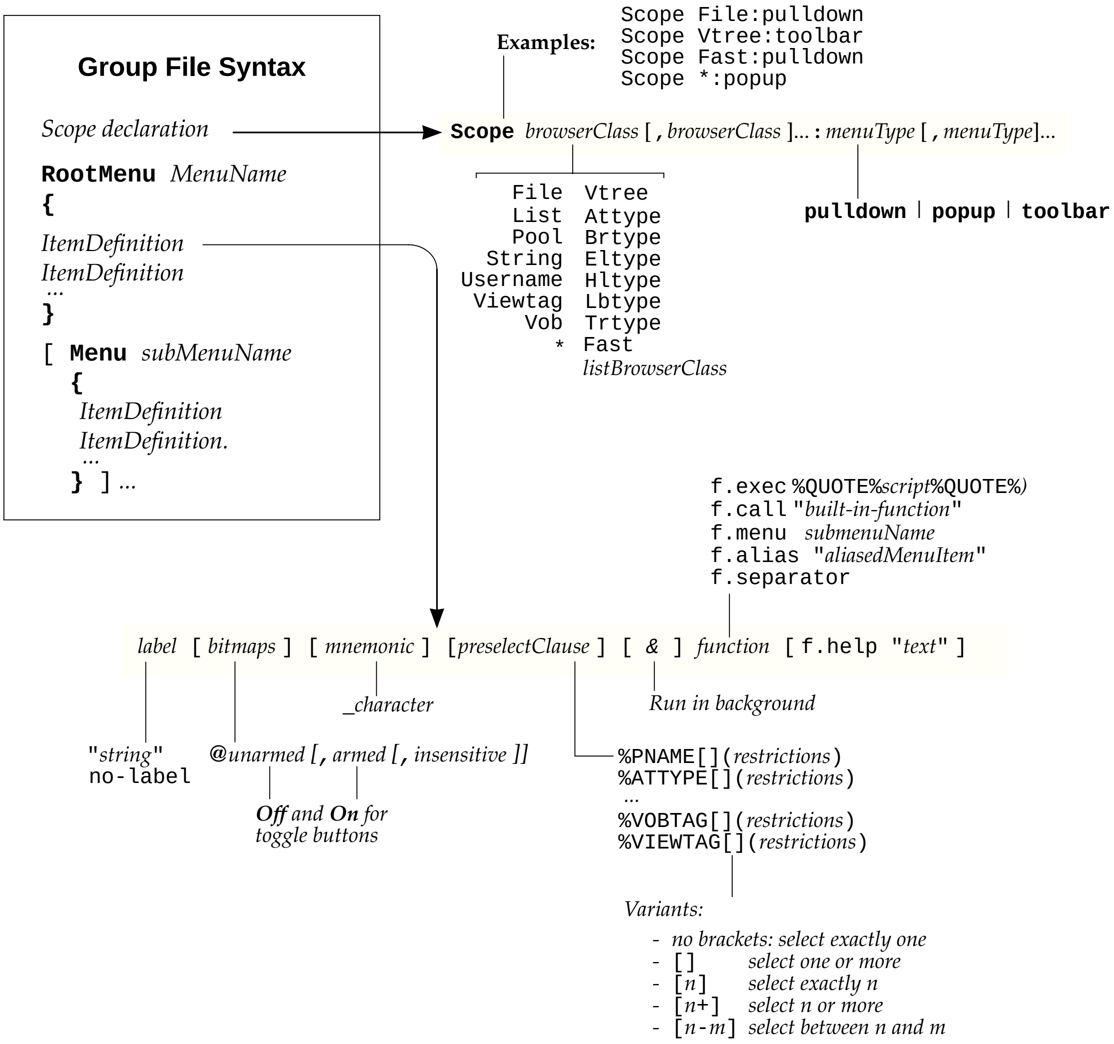


Figure 18-3 Group File Syntax

The following subsections describe first the *Scope* and *RootMenu* declarations that begin a group file, and then, the individual components of an item definition.

Scope

```
Scope browserClass [, browserClass ]... : menuType [ , menuType]...
```

Each menu — pulldown, popup, or toolbar — corresponds to a *.grp* file. The menu definitions (*.grp* files) all have the same structure and syntax, but they have different *scopes*. A menu can have one of four basic scopes:

Fast:pulldown
The menu is accessible from the Fast pulldown menu on the main panel. Menus with this scope can be duplicated (at runtime) as a *static menu* of buttons below the iconic toolbar.

browserClass:**pulldown**
The group appears as a stand-alone pulldown menu on any browser of class *browser* (file, *xx*type, vtree, and so on).

browserClass:**popup**
The group appears as a popup menu when the user presses *rightMouse* on any browser of class *browser*. You can scope only one menu as any browser's pop-up (the first one found wins).

browserClass:**toolbar**
The group appears as a toolbar on any browser of class *browser*. You can scope only one menu to any browser toolbar (the first one found wins).

The *browserClass* is one of the following as shown in Figure 18-4:

```
*
Fast
File
List
Pool
String
Username
Viewtag            First letter can be upper-case
Vob                or lower-case
Vtree
Attype
Brtype
Eltype
Hltype
Lbtype
Trtype
listBrowserClass   see %list, %listout macros
```

Figure 18-4 *browserClass* Listing

The *menuType* is one of:

```
pulldown
popup
toolbar
```

If a menu has no scope specifier, it defaults to *:pulldown and, therefore, appears as a pulldown menu on all browsers.

For example, the ClearCase-supplied group file *file_popup.grp* begins with the following line, which defines it as the popup menu for all file browsers:

```
Scope File:popup
```

Popup and Toolbar Scopes

For any browser class, only one menu can be explicitly scoped to the toolbar or pop-up menu. (See “Fast Menu Scope” on page 239 to learn how to create a pull-down menu that can be appended to the toolbar.) If you choose to replace a browser’s toolbar or pop-up menu, follow the procedure described in “Replacing an Existing Menu” on page 290. If you depart from this procedure, and use a group file with a different name than the standard menu, make sure the new file name sorts ahead of the existing one; *xclearcase* ASCII-sorts group files when it first reads them in.

Fast Menu Scope

The *Fast* scope is compatible only with menu type *pulldown*; Fast:popup and Fast:toolbar scopes have no effect.

The Fast menu appears only on the file browser that comes up when you execute *xclearcase*. It does not appear at all if no menu is scoped to it. If you close the original file browser, the Fast menu is lost for the duration of the *xclearcase* session (even if you start a new file browser with the Filet -> New file browser option on another browser).

The Fast menu’s distinguishing feature is the Static menu option that is prepended to each menu scoped there. You can set the Static menu option to duplicate a menu as a set of “fast access” buttons below the file browser’s iconic toolbar. (You could scope your “fast access” menu to File:toolbar, but since only one group file can be scoped to a browser’s toolbar, the standard iconic toolbar would be displaced.)

Typical usage — “Pick and choose” a set of commonly used menu items from elsewhere in the interface, and collect them together as a “fast access” menu scoped to Fast:pulldown. When constructing the “fast access” menu, use the *f.alias* function to clone menu items that reside elsewhere in the interface.

Figure 18-5, Figure 18-6, Figure 18-7, and Figure 18-8 illustrate the results of scoping group files, respectively, to File:pulldown, File:toolbar, File:popup, and Fast:pulldown.

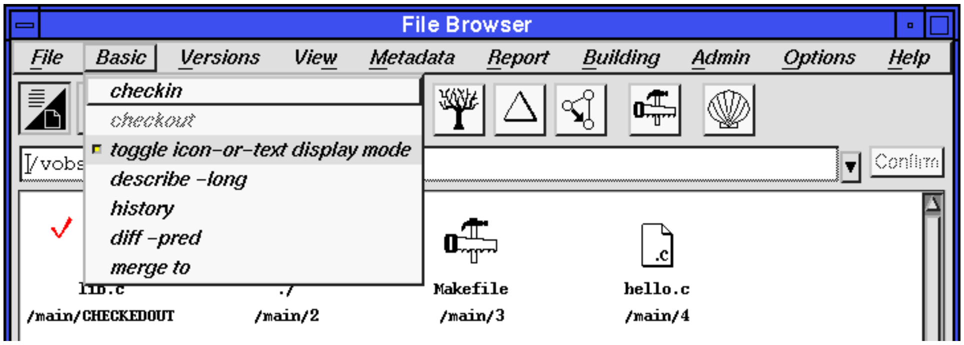


Figure 18-5 Basic Menu Scoped to File:pulldown

Note that it is the file name, not the menu name, that determines where in the menubar the View menu appears.

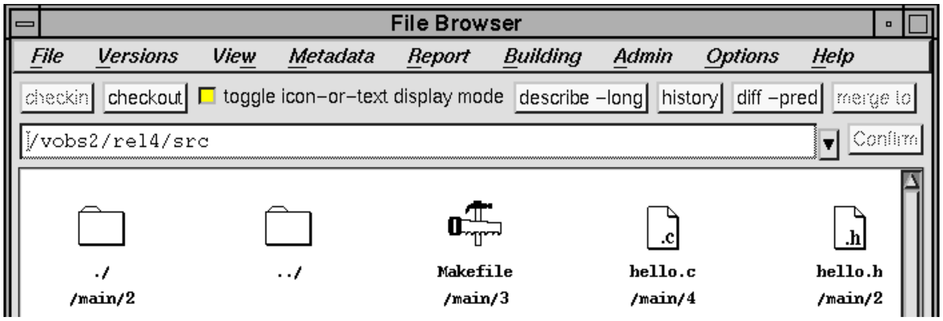


Figure 18-6 Basic Menu Scoped to File:toolbar

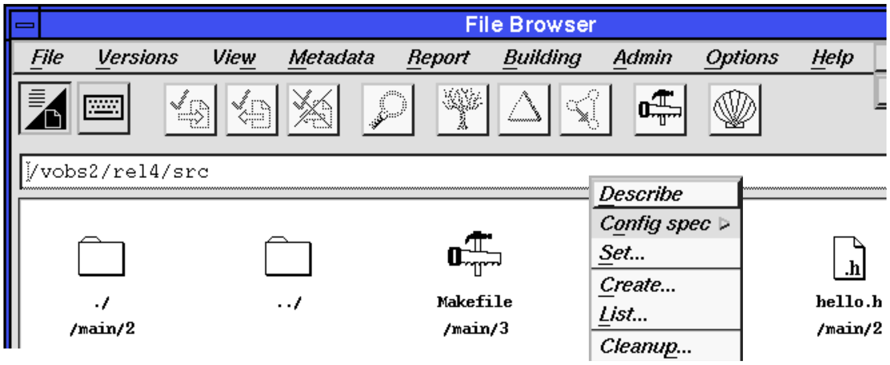


Figure 18-7 View Menu Scoped to File:popup

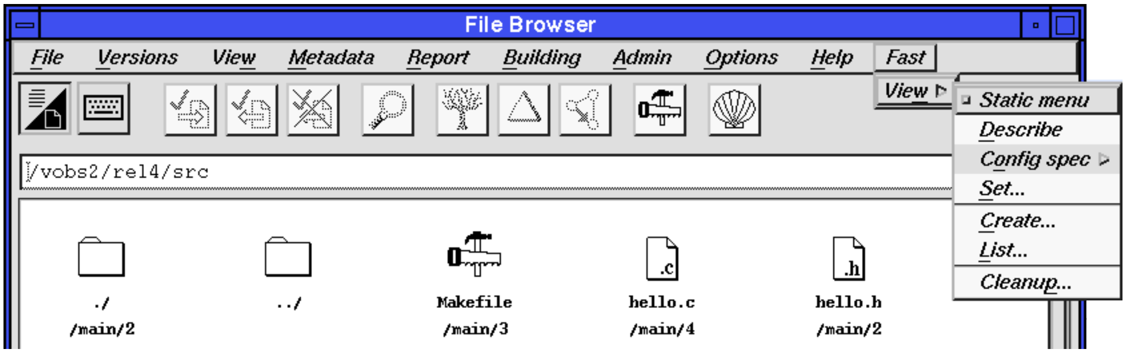


Figure 18-8 View Menu Scoped to Fast:pulldown

RootMenu Name

`RootMenu` *menuName* [*_mnemonicChar*]

For menus scoped pulldown, the *RootMenu* name is the character string that actually appears in the interface. The *RootMenu* name has no effect on menu location; that is determined by sorting the names of the group files themselves (See "How xclearcase Processes Group Files at Startup Time" on page 232.) See also "Menu Mnemonics" on page 243 for details on specifying an optional menu mnemonic character.

Item Labels

label [*bitmaps*] [*mnemonic*] [*preselectClause*] [&] *function* [f.help "*text*"]

```
"label string" or no-label
```

Item labels are quoted character strings (labels without blanks do not require quotes, but they are recommended for consistency). If the item is scoped to a toolbar, and its definition includes bitmaps, then the item label is not visible in the interface. In all other cases, this is a user-visible label for the menu item. Use no-label to suppress a label — with *f.separator*, in particular.

Bitmaps

label [***bitmaps***] [*mnemonic*] [*preselectClause*] [&] *function* [f.help "*text*"]

@unarmedFilename[,*armedFilename*[,*insensitiveFilename*]]

You can specify bitmap icons for *RootMenu* menu items scoped to a toolbar. Bitmaps for submenu items, or for items not scoped to a toolbar, are ignored.

The first two bitmaps represent the “unarmed” and “armed” (depressed) states. These two states correspond to “off” and “on” for two-state toggle buttons. (See the builtin function descriptions in section “f.call” on page 247.) The third bitmap is displayed when the item is insensitive, or “greyed out”. A toolbar button with a preselect clause needs all three bitmaps.

Bitmaps supplied in item definitions must be stored in */usr/atria/config/ui/bitmaps*; otherwise, they must be full pathnames (including suffixes). Do not confuse toolbar bitmaps with file browser icons, which are managed as described in *cc.icon* (and in the section “Icon Display in the File Browser” on page 292).

Menu Mnemonics

```
label [ bitmaps ] [ mnemonic ] [ preselectClause ] [ & ] function [ f.help "text" ]
```

```
_character
```

The underscore character is required; *character* is a single, printable character, which must occur in the label string. The first occurrence of *character* in the label string is underlined — but only if the character's letter case (upper or lower) matches the case in which it appears in the label string.

Menu mnemonics enable keyboard access to menus and menu items. See Table 3-2 in Chapter 3, "Using the ClearCase Graphical User Interface," for a description of how menu mnemonics are used to navigate menus.

A mnemonic character has no meaning for an item or menu scoped to a toolbar.

Preselect Clauses

```
label [ bitmaps ] [ mnemonic ] [preselectClause ] [ & ] function [ f.help "text" ]
```

```
% dataType[preselect-count](restrictions)
```

The *preselect clause* defines the conditions necessary to activate the menu item. The item is "greyed out" until the user selects at least one object that satisfies the preselect clause. If there is no preselect clause, the button or menu option is always enabled.

- A preselect clause has three parts:
- a single data type (or input macro) keyword; see "Input Macros" on page 262 for a information on how to construct this piece of a preselect clause.

- an optional “how many?” specifier, inside square brackets, which specifies the number of applicable data items the user must preselect in order to enable the menu item; some representative examples:

none	user must preselect exactly one item
[]	same as [1+]; user must preselect one or more items
[2]	user must preselect two items
[2+]	user must preselect two or more items
[2-4]	user must preselect between two and four items, inclusive
[0+]	no preselection requirement
[0]	no preselection permitted (item not enabled if any data is selected)

- a parenthesized list of restrictions (see the *restrictions* arguments to the various *input macros* in the section “Input Macros”). The actual restrictions are optional; the parentheses are not.

Preselect clauses are closely linked to the %selection macro. If an item definition includes a preselect clause, its execution script includes at least one %selection macro, which is replaced by the preselected data at execution time.

You cannot have the user preselect multiple data types (three pathnames and two label types, for example). If your menu item operation requires multiple kinds of data, or it requires data items in a particular sequence, you can get only one type of data, or one argument, via preselection. You must use one or more additional mechanisms to collect the remaining input; *xclearcase* input macros and *clearprompt* exist for this purpose.

Background or Foreground Processing

```
label [ bitmaps ] [ mnemonic ] [ preselectClause ] [ & ] function [ f.help "text" ]
```

&

Include an ampersand character to have the menu operation execute in the background. When a script executes in foreground mode (no &, the default), the watch cursor blocks other commands until the currently executing *f.exec* script (or *f.call*) is complete.

For backgrounded item definitions, all *f.exec* scripts except the last one behave like foreground scripts. The last (or only) *f.exec* script runs in the background. Any *f.call* functions that follow the last *f.exec* run in the foreground, without waiting for the *f.exec* to complete.

F-dot Functions

```
label [ bitmaps ] [ mnemonic ] [ preselectClause ] [ & ] function [ f.help "text" ]
```

Each item definition includes at least one of the following functions in Table 18-1.

Table 18-1 Group Item Functions

F-dot Function	Argument	Description
f.alias	*aliasedItem*	Duplicate an existing menu item.
f.call	*builtinCall*	Call a built-in function.
f.exec	*execString*	Execute the command or shell script when the button is pressed or menu option selected.
f.separator	none	Insert a dividing line in the group menu.
f.menu	*submenuName*	Specify a submenu (“cascading” menu).
f.help	*helpString*	Specify help text for item.

Quoting — The argument to any *f-dot* function can be quoted with either %QUOTE% keywords or quote characters (""). %QUOTE % keywords permit unescaped quote characters (") to appear in the argument. See also “Group File Processing and Macro Expansion” on page 254.

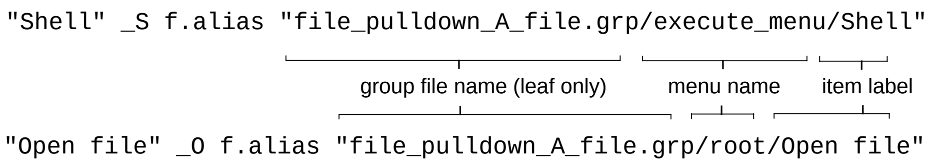
f.alias

```
f.alias "groupFileLeafName/menuSpecifier/itemLabel"
```

Duplicates an existing menu item from another group file. An “aliased” item inherits the following characteristics from an existing menu item definition:

- preselect clause
- foreground/background execution specifier
- all *f-dot* functions (*f.calls, f.execs, f.menu, f.help, f.title, f.alias*)

example as shown in Figure 18-9 (from *file_popup.grp)*



```
"Shell" _S f.alias "file_pulldown_A_file.grp/execute_menu/Shell"
"Open file" _O f.alias "file_pulldown_A_file.grp/root/Open file"
```

Figure 18-9 *f.alias* Example

Note: The menu name *root* in the second example. Use *root* if the menu item is defined in the target group file’s *RootMenu*.

You cannot override *f-dot* functions in the alias definition. Also, an *f.alias* function cannot be combined with other *f-dot* functions.

f.call

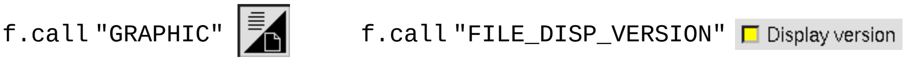
```
f.call "builtinFunction"
```

Calls one of the built-in *xclearcase* functions. Many of these calls toggle browser display parameters. (See */usr/atria/config/ui/grp/file_pulldown_G_options.grp*, for example).

There are also calls to invoke or “pop up” each of the *xclearcase* browsers. Others perform miscellaneous operations, like closing a browser or quitting *xclearcase*.

If the *builtinFunction* call includes an argument with literal quotes, quote the call with %QUOTE% keywords, instead of quote characters (").

Some of the *f.call* functions create two-state toggle buttons (rather than “push buttons”) automatically. Figure 18-10 shows iconic and textual examples of two-state toggle buttons



```
f.call "GRAPHIC"          f.call "FILE_DISP_VERSION"
```

Figure 18-10 Two-state Toggle Buttons (Iconic and Textual)

These builtin calls exist to let you position toggle buttons using the menu scope of your choice. Note that you do not get default graphical icons if you scope a toggle item to a toolbar; supply them in the menu item definition like the ClearCase-supplied group files do.

A “customized toggle button” is any toggle button *you* create with an *f.call* whose description in Table 18-3 begins “Enable/disable”. Adhere to the following guidelines to guarantee that the state of a customized toggle button is always predictable:

- In a single item definition, do not combine a toggle *f.call* with any other *f.call*s or *f.exec*s. You may be tempted to try and control multiple parameters from a single menu item. However, because *xclearcase* frequently adjusts these parameters internally in response to other user operations, the state of a toggle button that controls multiple parameters may become inaccurate with respect to one or more parameters.
- Do not create two toggle buttons for the same parameter on the same browser, as their states are likely to become confused.

Table 18-2 documents the built-in calls.

Table 18-2 Built-in Calls

Call[a]	**Description**[b]	**Applicable Browsers**
XXTYPE *pname*	Bring up type browser for VOB identified by *pname*.	all
CLOSE	Close current browser.	all
FILE *directory-pname*	Bring up file browser on *directory-pname*.	all
FILE_DISP_DATE	Enable/disable size and date modified display.	file
FILE_DISP_OWNER	Enable/disable owner and permissions display.	file
FILE_DISP_RULE	Enable/disable config spec rule display.	file
FILE_DISP_TYPE	Enable/disable object type information display.	file
FILE_DISP_VERSION	Enable/disable version information display.	file
FILE_SORT_BY_CHECKOUT	Enable (disable by-rule and by-type)/disable sort-by-checkout for file system objects in file browser.	file
FILE_SORT_BY_RULE	Enable (disable by-checkout and by-type)/disable sort-by-rule for file system objects in file browser.	file
FILE_SORT_BY_TIME	Enable/disable “minor”[c] sort-by-time.	file
FILE_SORT_BY_TYPE	Enable (disable by-rule and by-checkout)/disable sort-by-type for file system objects in file browser.	file
FORCE	Force update of current browser.	all
GRAPHIC	Disable/enable graphic (icon) display mode.	file, vtree

Table 18-2 (continued) Built-in Calls

Call[a]	**Description**[b]	**Applicable Browsers**
KEYBOARD	Enable/disable the keyboard input box for the current browser.	file, *xx*type, pool, VOB-tag, view-tag
LIST_DELETE_SELECTED	Delete selected items from the current list browser.	list
POOL *pname*	Bring up pool browser on VOB identified by *pname*.	all
print *output-pname* [*pagesize*] [*scale*]	Send PostScript image of vtree to *output-pname.* *pagesize* := a0 ǀ a1 ǀ a2 ǀ a3 ǀ a4 ǀ a5 ǀ **letter** ǀ legal *scale* := **SCALE_TO_PAGE** ǀ *percentScalingFactor* *percentScalingFactor* := an integer; default = 100	vtree
QUIT	Quit *xclearcase*.	all
SETVIEW *view-tag*	Set the process's current working view to *view-tag*.	all
SHOWTRANS	Bring up the transcript pad.	all
TYPE_DISP_ACTIVE	Enable/disable display of active objects.	*xx*type
TYPE_DISP_LOCKED	Enable/disable display of locked objects.	*xx*type
TYPE_DISP_OBSOLETE	Enable/disable display of locked-obsolete objects.	*xx*type
UPDATE	Update any browsers marked for update. By default, browsers are updated when a GUI command completes, or whenever 15 seconds elapse without any GUI activity.	all
USERNAME	Bring up username browser.	all
VIEWTAG	Bring up view-tag browser.	all
VOBTAG	Bring up VOB-tag browser.	all
VTREE *element-pname*	Bring up vtree browser on *element-pname.*	all
VTREE_DISP_ALL_LABEL	Enable/disable all labels display (disabled: max=5).	vtree
VTREE_DISP_ALL_VER	Enable/disable display of all versions (disabled: show "significant" versions only — branch points, labeled versions, and hyperlink endpoints).	vtree

Table 18-2 (continued) Built-in Calls

Call[a]	Description[b]	Applicable Browsers
VTREE_DISP_CHECKOUT	Enable/disable display of checked-out versions.	vtree
VTREE_DISP_MERGE	Enable/disable merge arrow display.	vtree

a. Arguments are required, unless enclosed in brackets ([]).

b. For two-state toggle functions, the first word in an "enable/disable" pair specifies the state that corresponds to "on" or "armed".

c. A "minor" sort parameter applies only to data items within the subsets created by the "major" parameters.

While *xclearcase* macros are most common in *f.exec* strings, they can appear in *f.call* arguments as well. Here is a common example:

```
f.call "VTREE %SELECTION()"
```

See the sections "Group File Processing and Macro Expansion" on page 254 and "xclearcase Macros" on page 258 for details on *xclearcase* macros.

f.exec

```
f.exec %QUOTE%execString%QUOTE%
```

Executes a command line or command script. The argument to *f.exec* is an executable command or shell script, enclosed either by double-quotes (" ") or by %quote% keywords.

By default, */bin/sh* is used. Include a first line like #!/bin/csh to specify another shell.

In a single item definition, you can combine an *f.exec* script with one or more additional *f.call*s and/or *f.exec*s. However, if an *f.exec* script returns a non-zero exit status, *f-dot* processing terminates for the current operation.

If the *execString* includes an argument with literal quotes, quote the *execString* with %QUOTE% keywords, instead of double-quote characters ("). In general, it is highly recommended that you use %QUOTE% for complicated scripts.

Because *xclearcase* adds *macro expansion* to *f.exec* string processing, quoting and character escaping are different than in ordinary scripts. For details on macro expansion, and on using the specific macro themselves, see the sections “Group File Processing and Macro Expansion” on page 254 and “xclearcase Macros” on page 258.

Environment Variables. If you set the environment variable clearcase_dbg_grp to a non-zero value, *xclearcase* sends debugging information to the transcript pad when executing commands.

If they are not already set, xclearcase sets the atriahome and grp_path environment variables to their default values when it starts up. (atriahome defaults to */usr/atria*; grp_path defaults to $home/*.grp*:$atriahome/*config/ui/grp*.) This means you can rely on the existence of these variables when writing scripts.

f.separator

```
no-label   f.separator
```

Inserts a separator line in a menu, or breaks the horizontal run of a toolbar button set. As for any item definition, a label is required; use the keyword no-label instead of a quoted label string.

f.menu

```
f.menu submenuName
```

Specifies a submenu to include in the current menu or submenu (no effect for a menu scoped to a toolbar). The *submenuName* specifies a menu that is named in a *Menu* declaration in the same group file.

Figure 18-11 shows a “cascading” menu — the ClearCase menu, and its submenu Checkouts. Figure 18-12 shows an abbreviated version of the group file that defines the *ClearCase* menu.

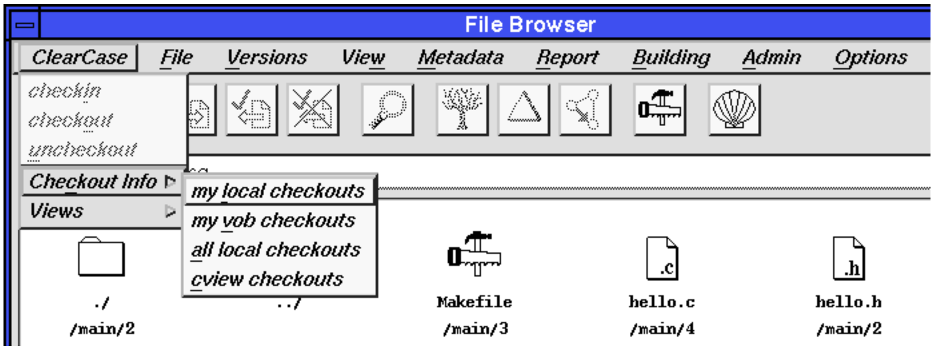
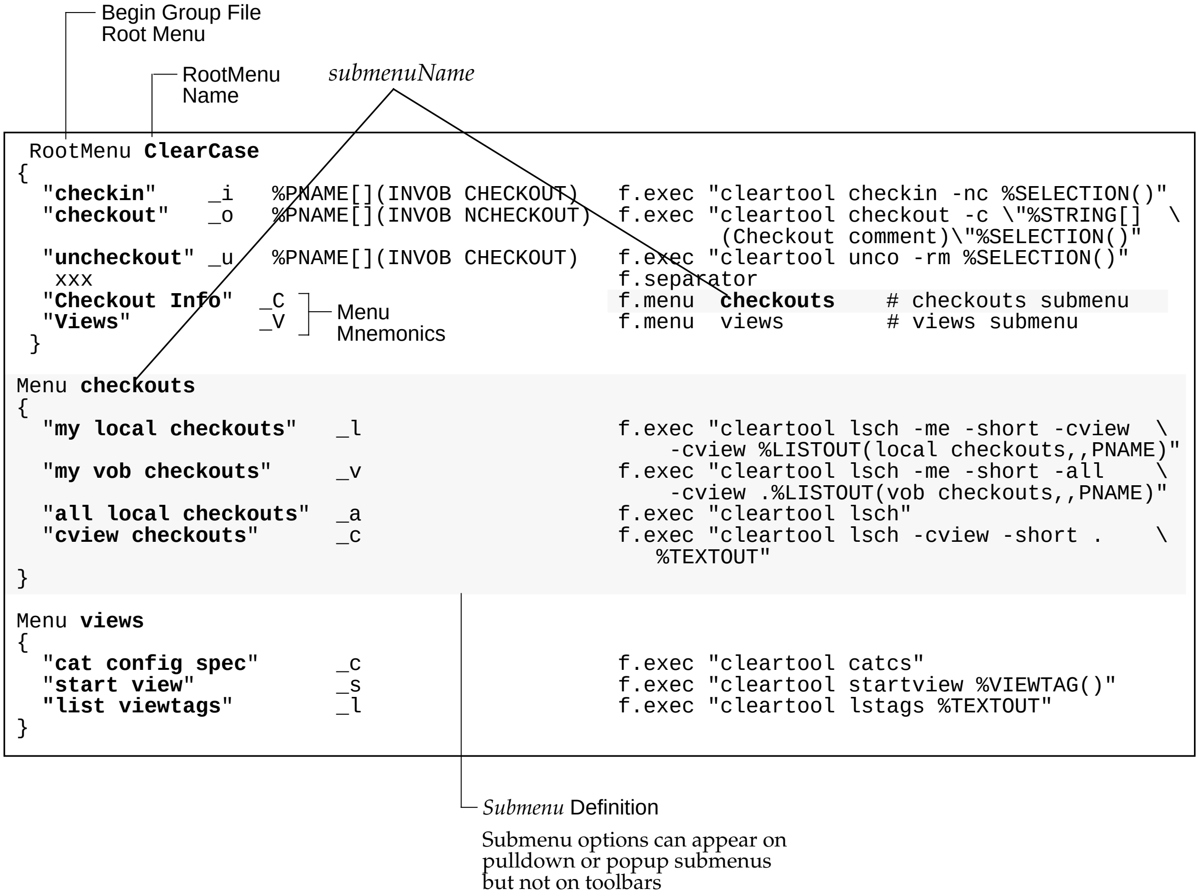


Figure 18-11 Cascading Menu



```
 RootMenu ClearCase
{
  "checkin"    _i   %PNAME[](INVOB CHECKOUT)   f.exec "cleartool checkin -nc %SELECTION()"
  "checkout"   _o   %PNAME[](INVOB NCHECKOUT)  f.exec "cleartool checkout -c \"%STRING[]  \
                                                       (Checkout comment)\"%SELECTION()"
  "uncheckout" _u   %PNAME[](INVOB CHECKOUT)   f.exec "cleartool unco -rm %SELECTION()"
   xxx                                         f.separator
  "Checkout Info"  _C                          f.menu  checkouts    # checkouts submenu
  "Views"          _V                          f.menu  views        # views submenu
 }

Menu checkouts
{
  "my local checkouts"   _l                    f.exec "cleartool lsch -me -short -cview  \
                                                   -cview %LISTOUT(local checkouts,,PNAME)"
  "my vob checkouts"     _v                    f.exec "cleartool lsch -me -short -all    \
                                                   -cview .%LISTOUT(vob checkouts,,PNAME)"
  "all local checkouts"  _a                    f.exec "cleartool lsch"
  "cview checkouts"      _c                    f.exec "cleartool lsch -cview -short .    \
                                                  %TEXTOUT"
}

Menu views
{
  "cat config spec"      _c                    f.exec "cleartool catcs"
  "start view"           _s                    f.exec "cleartool startview %VIEWTAG()"
  "list viewtags"        _l                    f.exec "cleartool lstags %TEXTOUT"
}
```

Figure 18-12 Group File with Cascading Menus

Note: The submenu name, referenced in the *f.menu* function and in the submenu definition, is not the label that appears in the interface. As with all item definitions, the quoted label string, *Checkout Info* in this case, is what the user actually sees.

f.help

```
label [ bitmaps ] [ mnemonic ] [ preselectClause ] [ & ] function [ f.help "text" ]
```

Specifies help text, which is displayed in a popup window when the user presses *rightMouse* over the menu item (if it is enabled). It is good practice to supply popup help text for all items that include *f.call* or *f.exec* functions. There is no need to supply help for *f.menu* declarations; users cannot access help text on non-command menu items.

When supplying help text, keep each line under approximately 70 characters, to prevent lines from wrapping in the default-sized “popup help” display window.

If the *helpText* includes literal quotes, quote the *execString* with %QUOTE% keywords, instead of quote characters (").

Group File Processing and Macro Expansion

Command script quoting and special character escaping are somewhat different than in standard scripts due to the addition of *xclearcase* macros (see also “xclearcase Macros” on page 258). Three passes are made over your *f.exec* scripts, two by *xclearcase* and one by the unix shell:

1. When *xclearcase* starts up, it scans all group files to build the menu interface.
2. When the user selects a menu item, *xclearcase* expands macros in the item’s execution script. (This discussion focuses on *f.exec* scripts, but applies as well to the less common *f.call* functions.)
3. Once *xclearcase* macros have been replaced with appropriate strings, the command, or command script, is passed to the unix shell, which performs its own substitutions and executes the commands.

If an item definition includes multiple *f.exec* scripts, each is evaluated and executed separately, first to last.

The template file */usr/atria/config/ui/grp/user.grp.template* illustrates many of the quoting, character escaping, and line continuation rules described in this section.

Pass 1: Scan Group Files

When *xclearcase* starts up, it scans the group files, checks syntax, and configures the various menus.

%quote%

The execution script syntax analysis is simplified if you use %quote% keywords, instead of quote characters (`"`), to enclose your *f.exec* scripts. Using %quote% eliminates the need to escape backslash characters (`\\n`, for example) and quote characters (`\"`) in scripts. Inside %quote% delimiters, there is no escape character, and no need for one.

xclearcase Line Continuation

Recall that an item definition is a single logical line. Therefore, a newline (<nl>) must be escaped with the backslash “line continuation character”, unless it falls inside quotes (", ’, or %QUOTE%).

Note: Continue to use line continuation characters in execution scripts as required by the unix shell.

The *xclearcase* Escape Character — %

The percent character (%) is the xclearcase macro escape character. It is used to prefix macros, and to escape literal characters that might confuse macro processing — % (percent character) and) (close parenthesis), in particular. If you want a % character to appear as a literal anywhere in an item definition (label string, prompt argument to an input macro, etc.) you must escape it, like this: %%.

Pass 2: Macro Expansion.

Macro expansion — string substitution — occurs when user selects the menu item. In the case of *input macros*, macro expansion includes prompting for user input.

During this pass, *xclearcase* is looking for:

- the macro escape character, **%**, which prefixes all *xclearcase* macros
- after any macro keyword, the parentheses **( )** that enclose macro arguments
- within parentheses, the commas that separate macro arguments

If you desire that **%**, **(**, or **)** characters survive this pass (to appear in a macro prompt string, for example), escape them with the **%** character, like this: **%% %(%)**.

If a macro argument includes a comma, quote the argument. (All macro arguments are strings, though only a few require explicit quotes.)

Macros can appear anywhere within the quoted *f.exec* string, and they can be nested as well. Macros are evaluated left-to-right; nested macros are evaluated from innermost to outermost.

Pass 3: Script Execution

Once macro expansion is complete, the unix shell makes pass 3, performing its own standard substitutions and executing the script. By default, */bin/sh* is used to execute your *f.exec* scripts. Include a first line like #!/bin/csh to specify another shell.

Figure 18-13 follows an item definition through all three passes, illustrating how the execution string and its *xclearcase* macros are processed.

Button Definition

continue item definition line for *xclearcase*

```
"checkout" _o %PNAME[](ELEM NCHECKOUT) \
   f.exec %QUOTE
          cleartool co -c "%STRING[](Checkout comment)" \
          %SELECTION()
          %QUOTE% \
   f.help "Checkout the selected elements."
```

continue command line for */bin/sh*

↓

Pass 1 — Scan Group File

When you start *xclearcase*, it reads and caches the item label, mnemonic, pre-select clause, and execution string, looking for syntax errors. It removes whatever kind of quotes enclose the executaion (*f.exec*) string. Here is the resulting one-line execution script:

```
cleartool co -c "%STRING[](Checkout comment)" \
%SELECTION()
```

↓

Pass 2 — Macro Expansion

User presses the checkout button, and *xclearcase* prepares for script execution by expanding macros, left to right (and innermost to outermost, for nested macros). During macro expansion, *xclearcase* first encounters the %string macro, which prompts for a checkout comment. Until the user presses Ok or Cancel (which aborts the entire operation), macro processing is suspended. Suppose the user supplies the comment, `Fix Bug 273-2`. Now the command line looks like this:

```
cleartool co -c "Fix Bug 273-2" \
%SELECTION()
```

Moving left to right, *xclearcase* encounters the %selection macro. Because the button was active, we know there was at least one pre-selected pathname satisfying the conditions `%PNAME[](ELEM NCHECKOUT)`. Assume the preselection was the single file `/usr/src/file2.c`. *xclearcase* substitutes this for %selection and macro expansion is complete:

```
cleartool co -c "Fix Bug 273-2" \
/usr/src/file2.c
```

↓

Pass 3 — Shell Script Execution

The command shell reads and executes the command normally (including handling the line continuation character).

Figure 18-13 How an Item Definition's *f.exec* String is Processed

xclearcase Macros

We have made numerous references to *xclearcase* macros. This section describes them more precisely and includes a set of tables to use as a reference.

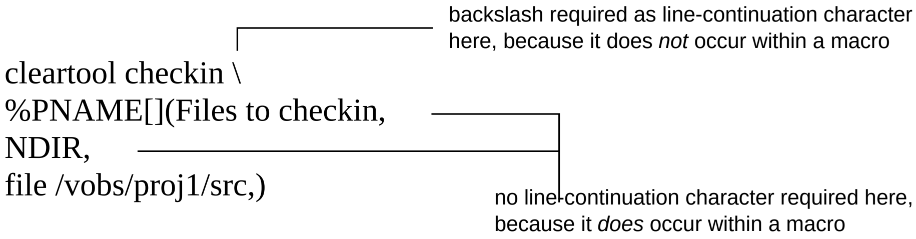
xclearcase macros provide a specialized but straightforward string-replacement mechanism. The overall syntax rules follow:

- **Nesting —** Macros can be nested to 32 levels, which are evaluated from innermost to outermost.
- **Arguments —** Character string arguments for prompts and titles need not be enclosed in quotes:

  ```
  %PNAME(Source file?,NELEM,file,ENABLE)
  ```

  This example specifies as a prompt the character string “Source file?”. Titles of browsers can be arbitrary character strings; in particular, they can include < SPACE > characters.

  Extra white space is allowed between the arguments of a macro, even < NL > characters. Also see Figure 18-14:



```
cleartool checkin \
%PNAME[](Files to checkin,
NDIR,
file /vobs/proj1/src,)
```

backslash required as line-continuation character here, because it does *not* occur within a macro

no line-continuation character required here, because it *does* occur within a macro

Figure 18-14 Character String Examples

- **Parentheses** — Macro arguments must be enclosed in parentheses. If you omit one or more arguments, you must use commas to indicate the "null" argument(s):

  ```
  %PNAME(Source file? ,NELEM,file,ENABLE)
                                  (Four explicit arguments)
  %PNAME(Source file? ,,,ENABLE)
                        (Four arguments, two of them "null")
  ```

  Even if a macro takes no arguments, you must include the parentheses:

  ```
  %SELECTION()              (Macro that takes no arguments)
  ```

- **Square brackets** — With *input macros* only, you can use square brackets to collect multiple data input items. See "Input Macros" on page 262.

Table 18-3 lists all the *xclearcase* macros. The following sections describe the *xclearcase* macros in detail.

Table 18-3 *xclearcase* Macros

Macro	Brief Description
Input Macros	
%ATTYPE[]()	Prompt for one or more attribute types
%BRTYPE[]()	Prompt for one or more branch types
%ELTYPE[]()	Prompt for one or more element types
%HLTYPE[]()	Prompt for one or more hyperlink types
%LBTYPE[]()	Prompt for one or more label types
%TRTYPE[]()	Prompt for one or more trigger types
%HYPERLINK[]()	Prompt for one or more hyperlink objects
%LIST[]()	Prompt for one or more list browser entries
%PNAME[]()	Prompt for one or more pathnames
%POOL[]()	Prompt for one or more VOB storage pools
%STRING[]()	Prompt for one or more character strings
%USERNAME[]()	Prompt for one or more user IDs
%VIEWTAG[]()	Prompt for one or more view-tags
%VOBTAG[]()	Prompt for one or more VOB-tags
Modifier Macros	
%ELEMENT()	Strip version IDs from pathnames
%ELEMSUFFIX()	Add extended naming symbol (@@) to pathnames
%MOUNT()	Convert VOB pathnames to VOB-tags (mount points)
%RELATIVE()	Convert names to relative pathnames
%SETVIEW()	Convert pathnames to view-extended pathnames

Table 18-3 (continued) *xclearcase* Macros

Macro	Brief Description
%SORT()	Return sorted list of pathnames
%VERMOD()	Convert pathnames to version IDs (branch/version)
%WHICH()	Return filename's full pathname from search path
%WILD()	Expand wildcard characters in a pathname argument
Output Macros	
%LISTOUT()	Redirect output to a list browser
%TEXTOUT()	Redirect output to a read-only text window
Memory Macros	
%SAVE()	Save a string in a named register for later recall
%REMOVE()	Remove named register storage
%RESTORE()	Restore a saved string from a named register
Miscellaneous Macros	
%SELECTION()	Insert pre-selected data in *f.exec* string
%TMPFILE()	Create a file in */usr/tmp* (or in $tmpdir)
%DIR()	Return directory associated with current browser
%NAME()	Return name associated with current browser

Input Macros

Input macros collect input from the user after he or she selects the menu item. Unlike the *f.call*s (vtree, file, vobtag, and so on) that launch new browsers, an input macro displays a text string prompt, and it does not "return" until the user selects data and presses Ok or aborts the operation with Cancel. Although the user can "interrupt" and perform other operations, the prompting browser waits until its original prompt is satisfied.

An input macro's prompt can appear in multiple browsers. For example, if the user has two file browsers up, a %pname macro's prompt will appear in both, as either is capable of returning the required data. Unlike its *f.call* counterpart, an input macro does not start a new browser if an appropriate one is already displayed.

Input Macros as Preselect Clauses

Input macros, in abbreviated form, double as the preselect clauses in item definitions. The macro tables that follow below document both uses; in particular, they include the various *restriction* parameters, which apply when composing both preselect clauses and *f.exec* scripts.

Input Macros and clearprompt

If your *f.exec* script calls a second script, which itself requires input from the user, use *clearprompt* in the second script. Note that some *cleartool* commands, when run from *xclearcase*, invoke *clearprompt* automatically to get user input. For example, a *checkin* command in an *f.exec* script uses *clearprompt* to collect a comment (if neither a comment nor -nc is supplied on the command line).

Input Macros and Keyboard Input

Most input macros have an optional *keyboard* argument, which you can use to enable keyboard input on the prompting browser. Always supply the ENABLE argument when the operation requires input strings that cannot, or may not, appear on the browser. For example, when defining a "make new directory" menu item, use ENABLE so the user can type in a new directory name.

In addition, you may want to use ENABLE even when the user could "point and click" from a browser; the keyboard input box lets users supply pathnames with wildcards, instead of having to select many individual items. In general, enable keyboard input whenever there is a chance the user will want to type in the required data items.

Input Macros and Browsers

Each ClearCase data type has a corresponding input macro whose job it is to prompt for that type of data as shown in Table 18-4.

Table 18-4 Data Types, Browsers, and Input Macros

Input macro	Kind of data returned by macro
%PNAME()	pathname
%HYPERLINK()	hyperlink
%LIST()	list data
%POOL()	storage pool
%ATTYPE(), %BRTYPE(), %ELTYPE(), %HLTYPE(), %LBTYPE(), %TRTYPE()	ClearCase type object
%STRING()	character string
%USERNAME()	UNIX username (login name)
%VIEWTAG()	view-tag
%VOBTAG()	VOB-tag

The prompt from an input macro appears at the bottom of the browser display and includes Ok and Cancel buttons. After Ok or Cancel, auto-displayed browsers disappear. In general, browsers stay up when started with an *f.call* function; otherwise, they come and go as needed to satisfy the input macros in *f.exec* strings.

Input Macros and Brackets

By default, an input macro seeks exactly one data item, and if the user selects more than one item, the operation fails with a Function not executed error message in the transcript pad.Use the optional square brackets on an input macro to collect multiple data input items. Note that the various numerical specifiers permitted inside brackets in preselect clauses do no apply to input macros.

Input Macros and Restrictions

If the user presses *Ok*, but the selected data does not satisfy the *restrictions* argument of an input macro, the transcript pad displays the script and the message Function not executed. (Note that the Ok button is not enabled until the user selects the number of data items that satisfies the [] construct.)

%ATTYPE

```
%ATTYPE[preselect-count]( restrictions )          preselector
%ATTYPE[]( prompt,restrictions,pname-in-VOB, keyboard ) macro
```

```
(also: %BRTYPE, %ELTYPE, %HLTYPE, %LBTYPE, %TRTYPE)
```

Prompt for one or more attribute, branch, element, hyperlink, label, or trigger types (see Table 18-5 and Example 18-1.)

Table 18-5 %ATTYPE Input Macro

Argument	Values	Default
prompt	Character string for user prompt	Select *xxx* type(s)
restrictions	ACTIVE or LOCK or OBSOLETE	any type object
pname-in-vob	Pathname in any VOB	VOB containing working dir
keyboard	ENABLE	not enabled

Example 18-1 ATTYPE

```
f.exec %QUOTE%
cleartool mkbranch %BRTYPE(Which branch?,,,ENABLE) %SELECTION()
%QUOTE%
```

Each *xx*type macro invokes a corresponding type browser on a particular VOB. Use the *restrictions* argument to limit the kind of data that the user can specify:

ACTIVE Prompt for (or allow preselection of) type objects that are not locked.

LOCK Accept only type objects that are locked.

OBSOLETE Accept only type objects that are obsolete. (See the lock manual page.)

%HYPERLINK

```
%HYPERLINK[preselect-count]( )                       preselector
%HYPERLINK[]( prompt,browser )                             macro
```

Prompt for one or more hyperlink objects (see Table 18-6 and Example 18-2.)

Table 18-6 `%HYPERLINK` Input Macro

Argument	Values	Default
prompt	Character string for user prompt	
browser	**vtree** *element-pname* [a]	

[a] Optional; if you omit it, user is prompted to select an element

Example 18-2 HYPERLINK

```
f.exec %QUOTE%
cleartool rmhlink \
  %HYPERLINK[](Select merge arrows to remove:,vtree
  %SELECTION())
%QUOTE%
```

%LIST

```
%LIST[preselect-count]( )                         preselector
%LIST[]( prompt,title )                                 macro
```

Prompt for data from a named list browser (see Table 18-7 and Example 18-3.)

Table 18-7 `%LIST` Input Macro

Argument	Values	Default
prompt	Character string for user prompt	
title	Title of list browser to use	

Example 18-3 LIST

```
f.exec %QUOTE%
%SAVE(COs,%LIST[](Select items to checkin:,My Checkouts))
cleartool checkin %RESTORE(COs)
%QUOTE%
```

Use %list to read *list data* from a named list browser, to which data was previously directed with %listout. (Only the %list macro can read the data in a list browser, and only %listout can create one.) The *title* argument must match the title used by %listout to create the list browser.

Use the %listout/%list combination when you need to display data and then have the user select from that data. The %listout and %list macros can appear in "back to back" *f.exec* scripts in the same item definition, or they can appear in separate item definitions — even in separate group files.

A list browser prompts the user to select one or more items (line of text). For each item, only the text preceding the first tab character is returned; any remainder is ignored. Pressing Ok submits the selection; pressing Cancel cancels the entire command operation. The user cannot edit the contents of a list browser.

%PNAME

```
%PNAME[preselect-count]( restrictions )          preselector
%PNAME[]( prompt,restrictions,browser,keyboard )       macro
```

Prompt (with file or vtree browser) for one or more pathnames (see Table 18-8.)

Table 18-8 `%PNAME` Input Macro

Argument	Values	Default
prompt	Character string for user prompt	Select pathname(s)
restrictions	DIR or NDIR	don't care
	ELEM or NELEM	don't care
	DOBJ or NDOBJ	don't care
	INVOB or NINVOB	don't care
	CHECKOUT or NCHECKOUT	don't care
	RESERVED or NRESERVED	don't care
browser	**file** *dir-pname* [a] or **vtree** *element-pname* [b]	file browser prompts for file in current directory
keyboard	ENABLE	not enabled

a. file — dir-pname is optional; if you omit it, current working directory is used

b. vtree — element-pname is optional; if you omit it, user is prompted to select an element

In Example 18-4, the File Browser comes up in */vobs/proj1/src*, not in the current working directory.

Example 18-4 PNAME

```
f.exec %QUOTE%
cleartool checkin \
%PNAME[](Files to checkin,
NDIR ELEM INVOB CHECKOUT RESERVED,
file /vobs/proj1/src)
%QUOTE%
```

Use the *restrictions* argument to limit the kind of data that the user can specify:

dir	all pathnames must be directories
ndir	all pathnames must not be directories
elem	all pathnames must be elements
nelem	all pathnames must not be elements
dobj	all pathnames must be derived objects
ndobj	all pathnames must not be derived objects
invob	all pathnames must be within some VOB
ninvob	all pathnames must not be within any VOB
checkout	all pathnames must be checked-out elements
ncheckout	all pathnames must not be checked-out elements
reserved	all pathnames must be elements with reserved checkouts
nreserved	all pathnames must not be elements with reserved checkouts

%POOL

```
%POOL[preselect-count]( )                            preselector
%POOL[]( prompt,pname-in-vob,keyboard )                    macro
```

Prompt for one or more storage pool names (see Table 18-9 and Example 18-5.)

Table 18-9 %POOL Input Macro

Argument	Values	Default
prompt	Character string for user prompt	
pname-in-vob	Pathname in any VOB	VOB containing working dir
keyboard	ENABLE	not enabled

Example 18-5 POOL

```
f.exec %QUOTE%
cleartool chpool %POOL(Select new pool:,,ENABLE) %SELECTION()
%QUOTE%
```

%STRING

```
%STRING[preselect-count]( )                        preselector
%STRING[]( prompt,default )                              macro
```

Prompt for one or more text lines (see Table 18-11 and Example 18-6.)

Table 18-10 %STRING Input Macro

Argument	Values	Default
prompt	Character string for user prompt	Enter string
default	Character string; pre-fills text input area	

Example 18-6 STRING

```
f.exec %QUOTE%
TAG='%STRING(Enter view tag name:)'
STGPNAME='%WILD(%STRING(Enter view storage pathname:,""))'
cleartool mkview -tag $TAG $STGPNAME
%QUOTE%
```

Use string browsers to prompt for simple text string arguments (for example, comments) or for other data strings— any data that cannot be captured by the more specific data type browsers.

If the %STRING macro includes square brackets, *xclearcase* may prompt the user for multiple lines of input. When the user presses *Ok*, the entire contents of the string browser is returned as a single text string: the last line is terminated with <NULL>; all other lines are terminated with <NL>.

%USERNAME

```
%USERNAME[preselect-count]( )                      preselector
%USERNAME[]( prompt,keyboard )                           macro
```

Prompt for one or more usernames (see Table 18-11 and Example 18-7.)

Table 18-11 %USERNAME Input Macro

Argument	Values	Default
prompt	Character string for user prompt	Select user name(s)
keyboard	ENABLE	Not enabled

Example 18-7 USERNAME

```
f.exec %QUOTE%
cleartool lscheckout -user \
  %USERNAME(Specify a user:,ENABLE) %TEXTOUT()
%QUOTE%
```

%VIEWTAG

```
%VIEWTAG[preselect-count]( restrictions )        preselector
%VIEWTAG[]( prompt,restrictions,keyboard )             macro
```

Prompt for one or more view-tags (see Table 18-9 and Example 18-8.)

Table 18-12 %VIEWTAG Input Macro

Argument	Values	Default
prompt	Character string for user prompt	Select viewtag(s)
restrictions	ACTIVE or INACTIVE	Don’t care
keyboard	ENABLE	not enabled

Example 18-8 VIEWTAG

```
f.exec %QUOTE%
cleartool startview %VIEWTAG()
%QUOTE%
```

%VOBTAG

`%VOBTAG[`*`preselect-count`*`](` *`restrictions`* `)`   *`preselector`*

`%VOBTAG[](` *`prompt,restrictions,keyboard`* `)`   *`macro`*

Prompt for one or more VOB-tags (see Table 18-9 and Example 18-9.)

Table 18-13 %VOBTAG Input Macro

Argument	Values	Default
prompt	Character string for user prompt	Select vobtag(s)
restrictions	MOUNTED or NMOUNTED	Don’t care
keyboard	ENABLE	Not enabled

Example 18-9 VOBTAG

```
f.exec %QUOTE%
cleartool umount %VOBTAG(,MOUNTED,ENABLE)
%QUOTE%
```

Note: The default *xclearcase* interface implements this operation differently:

```
%VOBTAG(MOUNTED) f.exec "cleartool umount %SELECTION()"
```

Modifier Macros

Modifier macros take pathname arguments, perform some kind of data manipulation (expansion, extraction, sorting, etc.), and return the resultant text strings.

Note: Unlike input macro arguments, arguments to modifier macros are all required.

%ELEMENT

`%ELEMENT(` *`pathnames`* `)` *`macro`*

Strip any ClearCase annotations from pathnames, leaving standard pathname (see Table 18-11 and Example 18-10.)

Table 18-14 %ELEMENT Modifier Macro

Argument	Values
pathnames	Any pathname(s) (non-element pathnames are returned unchanges)

Example 18-10 ELEMENT

```
f.exec %QUOTE%
cleartool checkin %ELEMENT(%SELECTION())
%QUOTE%
```

%ELEMSUFFIX

`%ELEMSUFFIX(` *`pathnames`* `)` *macro*

Append extended naming symbol (@@) to pathnames (see Table 18-11 and Example 18-11.)

Table 18-15 %ELEMSUFFIX Modifier Macro

Argument	Values
pathnames	Any pathname(s) (non-element pathnames are returned unchanges)

Example 18-11 ELEMSUFFIX

```
f.exec %QUOTE%
cleartool lsvtree %ELEMSUFFIX(
%ELEMENT(
%PNAME(Element whose version tree is to be listed,
INVOB ELEM)))
%QUOTE%
```

%MOUNT

`%MOUNT(` *`pathnames`* `)` *macro*

Return the VOB mount points (VOB-tags) of specified pathnames (see Table 18-11 and Example 18-12.)

Table 18-16 %MOUNT Modifier Macro

Argument	Values
pathnames	Any pathname(s) within one or more VOBs

Example 18-12 MOUNT

```
f.exec %QUOTE
%TEXTOUT(Mount List,"No mounted VOBs specified",6,4)
echo '%MOUNT(%PNAME[](Select VOB pathname%(s%),INVOB))'
%QUOTE%
```

The two parenthesis characters are escaped — %(and %) — to protect them from the macro expansion pass.

%RELATIVE

```
%RELATIVE( pathnames,dir )                                   macro
```

Convert full or relative pathnames into pathnames relative to directory (see Table 18-11 and Example 18-13.)

Table 18-17 %RELATIVE Modifier Macro

Argument	Values
pathnames	Any pathname(s) within one or more VOBs
dir	The directory to which the converted pathnames are to be relative

Example 18-13 RELATIVE

```
f.exec %QUOTE%
#!/bin/sh
PATH=$PATH:$HOME/%RELATIVE(
%PNAME(Select a dir anywhere under your home dir),$HOME)
%QUOTE%
```

%SETVIEW

```
%SETVIEW( pathnames )                                        macro
```

Convert pathnames to view-extended pathnames, based on the current set view (see Table 18-11 and Example 18-14.)

Table 18-18 %SETVIEW Modifier Macro

Argument	Values
pathnames	Any pathname(s)

The example invokes a text editor on the selected file.

Example 18-14 SETVIEW

```
f.exec %QUOTE%
%WHICH(GRP_PATH,editor.sh) %SETVIEW(%SELECTION())
%QUOTE%
```

Note: If the argument is already a view-extended pathname, it is unchanged by %setview, regardless of the view-tag on which it is based.

%SORT

`%SORT(` *`pathnames`* `)` *macro*

Sort pathnames as shown in Table 18-11 and Example 18-15.

Table 18-19 %SORT Modifier Macro

Argument	Values
pathnames	Any pathname(s)

Example 18-15 SORT

```
f.exec %QUOTE%
echo '%SORT(%PNAME[]())' %TEXTOUT(Sorted Pnames,,8,4)
%QUOTE%
```

`%VERMOD(` *`pathnames`* `)` *macro*

Convert standard pathnames to version-extended pathnames (see Table 18-11 and Example 18-16.)

Table 18-20 %VERMOD Modifier Macro

Argument	Values
pathnames	Any pathname(s) (non-element pathnames are returned unchanges)

Example 18-16 VERMOD

```
f.exec %QUOTE%
cleartool findmerge %SELECTION() -xmerge -log /dev/null \
  -whynot -fver %VERMOD(  %PNAME(
Select 'from' version: ['to' %VERMOD(%SELECTION())],ELEM INVOB,
vtree %SELECTION())
) | grep -v "Needs Merge"  %TEXTOUT("Merge Info",,8,2)
%QUOTE%
```

%WHICH

```
%WHICH( path-ev,filename )                          macro
```

Search specified path for the first occurrence of filename (see Table 18-11 and Example 18-17.)

Table 18-21 %WHICH Modifier Macro

Argument	**Values**
path-EV	Any environment variable that stores a directory search path (typically, grp_path)
filename	Any valid file name (a shell script or executable)

Example 18-17 WHICH

f.exec %QUOTE%

```
%WHICH(GRP_PATH,editor.sh) %SETVIEW(%SELECTION())
%QUOTE%
```

Like the unix *which*(1) command, %which returns (by searching *path-EV*) the full pathname of its *filename* argument. The %which macro exists to help support local customizations to the ClearCase-supplied *.sh* scripts that are installed in */usr/atria/config/ui/grp*.

The ClearCase-supplied group files include a number of *f.exec* scripts that invoke *.sh* files to perform specialized processing — finding and invoking the user's preferred text editor, for example. To customize a *.sh* file, first copy it to another directory. Make sure this directory occurs in the *path-EV*, and use %which to invoke your customized file.

For example, suppose you copy the file term_display.sh to $*home/.grp* and customize it. If your grp_path variable is set to $*home/.grp:/usr/atria/config/ui/grp* (which is the default), then the macro %WHICH(GRP_PATH,term_display.sh) finds and executes your version.

%WILD

`%WILD( *pathnames* )` *macro*

Expand wildcards in pathnames (see Table 18-11 and Example 18-18.)

Table 18-22 %WILD Modifier Macro

Argument	Values
path-expr	Pathname with one or more of these wildcard characters: (“), “*”, or “?”

Example 18-18 WILD

```
f.exec %QUOTE%
%PNAME(%WILD(%STRING()))
%QUOTE%
```

Output Macros

By default, the output generated by menu commands is collected in the transcript pad. The %listout and %textout macros redirect standard output to a list browser or read-only text window, respectively.

Both %textout and %listout redirect output for the entire *f.exec* operation, regardless of their location in the script. Either macro “expands” to the null string in the script; output redirection is established during the macro expansion “pre-pass”.

%LISTOUT

`%LISTOUT(` *`title`*`,`*`class`*`,`*`persistence`*`,`*`width`*`,`*`height`* `)` *`macro`*

Redirect output to a list browser (named or unnamed) (see Table 18-11 and Example 18-19.)

Table 18-23 `%LISTOUT` Output Macro

Argument	Values	Default
title	Character string	Enter string
class	Character string	class of active browser when macro invoked
persistence	TRANSIENT	browser stays up until user closes it
width	1-32 (window width in inches)	8
height	1-32 (window height in inches)	4

Example 18-19 LISTOUT

```
f.exec %QUOTE%
cleartool lsch -short . %LISTOUT(VOB Checkouts,,,6,4)
%QUOTE%
```

Use %listout to capture lists of pathnames or other VOB objects into a named list browser, from which data can later be selected (see %list) and supplied to other operations.

The *title* argument gives the list browser a name, which %list then uses to refer to the browser when using it to collect user input.

The optional *class* argument specifies a *list browser class*. You can scope a group file to a list browser class (giving the class an arbitrary name: Scope myList:toolbar, for example), thereby providing a set of menu items for the user to choose from when such a list browser is displayed. The group file menu appears on any list browser that %listout creates with the applicable class argument.

List data entries are separated or terminated by <nl> or null characters. Furthermore, within a single list entry (one line), only the text before the first tab character is returned by %list; the rest is ignored.

By default, a list browser created with %listout stays up until the user closes it. The TRANSIENT argument forces the list browser to close after either:

- a %list macro generates a prompt in that list browser, and the user responds to the prompt by pressing Ok or Cancel, or
- the menu item's command operation completes.

%TEXTOUT

`%TEXTOUT( title,default-text,width,height )` *macro*

Redirect standard output to read-only text-display window (see Table 18-11 and Example 18-20.)

Table 18-24 %TEXTOUT Output Macro

Argument	Values	Default
title	Title of text browser	
default-text	Character string	
width	1-32 (window width in inches)	8
height	1-32 (window height in inches)	4

Example 18-20 TEXTOUT

```
f.exec %QUOTE%
%TEXTOUT("Describe",,8,2)
cleartool lstype -attype -long %SELECTION() 2>&1"
%QUOTE%
```

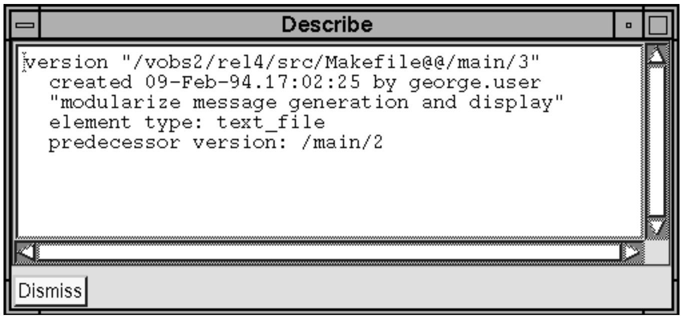
Unlike %listout, %textout redirects the standard output of commands to an untyped, read-only text window. Use %textout when your only objective is to display information.

The default text argument (*def-text*) specifies text to be displayed if the redirected operations generate no output.

Like %listout, %textout redirects *stdout* for the entire *f.exec* script. To redirect both *stderr* and *stdout* for a given command, use the */bin/sh* construct 2>&1.

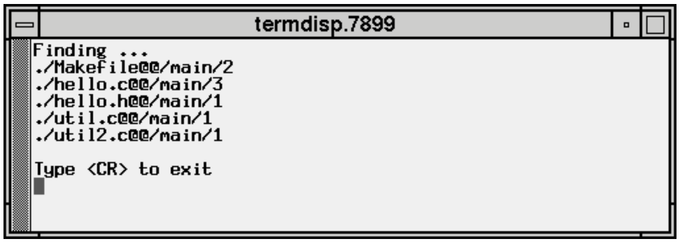
Text Output and Terminal Emulation Windows

For comparison with list browsers, Figure 18-15 shows a sample *text output window*, and Figure 18-16 shows a *terminal emulation window*. Neither prompts for, or accepts, user input; they are display-only devices.



```
version "/vobs2/rel4/src/Makefile@@/main/3"
  created 09-Feb-94.17:02:25 by george.user
  "modularize message generation and display"
  element type: text_file
  predecessor version: /main/2
```

Figure 18-15 A Text Output Window



```
Finding ...
./Makefile@@/main/2
./hello.c@@/main/3
./hello.h@@/main/1
./util.c@@/main/1
./util2.c@@/main/1

Type <CR> to exit
```

Figure 18-16 A Terminal Emulation Window

The text output window was generated by the Describe button and the terminal window by menu item Report ->Find query -> Whole VOB -> Versions with Label...

You can cancel output to a terminal emulation window with <Ctrl-C>.

Memory Macros

Use the %save and %restore macros to mimic user-defined shell variables inside other *xclearcase* macros. Because *xclearcase* simply expands macros into strings (before running the execution script), you cannot use shell variables or other shell constructs inside macros. However, in an *f.exec* script, you can use %save to store and name an arbitrary string. Later in the same script — or in a subsequent *f.exec* script — you can use %restore to apply the saved string in a command line (including inside an *xclearcase* macro).

By default, the temporary storage is release when the *f.exec* script terminates. Use the KEEP argument to retain the temporary variable until either (a) it is released by a %remove macro, or (b) the *xclearcase* session terminates.

%SAVE

`%SAVE( var-name,string,persistence )` *macro*

Save string into named temporary variable (can be referenced with %RESTORE (see Table 18-11 and Example 18-21.))

Table 18-25 %SAVE Memory Macro

Argument	Values	Default
var-name	Name of temporary variable	Enter string
string	Character string to be stored in variables	
persistence	KEEP	variable removed after this *f.exec* completes

Example 18-21 SAVE

```
f.exec %QUOTE%
%SAVE(dir, %PNAME(Select target dir, DIR INVOB),KEEP)
  .
  .
```

```
DIR='%RESTORE(dir)' %REMOVE(dir)
%QUOTE%
```

%REMOVE

`%REMOVE(` *`var-name`* `)` *macro*

Delete a persistent temporary variable created with %SAVE (see Table 18-11 and Example 18-22.)

Table 18-26 `%REMOVE` Memory Macro

Argument	Values
var-name	Name of temporary variable

Example 18-22 REMOVE

```
f.exec %QUOTE%
%SAVE(tmp_print_file, %TMPFILE(), KEEP)
   .
   .
%REMOVE(tmp_print_file)
%QUOTE%
```

%RESTORE

`%RESTORE(` *`var-name`* `)` *`macro`*

Retrieve the value of a temporary variable created with %SAVE (see Table 18-11 and Example 18-23.)

Table 18-27 `%RESTORE` Memory Macro

Argument	Values
var-name	Name of temporary variable

Example 18-23 RESTORE

```
f.exec %QUOTE%
%SAVE(dir, %PNAME(Select target directory, DIR INVOB))
  .
  .
DIR='%RESTORE(dir)'
%QUOTE%
```

Miscellaneous Macros

These macros take no arguments.

%SELECTION

`%SELECTION( )` *macro*

Return the current data selection(s) from the current browser (see Example 18-24.)

Example 18-24 SELECTION

```
f.exec %QUOTE%
cleartool uncheckout -rm %SELECTION()
%QUOTE%
```

The %selection() macro is replaced by the data that has been preselected by the user. %selection() can appear only in items in which the *f.exec* strings is preceded by an activation clause.

%DIR

`%DIR( )` *macro*

Return the pathname of the current browser's directory (see Example 18-25.)

Example 18-25 DIR

```
f.exec %QUOTE%
echo "Current working dir is %DIR()"
%QUOTE%
```

%NAME

```
%NAME( )macro
```

Return the name associated with the current browser (see Example 18-26.)

Example 18-26 NAME

```
f.exec %QUOTE%
%WHICH(GRP_PATH,term_display.sh) USE_MORE \
cleartool lshistory -long %NAME()
%QUOTE%"
```

For a vtree browser, expands to name of the element whose vtree is displayed. For all other browsers, expands to the current directory pathname.

%TMPFILE

```
%TMPFILE( )                                              macro
```

Create a temporary file and return its pathname (see Example 18-27.)

Example 18-27 TMPFILE

```
f.exec %QUOTE%
%SAVE(tmp_print_file, %TMPFILE(), KEEP)
%QUOTE%
```

This example creates a temporary file in $TMPDIR (or */usr/tmp*, if this EV is not set). The full pathname of the temporary file is placed in the *xclearcase* variable tmp_print_file.

Customization Procedures

Customization tasks fall into two general categories:

- add a new menu
- replace an existing menu

Here are the basic procedures for accomplishing each task.

Adding a New Menu

1. Copy to your work area (typically, $home/*.grp*) the predefined group file template */usr/atria/config/ui/grp/user.grp.template*, which is a syntax skeleton. Rename the file to *user.grp*, or to some other name with a *.grp* suffix.
2. Determine a scope for your new menu (Fast:pulldown, Vtree:pulldown, or File:pulldown, for example).

   The template group file defaults to Fast:pulldown scope. This positions the new menu on the file browser's Fast pulldown menu and, optionally, as a set of buttons on the file browser toolbar. See "Scope" on page 237 for more details.
3. Replace the template group file's menu item definitions with your own. Copy and modify an item definition from one of the ClearCase-supplied group files (ideally, one that does some or all of the desired operation).

   Each item definition must include at least a label (or no-label) and an *f-dot* function. (Most will include one or more *f.exec* scripts.) For a pull-down or pop-up menu item, it is good practice to add a menu mnemonic and, when applicable, a preselect clause. For a toolbar button, you may wish to add bitmap icons. If your menu will be used by others, supply *f.help* strings for all menu items.

   Before attaching a complex script to a menu item, see "Complex Execution Scripts", below.
4. Restart *xclearcase* and test the new menu items.

Because the group files are read once at startup time and cached, you must exit and restart *xclearcase* to activate any group file changes. In general, a menu is ignored (it does not appear in the interface) if its group file contains syntax errors; check the transcript pad for diagnostic messages.

By default, */bin/sh* is used to execute your *f.exec* scripts. Include a first line like #!/bin/csh to specify another shell.

Note: If you set the environment variable clearcase_dbg_grp to a non-zero value, *xclearcase* sends debugging information to the transcript pad when executing commands.

Replacing an Existing Menu

1. Find the ClearCase-supplied *.grp* file you want to customize (typically, to add new items) and copy it to *$home/.grp*, or to any directory in your grp_path that precedes */usr/atria/config/ui/grp*. Because *xclearcase* ignores any group file with the same name as one already scanned in, your copied group file now "eclipses" the standard group file.
2. Add your own item definitions.

   Note: Although you are now in a position to modify the predefined menu items, this is not advised. Such customizations are equivalent to modifying the source code, and have the potential to leave you "out of sync" with future ClearCase-supplied enhancements and bug fixes to the standard group files.
3. Restart *xclearcase* and test each new menu item.

Complex Execution Scripts

The technique described here is recommended if you are attaching either of the following to a menu item:

- a complex operation
- an operation that must be repeated for multiple menu items

Code the “meat” of the operation as a standard shell script (no ClearCase macros), and invoke the script from an *f.exec* script using the %which macro. The %which macro is described on page 278. The ClearCase-supplied group files include numerous examples of this technique (calling the various *.sh* scripts installed in */usr/atria/config/ui/grp*).

For example, the following item definition includes an *f.exec* script that calls *print_file.sh* to find a print command and then print the preselected files:

```
"Print" _P %PNAME[](NDIR) & \
f.exec %QUOTE%
       %WHICH(GRP_PATH,print_file.sh) %SETVIEW(%SELECTION())
       %QUOTE% \
f.help "Print the selected files."
```

Note: “external” scripts cannot use *xclearcase* macros to collect input and, therefore, must use *clearprompt* or some other mechanism to do so.

Resource Schemes

ClearCase provides a number of predefined color and font combinations for use with the graphical interface. Each predefined collection of resources is called a *scheme*. See the *schemes* manual page for details.

Note: The scheme files commonly set specific widget class resources. Therefore, *xclearcase* command line options (for example, -bg and -fg) may not work as expected, because they set resources at the most general level. You have the option to disable schemes, to modify them, or to set still more specific resources.

Icon Display in the File Browser

A file browser can display file system objects either by-name or graphically. In the latter case *xclearcase* selects an icon for each file system object as follows:

- In a process described in the *cc.magic* manual page, *xclearcase* uses an object's name, and/or its contents, to compile a list of one or more file types that correspond to the object.
- One by one, the file types are compared to the rules in predefined and user-defined *icon* files, as described in the *cc.icon* manual page. For example, the file type *c_source* matches this icon file rule:

  ```
  c_source : -icon c ;
  ```

As soon as a match is found, the search ends. The token following -icon identifies the icon's bitmap file.

- By default, *xclearcase* searches first for the bitmap file (which must be in *bitmap*(1) format) in the *.bitmaps* subdirectory of your home directory, then in */usr/atria/config/ui/bitmaps*. If the bitmap_path variable is set, *xclearcase* searches the directories there instead.
- If a valid bitmap file is found, *xclearcase* displays it; otherwise, the search for an icon continues with the next file type.

The name of an icon file should include a numeric suffix, which need not be specified in the icon file rule. The suffix tells *xclearcase* how much screen space to allocate for the icon. Each bitmap supplied with ClearCase is stored in a file with a .60 suffix (for example, lib.60), indicating a 60x60 icon.

Enabling a Customized Icon

The following steps install a customized icon for unix-style manual page source files.

1. Add a rule to your personal magic file ($home/*.magic*) that includes *manpage* among the file types assigned to all manual page source files:

   ```
   manpage src_file text_file file:  -name "*.[1-9]" ;
   ```

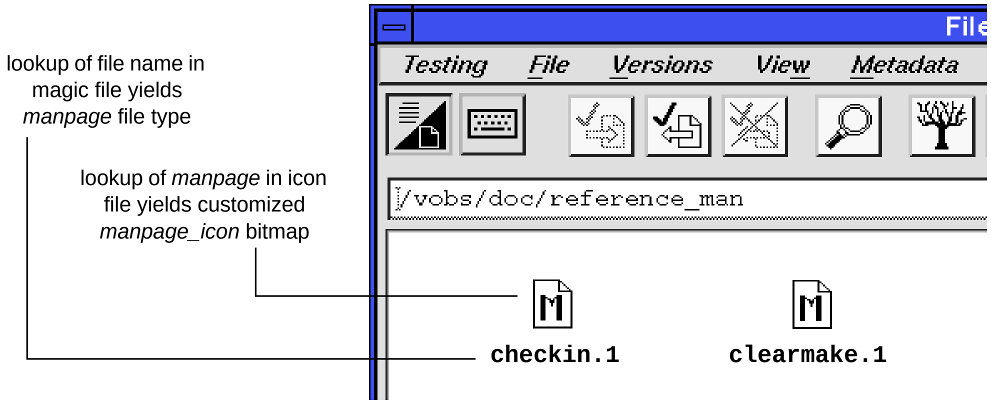
2. Add a rule to your personal icon file (in directory $home/*.icon*) that maps *manpage* to a user-defined bitmap file:

   ```
   manpage : -icon manual_page_icon ; manpage : -icon manual_page_icon ;
   ```

3. Create a *manpage* icon in your personal bitmaps directory ($home/*.bitmaps*) by revising one of the standard icon bitmaps with the X *bitmap* utility:

   ```
   % mkdir $HOME/.bitmaps
   % cd $HOME/.bitmaps
   % cp /usr/atria/config/ui/bitmaps/c.60 manual_page_icon.60
   % bitmap manual_page_icon.60
   ```

4. Test your work by having an *xclearcase* file browser display a manual page source file (Figure 18-17).



/vobs/doc/reference_man

lookup of file name in magic file yields *manpage* file type
lookup of *manpage* in icon file yields customized *manpage_icon* bitmap
File
Testing
File
Versions
View
Metadata
checkin.1
clearmake.1

Figure 18-17 User-Defined Icon Display

Chapter 19

Type Managers and Customized Processing of File Elements

This chapter discusses several features that allow you to classify files, and customize the way in which ClearCase manages different classes of files. These features include:

- file types
- element types and predefined type managers
- user-defined type managers
- icons for file types

Scenario

As an example, consider the various kinds of files involved in the *monet* project, discussed in several earlier chapters. Also see Table 19-1. (For simplicity, we won't attempt to classify all the files, just a representative sample.)

Table 19-1 Files Used in 'monet' Project

Kind of Files	Identifying Characteristics
Source Files	
C-language source file	`.c` file name suffix
C-language header file	`.h` file name suffix
FrameMaker^{TM} binary file	`<MakerFile>` as "magic number"
manual page source file	`.1 – .9` file name suffix
Derived Files	
ar(1) archive (library)	`.a` file name suffix
compiled executable	*<varies with system architecture>*

File Typing

In various contexts, ClearCase determines one or more *file types* for an existing file system object, or for a name to be used for a new object:

- When a *mkelem* command is entered without the -eltype option, file typing is performed on the new element's simple file name (*leaf name*).
- *xclearcase* sometimes displays file system objects as icons. To do so, it performs file typing on each object; then, it uses the file type to select an icon.

The file typing routines use predefined and user-defined *magic* files, as described in the *cc.magic* manual page. A magic file can use many different techniques to determine a file type, including file name pattern-matching, *stat*(2) data, and standard UNIX "magic numbers".

For example, the magic file listed in Figure 19-1 specifies several file types for each kind of file listed in Table 19-1.

```
c_src src_file text_file file:  -name "*.c";                       (1)
hdr_file text_file file:       -name "*.h" ;                       (2)
frm_doc doc file:              -magic 0, "<MakerFile" ;            (3)
manpage src_file text_file file: -name "*.[1-9]" ;                 (4)
archive derived_file file:       -magic 32, "archive" ;            (5)
sunexec derived_file file:     -magic 40,"SunBin" ;                (6)
```

Figure 19-1 Sample 'Magic' File

Element Types and Type Managers

ClearCase's ability to handle different classes of files differently hinges on the concept of *element type*. Each file element in a VOB must have an element type. An element gets its type when it is created with the *mkelem* command; you can change an element's type subsequently, with the *chtype* command. (An element is an *instance* of its element type, in the same way that an attribute is an instance of an attribute type, and a version label is an instance of a label type.)

Each element type has an associated *type manager*, a suite of programs that handle the storage and retrieval of versions from storage pools. Thus, the way in which a particular file element's data is handled involves two correspondences: (1) the file element has an element type; (2) the element type has a type manager. Figure 19-2 shows how these facilities work together.

Note: Each directory element also has an element type. But directory elements do not use type managers — the contents of a directory version are stored in the VOB database itself, not in storage pools.

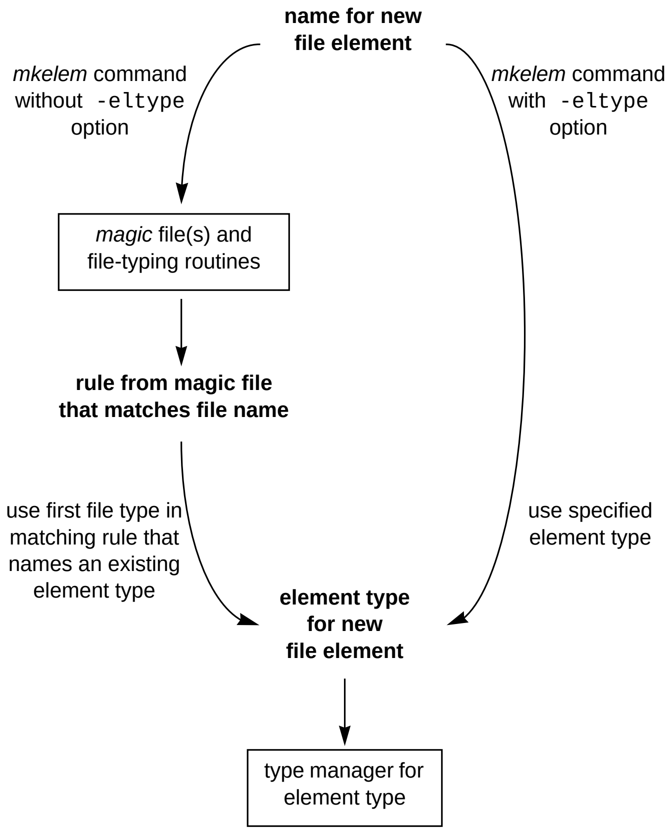

name for new file element
mkelem command without -eltype option
mkelem command with -eltype option
magic file(s) and file-typing routines
rule from magic file that matches file name
use first file type in matching rule that names an existing element type
use specified element type
element type for new file element
type manager for element type

Figure 19-2 Data Handling: File Type, Element Type, Type Manager

For example, a new element named *monet_adm.1* might get its element type as follows:

1. A user enters a “create element” command:

   ```
   % cleartool mkelem monet_adm.1
   ```

2. Since the user did not specify an element type (–eltype option), *mkelem* uses one or more magic files to determine the file type(s) of the specified name. Suppose that the magic file shown in Figure 19-1 is the first (or only) one to be used. In this case, rule (4) is the first one to match the name *monet_adm.1*, yielding this list of file types:

   ```
   manpage src_file text_file file
   ```

3. *mkelem* compares this list with the set of element types defined for the new element’s VOB. Suppose that text_file is the first file type that names an existing element type; in this case, *monet_adm.1* is created as an element of type text_file.

4. Data storage and retrieval for versions of element *monet_adm.1* will be handled by the type manager associated with the text_file element type; its name is *text_file_delta*:

   ```
   % cleartool describe -type text_file
   element type "text_file"
    01-Feb-93.09:11:32 by VOB administrator (vobadm.dvt@sol)
     "Predefined element type used to represent a text file."
     type manager: text_file_delta
     supertype: file
     meta-type of element: file element
   ```

Note: ClearCase supports a “search path” facility, using the environment variable MAGIC_PATH. See the cc.magic manual page for details.

File-typing mechanisms are defined on a per-user (or per-site) basis; element types are defined on a per-VOB basis.A new element named *monet_adm.1* was created as a text_file element in this case; in a VOB with a different set of element types, the same magic file might have caused it to be created as a src_file element.

Note: Ensuring element-type consistency among VOBs is a manual administrative task.

Other Applications of Element Types

Element types allow differential and customized handling of files beyond the selection of type managers discussed above. Following are some examples.

Using Element Types to Configure a View

Creating all C-language header files as elements of type hdr_file allows flexibility in configuring views. Suppose that one developer has been reorganizing

all of the project's header files, working on a branch named *header_reorg* to avoid destabilizing others' work. To compile with the new header files, another developer can use a view reconfigured with one additional rule:

```
element * CHECKEDOUT
element -eltype hdr_file * /main/header_reorg/LATEST
element * /main/LATEST
```

Processing Files by Element Type

Suppose that a coding-standards program named *check_var_names* is to be executed on each C-language source file. If all such files have element type c_src, then a single *cleartool* command does the job:

```
%  cleartool find -avobs -visible -element 'eltype(c_src)' \
        -exec 'check_var_names $CLEARCASE_PN'
```

Predefined and User-Defined Element Types

Some of the element types discussed in the sections above (for example, text_file) are *predefined*. Others (for example, c_src and hdr_file) are not predefined — the examples above work only if *user-defined* element types with these names are created with the *mkeltype* command.

When a new VOB is created, it automatically gets a full set of the predefined element types. Each one is associated with one of the type managers provided with ClearCase. The *mkeltype* manual page describes the predefined element types and their type managers.

When you create a new element type with *mkeltype*, you must specify an existing element type as its *supertype*. By default, the new element type uses the same type manager as its supertype; in this case, the only distinction between the new and old types is for the purposes described in “Other Applications of Element Types” on page 300. For differential data handling, use the –manager option to create an element type that uses a different type manager from its supertype.

Directory */usr/atria/examples/mkeltype* contains shell scripts that create a hierarchy of element types.

Predefined and User-Defined Type Managers

Just as ClearCase has predefined element types, it also has predefined type managers. They are described in the *type_manager* manual page. Each type manager is implemented as a suite of programs in a subdirectory of */usr/atria/lib/mgrs* — the name of the subdirectory is the name of the type manager.

The mkeltype -manager command creates an element type that uses an *existing* type manager. You can further customize ClearCase by creating totally *new* type managers (and then creating new element types that use them). Architecturally, type managers are mutually independent. But new type managers can use symbolic links to “inherit” some of the functions of existing ones.

The *type_manager* manual page describes the basic components of a type manager, and outlines the process of creating a new type manager. File */usr/atria/lib/mgrs/mgr_info.h* provides comprehensive information on type managers. We suggest that you familiarize yourself with these sources before proceeding to the following sections, which present an extended example of creating and using a new type manager.

Type Manager for Manual Page Source Files

One kind of file listed in Table 19-1 is “manual page source file” (a file coded in *nroff*(1) format). A type manager for this kind of file might have these characteristics:

- stores all versions in compressed form in separate data containers, like the *z_whole_copy* type manager
- implements version-comparison (*compare* method) by diff’ing formatted manual pages instead of the source versions themselves

The basic strategy is to use most of the *z_whole_copy* type manager’s methods. The *compare* method will use *nroff*(1) to format the versions before displaying their differences.

Creating the Type Manager Directory

The name *mp_mgr* (“manual page file”) is appropriate for this type manager. The first step is to create a subdirectory with this name:

```
# mkdir /usr/atria/lib/mgrs/mp_mgr
```

Inheriting Methods from Another Type Manager

Most of the *mp_mgr* methods will be inherited from the *z_whole_copy* type manager, through symbolic links. You might enter the following commands as the *root* user in a Bourne shell:

```
# MP=$ATRIAHOME/lib/mgrs/mp_mgr
# for FILE in create_element create_version construct_version \
      create_branch delete_branches_versions \
      merge xmerge xcompare
> do
> ln -s ../z_whole_copy/$FILE $MP/$FILE
> done
#
```

Any methods that will not be supported by the new type manager can be omitted from this list. The lack of a symbolic link will cause ClearCase to generate an Unknown Manager Request error.

The following sections describe two of these inherited methods, *create_version* and *construct_version*, which can serve as models for user-defined methods. Both are actually implemented as scripts in the same file, */usr/atria/lib/mgrs/z_whole_copy/Zmgr*.

The 'create_version' Method

The *create_version* method is invoked when a *checkin* command is entered. The *create_version* method of the *z_whole_copy* type manager:

- compresses the data in the checked-out version
- stores the compressed data in a data container located in a source storage pool
- returns an exit status to the calling process, indicating what to do with the new data container

File */usr/atria/lib/mgrs/mgr_info.h* shows what arguments will be passed to the method from the calling program (usually *cleartool* or *xclearcase*):

```
/***************************************************************************
 * create_version
 * Store the data for a new version.
 * Store the version's data in the supplied new container, combining it
 * with the predecessor's data if desired (e.g for incremental deltas).
 *
 * Command line:
 *  create_version create_time new_branch_oid new_ver_oid new_ver_num
 *               new_container_pname pred_branch_oid pred_ver_oid
 *               pred_ver_num pred_container_pname data_pname
```

The only arguments that require special attention are new_container_pname (5th argument), which specifies the pathname of the new data container, and data_pname (10th argument), which specifies the pathname of the checked-out file.

File */usr/atria/lib/mgrs/mgr_info.sh* lists the appropriate exit statuses and provides a symbolic name for the *create_version* method:

```
# Any unexpected value is treated as failure
MGR_FAILED=1

# Return Values for store operations
MGR_STORE_KEEP_NEITHER=101
MGR_STORE_KEEP_JUST_OLD=102
MGR_STORE_KEEP_JUST_NEW=103
MGR_STORE_KEEP_BOTH=104
 .
 .
MGR_OP_CREATE_VERSION="create_version"
```

Figure 19-3 shows the code that implements the *create_version* method.

```
shift 1                                                         (1)
if [ -s $4 ] ; then                                             (2)
    echo '$0: error: new file is not of length 0!'              (3)
    exit $MGR_FAILED                                            (4)
fi                                                              (5)
if cat $9 | compress >> $4 ; ret=$? ; then : ; fi               (6)
if [ "$ret" = "2" -o "$ret" = "0" ] ; then                      (7)
    exit $MGR_STORE_KEEP_BOTH                                   (8)
else                                                            (9)
    exit $MGR_FAILED                                            (10)
fi                                                              (11)
```

Figure 19-3 ’create_version’ Method

The Bourne shell allows only nine command-line arguments. The shift 1 in Line 1 discards the first argument (create_time), which is unneeded. Thus, the pathname of the checked-out version (data_pname), originally the 10th argument, becomes $9.

In Line 6, the contents of data_pname are compressed, then appended to the new, empty data container: new_container_pname, originally the 5th argument, but shifted to become $4. (Lines 2-5 verify that the new data container is, indeed, empty.)

Finally, the exit status of the *compress* command is checked, and the appropriate value is returned (Lines 7–11). The exit status of the *create_version* method indicates that both the old data container (which contains the predecessor version) and the new data container (which contains the new version) should be kept.

The ‘construct_version’ Method

An element’s *construct_version* method is invoked when standard UNIX software reads a particular version of the element (unless the contents are already cached in a cleartext storage pool). For example, the *construct_version* method of element *monet_admin.1* is invoked by the *view_server* when a user enters these commands:

```
% cp monet_admin.1 /usr/tmp   (read version selected by view)
% cat monet_admin.1@@/main/4       (read a specified version)
```

It is also invoked during a *checkout* command, which makes a view-private copy of the most recent version on a branch. The *construct_version* method of the *z_whole_copy* type manager:

- uncompresses the contents of the data container
- returns an exit status to the calling process, indicating what to do with the new data container

File */usr/lib/lib/mgrs/mgr_info.h* shows what arguments will be passed to the method:

```
/****************************************************************************
 * construct_version
 * Fetch the data for a version.
 * Extract the data for the requested version into the supplied pathname, or
 * return a value indicating that the source container can be used as the
 * cleartext data for the version.
 *
 * Command line:
 *  construct_version source_container_pname data_pname version_oid
```

File */usr/atria/lib/mgrs/mgr_info.sh* lists the appropriate exit statuses and provides a symbolic name for the *construct_version* method:

```
# Any unexpected value is treated as failure
MGR_FAILED=1

# Return Values for construct operations
MGR_CONSTRUCT_USE_SRC_CONTAINER=101
MGR_CONSTRUCT_USE_NEW_FILE=102
 .
 .
MGR_OP_CONSTRUCT_VERSION="construct_version"
```

Figure 19-4 shows the code that implements the *construct_version* method.

```
if cat $1 | uncompress >> $2; then
                                         (1)
    exit $MGR_CONSTRUCT_USE_NEW_FILE
                                         (2)
else
                                         (3)
    exit $MGR_FAILED

                                         (4)
fi
                                         (5)
```

Figure 19-4 ’construct_version’ Method

In Line 1, the contents of source_container_pname are uncompressed, then stored in the cleartext container, data_pname. The remaining lines return the appropriate value to the calling process, depending on the success or failure of the *uncompress* command.

Implementing a New ‘compare’ Method

The *compare* method is invoked by a cleartool diff command. This method will:

- format each version using *nroff*(1), producing a pure-ASCII text file
- compare the formatted versions, using *cleardiff*

File */usr/atria/lib/mgrs/mgr_info.h* shows what arguments will be passed to the method from *cleartool* or *xclearcase*:

```
/****************************************************************************
 * compare
 * Compare the data for two or more versions.
 * For more information, see man page for cleartool diff.
 *
 * Command line:
 *  compare [-tiny | -window] [-serial | -diff | -parallel] [-columns n]
 *         [pass-through-options] pname pname ...
```

This listing shows that a user-supplied implementation of the *compare* method must accept all of the command-line options supported by the ClearCase *diff* command. Our strategy will be simply to pass the options on to *cleardiff*, without attempting to interpret them. After all options are processed, the remaining arguments specify the files to be compared.

File */usr/atria/lib/mgrs/mgr_info.sh* lists the appropriate exit statuses and provides a symbolic name for the *compare* method:

```
# Return Values for COMPARE/MERGE Operations
MGR_COMPARE_NODIFFS=0
MGR_COMPARE_DIFF_OR_ERROR=1
  .
  .
MGR_OP_COMPARE="compare"
```

The Bourne shell script listed in Figure 19-5 implements the *compare* method. Implementing the *xcompare* method as a slight variant of *compare* is left as an exercise for the reader.

```
#!/bin/sh -e
MGRDIR=${ATRIAHOME:-/usr/atria}/lib/mgrs

#### read file that defines methods and exit statuses
. $MGR_DIR/mgr_info.sh

#### process all options: pass them through to cleardiff
OPTS=""
while (expr $1 : '\-' > /dev/null) ; do
    OPTS="$OPTS $1"
    if [ "$1" = "$MGR_FLAG_COLUMNS" ] ; then
        shift 1
        OPTS="$OPTS $1"
    fi
    shift 1
done

#### all remaining arguments ($*) are files to be compared
#### first, format each file with NROFF
COUNT=1
TMP=/usr/tmp/compare.$$
for X in $* ; do
    nroff -man $X | col | ul -Tcrt > $TMP.$COUNT
    COUNT=‘expr $COUNT + 1‘
done

#### then, compare the files with cleardiff
cleardiff -quiet $OPTS $TMP.*

#### cleanup and return appropriate exit status
if [ $? -eq MGR_COMPARE_NODIFFS ] ; then
    rm -f $TMP.*
    exit MGR_COMPARE_NODIFFS
else
    rm -f $TMP.*
    exit MGR_COMPARE_DIFF_OR_ERROR
fi
```

Figure 19-5 Script for ‘compare’ Method

Testing the Type Manager

A new type manager can be tested only by actually using it on some ClearCase host. This process need not be obtrusive. Since the type manager has a new name, no existing element type — and hence, no existing element — uses it automatically. To place the type manager in service, create a new element type, create some test elements of that type, and run some tests.

The following testing sequence continues the *mp_mgr* example.

Creating a Test Element Type. To make sure that an untested type manager is not used accidentally, associate it with a new element type, *manpage_test*, whose use is restricted to yourself.

```
% cleartool mkeltype -nc -supertype compressed_file \
         -manager mp_mgr manpage_test
% cleartool lock -nusers $USER -eltype manpage_test
```

Creating and Using a Test Element. These commands create a test element that uses the new type manager, and tests the various data-manipulation methods:

```
% cd directory-in-test-VOB
% cleartool checkout -nc .
                              (tests ‘create_element’ method)
% cleartool mkelem -eltype manpage_test -nc -nco test.1
% cleartool checkout -nc test.1
                           (tests ‘construct_version’ method)
% vi test.1                        (edit checked-out version)
% cleartool checkin -c "first" test.1
                             (tests ‘create_ version’ method)
% cleartool checkout -nc test.1
                          (tests ‘construct_ version’ method)
% vi test.1
                                   (edit checked-out version)
% cleartool checkin -c "second" test.1
                             (tests ‘create_ version’ method)
% cleartool diff test.1@@/main/1 test.1@@/main/2
                                     (tests ‘compare’ method)
```

Installing and Using the Type Manager

After a type manager has been fully tested, you can “make it official” with the following procedure.

Installation. A VOB is a network-wide resource — it can be mounted on any ClearCase host. But a type manager is only a host-wide resource — a separate copy must be installed on each host where ClearCase client programs execute. (It need not be installed on hosts that serve only as repositories for VOBs and/or views.)

To install the type manager on a particular host, create a subdirectory in */usr/atria/lib/mgrs*, then populate it with the programs that implement the methods. You might create across-the-network symbolic links to a "master copy" on a server host.

Creation of Element Types. Create one or more element types that use the type manager, just as you did in <Emphasis>Testing the Type Manager on page 308 (but don't include "test" in the name of the element type!). For example, you might name the element type named *manpage* or *nroff_src*.

Conversion of Existing Elements. It is likely that you'll want to have at least a few existing elements use the new type manager. The *chtype* command does the job:

```
% cleartool chtype -force manpage pathname(s)
```

Permission to change an element's type is restricted to the element's owner, the VOB owner, and the *root* user.

Revision of Magic Files. If you want the new element type(s) to be used automatically for certain newly created elements, create (or update) a *local.magic* file in each host's */usr/atria/config/magic* directory:

```
manpage src_file text_file file: -name "*.[1-9]" ;
```

Public Relations. Advertise the new element type(s) to all ClearCase users, describing the features and benefits of the new type manager. Be sure to include directions on how to gain access to the new functionality automatically (through filenames that match 'magic' file rules) and explicitly (with mkelem -eltype).

Icon Usage by GUI Browsers

An *xclearcase* browser can display file system objects either by-name or graphically. In the latter case, *xclearcase* selects an icon for each file system object as follows:

- The object's name or its contents determines a list of file types, as described in "File Typing" on page 296.
- One by one, the file types are compared to the rules in predefined and user-defined *icon* files, as described in the *cc.icon* manual page. For example, the file type *c_source* matches this icon file rule:

  ```
  c_source : -icon c ;
  ```

  As soon as a match is found, the search ends. The token following -icon names the file that contains the icon to be displayed.
- *xclearcase* searches for the file, which must be in *bitmap*(1) format, in directory *$HOME/.bitmaps*, or */usr/atria/config/ui/bitmaps*, or the directories specified by the environment variable BITMAP_PATH.
- If a valid bitmap file is found, *xclearcase* displays it; otherwise, the search for an icon continues with the next file type.

The name of an icon file should include a numeric suffix, which need not be specified in the icon file rule. The suffix tells *xclearcase* how much screen space to allocate for the icon. Each bitmap supplied with ClearCase is stored in a file with a .60 suffix (for example, lib.60), indicating a 60x60 icon. These bitmaps themselves are all 40x40, however; the discrepancy allows for future revisions of *xclearcase* to annotate the icons.

The following steps show how you can have *xclearcase* display manual page source files with a customized icon. In accordance with the preceding sections of this chapter, all manual pages will have file type *manpage*.

1. Add a rule to your personal magic file (in directory *$HOME/.magic*) that includes *manpage* among the file types assigned to all manual page source files:

```
manpage src_file text_file file:  -name "*.[1-9]" ;
```

2. Add a rule to your personal icon file (in directory *$HOME/.icon*) that maps *manpage* to a user-defined bitmap file:

```
manpage : -icon manual_page_icon ;
```

3. Create a *manpage* icon in your personal bitmaps directory (*$HOME/.bitmaps*) by revising one of the standard icon bitmaps with the standard X *bitmap* utility:

```
% mkdir $HOME/.bitmaps
% cd $HOME/.bitmaps
% cp $ATRIAHOME/config/ui/bitmaps/c.60 manual_page_icon.60
% bitmap manual_page_icon.60
```

4. Test your work by having an *xclearcase* graphical directory browser display a manual page source file (Figure 19-6).

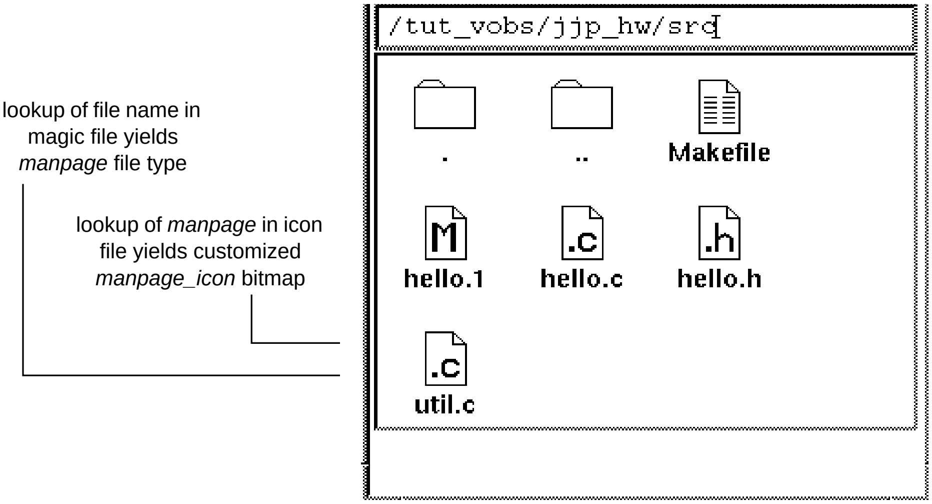

lookup of file name in magic file yields *manpage* file type
lookup of *manpage* in icon file yields customized *manpage_icon* bitmap
/tut_vobs/jjp_hw/src
.
..
Makefile
hello.1
hello.c
hello.h
util.c

Figure 19-6 User-Defined Icon Displayed by *xclearcase*

Chapter 20

Using Triggers, Attributes, and Locks to Implement Development Policies

This chapter presents brief scenarios, showing how common development policies can be implemented and enforced with ClearCase.

Scenario: Requiring Good Documentation of Changes

Policy #1 “All changes to sources should be recorded and comments should be provided.”

Each ClearCase command that modifies a VOB automatically creates one or more *event records*. Many such commands (for example, *checkin*) prompt for a comment. The event record automatically includes the user’s name, date, comment, host machine, and description of what was changed.

To prevent users from subverting the system by providing empty or meaningless comments, you might create a *pre-operation trigger type* to monitor the *checkin* operation. The trigger action script could analyze the user’s comment (passed in an environment variable), disallowing “bad” ones (for example, those shorter than 10 words).

Trigger definition:

```
% mktrtype -element -global -preop checkin \
       -exec comment_policy.sh CommentPolicy
```

Trigger action script:

```
#!/bin/sh
#
#      comment_policy
#
ACCEPT=0
REJECT=1
WORDCOUNT=‘echo $CLEARCASE_COMMENT | wc -w‘

if [ $WORDCOUNT -ge 10 ] ; then
      exit $ACCEPT
else
      exit $REJECT
fi
```

Scenario: Tracking State Transitions

Policy #2 “The system must track the progress of each source file through the official approval stages.”

A process-control system must be able to track the progress of individual files. Ideally, the system will shepherd the file through various intermediate states, until it is finally declared “ready”. Attributes fit this model naturally — you can create a string-valued attribute type, *Status*, which accepts only a specified set of values.

Attribute definition:

```
% cleartool mkattype -c "standard file levels" \
        -enum '"inactive","under_devt","QA_approved"' Status
Created attribute type "Status".
```

A *Status* attribute will be applied to many different versions of an element. Early versions on a branch might get the attribute with "inactive" and "under_dvt" values; later versions the "QA_approved" value. The same value might be used for several versions, or moved from an earlier version to a later version.

To enforce conscientious application of the status attribute to versions of all source files, you might create an *CheckStatus* trigger type much like the *CommentPolicy* trigger type in the preceding scenario. The associated trigger action script would disallow *checkin* of versions that had no *Status* attribute.

Trigger definition:

```
% mktrtype -element -global -preop checkin \
       -exec check_status.sh CheckStatus
```

Trigger action script:

```
#!/bin/sh
#
#      check_status
#

ACCEPT=0
REJECT=1
ATTYPE="Status"

if [ "`cleartool find $CLEARCASE_XPN \
       -version 'attype($ATTYPE)' -print`" ]
then
       exit $REJECT
else
       exit $ACCEPT
fi
```

Scenario: Recording a Released Configuration

Policy #3 “All the versions that went into the building of Release 2 — and only those versions — should be labeled REL2.”

After Release 2 is built and tested, you can create label type *REL2,* using the *mklbtype* command. You can then attach *REL2* as a version label to the appropriate source versions, using the *mklabel* command.

What are the appropriate versions? If Release 2 was built “from the bottom up” in a particular view, you can label the versions selected by that view:

```
% cleartool mklabel -recurse REL2 top-level-directory
```

Alternatively, you can use the configuration records (CRs) of the release’s derived objects to drive the labeling process:

```
       .. on June 17:
% clearmake vega
       .. sometime later, after QA approves the build:
% cleartool mklbtype REL2
       .. and then:
% cleartool mklabel -config vega@@17-Jun.18:05 REL2
```

Using CRs to attach version labels assures accurate and complete labeling, even if users have created new versions since the release build. You need not stop development while quality-assurance and release procedures are performed.

To prevent version label REL2 from being used further, lock the label type:

```
% cleartool lock -lbtype REL2
       ... and then:
% cleartool lock -nusers vobadm -lbtype REL2
```

The -nusers option provides a controlled “escape hatch” — the object will be locked to all users, except the specified ones.

Scenario: Isolating Work on a Bugfix

Policy #4 “Fixes to a past release must be performed in isolation, starting with the exact configuration of versions that went into that release.”

This policy fits perfectly with ClearCase’s *baselevel-plus-changes* development model. First, a *REL2* label must be attached to the release configuration, as described in the preceding scenario. Then, a view configured with the following config spec implements the policy:

```
element * CHECKEDOUT
element * .../rel2_bugfix/LATEST
element * REL2 -mkbranch rel2_bugfix
```

If all fixes are performed in one or views with this configuration, all the changes will be isolated on branches of type *rel2_bugfix*. The -mkbranch clause causes such branches to be created automatically, as needed, when element are checked out.

This config spec selects versions from *rel2_bugfix* branches, where branches of this type exist; it creates such a branch whenever a *REL2* version would be checked out.

Scenario: Isolating All Users from Each Other

Policy #5 “Users should be isolated from changes.”

ClearCase *views* provide isolation from other users’ changes. Using just a few configuration rules, you can specify exactly which changes you wish to see, and which you wish to exclude. Some examples:

- Your own work, plus all the versions that went into the building of Release 2:

  ```
  element * CHECKEDOUT
  element * REL2
  ```

- Your own work on the *main* branch, plus the checked-in versions as of Sunday evening

  ```
  element * CHECKEDOUT
  element * /main/LATEST -time Sunday.18:00
  ```

- Your own work, along with new versions created in the *graphics* directory, plus the versions that went into last night's build

  ```
  element * CHECKEDOUT
  element graphics/* /main/LATEST
  element * -config myprog@@12-Jul.00:30
  ```

- Your own work, plus whatever versions either you (jones) or mary have checked in today, plus the most recent QAed versions:

  ```
  element * CHECKEDOUT
  element * /main/{ created_since(06:00) && \
            ( created_by(jones) || created_by(mary) ) }
  element * /main/{QAed=="TRUE"}
  ```

- You can use the config spec "include" facility to set up standard sets of configuration rules to be incorporated by users into their personal config specs:

  ```
  element * CHECKEDOUT
  element msg.c /main/18
  include /usr/cspecs/rules_for_rel2_maintenance
  ```

Scenario: Freezing Certain Data

Policy #6 "Public header files may not be changed until further notice."

The *lock* command is designed to enforce such "temporary" policies:

- Lock all header files:

  ```
  % cleartool lock src/pub/*.h
  ```

- Lock the headers for all users except Mary and Fred:

  ```
  % cleartool lock -nusers mary,fred src/pub/*.h
  ```

- Lock *all* header files, not just public ones:

  ```
  % cleartool lock -eltype c_header
  ```

- Lock an entire VOB:

  ```
  % cleartool lock -vob /vobs/myproj
  ```

Scenario: Customized Change Notification

Policy #7 "Whenever anyone changes the GUI, mail should be sent to the Doc Group, for a documentation update."

Post-operation triggers are designed to support such notification-oriented policies. First, create a trigger type that sends mail, then attach it to the relevant elements.

Trigger definition:

```
% cleartool mktrtype -nc -element -postop checkin \
       -exec informwriters.sh InformWriters
Created trigger type "InformWriters".
```

Attaching triggers to existing elements:

- Place the trigger on the *inheritance list* of all existing directory elements within the GUI source tree:

  ```
  % cleartool find /vobs/gui_src -type d \
         -exec 'mktrigger -recurse -nattach InformWriters
   $CLEARCASE_PN'
  ```

- Place the trigger on the *attached list* of all existing file elements within the GUI source tree:

Trigger action script:

```
#!/bin/sh
#
#check_status
#

DOC_GROUP="john george"

mail $DOC_GROUP << 'ENDMAIL'
New version of element created: $CLEARCASE_PN
                            by: $CLEARCASE_USER
Comment string:
---------------
$CLEARCASE_COMMENT

ENDMAIL
```

Scenario: Enforcing Quality Standards

Policy #8 “C-language source files may not be checked-in unless they pass our quality metrics.”

Pre-operation triggers can run any user-defined programs, and can cancel the operations that trigger them. You might have your own metrics program, or you might run *lint*(1). Suppose that you have defined an element type, *c_source*, for C language files (*.c).

Trigger definition:

```
% cleartool mktrtype -element -global \
         -eltype c_source \
         -preop checkin \
         -exec 'apply_metrics.sh $CLEARCASE_PN' ApplyMetrics
```

This trigger type *ApplyMetrics* is global — it will fire on a checkin of any element of type *c_source*. (When a new *c_source* element is created, it will be monitored automatically.) If a user attempts to check in a *c_source* file that fails the *apply_metrics.sh* test, the *checkin* operation will be disallowed.

Note: The *apply_metrics.sh* script could read the value of $CLEARCASE_PN from its environment. But having it accept a file name argument provides flexibility: the script can be invoked as a trigger action, but developers can also use it “manually”.

Scenario: Associating Changes with Change Orders

Policy #9 “All changes (ECOs) should be tagged with the number of the bug they are intending to fix.”

Define *ECO* as an integer-valued attribute type. Define a global trigger type, *EcoTrigger*, which fires whenever a new version is created, using a built-in operations to set the *ECO* attribute to a bug number (which is read from the developer’s environment).

Attribute definition:

```
% cleartool mkattype -c "bug number associated with change" \
        -vtype integer ECO
Created attribute type "ECO".
```

Trigger definition:

```
% cleartool mktrtype -element -global -postop checkin \
        -mkattr 'ECO=$ECONUM' EcoTrigger
Created trigger type "EcoTrigger".
```

Alternative Implementation, Using Branches

Create a distinct branch type for each ECO number (for example, *bug253*). Fixes for a bug go onto the appropriate branch, and can be merged into whichever releases wished to incorporate those changes.

Here is the config spec for a view in which to work on ECO #253, a fix to Release 4.3:

```
element * CHECKEDOUT
element * .../bug253/LATEST
element * REL4.3 -mkbranch bug253
```

Using branches allows different bugfix projects to be isolated from one another. All the work can still can be integrated, by using multiple configuration rules:

```
element * CHECKEDOUT
element * .../bug253/LATEST
element * .../bug247/LATEST
element * .../bug239/LATEST
```

Scenario: Requirements Tracing

Policy #10 “Each source code module should have a pointer to an associated design document.”

Requirements tracing applications can be implemented with ClearCase *hyperlinks*, which associate pairs of VOB object. The association should be at the version level (rather than the branch or element level): each version of a source code module should be associated with a particular version of a related design document (see Figure 20-1.)

ClearCase’s hyperlink inheritance feature makes the implementation easy:

- When the source module, *hello.c*, and the design document, *hello_dsn.doc*, are both updated, a new hyperlink is created connecting the two updated versions.
- When either the source module or the design document gets a minor update, no hyperlink-level change is required — the new version automatically *inherits* the hyperlink connection of its predecessor.
- When either the source module or the design document gets a significant update that renders the connection invalid, a *null-ended* hyperlink effectively severs the connection.

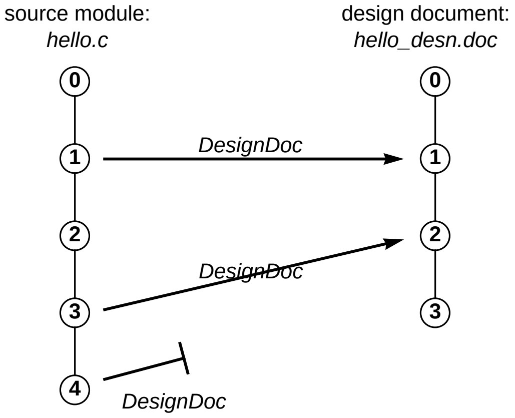
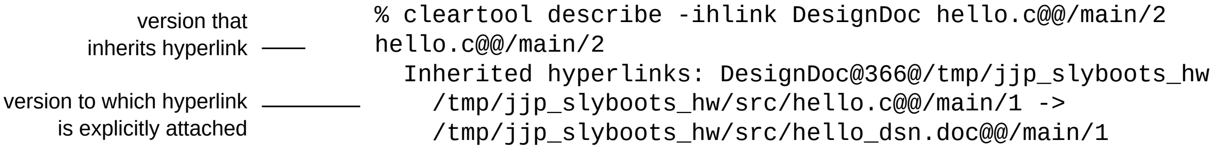


Figure 20-1 Requirements Tracing

Using the *-ihlink* option when describing version */main/*2 of the source module lists the hyperlink inherited from version */main/1* (Figure 20-2):



```
% cleartool describe -ihlink DesignDoc hello.c@@/main/2
hello.c@@/main/2
  Inherited hyperlinks: DesignDoc@366@/tmp/jjp_slyboots_hw
    /tmp/jjp_slyboots_hw/src/hello.c@@/main/1 ->
    /tmp/jjp_slyboots_hw/src/hello_dsn.doc@@/main/1
```

Figure 20-2 Hyperlink Inheritance

Scenario: Change Sets

Policy #11 "A set of files form a group and should be checked out together."

Related files often live in a common directory and can be checked-out with a wildcard:

```
% cleartool checkout -c "Fix rendering bug" graphics/*.c
```

When the files are not co-located, you can implement this policy by using an attribute to annotate elements, and by using a *find* query to supply a list of arguments to *checkout*.

```
% cleartool mkattype -c "group identifier" ElemGroup
Created attribute type "ElemGroup".

% cleartool mkattr ElemGroup '"graphics subsystem"' \
        file1@@ \
        subd/file2@@ \
        ../subd/file3@@ ...
Created attribute "ElemGroup" on "file1@@/main/14".
Created attribute "ElemGroup" on "subd/file2@@/main/8".
Created attribute "ElemGroup" on "../subd/file3@@/main/10".
  ...
< ... whenever checkout is to be performed:

% cleartool checkout -c "Fix rendering bug" \
       ‘cleartool find . -element ' \
       ElemGroup="graphics"' -nxname -print‘
```

Chapter 21

Using ClearCase to Organize and Implement Parallel Development

This chapter shows one way in which ClearCase can be used to organize development work, including both creation of a new release and concurrent maintenance of the previous release.

The approach taken in this chapter is by no means the only one that ClearCase supports. We believe that it addresses typical organizational needs in a straightforward, sensible way.

Project Overview

Release 2.0 development of the *monet* project is to include several kinds of work:

- **patches** — a small number of high-priority bugfixes to Release 1.0
- **minor enhancements** — some commands need new options; some option names need to be shortened (*-recursive* becomes *-r*); some algorithms need performance work
- **major new features** — graphical user interface; many new commands; internationalization support

These three *development streams* can proceed largely in parallel (Figure 21-1), but major dependencies and milestone dates must be considered:

- Several Release 1 patch releases will ship before Release 2.0 is complete.
- The new features will take longer to complete than the minor enhancements.
- Certain new features depend on the minor enhancements.

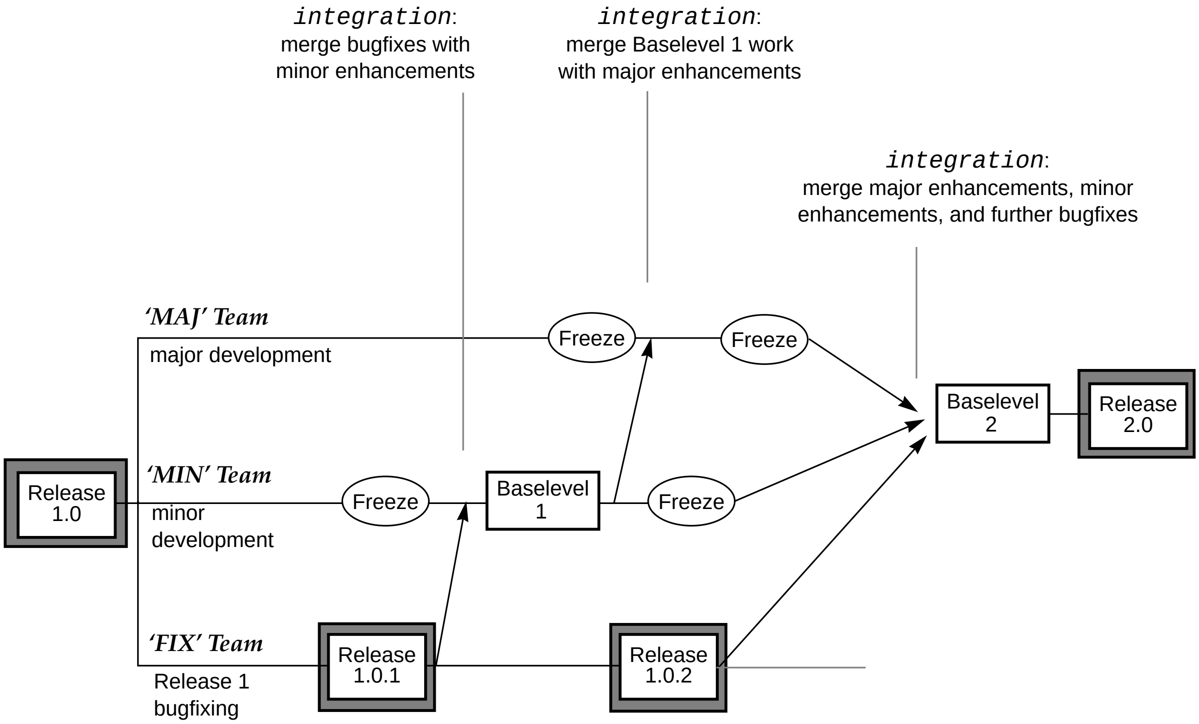


Figure 21-1 Project Plan for Release 2.0 Development

The overall plan adopts a *baselevel-plus-changes* approach. Periodically, developers stop writing new code, and spend some time integrating their work, building, and testing. The result is a *baselevel*: a stable, working version of the application. ClearCase makes it easy to integrate product enhancements, incrementally and frequently. The more frequent the baselevels, the easier the tasks of merging parallel development work and testing the results.

After a baselevel is produced, "real" work resumes; any new development efforts begin with the set of source versions that went into the baselevel build.

With ClearCase, a baselevel can be defined by assigning the same version label (for example, *R2_BL1* for “Release 2, Baselevel 1”) to all the versions that go into, or are produced by, the baselevel build.

The development staff will be divided into three teams, each working on a different development stream: the MAJ team (major enhancements), the MIN team (minor enhancements) and the FIX team (Release 1 bugfixes and patches).

Note: Some developers might belong to multiple teams. Such developers would switch views, depending on their current task

Figure 21-2 shows the development area for the *monet* project. At the beginning of Release 2 development, the most recent versions on the *main* branch are all labeled *R1.0*.

```
/proj/monet/                          (project top-level directory)
src/                                                     (sources)
include/                                           (include files)
lib/                             (archives — sources and .a files)
```

Figure 21-2 Source Tree for *monet* Project

Development Strategy

This section discusses the ClearCase-related issues to be resolved before development begins.

Project Leader and ClearCase Administrator

In most development efforts, the project leader and the system administrator are different people. The leader of this project will be a user named *meister*. The administrator will be the *vobadm* user, introduced in earlier chapters as the creator and owner of the *monet* and *libpub* VOBs.

Use of Branches

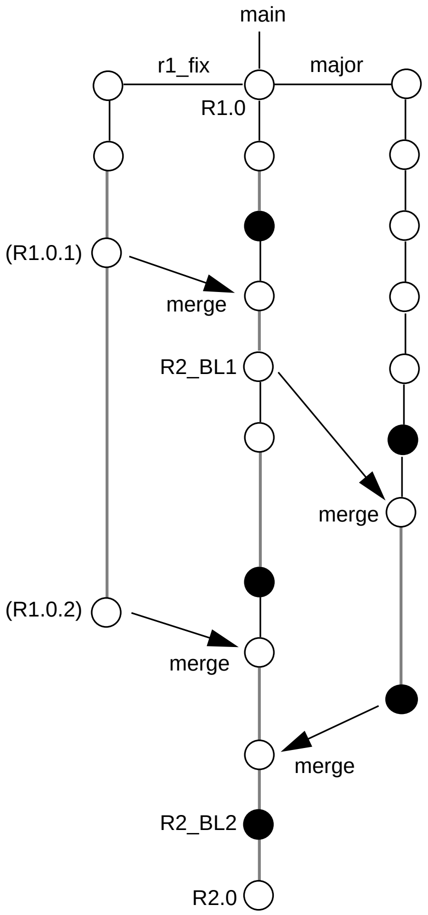
In general, different kinds of work should be performed on different branches. High-priority Release 1 bugfixing, for example, should take place on its own branch. This isolates it from new development. It also enables creation of bugfix (“patch”) releases that do not include any of the Release 2 enhancements — and incompatibilities.

Since the MIN team will produce the first baselevel release essentially on its own, the project leader decides to give the *main* branch to this team. The MAJ team will develop major enhancements on a subbranch, and will not be ready to integrate for a while;[1] the FIX team will perform Release 1 bugfixing on another subbranch, and can integrate its bugfix changes at any time.

Note: The project leader has arranged matters so that the first baselevel will be created from versions on the *main* branches of their elements. This is not a requirement, however — you can create a release that uses versions on any branch, or combination of branches.

Figure 21-3 shows the evolution of a typical element during Release 2 development, and indicates correspondences to the overall project plan (Figure 21-1).

[1] Each major enhancement can be developed on its own subbranch, to make integration and testing more manageable. For simplicity, this chapter discusses use of a single major-enhancements branch.



1. (all branches) Start minor and major enhancements, along with R1.0 bugfixing

2. (main) freeze minor enhancements work

3. (main) merge bugfixes from Release 1.0.1 into minor enhancements

4. (main) Baselevel 1 release

5. (major) freeze major enhancements work

6. (major) Merge Baselevel 1 changes into major enhancements

7. (main) freeze minor enhancements work

8. (main) merge additional bugfixes into minor enhancements

9. (major) freeze major enhancements work

10. (main) merge major enhancements work with minor enhancements work

11. (main) Baselevel 2 release

12. (main) final testing period

13. (main) Release 2.0

Figure 21-3 Development Milestones: Evolution of a Typical Element

Work Environment Planning — Views

Each developer will work in his or her own view, editing programs, building software, testing, and so on. Though the views are separate, they will all be configured with the same config spec. This produces a development environment in which developers on the same team:

- “see” the development tree in the same way
- are totally isolated from work performed on other branches
- are isolated from each other when they *checkout* elements and make changes to them
- share each others’ source code, as versions are checked in on their common branch
- share each others’ derived objects (through *clearmake*’s *wink-in* capability), when appropriate

The MAJ team will work on a branch named *major*, and will use this config spec:

```
element * CHECKEDOUT                                (1)
element * .../major/LATEST                          (2)
element * R1.0 -mkbranch major                      (3)
element * /main/LATEST -mkbranch major              (4)
```

The MIN team will work on the *main* branch, and so can use the default config spec:

```
element * CHECKEDOUT                                (1)
element * /main/LATEST                              (2)
```

The FIX team will work on a branch named *r1_fix*, and will use this config spec:

```
element * CHECKEDOUT                                 (1)
element * .../r1_fix/LATEST                          (2)
element * R1.0 -mkbranch r1_fix                      (3)
element * /main/LATEST -mkbranch r1_fix              (4)
```

For the MAJ and FIX teams, use of the *auto-make-branch* facility in Rules 3 and 4 enforces consistent use of subbranches. It also relieves developers of having to create branches explicitly, and ensures that all branches are created at the version labeled *R1.0*.

Creating Branch Types

The project leader creates the *major* and *r1_fix* branch types required for the config specs listed above:

```
% cleartool mkbrtype -vob /proj/monet r1_fix major
Comments for "r1_fix":
development branch for monet R1 bugfixes
.
Created branch type "r1_fix".
Comments for "major":
development branch for monet R2 major enhancements
.
Created branch type "major".
% cleartool mkbrtype -vob /proj/libpub r1_fix m
(same interaction as above)
```

Note: Since each VOB has its own set of branch types, the branch types must be created separately in the *monet* VOB and the *libpub* VOB. Throughout the remainder of this chapter, we will note where such separate processing is required, without providing details.

Creating Project-Standard Config Specs

To ensure that all developers in a team configure their views the same way, the project leader creates files containing standard config specs:

- */public/config_specs/MAJ* contains the MAJ team's config spec
- */public/config_specs/FIX* contains the FIX team's config spec.

(These standard config spec files are stored in a standard UNIX directory, so that there is no possibility of users getting different versions of them.)

Creating, Configuring, and Registering Views

Each developer creates a view under his or her home directory. For example, developer *allison* enters these commands:

```
% mkdir $HOME/view_store
% cleartool mkview -tag allison_major
 $HOME/view_store/arbmaj.vws
View /net/phobos/usr/people/arb/view_store/arbmaj.vws
 created
It has the following rights:
User : allison  : rwx
Group: mon      : rwx
Other:          : rwx
```

A new view has the default config spec. Thus, developers on the MAJ and FIX teams must reconfigure their views, using the standard file for their team. For example:

```
% cleartool setcs -tag allison_major /public/config_specs/MAJ
```

If the project leader changes the standard file, developers can pick up the changes by entering this command again.

Development Begins

To begin the project, a developer sets his or her properly-configured view, checks out one or more elements, and gets to work. For example, developer *david* on the MAJ team enters:

```
% cleartool setview david_major
% cd /proj/monet/src
% cleartool checkout -nc opt.c prs.c
Created branch "major" from "opt.c" version "/main/6".
Checked out "opt.c" from version "/main/major/0".
Created branch "major" from "prs.c" version "/main/7".
Checked out "prs.c" from version "/main/major/0".
```

The auto-make-branch facility causes each element to be checked out on the *major* branch (see Rule 4 in the MAJ team's config spec, on page 330). If a developer on the MIN team enters this command, the elements are checked out on the *main* branch, with no conflict.

ClearCase is fully compatible with standard UNIX development tools and practices. Thus, developers use their familiar editing, compilation, and debugging tools (including personal scripts and aliases) while working in their views.

Developers use *checkin* periodically to make their work automatically visible to other developers on the same team (that is, others whose views select the most recent version on the team's branch). This allows intra-team integration and testing to proceed throughout the development period.

Techniques for Maintaining Privacy

Although this chapter is intended to illustrate a single approach to organizing development, we briefly note here some techniques that individual developers might use to isolate themselves from changes made by other members of their team.

- **Time rules —** If another team member checks in an incompatible change, a developer can "turn back the clock" to a time before those changes were made.
- **Further subbranching —** A developer can create a private subbranch in one or more elements (for example, */main/major/jackson_wk*) to isolate herself from other team members' new versions of those elements. This requires a config spec change, to "prefer" versions on the */main/major/jackson_wk* branch to versions on the */main/major* branch.
- **Viewing only one's own revisions —** A developer can use a ClearCase query to configure a view which sees only her own revisions to the source tree.

Creating Baselevel 1

The MIN team has implemented and tested the first group of minor enhancements, and the FIX team has produced a patch release, whose versions are labeled R1.0.1. It is time to combine these efforts, to produce *Baselevel 1* of Release 2.0 (Figure 21-4).

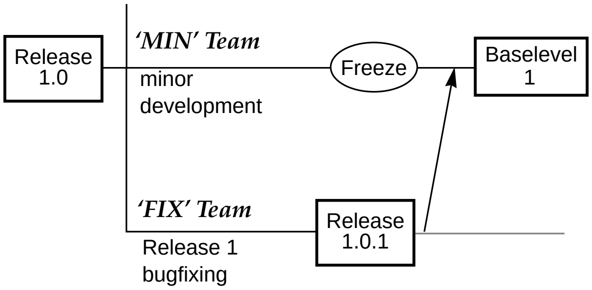


Figure 21-4 Creating Baselevel 1

Merging of Data on Two Branches

The project leader asks the MIN developers to merge the R1.0.1 changes from the *r1_fix* branch into their own branch (*main*). All the changes can be merged with a single invocation of *findmerge*:

```
% cleartool findmerge /proj/libpub /proj/monet/src \
        -fversion .../r1_fix/LATEST -merge -xmerge
  .
  . <lots of output>
  .
```

Integration and Test

After all merges have been performed, the */main/LATEST* versions of certain elements represent a combination of the efforts of the MIN and FIX teams. Members of the MIN team now compile and test the *monet* application to find and fix incompatibilities in the two teams' work.

The developers on the MIN team choose to perform the integration in a single, shared view. The project leader creates the view storage area in a location that is NFS-accessible to all developers' hosts:

```
% umask 2
% mkdir /netwide/public
% cleartool mkview /netwide/public/integrate.vws
View /netwide/public/integrate.vws created
It has the following rights:
User : meister   : rwx
Group: mon      : rwx
Other:          : r-x
```

The umask value of 2 allows all members of the *mon* group to use the view. Each developer registers this view on his or her host, under the name *base1_vu*. For example, suppose that *sol:/netwide/public* is mounted at */public* on all user's workstations:

```
% cleartool mktag /public/integrate.vws base1_vu
```

Since all integration work takes place on the *main* branch, there is no need to change the configuration of the new view from the ClearCase default. MIN team developers set this view (*cleartool setview base1_vu*) and coordinate builds and tests of the *monet* application. Since the developers are sharing a single view, they are careful not to "clobber" each other's view-private files. Any new versions created to fix inconsistencies (and other bugs) go onto the *main* branch.

Labeling Sources

The *monet* application's minor enhancement work and bugfix work are now integrated, and a clean build has been performed in view *base1_vu*. To create the baselevel, the project leader assigns the same version label, *R2_BL1*, to the */main/LATEST* versions of all source elements. He begins by creating an appropriate label type:

```
% cleartool mklbtype -vob /proj/monet -c "Release 2, \
        Baselevel 1" R2_BL1
Created label type "R2_BL1".
```

Note: Also creates and locks label type in *libpub* VOB.

For security, he locks the label type, preventing all developers (except himself) from using it:

```
% cleartool lock -nusers meister -vob /proj/monet -lbtype \
        R2_BL1
Locked label type "R2_BL1".
```

Note: Also creates and locks label type in *libpub* VOB.

Before applying labels, he verifies that all elements are checked in on the *main* branch (checkouts on other branches are still permitted):

```
% cleartool lscheckout -avobs | egrep '(monet|libpub)'
```

Null output indicates that all elements for the *monet* project are checked in. Now, the project leader attaches the *R2_BL1* label to the currently-selected version (*/main/LATEST*) of every element in the two VOBs:

```
% cleartool mklabel -recurse R2_BL1 /proj/monet /proj/libpub
Created label "R2_BL1" on "/proj/monet" version "/main/1".
Created label "R2_BL1" on "/proj/monet/src" version "/main/3".
 <many more label messages>
```

Deleting the Integration View

The view registered as *base1_vu* is no longer needed, so the project leader deletes it:

```
% cleartool rmview -force -tag base1_vu
```

Synchronizing Ongoing Development

The MAJ team has been working undisturbed throughout the Baselevel 1 integration period. At some point following this milestone, the MAJ team pauses to merge the Baselevel 1 changes into its work (Figure 21-5). This provides access to the minor enhancements that the MAJ team needs for further development. It also provides an early opportunity for MAJ team members to determine whether they have made any incompatible changes.

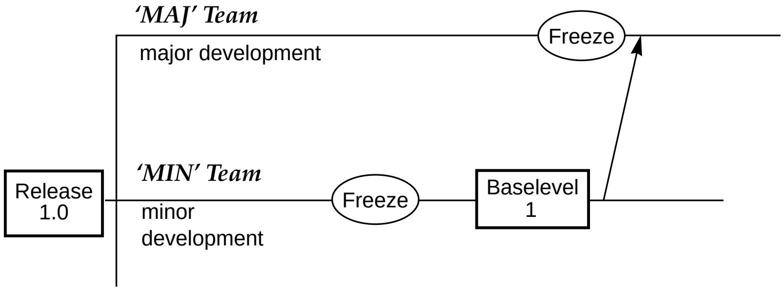


Figure 21-5 Updating Major Enhancements Development

Accordingly, the project leader declares a freeze of major enhancements development.[1] MAJ team members checkin all elements, and verify that the *monet* application builds and runs, making small source changes as necessary. When all such changes have been checked in, the team has a consistent set of */main/major/LATEST* versions.

Preparing to Merge

The project leader makes sure that no element is checked out on the *major* branch:

```
% cleartool lscheckout -avobs | egrep \
        '(monet|libpub).*/major/'
```

Note: Any MAJ team members who wish to continue with non-merge work can create a subbranch at the "frozen" version (or work with a version that is checked out *unreserved*).

The project leader determines which elements need to be merged:

```
% cleartool setview major_vu         (use any MAJ team view)
% cleartool findmerge /proj/monet /proj/libpub -version \
        /main/LATEST -print
  .
```

[1] Developers working on other major enhancements branches might merge at other times, using the same merge procedures described in this section.

```
  . <lots of output>
  .
A 'findmerge' log has been written to
"findmerge.log.04-Feb-94.10:01:23"
```

The *findmerge* log file is in the form of a shell script: it contains a series *cleartool findmerge* commands, each of which will actually perform the required merge for one element:

```
% cat findmerge.log.04-Feb-94.10:01:23
cleartool findmerge -dir /proj/monet/src@@/main/major/3 -fver /main/LATEST -merge
cleartool findmerge /proj/monet/src/opt.c@@/main/major/1 -fver /main/LATEST -merge
cleartool findmerge /proj/monet/src/prs.c@@/main/major/3 -fver /main/LATEST -merge
  .
  .
cleartool findmerge -dir /proj/lubpub/src@@/main/major/1 -fver /main/LATEST -merge
cleartool findmerge /proj/libpub/src/dcanon.c@@/main/major/3 -fver /main/LATEST -merge
cleartool findmerge /proj/libpub/src/lineseq.c@@/main/major/10 -fver /main/LATEST -merge
```

Assigning merge tasks to individual developers amounts to parceling out the contents of this log file.

The project leader then locks the *major* branch, allowing it to be used only by the developers who will be performing merges:

```
% cleartool lock -nusers meister,allison,david,sakai \
        -brt major
Locked branch type "major".
```

Note: Also locks branch type in *libpub* VOB.

Performing Merges

Since the MAJ team is not immediately heading for a baselevel, it is not essential that all merges be performed (and the results tested) in a shared view. Each MAJ developer can continue working in his regular view.

Periodically, the project leader sends an excerpt from the *findmerge* log to an individual developer, who executes the commands and monitors the results. (The developer can sends the resulting log files back to the project leader, as confirmation of the merge activity.)

A merged version of an element will include changes from three development streams: Release 1 bugfixing, minor enhancements, and major enhancements (Figure 21-6).

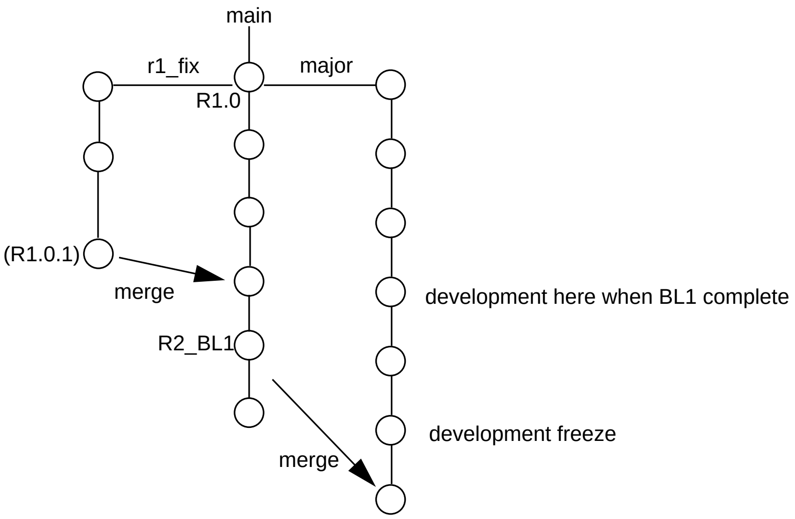


Figure 21-6 Merging Baselevel 1 Changes into the 'major' Branch

The project leader verifies that no more merges need to be performed, by entering a *findmerge* command with the *-whynot* option:

```
% cleartool findmerge /proj/monet /proj/libpub -version \
        /main/LATEST -whynot -print
  .
  .
No merge "/proj/monet/src" [/main/major/4 already merged from /main/3]
No merge "/proj/monet/src/opt.c" [/main/major/2 already merged from /main/12]
   ..
```

He ends the merge period by removing the lock on the *major* branch:

```
% cleartool unlock -brtype major
Unlocked branch type "major".
```

Note: Also locks branch type in *libpub* VOB.

Creating Baselevel 2

The MIN team has reached its second development freeze, and the MAJ team will do so shortly (Figure 21-7). Baselevel 2 will integrate all three development streams, thus requiring two sets of merges:

- Bugfix changes from the most recent patch release (versions labeled *R1.0.2*) are to be merged to the *main* branch.
- Major enhancements are to be merged from the major branch to the *main* branch. (This is the opposite direction from the merges described in "Synchronizing Ongoing Development" on page 336.)

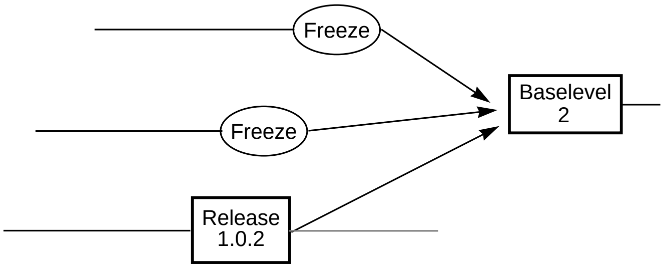


Figure 21-7 Baselevel 2

ClearCase supports multi-way merges, so both the bugfix changes and the major enhancements could be merged into the *main* branch at the same time. In general, though, it is easier to verify the results of two-way merges.

Merging from the *r1_fix* Branch

The first set of merges is virtually identical to those described in "Merging of Data on Two Branches" on page 334. Thus, we omit the details here.

Preparing to Merge from the *major* Branch

After the integration of the *r1_fix* branch is completed, the project leader gets ready to manage the merges from the major branch.[1] These merges will be performed in a tightly-controlled environment, because of the approaching Baselevel 2 milestone, and because the major branch is to be abandoned.

The project leader verifies that everything is checked in on *both* the *main* branch and *major* branches:

```
% cleartool lscheckout -recurse /proj/monet \
        /proj/libpub | grep -v 'r1_fix'
%
```

This command filters out checkouts on the *r1_fix* branch, which are still permitted. Thus, null output from this command indicates that no element is checked-out on either its *main* branch or its *major* branch.

Next, the project leader determines which elements require merges:

```
% cleartool setview minor_vu(use any MIN team view)
% cleartool findmerge /proj/monet /proj/libpub \
        -version .../major/LATEST -print
   .
   . <lots of output>
   .
A 'findmerge' log has been written to
"findmerge.log.26-Feb-94.19:18:14"
```

All development on the major branch is to cease after this baselevel. Thus, the project leader locks the *major* branch to all users, except those who will be performing the merges; this will allow ClearCase to record the merges with a hyperlink of type *Merge*:

```
% cleartool lock -brtype -nusers allison,david major
Locked branch type "major".
```

Note: Also locks branch type in *libpub* VOB.

The *main* branch will be used for Baselevel 2 integration by a small group of developers. Accordingly, the project leader had *vobadm* lock the *main* branch to everyone else:

```
% cleartool lock -nusers meister,allison,david,sakai \
        -brtype main
Locked branch type "main".
```

[1] It is probably more realistic to build and verify the application, then apply version labels before proceeding to the next merge.

Note: Also locks branch type in *libpub* VOB.

(Only the VOB owner or *root* can lock the *main* branch.)

Performing the Merges from the 'major' Branch

Since the *main* branch is the destination of the merges, developers work in a view with the default config spec. The situation is similar to that on described in <Emphasis>Preparing to Merge on page 337; this time, the merges are to take place in the opposite direction: from the major branch to the main branch. Accordingly, the *findmerge* command is very similar:

```
% cleartool findmerge /proj/monet /proj/libpub
        -version /main/major/LATEST -print
  .
  . <lots of output>
  .
A 'findmerge' log has been written to
"findmerge.log.23-Mar-94.14:11:53"
```

After checkin, a typical merged element appears as in Figure 21-8.

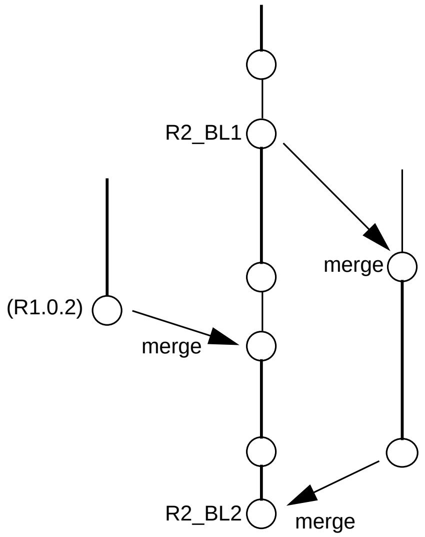


Figure 21-8 Element Structure after the Pre-Baselevel-2 Merge

Decommissioning the 'major' Branch

After all data has been merged to the *main* branch, no further development is to take place on the major branch. At that time, the project leader enforces this policy by *obsoleting* the *major* branch:

```
% cleartool unlock -brtype major
Unlocked branch type "major".
% cleartool lock -obsolete -brtype major
Locked branch type "major".
```

Note: Also locks branch type in *libpub* VOB.

Integration and Test

Structurally, the Baselevel 2 integration-and-test phase is identical to the one for Baselevel 1 (see “Integration and Test” on page 334). At the end of the integration period, the project leader attaches version label *R2_BL2* to the */main/LATEST* version of each element in the *monet* and *libpub* VOBs (the Baselevel 1 version label was *R2_BL1*).

Final Validation - Creating Release 2.0

Baselevel 2 has been released internally, and further testing has flushed out only minor bugs. These bugs have been fixed by creating new versions on the *main* branch (Figure 21-9).

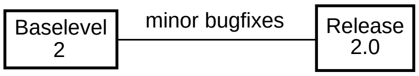


Figure 21-9 Final Test and Release

Prior to customer shipment, the *monet* application will go through a tightly-controlled validation phase:

- All editing, building, and testing will be restricted to a single, shared view.
- All builds will be performed from sources with a particular version label (*R2.0*), and only the project leader will have permission to make changes involving that label.
- Only high-priority bugs will be fixed, using this procedure:
  - The project leader authorizes a particular developer to fix the bug, by granting her permission to create new versions (on the main branch).
  - The developer’s *checkin* activity is automatically tracked by a ClearCase trigger.
  - After the bug is fixed, the project leader moves the *R2.0* version label to the fixed version, and revokes the developer’s permission to create new versions.

Labeling Sources

The project leader labels the */main/LATEST* versions throughout the entire *monet* development tree:

```
% cleartool setview meister_vu
                        (set a view with default config spec)
% cleartool mklbtype -c "Release 2.0" R2.0
                                        (create a label type)

% cleartool lock -nusers meister -lbtype R2.0
                               (restrict usage of label type)
Locked label type "R2.0".
                          (label the entire development tree)
% cleartool mklabel -recurse R2.0 /proj/monet
 <many label messages>
```

Note: Also locks branch type and labels versions in *libpub* VOB.

During the final test phase, he will move the label forward, using *mklabel* –replace, if any new versions are created.

Further Restricting Use of the *main* Branch

At this point, use of the *main* branch is restricted to a few users: those who performed the merges and integration leading up to Baselevel 2 (see page 341). Now, the project leader has *vobadm* close down the *main* branch to everyone except himself, *meister*:

```
% cleartool unlock -brtype main
Unlocked branch type "main".
% cleartool lock -brtype -nusers meister main
Locked branch type "main".
```

The *main* branch will be opened only for last-minute bugfixes (see “Implementing a Final Bugfix” on page 347.)

Setting Up the Test View

The project leader creates a new shared view, *r2_vu*, and gives it a special, one-rule config spec:

```
% umask 2
% cleartool mkview -tag r2_vu /public/integrate_r2.vws
% cleartool edcs -tag r2_vu
```

```
 <edit config spec to contain ... >
element * R2.0                                     (1)
```

This config spec guarantees that only properly-labeled versions will be included in final-validation builds.

Setting Up the Trigger to Monitor Bugfixing

The project leader places a trigger on all elements in the *monet* VOB; the trigger will fire whenever a new version of any element is checked in. First, he creates an executable shell script in a standard UNIX directory (not within a VOB):

```
% vi /public/ccase_triggers/notify_leader
```

```
 <edit script to contain ... >

#!/bin/sh
PATH=/bin:/usr/ucb

mail meister <<!
Subject: Checkin $CLEARCASE_PN by $CLEARCASE_USER
$CLEARCASE_XPN
Checked in by $CLEARCASE_USER.

Comments:
$CLEARCASE_COMMENT
!

% chmod +x /public/ccase_triggers/notify_leader
```

Then, he has *vobadm* create a global element trigger type in the *monet* and *libpub* VOBs, specifying the script as the trigger action:

```
% cleartool mktrtype -nc -vob /proj/monet -element -global \
      -postop checkin -brtype main \
      -exec '/public/ccase_triggers/notify_leader' r2_checkin
Created trigger type "r2_checkin".
```

Note: Also creates trigger type in *libpub* VOB.

(Only the VOB owner or *root* can create trigger types.)

Implementing a Final Bugfix

This section demonstrates the final validation environment in action. Developer *allison* discovers a serious bug, and requests permission to fix it. The project leader grants her permission to create new versions on the *main* branch, by having *vobadm* enter these commands.

```
% cleartool unlock -brtype main
Unlocked branch type "main".
% cleartool lock -nusers allison,meister -brtype main
Locked branch type "main".
```

The developer fixes the bug in a view with the default config spec, and tests the fix there. This involves creation of two new versions of element *prs.c* and one new version of element *opt.c*. On each *checkin*, the *r2_checkin* trigger automatically sends mail to the project leader. For example:

```
Subject: Checkin /proj/monet/src/opt.c by allison
/proj/monet/src/opt.c@@/main/9
Checked in by allison.

Comments:
fixed bug #459: made buffer larger
```

When regression tests verify that the bug has been fixed, the project leader revokes *allison*'s permission to create new versions. Once again, the commands are executed by *vobadm*:

```
% cleartool unlock -brtype main
Unlocked branch type "main".
% cleartool lock -brtype -nusers meister main
Locked branch type "main".
```

Then, the project leader moves the version labels to the new versions of *prs.c* and *opt.c*, as indicated in the mail messages. For example:

```
% cleartool mklabel -replace R2.0 \
       /proj/monet/src/opt.c@@/main/9
Moved label "R2.0" on "prs.c" from version "/main/8" to
"/main/9".
```

Rebuilding from Labels

After the labels have been moved, developers rebuild the *monet* application once again, to prove that a good build can be performed using only those versions labeled *R2.0.*

Wrapping Up

When the final build in the *r2_vu* passes the final test, Release 2.0 of *monet* is ready to ship. After the distribution medium has been created, from derived objects in the *r2_vu*, the project leader has *vobadm* clean up and get ready for the next release:

- *vobadm* removes the checkin triggers from all elements by deleting the global element trigger type:

  ```
  % cleartool rmtype -trtype r2_checkin
  Removed trigger type "r2_checkin".
  ```

- *vobadm* reopens the *main* branch to everyone:

  ```
  % cleartool unlock -brtype main
  Unlocked branch type "main".
  ```

Index

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2369-001.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389