CASEVision™/ClearCase Concepts Guide

Document Number 007-1612-020

CONTRIBUTORS

Written by John Posner and Jeffery Block
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 Concepts Guide
Document Number 007-1612-020

Contents

Figures

Preface

CASEVision™/ClearCase is a version control and configuration management system, designed for development teams working in a local area network. Versions of ClearCase running on different hardware platforms are fully compatible.

This manual is for users who are new to ClearCase. Chapter 1, “Introduction to ClearCase” provides a broad overview, showing how ClearCase’s components work together to provide a complete configuration management solution. Subsequent chapters describe the components in greater detail.

Typographical Conventions

This manual uses the following typographical conventions:

- *This font* is used for names of ClearCase commands, names of UNIX commands, and glossary terms. It is also used for file names and user-supplied names.
- This font is used for examples. Where user input needs to be distinguished from program output, this font is used for user input.
- When full or partial command syntax appears in a paragraph, this font is used.
- Nonprinting characters and keyboard characters are indicated with this font, and enclosed in angle brackets: <EOF>, <Return>.

Chapter 1

Introduction to ClearCase

ClearCase is a comprehensive software *configuration management* system. It manages multiple variants of evolving software systems, tracks which versions were used in software builds, performs builds of individual programs or entire releases according to user-defined version specifications, and enforces site-specific development policies.

These capabilities enable ClearCase to address the critical requirements of organizations that produce and release software:

- **Effective development** — ClearCase enables users to work efficiently, allowing them to fine-tune the balance between *sharing* each other's work and *isolating* themselves from destabilizing changes. ClearCase automatically manages the sharing of both source files and the files produced by software builds.
- **Effective management** — ClearCase tracks the software build process, so that users can determine *what* was built, and *how* it was built. Further, ClearCase can instantly recreate the source base from which a software system was built, allowing it to be rebuilt, debugged, and updated — all without interfering with other programming work.
- **Enforcement of development policies** — ClearCase enables project administrators to define development policies and procedures, and to automate their enforcement.

ClearCase Data Structures

Figure 1-1 shows a development environment managed by ClearCase. At its heart is a permanent, secure data repository. It contains data that is shared by all users: this includes current and historical versions of source files, along with *derived objects* built from the sources by compilers, linkers, and so on. In addition, the repository stores detailed "accounting" data on the development process itself: who created a particular version (and when, and why), what versions of sources went into a particular build, and other relevant information.

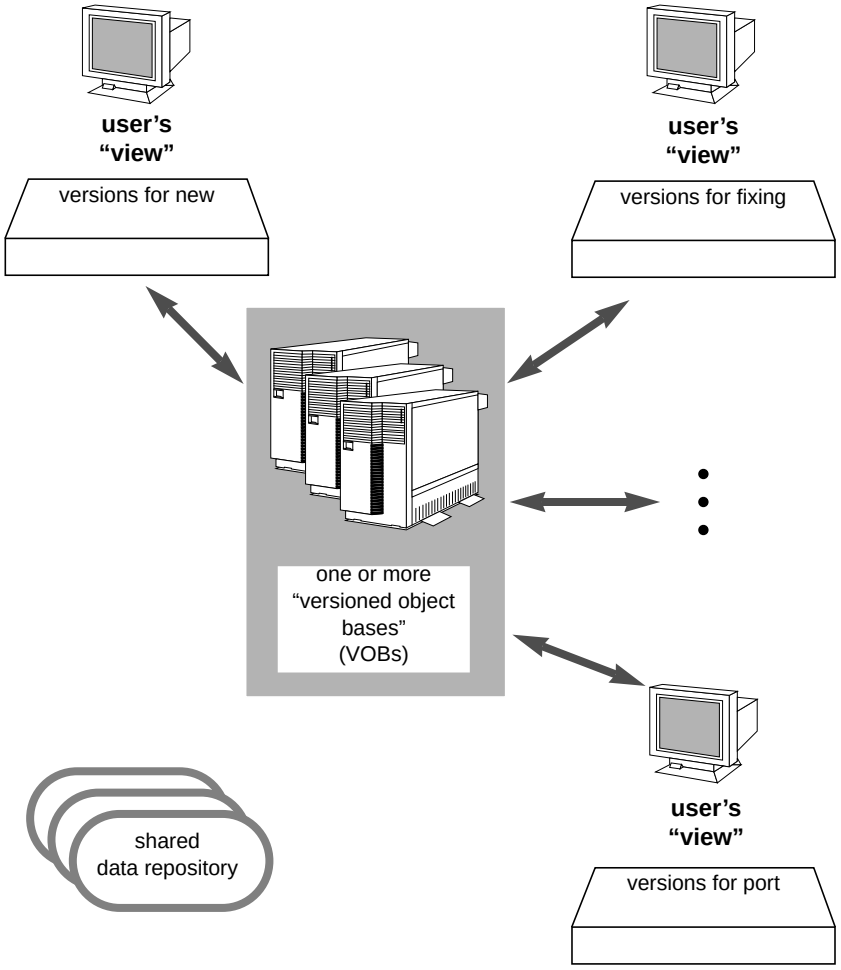


Figure 1-1 ClearCase Development Environment

Only ClearCase commands can modify the permanent data repository. This ensures orderly evolution of the repository and minimizes the likelihood of accidental damage or malicious destruction.

Conceptually, the data repository is a globally accessible, central resource. The implementation, however, is modular: each source (sub)tree can be a separate *versioned object base (VOB)*. VOBs can be distributed throughout a local area network, accessed independently or linked into a single logical tree. To system administrators, modularity means flexibility; it facilitates load-balancing in the short term, and enables easy expansion of the data repository over the long term.

Note: The repository can even be distributed over a wide-area network, or to sites that have no live data connection at all. This capability is implemented by Atria's MultiSite product, available separately.

Views and Transparent Access

As Figure 1-1 illustrates, users access the ClearCase data repository through *views*. A view is a software development work environment that is similar to — but greatly improves on — a traditional "development sandbox". Each view can easily be configured to access just the right source data from the central repository:

- the up-to-date versions for development of the next major release
- the versions that went into the port of Release *X.Y* to hardware architecture *Z*
- the versions being used to fix bug #*ABC* in Release *D.E*

A view is an isolated "virtual workspace", which provides dynamic access to the entire data repository. The changes being made to a source file in a particular view are invisible to other views; software builds performed in a view do not disturb the work taking place in other views.

Working in views, ClearCase users access version-controlled data using standard pathnames and their accustomed commands and programs. The view accesses the appropriate data automatically and transparently.

A view's isolation does not render it inaccessible; a view can be accessed from any host in the local area network. For example, a *distributed build* involves execution of build scripts on several hosts at once, all in the same view. Similarly, a view can be shared by several users, working on a single host or on multiple hosts. One user might "peek" into another's view, just to see what changes are being made to a particular file.

Version Control

The most basic requirement for a software *configuration management* system is *version control* — maintaining multiple versions of software development objects. Traditional version-control systems handle text files only; ClearCase manages *all* software development objects: *any* kind of file, and directories and links, as well.

Versions of text files are stored efficiently as *deltas*, much like SCCS or RCS versions. Versions of non-text files are also stored efficiently, using data compression. Version control of directories enables the tracking of changes to the *organization* of the source code base, which are just as important as changes to the contents of individual files. Such changes include creation of new files, renaming of files, and even major source tree "cleanups".

Versioned Object Bases (VOBs)

ClearCase development data is organized into any number of *versioned object bases* (*VOBs*). Each VOB provides permanent storage for all the historical versions of all the source objects in a particular directory tree. As seen through a ClearCase *view*, a VOB seems to *be* a standard directory tree — the "right" versions of the development objects appear, and all other versions are hidden (Figure 1-2). How this works is described in section "Environment Management" on page 10.

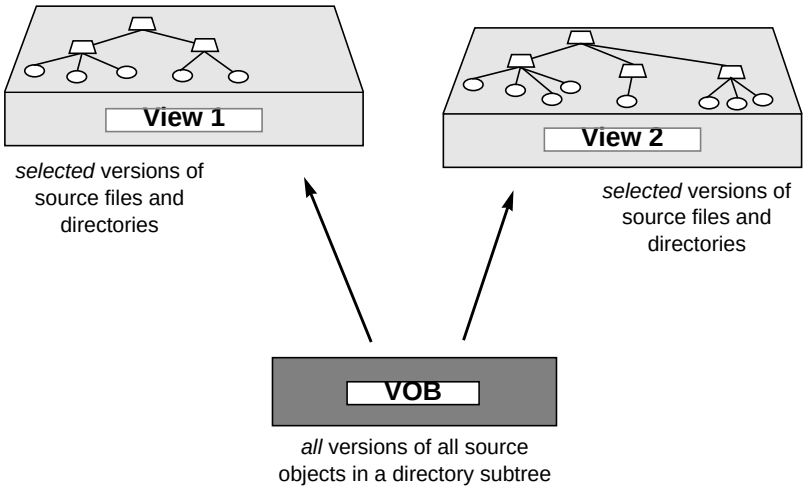


Figure 1-2 VOBs Appear in Views as Ordinary Directory Trees

A version-controlled object in a VOB is called an *element*; its versions are organized into a *version tree* structure, with *branches* and *subbranches* (Figure 1-3. As this figure shows, branches have user-defined names, typically chosen to indicate their role in the development process. All versions have integer ID numbers; important versions can be assigned *version labels*, to indicate development milestones — for example, a product release.

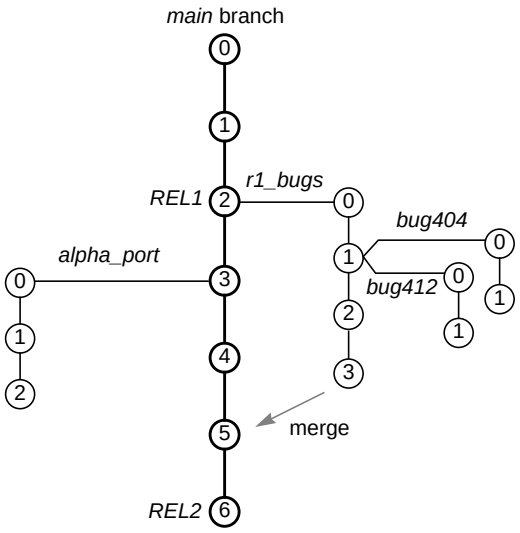


Figure 1-3 Version Tree of an Individual Element

Parallel Development

Each (sub)branch in an element's version tree represents an independent "line of development". This enables *parallel development* — creating and maintaining multiple variants of a software system concurrently. Creation of a variant might be a major project (porting an application to a new platform), or a minor detour (fixing a bug; creating a "special release" for an important customer).

The overall ClearCase parallel development strategy is as follows:

- **Establish a baselevel** — Development work on a new variant begins with a consistent set of source versions, identified (for example) by a common version label.
- **Use dedicated branches** — All changes for a particular variant are made on newly-created branches with a common name.

- **Isolate changes in views** — Development work for a particular variant takes place in one or more views that are configured to "see" the versions on the dedicated branches.

For example, changes to several source files might be required to fix bug #819, which was reported in Release 2.6. For each file element, the changes are made on a new branch (named *fix819*), created at the "baseline" version (labeled *RLS2.6*). The view in which a user works to fix the bug sees the *fix819* branch versions, or else "falls back" to the baseline *RLS2.6* version ("View 1" in Figure 1-4). For contrast, this figure also illustrates another view, configured to select different versions of the same file elements.

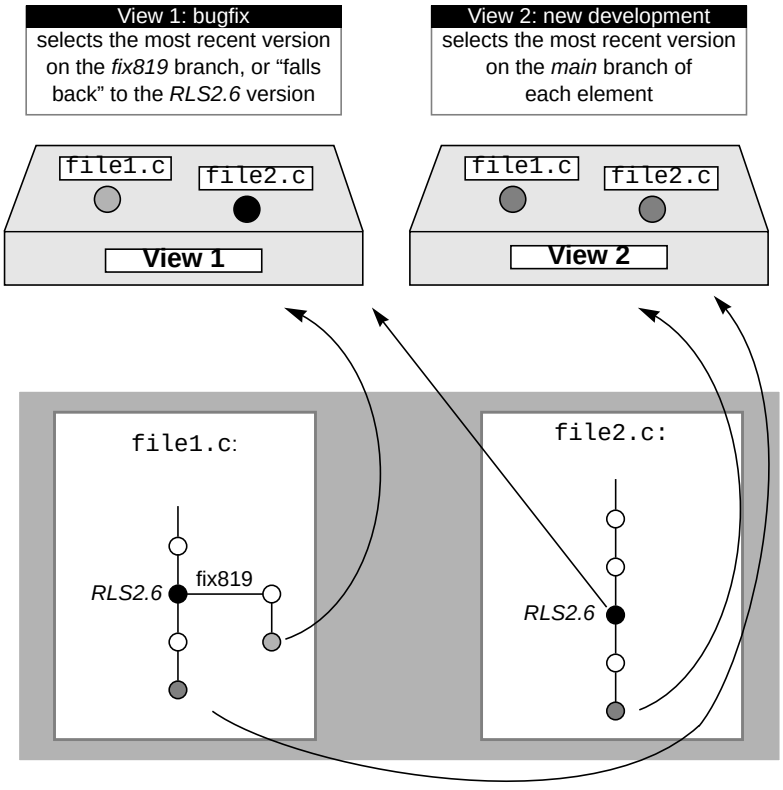


Figure 1-4 Parallel Development

This strategy enables any number of views — and thus any number of development projects — to be active concurrently. All the views access the required source versions from the shared data repository.

Merging Branches

There is an additional important aspect of the ClearCase parallel development strategy. Work performed on subbranches should periodically be reintegrated (*merged*) into the *main* branch, the principal line of development. ClearCase includes tools that automate this process.

Extended Namespace

Most of the time, a user needs just the one version of an element that appears in his view. In some situations, however, he needs convenient access to other versions. Examples include merging the changes made on a subbranch into the *main* branch, and searching *all* the versions of an element for an old phrasing of an error message.

ClearCase makes access to historical versions easy, by extending the standard file/directory namespace. In essence, the entire version tree of every element is embedded under its standard pathname. Most of the time, the version tree remains hidden; but special *version-extended pathnames* allow any program to access any (or all) of an element's versions (Figure 1-5).

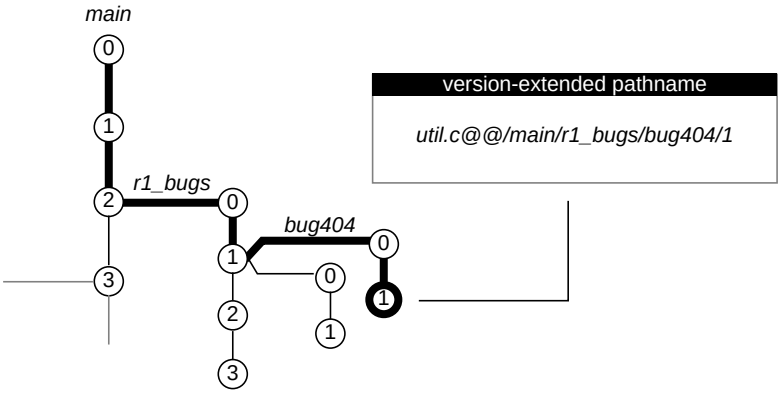


Figure 1-5 Version-Extended Pathnames

Environment Management

A software *configuration management* system must provide a flexible, efficient collection of "development environments", or "workspaces", in which users can do their work. ClearCase *views* fulfill this role, providing these services:

- access to the appropriate versions of development sources
- private data storage for use in day-to-day development tasks
- isolation from activity taking place in other views
- automatic and user-requested facilities for sharing data with other views, when appropriate

Views and Transparent Access

As described in "ClearCase Data Structures" on page 2, a *view* directly accesses the version-controlled elements in the permanent, shared data repository. There is no need to copy the versions required for a particular project to a view; instead, the correct versions are accessed dynamically. A particular version of each element is selected according to user-specified rules in the view's *config spec* ("configuration specification"): a file element appears to be an ordinary file; a directory element appears to be an ordinary directory.

The overall effect of automatic version selection is *transparency*: the version-control system becomes invisible, so that a VOB appears to be a standard directory tree (Figure 1-6). This key feature enables ClearCase to work smoothly with standard system software, third-party commercial applications, and a development team's "home-grown" software tools. Users do not have to discard their accustomed ways of working, or their existing tools. For example, such standard programs as *grep*, *more*, *ls*, and *cc* will work the same way on ClearCase data as on non-ClearCase data.

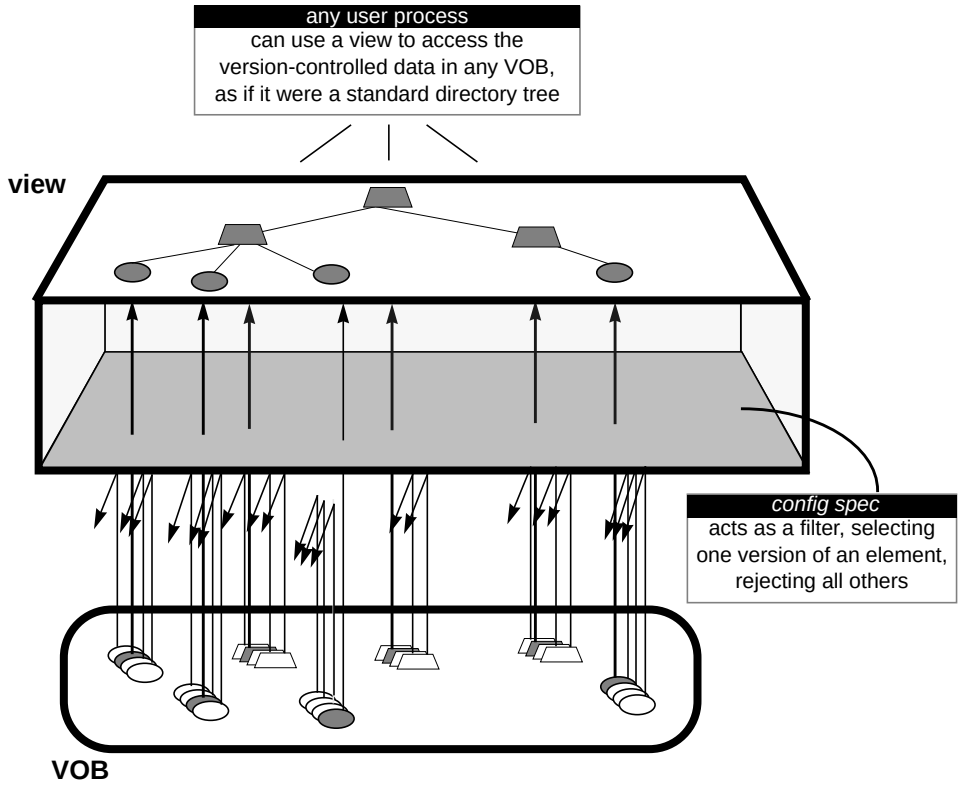


Figure 1-6 Version Selection by a View

Views are dynamic — *config spec* rules are continually reevaluated. This means that a view is open-ended; as new data is added to the central repository, it is immediately accessible to all views. It also means that a view's configuration can be instantly modified — for example, to "shut out" a recent destabilizing change to the repository.

The View as Isolated Workspace

In addition to providing automatic version selection, a view provides an isolated workspace in which users perform such tasks as editing source files, compiling and linking object modules, and testing executables. Any number of users can work in the same source directory, building the same programs; they will never interfere with each other, as long as they work in different views. Conversely, two or more users working together closely can share a single view.

A view's isolation is achieved, in part, by its having a *private storage area*. This area is principally used to store:

- source files that are being edited by the user(s) working in the view
- *derived objects* produced by software builds—object modules, executables, and so on

This area is also used for incidentals, such as text-editor backup files and cut-and-paste temporary files.

The 'Virtual Workspace'

All the files in view-private storage appear to be in the appropriate VOB directory, even though they are (typically) stored on the user's workstation, rather than in the central data repository. That is, the view combines objects in view-private storage with objects in the shared repository to form an isolated "virtual workspace".

Figure 1-7 shows a listing of a VOB directory, as it appears in the "virtual workspace" created by a view.

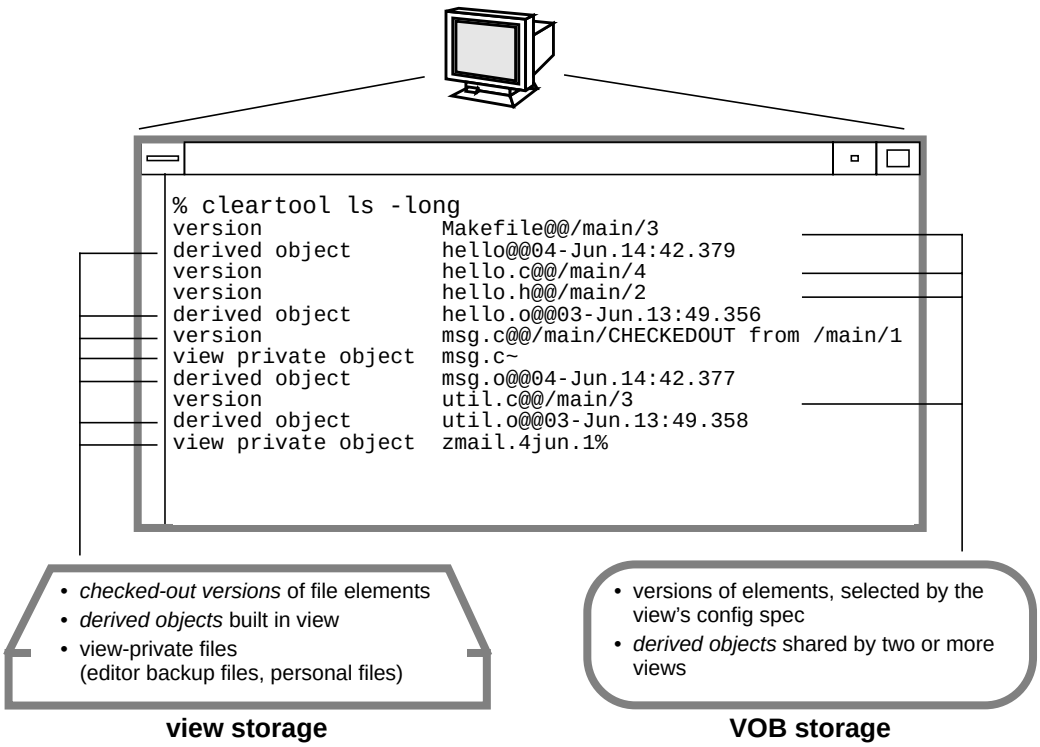


Figure 1-7 View Development Environment

Example: Editing Source Files in a View

A user, working in a view, enters a *checkout* command to make a source file editable (Figure 1-8). This seems to change a file element in the data repository from read-only to read-write. In reality, ClearCase copies the read-only repository version to a writable file in the view's private storage area. This writable file, the *checked-out version*, appears in the view at the same pathname as the file element; the view accesses this editable, checked-out version until the user enters a *checkin* command, which updates the repository and deletes the view-private file.

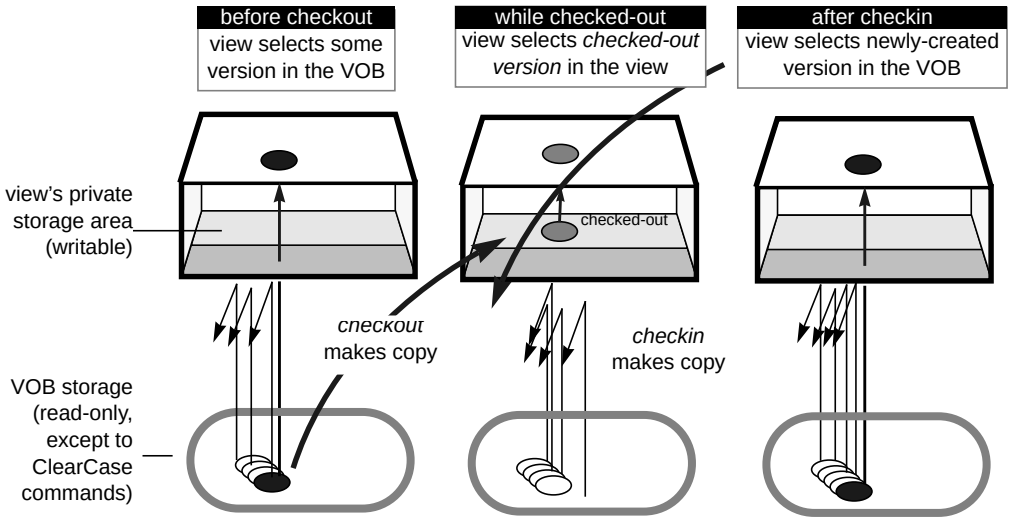


Figure 1-8 Checkout/Checkin and View-Private Storage

The View as Shared Resource

Subject to access permissions, a view is a shared resource, available on all hosts in the local area network. Each view is globally accessible through a simple name, its *view-tag*. An individual user uses the same view on several hosts during a distributed software build. During an integration period, several users might share a single view, each using his or her own workstation.

View-Extended Naming

A user sometimes needs to compare (or otherwise manipulate) the data seen through two or more views. Access to multiple views in a single command is made possible through *view-extended naming*. "Extended Namespace" on page 9 describes how ClearCase extends the file system "downward" by embedding an element's entire version tree under its pathname. Similarly, ClearCase extends the file system "upward" by creating a virtual super-root directory (the *viewroot*) which conceptually contains all active views. A

view-extended pathname accesses the version of a particular element that is seen by a particular view (Figure 1-9).

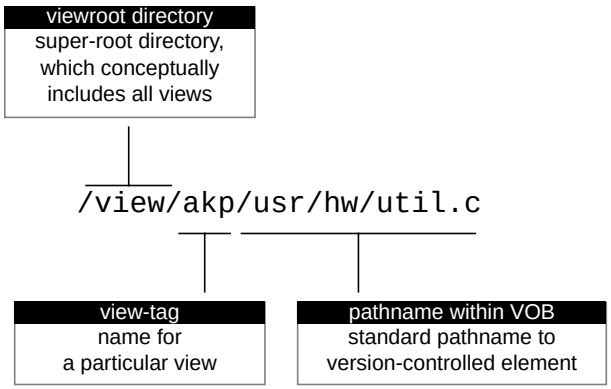


Figure 1-9 View-Extended Pathname

Build Management

ClearCase supports *makefile-based* building of software systems. This means users can continue to build systems using their accustomed procedures. They can even use the same tools — for example, a host's system-supplied *make* program or a third-party build utility. ClearCase's own build program, *clearmake*, provides compatibility with other *make* variants, along with powerful enhancements.

Build Auditing

clearmake's fundamental enhancement is *build auditing*: monitoring of file system activity during a software build, at the system-call level. *clearmake* implements this capability by working with ClearCase's virtual file system extension, the *multiversion file system* (MVFS).

Build auditing enables complete and automatic documentation of software builds. A build's "bill-of-materials" and "assembly instructions" are preserved in *configuration records* (Figure 1-10)

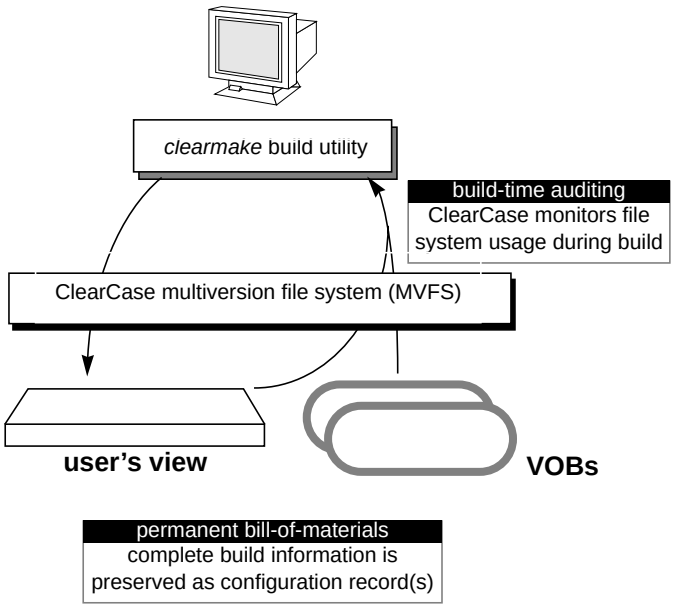


Figure 1-10 Build Auditing and Configuration Records

The files produced by a build (object modules, executables, libraries, and so on) are cataloged in the central repository as *derived objects*.

Users can compare different builds of the same program — different derived objects built at the same pathname — through their configuration records. Moreover, *clearmake* automatically uses configuration records during subsequent builds, to implement additional build enhancements.

Build Avoidance

Standard *make* programs support incremental building of software systems through *build avoidance*. A "make" of an entire system actually rebuilds only those components that *need* to be rebuilt, because they are out-of-date with respect to the corresponding source files.

clearmake's build-avoidance scheme is more sophisticated, and specifically designed for use in parallel development situations. Typically, each user modifies only a few source files at a time to produce a variant of a software system. If the *same* version of a particular source file is used by several programmers, it would be compiled to exactly the same object module in each of their views. *clearmake* uses configuration records to detect such situations; instead of performing redundant builds, it causes a single derived object to be shared among the views (Figure 1-11). This facility, termed *wink-in*, saves both disk storage and build time.

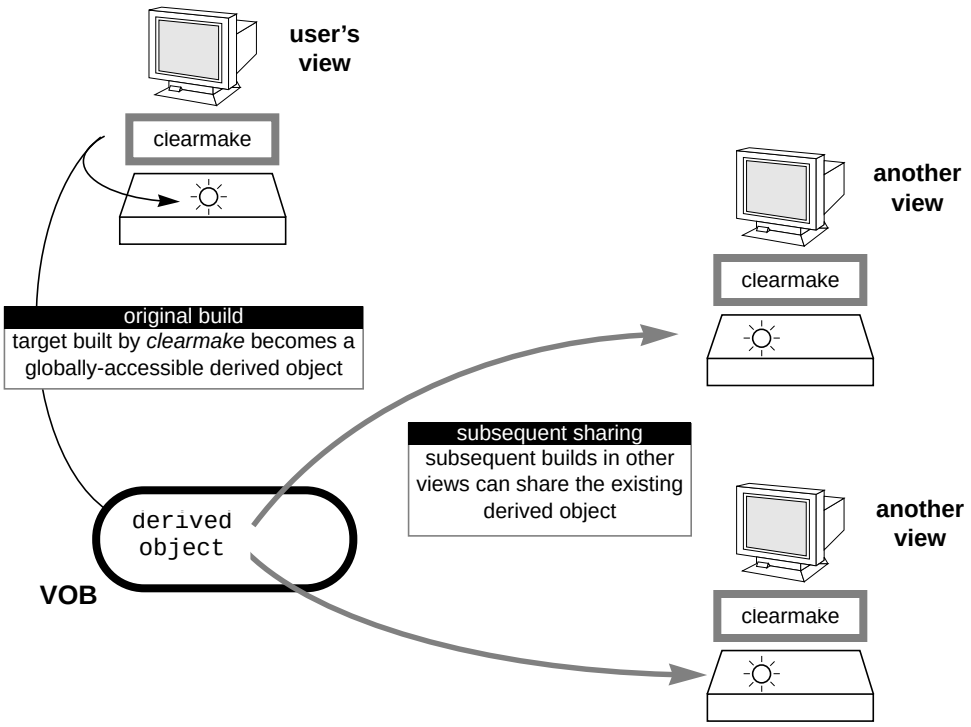


Figure 1-11 Derived Object Sharing

Automatic Dependency Detection

Configuration records enable automatic checking of source dependencies as part of build avoidance. All such dependencies (for example, on C-language header files) are logged in a build's configuration record, whether or not they are explicitly declared in a *makefile*.

Build Script Checking

Configuration records also enable the build-avoidance algorithm to include checking of a target's build script. If the build script has changed, *clearmake* rebuilds the target. Many *make* variants ignore build-script changes, and thus fail to perform a rebuild when it is actually required.

Building on Remote Hosts

clearmake supports efficient building of large software systems through its ability to execute multiple build scripts in parallel, and to distribute build script execution to a group of hosts in the local area network. A tunable load-balancing scheme optimizes uses of network resources during a *distributed build*.

A build can even take place on a host where ClearCase itself has not been installed. This feature is particularly valuable for organizations that support multiple hardware/software architectures, including some not supported by ClearCase.

Process Control

ClearCase provides mechanisms for monitoring and controlling the development process itself. ClearCase does not attempt to impose its own particular policies or procedures — instead, it includes a flexible, powerful toolset, which administrators can use to implement an organization's existing policies.

Process management comprises several functional areas, which ClearCase addresses both with static mechanisms (control structures) and dynamic mechanisms (procedures). Some of the mechanisms are completely automatic; others are created and/or controlled by users and administrators.

Information Capture and Retrieval

ClearCase automatically logs each change to the data repository in the form of an *event record*, providing an audit trail of development activities. ClearCase includes commands for creating reports based on these records, with many selection and filtering options. Such reports can include the *event records* for a single version-controlled object, for any user-specified set of objects, or for entire VOBs. *Event records* can be retained indefinitely, or can be "scrubbed" selectively on a periodic basis, in order to conserve disk space.

When a user merges the changes made on one branch of an element into another branch, ClearCase automatically writes an *event record*, and also connects the merged versions with a *merge arrow* (see Figure 1-3). This makes it easy to track (and often, to fully automate) the process of integrating work performed on subbranches back into the main line of development.

Merge arrows are a special case of a more general mechanism for indicating a relationship between two objects. Any two objects in the central repository can be connected with a logical arrow called a *hyperlink*. This capability addresses such process-control needs as *requirements tracing*.

Meta-Data Annotations

To supplement the information automatically captured by ClearCase, users can explicitly annotate file system objects. Such annotations are termed *meta-data*. The hyperlinks and merge arrows discussed just above are one form of meta-data; so are the version labels first discussed on "Version Control" on page 5. *Attributes* provide yet another annotation facility, in the form of name/value pairs. For example, an attribute named *CommentDensity* might be attached to each version of a source file, to indicate how well the code is commented. Each such attribute might have the value `"unacceptable"`, `"low"`, `"medium"`, or `"high"`.

Notification Procedures

Virtually any operation that modifies the data repository can *trigger* the execution of a user-defined procedure. A typical use for this capability is to notify one or more users that the operation took place. For example, a trigger on the *checkin* operation might send mail to the QA department, explaining that a particular user modified a particular file. Special environment variables make the relevant information available to the script or program that implements the user-defined procedure.

In addition to performing notification tasks, triggers can automate a wide variety of process management functions — for example:

- adding meta-data annotations to the objects that were just modified
- logging information that is not included in the *event records* that ClearCase creates automatically
- initiating a build procedure and/or source-code-analysis procedure whenever certain objects are modified

Policy Enforcement

Every organization has its own "rules of the road", which provide guidance (gentle or otherwise) as to where, when, and how development activities are to take place. ClearCase's trigger mechanism, introduced in the preceding section, provides a flexible tool for implementing development policies. In particular, a trigger can impose any user-defined requirement or prerequisite on any operation that modifies the data repository.

For example, a trigger might fire whenever a user attempts to *checkin* a new version of a critical file. The trigger procedure can subject the user and/or the file to any kind of test — and if the test fails, the procedure can cancel the *checkin*.

Access Control

Various objects in the data repository can be *locked*, which prevents them from being modified or used. Locks can be fine-grained (for example, locking a particular branch of a particular element) or general (for example, locking an entire VOB). A typical application is locking just the *main* branch of all elements during a software integration period, except to those few users who will be performing the integration work.

Access modes, or permissions, apply to all elements. Permissions control reading, writing, and executing of objects at the traditional levels of granularity: user (owner), group, and other. They also apply to the physical storage in the underlying file system. Protections effectively thwart attempts to circumvent ClearCase and tamper with the raw data storage.

ClearCase Client-Server Architecture

ClearCase is a "groupware" product, with a distributed client-server architecture. Both the *programs* that implement ClearCase functions and the development group's *data* can be distributed throughout a local area network. This makes ClearCase *scalable* — as workstations are added to the network to accommodate additional users, ClearCase's data-storage and data-processing resources increase, as well.

Note: Using the MultiSite extension to ClearCase to wide-area networks, as well — even networks whose only data communications channel is magnetic tape transfer.

Figure 1-12 shows a typical distribution of ClearCase programs and development data in a network. The data storage is organized as follows:

- The permanent, shared data repository is implemented as a collection of versioned object bases (VOBs). Several VOBs can be located on the same host; the practical limit is a function both of disk space and of processing resources.

- Users have individual (or shared) work areas, *views*, each of which has a private data storage area. A view's storage area is typically located on a user's individual workstation. Central server hosts can also be used — for example, for a shared view or a view in which an entire application will be rebuilt "from scratch".
- For increased flexibility, the data storage for an individual VOB or view can be distributed across two or more hosts.

Users access this data with ClearCase *client programs* (for example, the *clearmake* build utility), along with standard operating system facilities (text editors, compilers, debuggers) and third-party applications. Access to the data stored in VOBs and views is mediated by ClearCase *server programs*. Client and server processes communicate with each other using remote procedure call (RPC) facilities. This makes ClearCase network-transparent — users need not be concerned with the physical location of data storage; ClearCase servers make the data available globally.

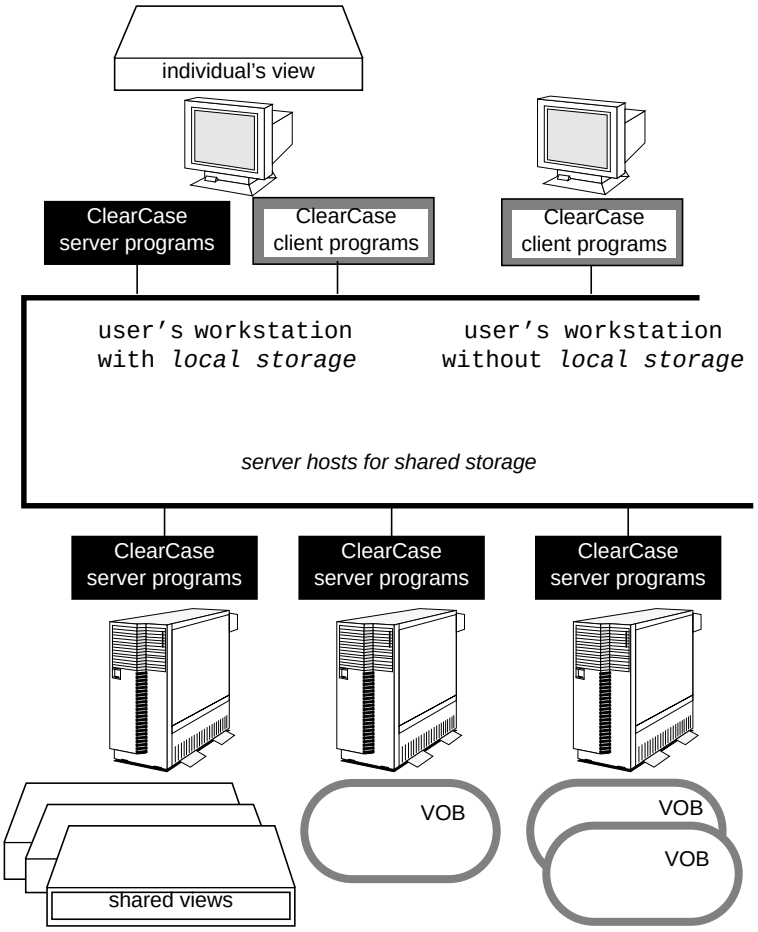


Figure 1-12 ClearCase Distributed Client-Server Architecture

ClearCase Interfaces

ClearCase has both a command-line interface (CLI) and a graphical user interface (GUI). The CLI is implemented as a set of executables, stored in */usr/atria/bin*. (Each user should add this directory to his or her search path.)

The "first among equals" of the CLI utilities is *cleartool*; through a set of subcommands, it provides the functions performed most often by users: checkout, checkin, list history, display version with annotations, and so on. *cleartool* uses multicharacter mnemonic options:

```
% cleartool checkin -identical -nc util.c hello.h
"Checkin files util.c and hello.h, without any comments; do
the work even if the new version is identical to its
predecessor."
```

```
% cleartool lshistory -since yesterday.17:00 -recurse /vobs/proj/src
"List all events that occurred since 5 pm yesterday,
pertaining to objects within the directory tree at
/vobs/proj/src."
```

```
% cleartool merge -to msg.c -version /main/alpha_port/LATEST
```

"Merge the most recent version on the *alpha_port* branch of file *msg.c* into the version I'm editing."

The ClearCase GUI includes several point-and-click programs:

- *xclearcase* provides a "master control panel" that is both easy to use and thoroughly customizable. Users can examine and select both their file system data and ClearCase meta-data, with a variety of *browsers*.
- *xlsvtree* displays the version tree of an element, making it easy both to determine how an element has evolved, and to select particular versions for comparison or merging.
- *xcleardiff* is a flexible tool for comparing and/or merging the contents of multiple versions of an element, or any other files.

Figure 1-13 illustrates some of the features of the ClearCase GUI.

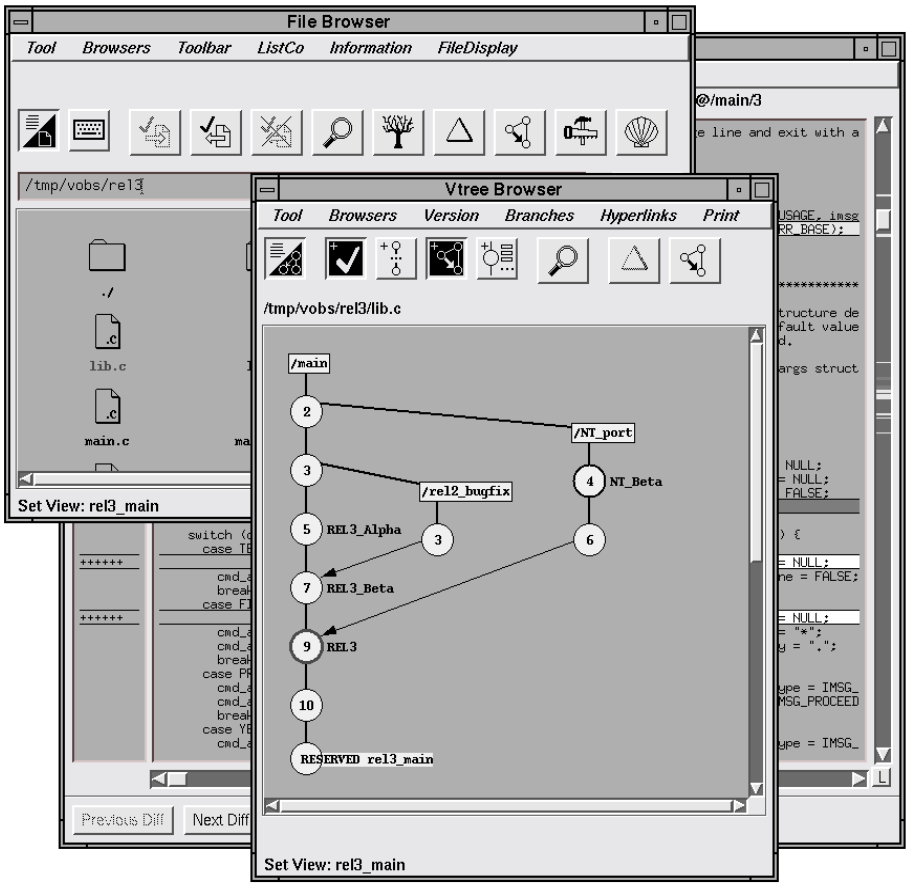


Figure 1-13 ClearCase Graphical User Interface

ClearCase Documentation

In addition to this manual, the *CASEVision™/ClearCase Concepts Guide*, the ClearCase printed documentation set includes:

CASEVision™/ClearCase Tutorial
: Important information on setting up a user's environment, along with a step-by-step tour through ClearCase's most important features.

CASEVision™/ClearCase User's Guide
: Background information and step-by-step procedures for use by individual users.

CASEVision™/ClearCase Administration Guide
: Background information and step-by-step procedures for use by ClearCase system administrators.

CASEVision™/ClearCase Reference Pages
: All the ClearCase manual pages, for programs, data structures, and administrative utilities.

Documentation

All CLI utilities can display usage syntax summaries. The *cleartool* utility's *help* subcommand can display a usage message for individual subcommands:

```
% cleartool help checkout
Usage: checkout | co [-reserved | -unreserved]
    [-branch branch-pname]
    [[-data] [-out dest-pname] | -ndata]
    [-c comment | -cq | -cqe | -nc] pname ...
```

The ClearCase manual pages reside within the ClearCase installation area. Users can access these manual pages with the *cleartool man* subcommand, or with the standard UNIX *man*(1) facility.

Each of the ClearCase GUI programs has its own context-sensitive help facility. Installation instructions, release notes, and supplementary technical notes are also provided in the ClearCase software.

Chapter 2

Version Control - ClearCase VOBs

ClearCase supports a well-organized, controlled development environment by maintaining two kinds of data storage:

- **Permanent data repository** — a globally-accessible, shared data resource that can be modified only by ClearCase commands. The repository contains both historical and current development data.
- **Working data storage** — any number of distinct areas which provide "scratchpad storage" for day-to-day development activities. A typical area belongs to an individual user, or to a small group working on the same task.

This chapter and the next describe the contents of *versioned object bases*, which implement the permanent data repository. Working data storage, implemented by *views*, is discussed briefly in this chapter and more fully in Chapter 4, "ClearCase Views."

Versioned Object Bases (VOBs)

A *versioned object base*, or *VOB*, is the permanent data repository for a development tree or subtree. A VOB stores file system objects: directories, files, symbolic links, and hard links. (It also stores non-file-system information, *meta-data*, which we discuss in Chapter 3, "ClearCase Meta-Data."

Using VOBs: Clients, Servers, and Views

Many version-control systems require users to perform their day-to-day work on copies of data, only occasionally accessing the permanent data repository. ClearCase allows users to work directly with the repository — that is, directly with VOBs. Direct access is implemented coherently and

securely in the multiple-user, multiple-host environment by combining several mechanisms (Figure 2-1):

- **View context** — Any program, not just a ClearCase program, can read a VOB's data. But a program must use a ClearCase *view* to access a VOB; otherwise, the VOB appears to be empty. Through the view, the VOB appears to be a standard directory tree.
- **Client programs** — A VOB can be modified only by special ClearCase client programs. Most version-control operations are implemented by *cleartool* (command-line interface), and by (graphical user interface). Audited builds are performed with *clearmake* and *clearaudit*.
- **VOB activation** — Typically, a VOB is located on a remote host, rather than on the user's own host. A VOB is made available on the user's host by *activating* it there, through operating system networking facilities. For example, on a UNIX system, a VOB is activated by mounting it as a file system of type MVFS — ClearCase's *multiversion file system* type.
- **Server programs** — Only ClearCase *server programs*, running on the host where the VOB physically resides, perform the "real" work: retrieving data from the VOB and updating it. Client programs communicate with server programs using operating system remote-procedure-call (RPC) facilities.

Typically, a user works on his or her own *client host*, accessing VOBs that are physically located on one or more remote *VOB server hosts*.

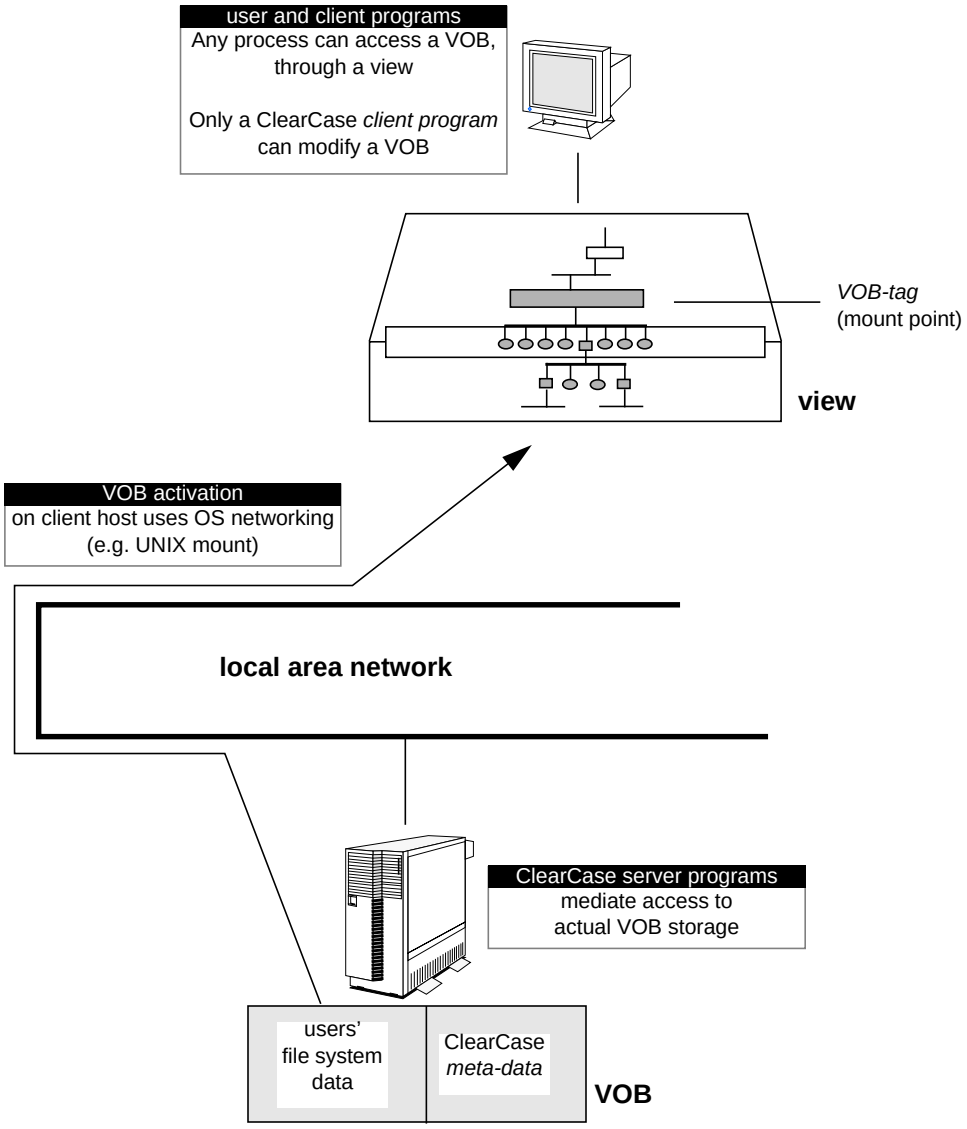


Figure 2-1 How ClearCase Users Access a VOB

VOB Data Structures

A VOB is implemented as a *VOB storage directory,* a directory tree whose principal contents are a set of storage pools and an embedded database (Figure 2-2).

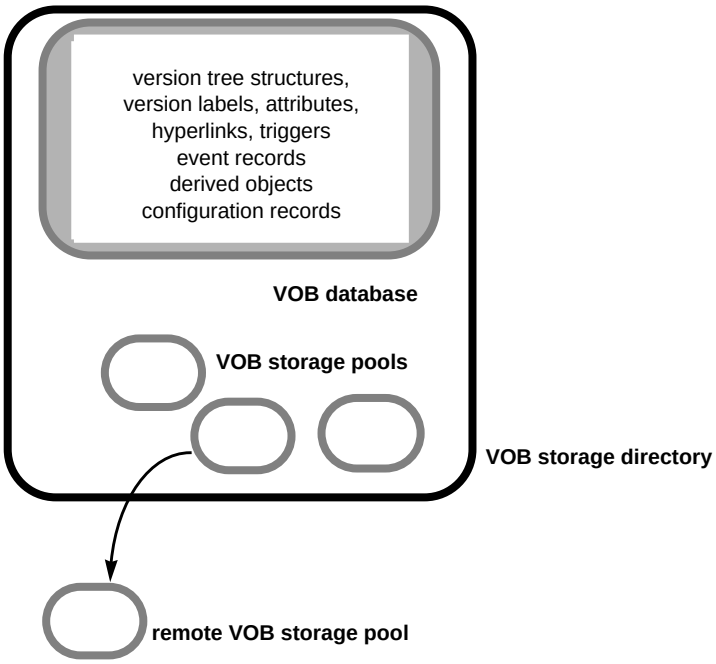


Figure 2-2 VOB Storage Directory

VOB Storage Pools

A *VOB storage pool* is a subdirectory that stores users' file system data. Some storage pools provide a repository for historical and currently-used versions of source files. Other storage pools provide a repository for object modules, executables, and other *derived objects* created during the development process.

Each VOB is created with an initial set of pools, located within the VOB storage directory. These can be supplemented (or replaced) with pools located on the same disk, on another local disk, or on a remote host. This affords administrators great flexibility, enabling data storage to be placed on hosts where ClearCase itself is not installed (for example, a very fast file server machine).

ClearCase can store individual versions of files in several ways, using such techniques as data compression and line-by-line *deltas*. The data storage/retrieval facility is extensible — users can supply their own *type managers*, implementing customized methods for storing and retrieving development data. (See "Element Types and Type Managers" on page 44.)

VOB Database

Each VOB has a *VOB database*, a subdirectory containing information managed by the database management system embedded in ClearCase. The VOB database stores version-control information, along with a wealth of other information, including:

- user-defined annotations on source file versions
- complete "bill-of-materials" records of software builds
- *event records* that chronicle the changes that users have made to the VOB: creation of new versions, adding of annotations, renaming of source files, and so on

This information is termed *meta-data*, to distinguish it from file system data. We describe it in more detail in Chapter 3, "ClearCase Meta-Data.".

Elements, Branches, and Versions

Each ClearCase VOB stores version-controlled file system objects, termed *elements*. An element is a file or directory for which ClearCase maintains multiple *versions*. The versions of an element are logically organized into a hierarchical *version tree*, which can include multiple *branches* and *subbranches* (Figure 2-3).

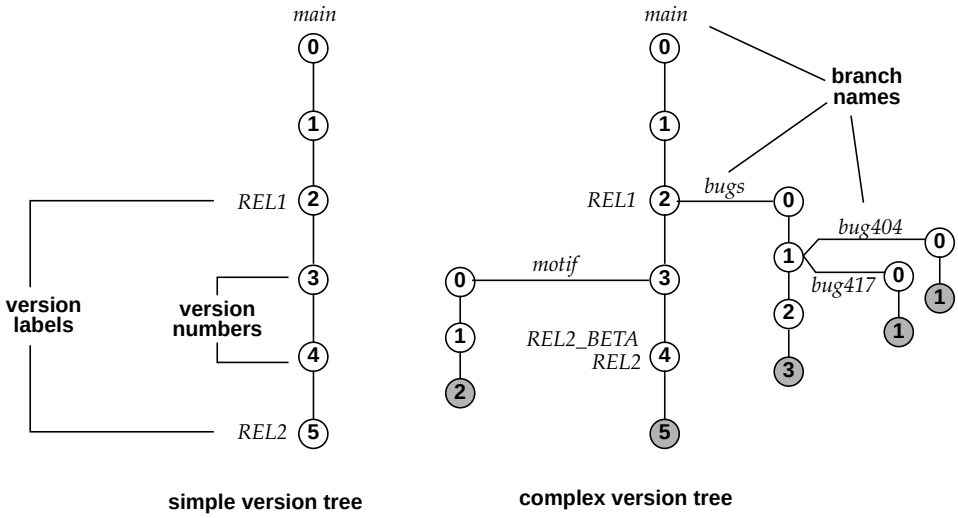


Figure 2-3 Version Trees of ClearCase Elements

Some elements may have version trees with a single branch — the versions form a simple linear sequence. But typically, users define additional branches in some of the elements, in order to isolate work on such tasks as bug fixing, code reorganization, experimentation, and platform-specific development.

Figure 2-3 illustrates several features of ClearCase version trees:

- Each element is created with a single branch, named *main*, which has an empty version, numbered 0.
- ClearCase automatically assigns integer *version numbers* to versions. Each version can also have one or more user-defined *version labels* (for example, *REL1*, *REL2_BETA*, *REL2*).
- One or more branches can be created at any version, each with a user-defined name. (Branches are not numbered in any way.)
- ClearCase supports multiple branching levels.
- *Version 0* on a branch is identical to the version at the branch point.

Version-IDs

Each version of an element has a unique *version-ID*, which indicates its location in the version tree. A version-ID takes the form of a *branch-pathname* followed by a *version-number* (Figure 2-4).

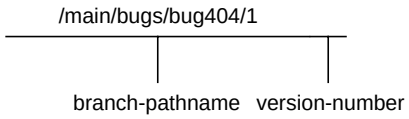


Figure 2-4 Version-ID of a Version

For example, the shaded versions at the end of the branches in Figure 2-3 have these version-IDs:

```
/main/5
/main/motif/2
/main/bugs/3
/main/bugs/bug404/1
/main/bugs/bug417/1
```

Version-IDs look like pathnames because an element's version tree has the same hierarchical structure as a directory tree. Version-IDs work like pathnames, too, as described below.

Letting the View Select Versions

Typically, users wish to access only one version of an element at a time. A ClearCase *view* provides a work environment in which at most one version (or perhaps no version at all) of each version-controlled object appears. Thus, a particular view might resolve a simple file name or a full pathname:

```
util.c
   ... or ...
/vobs/vega/src/util.c
```

... to version */main/5* of file element *util.c*. Another view might resolve the same pathname to version */main/motif/2*.[1]

The ability to reference any version with a standard operating system pathname is a very important ClearCase feature, termed *transparency*. This is what makes a VOB appear to be a standard directory tree, as illustrated in Figure 2-1. Standard operating system utilities and third-party development tools "just work" with VOB data, without requiring any modification, conversion, or wrapper routines. For more on transparency, see Chapter 4, "ClearCase Views.".

Accessing Any Version with a Version-Extended Pathname

Any version of an element can be uniquely specified by appending its version-ID to its standard pathname, forming a *version-extended pathname*. Figure 2-5 shows an example.

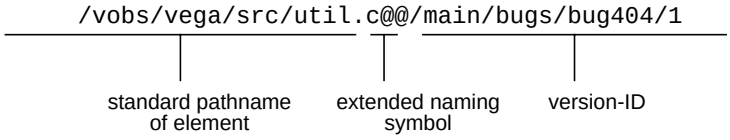


Figure 2-5 Version-Extended Pathname

A version-extended pathname can also use a version label; this enables users to reference versions mnemonically:[1]

```
% cat util.c@@/REL2
% diff messages.c messages.c@@/REL3_BSLVL2
% /usr/local/bin/validate bgrs.h@@/NEW_STRUCT_TEST
```

[1] Note that with ClearCase, a full pathname is not an absolute identifier. The same full pathname can indicate different objects, depending on the view context.

[1] In most cases, this syntax is valid even if the labeled version is not on the element's *main* branch.

Version-extended pathnames can be used in standard commands, either alone or in conjunction with standard pathnames. For example, this command searches for a character string in the "current" version and in two "historical" versions:

```
% grep "InLine" monet.c monet.c@@/main/13 monet.c@@/RLS2.5
```

More Extended Pathnames

The *extended naming symbol (*by default, `@@`) suppresses ClearCase's automatic version-selection mechanism. It causes the MVFS to interpret the pathname as a reference to the element itself, or to some location in its version tree:

```
util.c (the version selected by the view)
util.c@@ (the element itself)
util.c@@/main (one of the element's branches)
util.c@@/main/2 (a specific version of the element)
```

The entire version tree of an element is embedded under its standard pathname. This enables users to access any (or all) historical versions through pathnames, using any program. The following variant of the above *grep* command shows the power of this file system extension:

```
% grep "InLine" monet.c@@/main/*
```

In this command, a standard shell wildcard character (*) specifies all the versions on an element's *main* branch.

How New Versions Are Created

This section describes ClearCase facilities for managing the creation of new versions and branches in version-controlled elements.

The Checkout-Edit-Checkin Model

Like many version-control systems, ClearCase uses a checkout-edit-checkin model to manage the growth of elements' version trees:

1. In the "steady-state", an element is read-only — users can neither modify it or remove it.
2. To modify an element, a user establishes a view context, then enters a *checkout* command. This seems simply to change the element from read-only to read-write; in actuality, it makes an editable copy — a *checked-out version*. (For details, see "Revising a Source File / Checkout-Edit-Checkin" on page 78.)
3. The user revises the checked-out version using any available system-supplied or third-party tools.
4. The user enters a *checkin* command. This creates a new, permanent version of the element, which then reverts to the "steady-state" of being read-only.

Reserved and Unreserved Checkouts

When entering a *checkout* command, the user implicitly expresses an intention to create a new version of the element at some future time. ClearCase supports either a "first-come-first-served" approach or an "exclusive privilege" approach to managing the creation of new versions.

A *reserved* checkout grants the exclusive privilege to create the next version on the branch; a branch can have only one reserved checkout at a time. But a branch can have any number of *unreserved* checkouts. If several users, in different views, have unreserved checkouts of a branch, the first one to perform a *checkin* "wins"; the others must combine the changes in the newly-created version into their own work before they can perform a *checkin*. (See "Merging Versions of an Element" on page 40.)

Creating Versions in a Parallel Development Environment

In a parallel development environment, the same element can be modified by several projects concurrently; each project creates versions on its own branch, isolated from the changes being made by other projects on other branches.

Figure 2-6 shows an element with two subbranches: the *r2_fix* branch is for maintenance work; the *gopher_port* branch is for porting the application to a new architecture. This figure also shows how the *checkout* command supports parallel development, by localizing its effects:

- The *checkout* command operates on a particular branch of an element, not on the entire element. Several different branches might be checked out at the same time, each for a different programming task. (*checkout* always checks out the last version on a branch; revising an intermediate version requires creation of a subbranch at that version.)
- An element's branch is checked out to one particular view. The checked-out version is visible only in that view; the *checkout* command leaves all other views unaffected. Each view can checkout only one branch at a time of any given element, because the view can see only one object at the element's pathname.

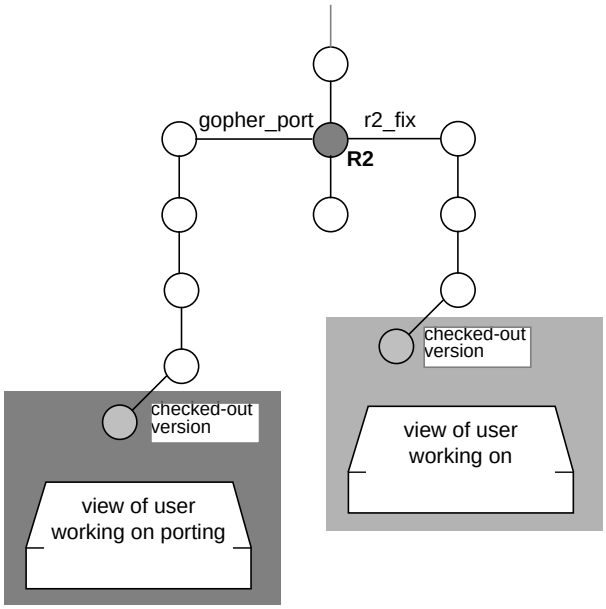


Figure 2-6 Parallel Development

Merging Versions of an Element

In a parallel development environment, different branches of an element can remain active for arbitrarily long periods. The contents of the branches inevitably diverge more and more as time passes. In some cases, this may be acceptable; but most organizations wish to keep an element's *main* branch up-to-date — with bugfixes, with architecture-dependent constructs, and so on. This means that changes made on subbranches must periodically be *merged* back into the *main* branch. It is also typical for data to be merged onto other branches — for example, a branch used to develop a port of an application.

ClearCase includes flexible and powerful merge tools. Often, the process of merging all the work performed for a particular development project is completely automatic. When conflicting changes to the same section of code prevent a merge from being completely automatic, users can invoke the *xcleardiff* graphical merge utility to resolve the conflicts and perform any necessary cleanup (Figure 2-7).

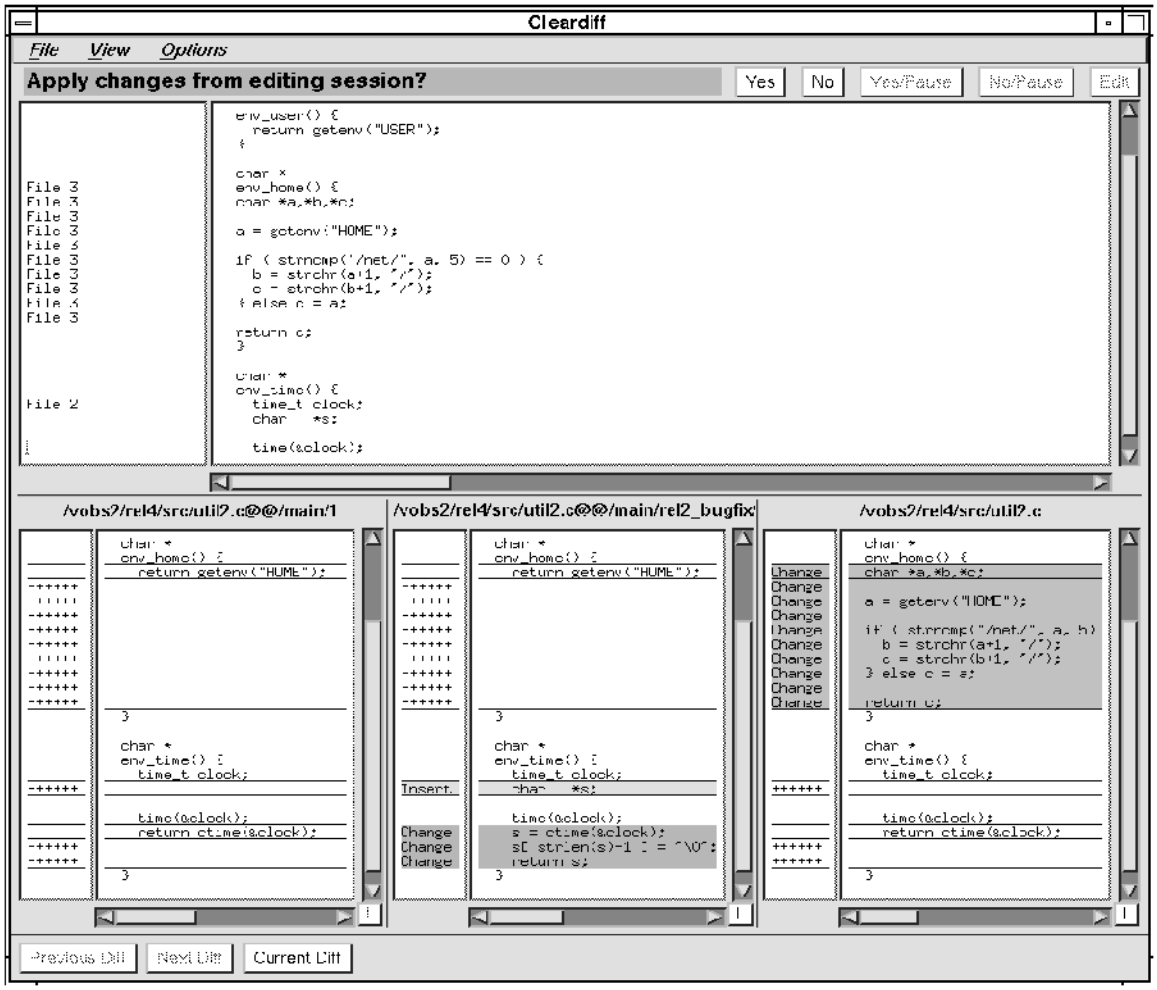


Figure 2-7 Graphical Merge Utility: 'xcleardiff'

ClearCase supports many variants of the basic merge scenario. For example:

- Any version of an element can be merged into any other version; the target need not be on the *main* branch.
- A *selective merge* can incorporate some, but not all, of the changes made on one branch into another branch.
- A *subtractive merge* can remove the changes made in a set of versions.

The Version as Compound Object

A version is actually a two-part compound object:

- **Object in the VOB database** — There is a *version* object in the VOB database. Pointers to branch and element objects establish the location of the version object in some element's version tree. The version object can have user annotations (for example, version labels); it is referenced by various ClearCase-generated event records and build configuration records.
- **File in a VOB storage pool** — There is a *data container* file in one of the VOB's source storage pools. This file contains the version's file system data.

Figure 2-8 illustrates the relationship between the two components of a version. In general, ClearCase commands operate directly on the database object; standard programs operate on the associated data container.

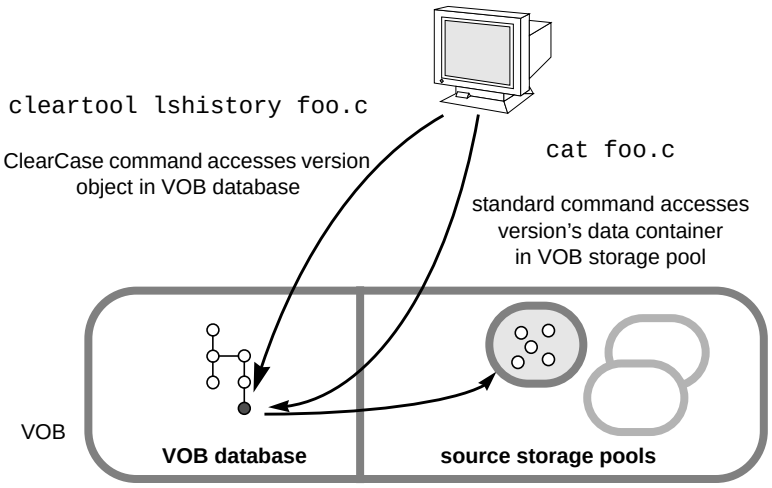


Figure 2-8 The Version as Compound Object

Special Cases

In general, users need not be concerned with the dual nature of a version. Some ClearCase features, however, are best understood by keeping the duality in mind. For example:

- Users can create versions that consist only of a VOB database object (along with its user annotations and ClearCase-generated records). Similarly, the data container of an existing version can be discarded, leaving only the database object.
- Administrators can move the data containers of existing versions to different storage pools.
- A view can be configured so that the file system data of some versions disappear, but the corresponding database objects remain accessible to ClearCase client programs.

Element Types and Type Managers

Each element has an *element type*, which can be used for either or both of these purposes:

- to provide a logical or functional classification for the element
- to determine how the element's versions are to be stored and retrieved

ClearCase has several predefined element types, each of which represents a different physical file format for data container file(s) in a source storage pool. Each element type has its own data-access method, which optimizes either performance or storage requirements:

file — arbitrary sequence of bytes; each version is stored in whole-copy format ("as-is") in a separate data container file

text_file — sequence of non-NULL bytes, separated into "text lines" by `<NL>` characters; all versions are stored efficiently as *deltas* (incremental differences) in a single structured data container file

compressed_file and *compressed_text_file* — variants of *file* and *text_file* that use standard UNIX data-compression techniques; compression typically reduces the disk storage requirement by 25%–50%

The element types *file* and *compressed_file* can be used to version-control any file, including executables, programming libraries, databases, bitmap images, and non-ASCII desktop-publishing documents.

Cleartext Storage Pools

ClearCase implements a performance optimization for all file elements whose versions are not stored in whole-copy format. This includes text files (stored in delta format) and compressed files. ClearCase attempts to amortize the cost of "extracting" a particular version (by applying deltas, or by uncompressing) over multiple accesses to the version:

- The first time an existing version is extracted from its data container, the "clear text" of the version is stored in a data container in a *cleartext storage pool*.
- On subsequent accesses to the same version, the cleartext data container is used again, eliminating the need for another "extraction".

Thus, a cleartext storage pool acts as a cache of the file system data for recently-accessed versions.

User-Defined Element Types

Users can define their own element types to classify elements functionally, enabling such operations as:

- easily identifying and defining operations on all of a project's header files
- using a particular icon to represent header files in the ClearCase GUI
- using different version-selection criteria for different element types
- requiring that C-language source files (but not header files) be subjected to a *lint*(1) check when they are modified

ClearCase includes an automatic *file typing* mechanism, which enables element types to be applied efficiently and consistently. For example, new file elements whose names end with *.c* might automatically be assigned the user-defined element type *c_source*.

Type Managers

Another reason for ClearCase sites to define their own element types is to handle different physical or logical file formats. Each element type has an associated suite of programs, its *type manager*, which handles the task of accessing the element's versions in a VOB storage pool. For example, the type manager *text_file_delta* implements the efficient storage of version-to-version changes as *deltas* in a single, structured *data container* file.

A user-defined element type can have its own, user-supplied type manager suite. For example:

- A type manager for elements of type *bitmap* might compute incremental differences between versions of a bitmap file, and store all the versions in a single, structured data container file.
- A type manager for elements of type *frame_file* might use special routines to compare two versions of a FrameMaker® document.
- A type manager for elements of type *manual_page* might compare two versions of a manual page source file by first formatting them with the standard *nroff* program.

Version Control of Links

Most implementations of UNIX support two kinds of links:

- A *hard link* is a directory entry that provides an alternative name for an existing file. The file's "original" name and all its additional hard links are completely equivalent.
- A *symbolic link* is a directory entry whose contents is a character string that specifies a full or relative pathname. The link acts as a pointer to a file or directory (or another symbolic link). Many UNIX functions and commands "chase" this pointer: a reference to a symbolic link becomes a reference to the object at the full or partial pathname specified by the link.

As counterparts to standard UNIX links, ClearCase supports *VOB hard links* and *VOB symbolic links*:

- A *VOB hard link* is a directory entry that provides an additional name for an existing element in the same VOB.
- A *VOB symbolic link* is an object whose contents is a character string that specifies a pathname. The pathname can specify a location in the same VOB, in a different VOB, or a non-VOB location.

An essential difference between these kinds of objects is that VOB symbolic links do not have version trees (since they do not name elements). Version-control of VOB symbolic links is accomplished through directory versioning, as described in the next section.

Version Control of Directories

Some version-control systems are good at tracking the changes to the *contents* of files, but have little or no ability to handle changes to the *names* of files. Name changes are a fact of life in long-lived projects. So are such changes as creating new files, deleting obsolete files, moving a file to a different directory, merging source files together, and completely reorganizing a multiple-level directory structure. ClearCase addresses these needs by providing version-control of directories.

Directory Elements

Each version of a ClearCase *directory element* is analogous to a standard UNIX directory:

- A UNIX directory is a list of names, each of which points (through an *inode number*) to a file system object — file, directory, and so on.
- A version of a ClearCase directory element is a list of names, each of which points (through a ClearCase-internal *object-ID*, or *OID*) to a file element, a directory element, or a VOB symbolic link (Figure 2-9).

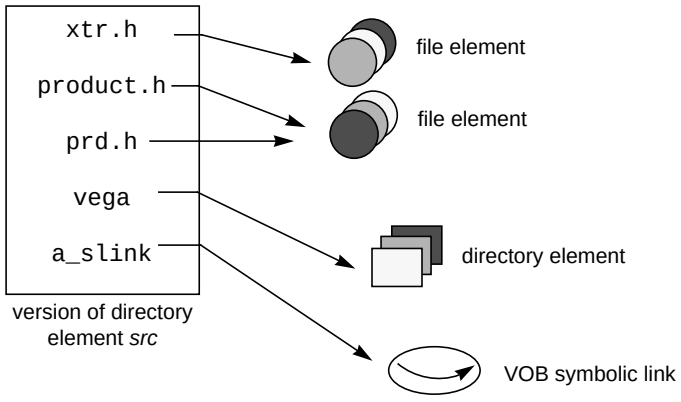


Figure 2-9 Directory Element

In many respects, a directory element resembles a file element:

- It has a version tree.
- Any directory version can be accessed with a version-extended pathname. For example, */vobs/vega/src@@/main/3* references a particular version of directory element *src*.
- A ClearCase view selects a particular version of each directory element. The selected versions of all directory elements in a VOB constitute a *namespace*; different views can instantiate different namespaces.

Figure 2-10 shows the critical difference: each version of a file element is a regular file; each version of a directory element is a list of element and VOB symbolic link names.

Accessing Files through Directory Versions

A version of a file element is accessed by first traversing a hierarchy of directory versions. Accessing a particular version of element */vobs/vega/src/util.c* involves:

- accessing some version of */vobs/vega* (the VOB's top-level or *root* directory)
- accessing a version of directory element *src*
- accessing the desired version of *util.c*

Although directory versions are essential for managing long-lived projects, users need not think about them on a day-to-day basis. ClearCase's *transparency* feature, discussed for file elements in "Letting the View Select Versions" on page 35, works for directory elements, also. Thus, a view would resolve the standard pathname */vobs/vega/src/util.c* automatically by selecting a version of directory *vega*, a version of directory *src*, and a version of file *util.c*. (For more, see "Transparency and Its Alternatives" on page 70.)

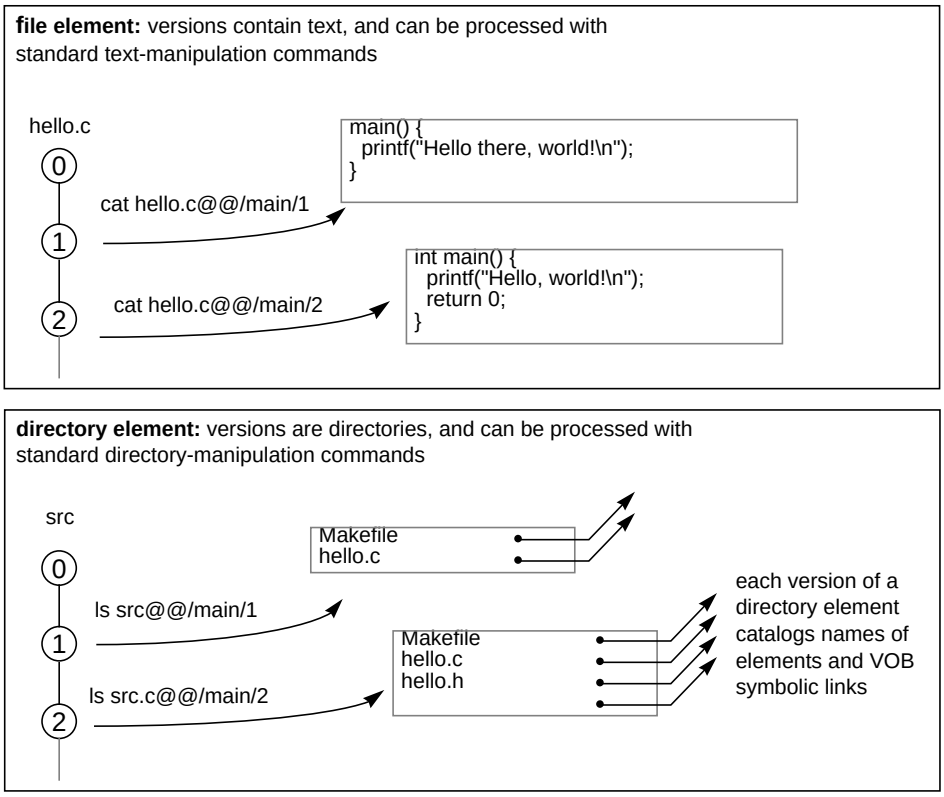


Figure 2-10 File Element vs. Directory Element

Chapter 3

ClearCase Meta-Data

Each ClearCase VOB stores a variety of information related to, but distinct from, the contents of file system objects. This information is termed *meta-data*.

Operating systems also support certain kinds of meta-data. For example, a file's time-modified stamp is information *pertaining to* the file that is not actually *in* the file.

Meta-Data: Records and Annotations

As users enter ClearCase commands, meta-data *records* are created automatically. These records include "who, what, when, where, and why" information regarding such operations as:

- creation of new elements, and new versions of existing elements
- creation of user-defined annotations
- software builds

In particular, the record of a software build — a *configuration record* — includes a "bill of materials", listing (along with other information) exactly which versions of source elements were used. This provides both complete build auditing and guaranteed rebuildability.

Users can also create meta-data *annotations* explicitly, attaching them to file system objects. A user-defined annotation might indicate:

- ... that a particular version of an element was used in Release 4.31
- ... that the benchmark score on a particular executable was 19.3
- ... that two elements in different directories need to be kept synchronized

Storage in the VOB Database

A VOB's meta-data is stored in its *VOB database*, within the VOB storage directory. ClearCase includes commands for listing (and in some cases comparing) the contents of previously created meta-data records (Figure 3-1).

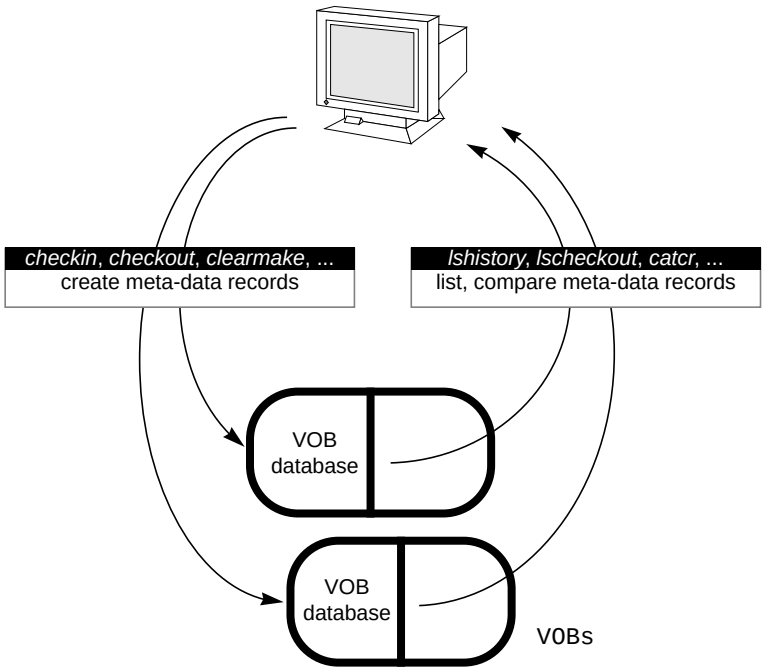


Figure 3-1 Meta-Data: ClearCase-Generated Records

Users can access file system objects through their meta-data. This makes it possible (and easy) to define, reproduce, and maintain configurations of file system objects. For example, "all the versions labeled *REL4.31*" might constitute the configuration of files that was used to build a particular product release.

Meta-data also plays a fundamental role in ClearCase's tools for implementing development policies and procedures. For example, a project's source files might be partitioned into several classes using meta-data annotations. Then, different development policies might be applied to each class of source file.

User-Defined Meta-Data

ClearCase supports numerous kinds of user-defined meta-data, all of which share the same implementation scheme:

1. Some user (typically, an administrator or project leader) *defines* a meta-data item for use within a particular VOB, by creating a *type* object in that VOB.
2. Thereafter, users can create *instance*s of the type object. This is typically described as "attaching", "assigning", or "instancing" the meta-data; it creates a pointer in some VOB object, referencing the type object.[1]

For example, Figure 3-2 illustrates the mechanism by which a version label is attached to a version object.

[1] In some cases, instancing a type object does more than just annotate an existing object — it creates an entirely new object, to which additional meta-data annotations can be attached.

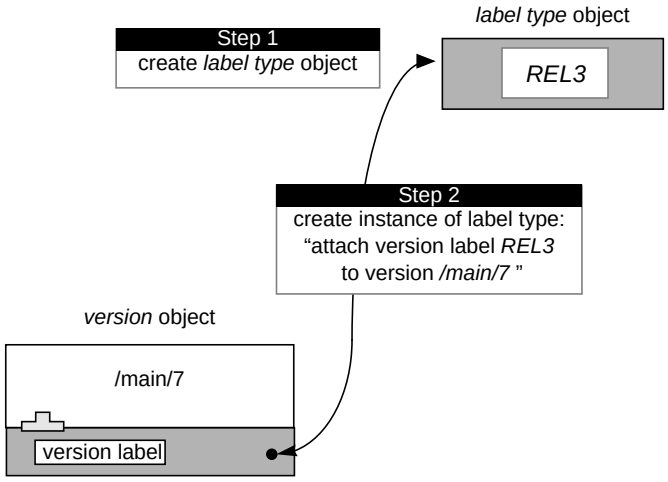


Figure 3-2 Attaching Meta-Data to a VOB Object

The following sections briefly describe the several kinds of user-defined meta-data. Included are cross-references to discussions, in other chapters, of how users and administrators employ these mechanisms.

Version Labels / Label Types

A mnemonic identifier called a *version label* can be attached to any version of an element. A version label (for example, *REL3.1.5*) can often be used instead of the system-assigned *version-ID* (for example, */main/12*) to specify a particular version of an element.

As Figure 3-3 indicates, the same version label can be used in many elements. It is also true that multiple labels can be assigned to the same version; for example, the same version of a rarely-modified element might “accumulate” many labels: *REL3.0*, *REL3.1*, *REL3.1.5*, and so on.

Like branches, version labels play a pivotal role in organizing and describing the development environment. A release engineer might assign version label *REL3.1.5* to the set of source versions used in the build of a particular release. At any subsequent point, this configuration of versions can be referenced in a variety of contexts as "all the versions labeled *REL3.1.5*" (the shaded versions in Figure 3-3).

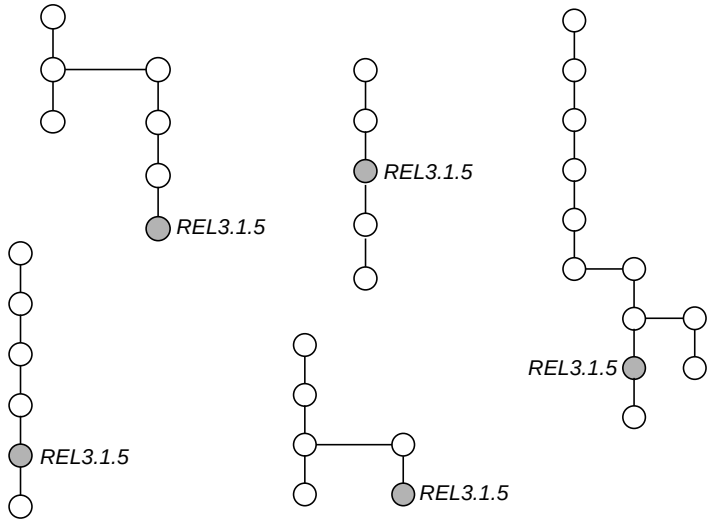


Figure 3-3 Configuration of Versions Selected by Version Label

As illustrated in Figure 3-2, a version label is an instance of a *label type* object.

Attributes / Attribute Types

An *attribute* is a user-defined annotation in the form of a name/value pair. Attributes can be attached to many kinds of objects (whereas version labels can be attached only to versions). Following are some examples of attributes:

Name	Kind of Value	Example
QAed	*string*	“Yes”
CodeQuality	*integer*	3
QAedBy	*string*	"george"
ECOnum	*integer*	`8987`
Benchmark	*real (floating point)*	34.1
LastQAed	*time*	4-Apr.08:45

Attributes have a multitude of applications, including:

- **Bug tracking** — Each version involved in the fix for Bug #8987 might be assigned the attribute BugNumber/8987.
- **Quality control** — Every version of a module might be assigned a CodeQuality attribute, with values between 1 (worst) and 10 (best). Every element might be assigned a TestProc attribute, whose value names a corresponding test procedure.
- **Requirements tracing** — Attributes can be attached to the hyperlinks that implement requirements tracing, to provide additional information about the relationship. (See “Hyperlinks / Hyperlink Types” on page 57.)

An attribute is an instance of an *attribute type* object (Figure 3-4).

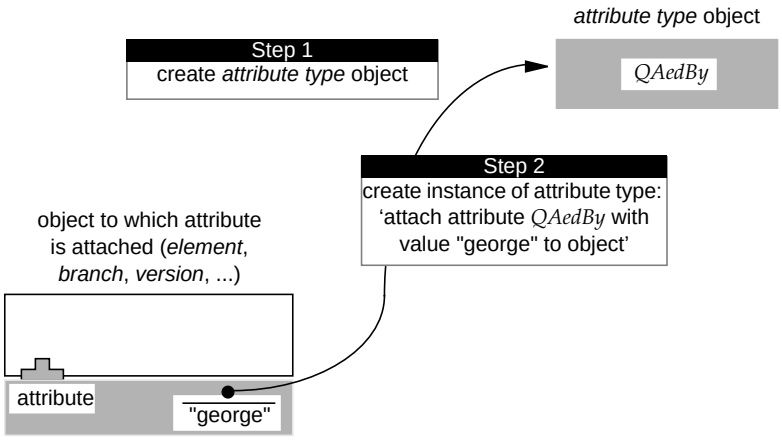


Figure 3-4 Attribute and Attribute Type

Hyperlinks / Hyperlink Types

A *hyperlink* is a user-defined logical arrow that connects two elements, branches, versions, or VOB symbolic links. The objects can be in the same VOB, or in different VOBs. Either end of a hyperlink can be annotated with a text string. Figure 3-5 illustrates a common application of hyperlinks, *requirements tracing*.

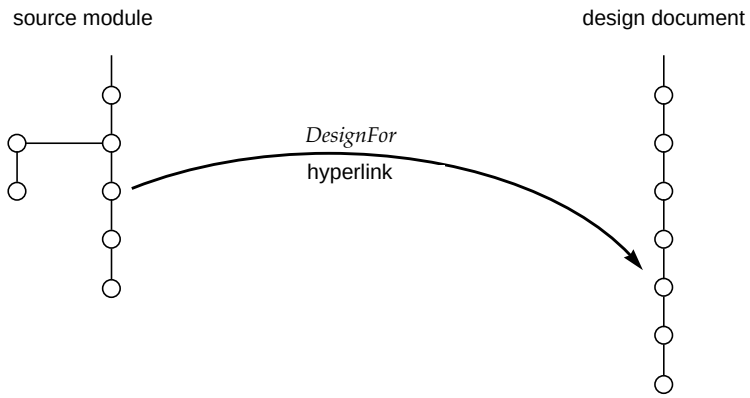


Figure 3-5 Hyperlink for Requirements Tracing

In this example, the version of a source module that implements a new algorithm is logically connected by a *DesignFor* hyperlink to the version of a design document that describes the algorithm. Subsequent versions of both elements implicitly *inherit* the connection between the versions that are linked explicitly.

Some additional examples:

- **Informal dependencies** — A set of hyperlinks defines a "network" of elements that must be modified as a group, even though there are no source-code dependencies among them. A hyperlink connects a source file element that defines run-time error messages to a document file element that contains a user manual's "Error Messages Appendix".
- **Merge arrows** — ClearCase itself uses a hyperlink to indicate that two versions of an element have had their contents merged.

Hyperlinks as Objects

A hyperlink is an instance of a *hyperlink type* object (Figure 3-6). Each hyperlink is also a VOB object in its own right, with a unique *hyperlink-ID*; certain ClearCase commands accept names of hyperlinks in much the same way as they accept file system pathnames:

```
% cleartool describe util.c (describe file system object)
version "util.c@@/main/4"
   created 10-Dec-93.18:29:11 by (jjp.users@phobos)
   element type: text_file
      predecessor version: /main/3
      Labels:
      REL3

% cleartool describe Merge@277 (describe hyperlink object)
hyperlink "Merge@277@/usr/hw"
   created 10-Dec-93.17:50:54 by akp.dvt@neptune
   Merge@277@/usr/hw
      /usr/hw/src/util.c@@/main/rel2_bugfix/1
      -> /usr/hw/src/util.c@@/main/3
```

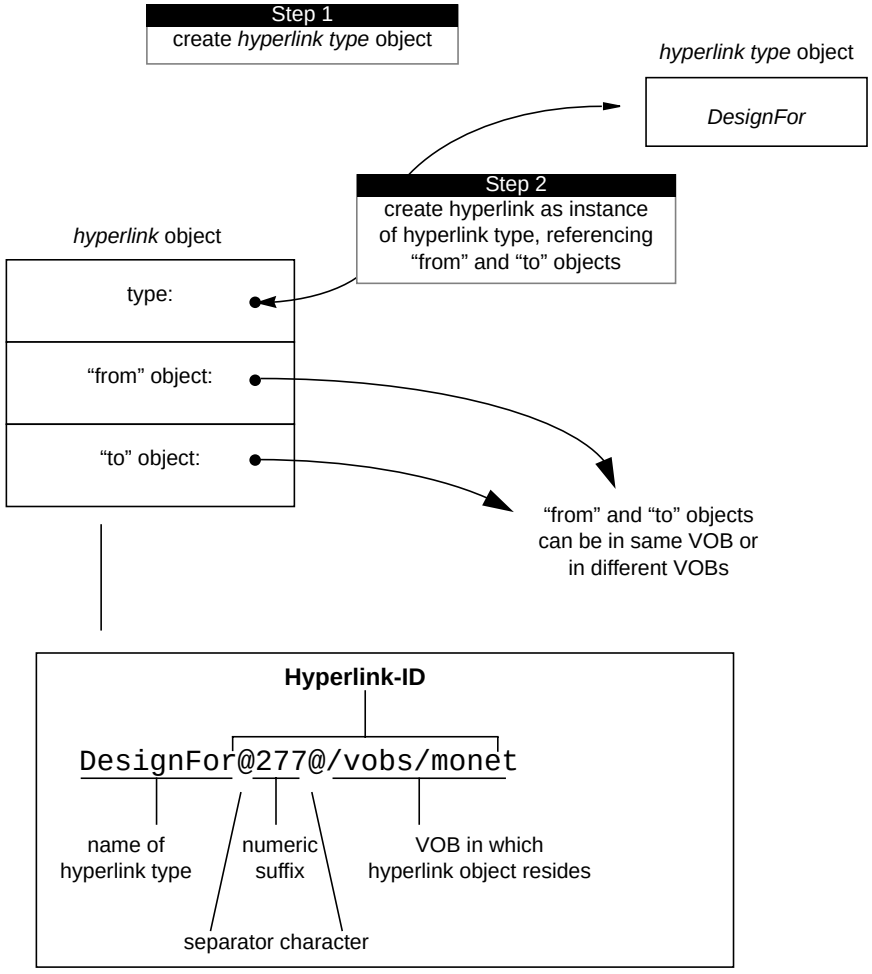


Figure 3-6 Hyperlink, Hyperlink Type, and Hyperlink-ID

Triggers and Trigger Types

A *trigger* is a "monitor" that tracks development work. When a user-specified ClearCase operation is performed, the trigger *fires* automatically, executing one or more user-specified *trigger actions*. Triggers can help document the development process, improve communications within the development group, and implement process-management policies. For example, a trigger might:

- send mail to a development manager and/or technical writer when a new version of file element *cmd_parser.c* is created
- send mail to all users when any header (*.h*) file is modified
- send mail to a project manager when a new branch is created in any element within directory tree */vobs/vega*
- attach an attribute to a new version of any source file, indicating the ECO number of the project for which the change was made
- perform a *lint*(1) check (and require that it succeed) before allowing *checkin* of a file of type *c_source*
- require that any new branch type created by a user include the user's *login* name

A trigger is an instance of a *trigger type* object. Triggers can be placed on several kinds of objects:

- **Individual elements** — A trigger attached to an element fires each time a ClearCase command operates on the element.
- **All elements** — A *global trigger type* is implicitly attached to all the elements in a VOB, both current and future.
- **Individual type objects** — A trigger attached to a *type object* fires whenever the object is used or modified. For example, attaching a version label to any version of any element would fire a trigger that is attached to the corresponding *label type*.

A trigger can be fine-tuned, so that it fires only on certain ClearCase operations, or when certain kinds of meta-data are involved. For more on the design and use of triggers, see Chapter 6, "ClearCase Process Control."

Elements, Branches, and Their Types

Elements and branches were introduced in the discussion of ClearCase version-control, in Chapter 3, "ClearCase Meta-Data." There, we stated that "each element has an element type". Now, we can refine that statement:

"each *element* is an instance of an *element type*"

That is, a (predefined or user-defined) element type object must first exist; then, users can create one or more element objects as instances of the type. Similarly, each *branch* is a VOB object, created as an instance of an existing *branch type* object.

Thus, elements and branches are implemented in the VOB database as instances of *type* objects, in exactly the same way as version labels, attributes, hyperlinks, and triggers.

ClearCase-Generated Meta-Data

Many ClearCase commands automatically chronicle their work by writing *records* to one or more VOB databases. Users can examine these records to determine what development work has taken place. *clearmake*, the ClearCase build utility, implements a build-avoidance scheme that uses records created by previous builds.

Event Records

Each ClearCase command that makes a change in a VOB creates an *event record* that describes the change, including "who, what, when, and where" information. The event record is permanently[1] stored in the VOB database, and is logically associated with the particular VOB object that was changed by the command.

[1] An event record is deleted when its associated object is deleted. ClearCase includes a *vob_scrubber* utility that deletes unwanted event records, enabling administrators to inhibit unwarranted VOB database growth.

For example, when the *checkin* command creates a new version, it also writes a create version event record to the VOB-fmt database:

```
27-Jul.15:11  akp  create version "tb004.doc@@/main/45" "gordon's corrections"
```

ClearCase includes a simple report-writing facility, which can be used to extract certain information from a selected set of event records. For example, the following report interleaves information drawn from the event records of a set of elements specified with a wildcard (cmd_*.c), using a custom format (-fmt option):

```
% cleartool lshistory -fmt "%Sd  %e  %n\n" cmd_*.c
24-Mar-94  checkout version cmd_protect.c
24-Mar-94  create version  cmd_utl.c@@/main/multisite_v1/5
24-Mar-94  checkout version  cmd_list.c
24-Mar-94  create version  cmd_list.c@@/main/ibm_port/0
24-Mar-94  create branch  cmd_list.c@@/main/ibm_port
23-Mar-94  destroy sub-branch "david_work" of branch  cmd_pool.c@@/main
22-Mar-94  create version  cmd_registry.c@@/main/16
22-Mar-94  create version  cmd_view.c@@/main/138
22-Mar-94  create version  cmd_utl.c@@/main/multisite_v1/4
22-Mar-94  create version  cmd_utl.c@@/main/112
22-Mar-94  checkout version  cmd_utl.c
21-Mar-94  create version  cmd_pool.c@@/main/30
21-Mar-94  create version  cmd_annotate.c@@/main/17
18-Mar-94  create version  cmd_utl.c@@/main/multisite_v1/3
18-Mar-94  create version  cmd_utl.c@@/main/111
17-Mar-94  create version  cmd_pool.c@@/main/29
16-Mar-94  create version  cmd_view.c@@/main/137
16-Mar-94  create version  cmd_findmerge.c@@/main/23
16-Mar-94  create version  cmd_findmerge.c@@/main/22
16-Mar-94  create version  cmd_find.c@@/main/38
14-Mar-94  create version  cmd_registry.c@@/main/15
14-Mar-94  create version  cmd_findmerge.c@@/main/21
14-Mar-94  destroy sub-branch "akp_tmp" of branch  cmd_view.c@@/main
14-Mar-94  create version  cmd_registry.c@@/main/akp_tmp/1
11-Mar-94  create version  cmd_findmerge.c@@/main/20
```

Configuration Records

When the ClearCase build utility *clearmake* executes a build script, it creates a *configuration record* (*CR*) — the "bill of materials" for that particular execution of the build script. Like an event record, a CR contains "who, what, when, and where" information. It also shows how the file was built, by listing versions of source elements used, build options, makefile macros, the build script, and more. The CR is logically associated with the file(s) created by the build script, which are termed *derived objects* (*DOs*).

ClearCase includes tools for displaying and comparing CRs, through their associated DOs:

```
% cleartool catcr hello.o
Target hello.o built on host "neptune" by akp.user
Reference Time 19-Nov-93.19:30:12,
this audit started 19-Nov-93.19:30:13
View was neptune:/home/akp/akp.vws
Initial working directory was neptune:/usr/src
----------------------------
MFS objects:
----------------------------
/usr/src/hello.c@@/main/4 <19-Nov-93.19:30:05>
/usr/src/hello.h@@/main/2 <19-Nov-93.19:30:07>
/usr/src/hello.o@@19-Mov.19:30.364
----------------------------
Build Script:
----------------------------
 cc -c hello.c
----------------------------
```

For a full description of configuration records and derived objects, see Chapter 5, "Building Software with ClearCase."

The ClearCase Query Language

Chapter 2, "Version Control - ClearCase VOBs" introduced *extended naming*, a method for accessing version-controlled data. ClearCase also has a more general *query* facility, which locates objects using their associated meta-data. Queries can be used in many different contexts to locate one or more elements, branches, or versions. For example:

- A view is typically configured in terms of particular *branch names* and *version labels*.
- A quality-assurance engineer might use queries to track source versions that have achieved a particular level of coding excellence, as indicated by the value of their *CodeQuality* attribute.
- A user might use a query on *CoordinateWith* hyperlinks to determine which elements in the current source tree must be kept synchronized with elements in another tree.

Following are some examples of ClearCase queries. Note that queries can involve both ClearCase-generated and user-defined meta-data:

```
created_by(akp)
```

locates versions that were created by user *akp*

```
created_by(drp) && created_since(yesterday)
```

locates versions that were created recently by user *drp*

```
lbtype(REL1) || lbtype(REL1.01)
```

locates versions that have been labeled either *REL1* or *REL1.01*

```
merge(REL3, /main/rel2_bugfix)
```

locates elements for which a merge has been performed, incorporating data from the version labeled *REL3* into a version on the *rel2_bugfix* branch. This operation is recorded in the VOB database by a hyperlink of type *Merge*.

```
BenchMk<45 && (! trtype(bench_trigger))
```

locates versions to which the *BenchMk* attribute has been attached with a value less than 45, and to which a trigger named *bench_trigger* has not been attached.

Chapter 4

ClearCase Views

This chapter describes ClearCase's *view* facility, which implements independent, user-configurable workspaces for software development.

Requirements for a Development Workspace

Long-lived development projects typically involve many users, each working with tens, hundreds, or thousands of source files. The users work at different rates, and on different tasks: new coding for the next release; fixing bugs in one or more old releases; porting to new platforms; and so on.

Each user needs a certain set of sources with which to work. Different tasks (for example, bugfixing vs. new development) require different versions of the sources — that is, different *configurations* of the source tree(s).

Moreover, the user's workspace needs to be isolated, for purposes of editing, compiling, testing, and debugging. Ideally, the isolation should be relative, not absolute:

- Others should be able to track the user's work, and selectively integrate it into their own work.
- Conversely, others should be able to shut out (until a subsequent integration period) those changes that prove destabilizing to their own work.

Figure 4-1 illustrates such a development workspace.

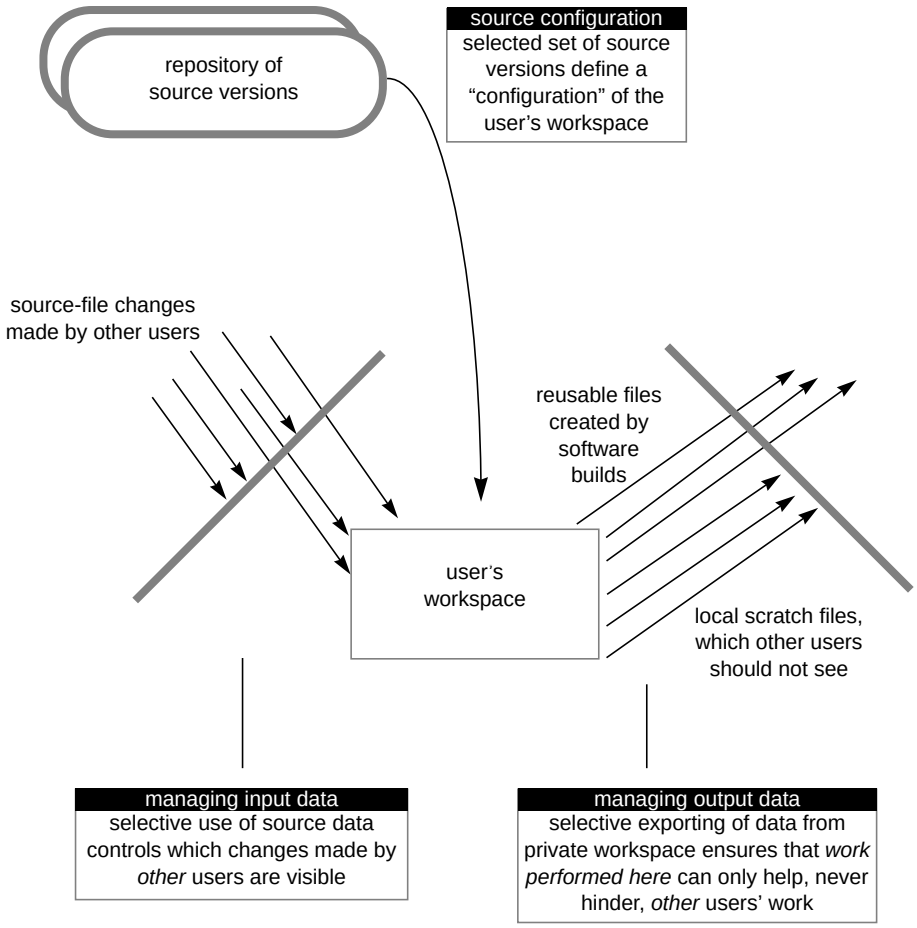


Figure 4-1 Requirements for a Development Workspace

ClearCase Development Workspaces: Views

ClearCase supports the creation of any number of independent *views*. Each view is a flexible, resource-conservative workspace that combines these services:

- **Access to permanent, shared storage** — A view accesses the correct set of source versions for the task at hand (bugfix, port to a new architecture, new development, and so on). It automatically selects a particular version of each ClearCase element, according to user-specified rules. Together, a "matched set" of versions constitutes a particular *configuration* of the central data repository.
- **Private storage** — A view provides an isolated workspace with private storage, enabling individual users or small groups to work independently of each other.

A view can be completely private to an individual user, or shared among users. A view can be accessed on a single host, or from any host in a local area network. Views are compatible with all software development tools, not just ClearCase's own programs. Users working in views can use system-supplied programs, home-grown scripts, and third-party tools to process ClearCase data.

View Implementation and Usage

In its implementation, a view is somewhat similar to a VOB (see "VOB Data Structures" on page 32). Each view has a *view storage directory*, which contains a *view database* and a *private storage area*. Access to the physical storage is mediated by a ClearCase server process, the *view_server*.

But in its usage, a view differs greatly from a VOB. As part of the central data repository, a VOB is intended to be seen by (some or) all users. By contrast, a view is not principally a repository; rather, it is a tool for *seeing* the repository. Moreover, a view is much more likely to be private (intended for one user) than public.

Processes and View Contexts

A process that works with ClearCase data must use a view. That is, any reference to a location within a VOB must be interpreted in the *context* of some view. Typically, a user enters a command to "set a view", which establishes a view context both for the current process and for any child (subsidiary) processes created subsequently. All those processes will see a particular configuration of the data repository.

A user might set different views in different windows, in order to work with two or more configurations at once. It is even possible to use multiple views in a single command — see "View-Extended Pathnames" on page 71.

Transparency and Its Alternatives

After setting a particular view, a user can use standard pathnames to access all ClearCase data; the view takes care of resolving the pathnames, component by component (Figure 4-2).

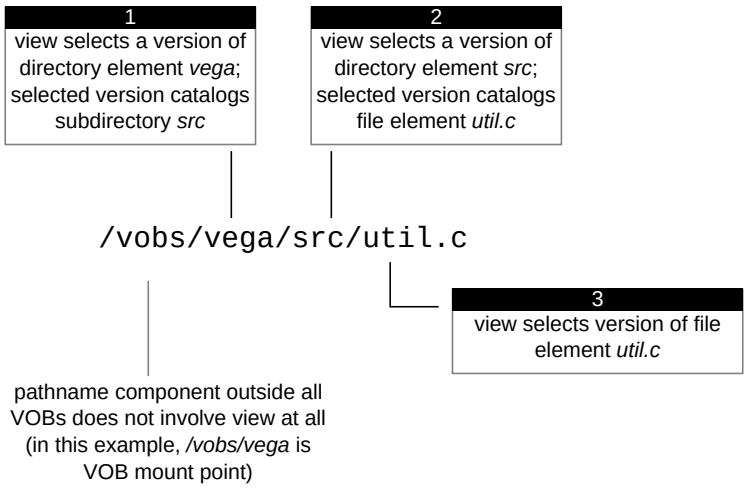


Figure 4-2 View Transparency: Resolving a Pathname

The net effect is to make all VOBs appear to be standard directory trees; an element with a complex version tree appears to be a simple file or directory. That is, automatic version-selection by a view makes the ClearCase versioning mechanism *transparent*.

Overriding Automatic Version-Selection

Automatic version selection by a view does not prevent users from explicitly accessing other versions. A ClearCase *extended pathname* can specify any version of any element, or the version of an element selected by some other view.

Version-extended pathnames, introduced in "Accessing Any Version with a Version-Extended Pathname" on page 36, specify versions by their version-tree locations. For example:

```
% cat util.c@@/main/bug404/1
```

View-Extended Pathnames

Users can also use *view-extended pathnames*, which reference the particular versions of elements selected by *other* views. All views are embedded in ClearCase's extended file name space, under a "super-root" directory, the *viewroot*. In this directory, each view appears as a separate subdirectory, named by its unique *view-tag* (Figure 4-3).

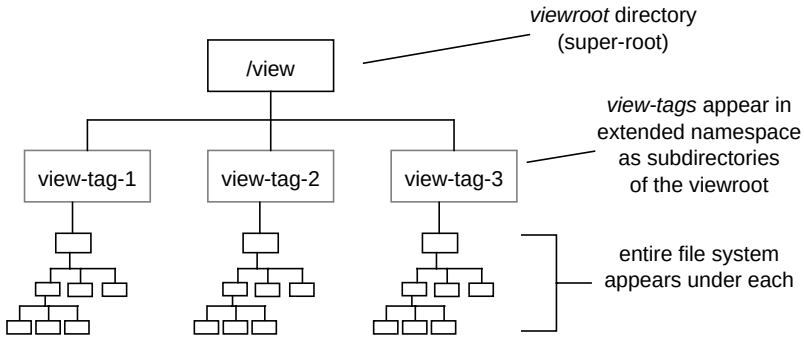


Figure 4-3 Viewroot Directory as a Super-Root

Through a view-extended pathname, any view can be used to access any available VOB. Figure 4-4 shows the components of a view-extended pathname.

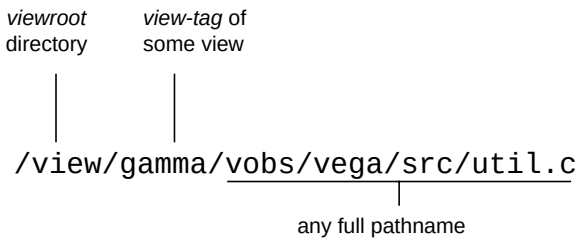


Figure 4-4 View-Extended Pathname

All of the following pathnames might be used in a single command (for example, to compare the specified versions):

```
util.c (version seen in current view)
/view/gamma/vobs/vega/src/util.c (version seen in view 'gamma')
/view/akp/vobs/vega/src/util.c (version seen in view 'akp')
```

Mechanics of Version-Selection

Each ClearCase VOB contains all the versions of the file system objects in a particular source tree. Version-selection by a view is a dynamic *filtering* of the contents of all available VOBs (not a static copying or linking operation). The actual contents of a selected version remain in VOB storage; the version is merely mapped into the view's file name space (Figure 4-5).

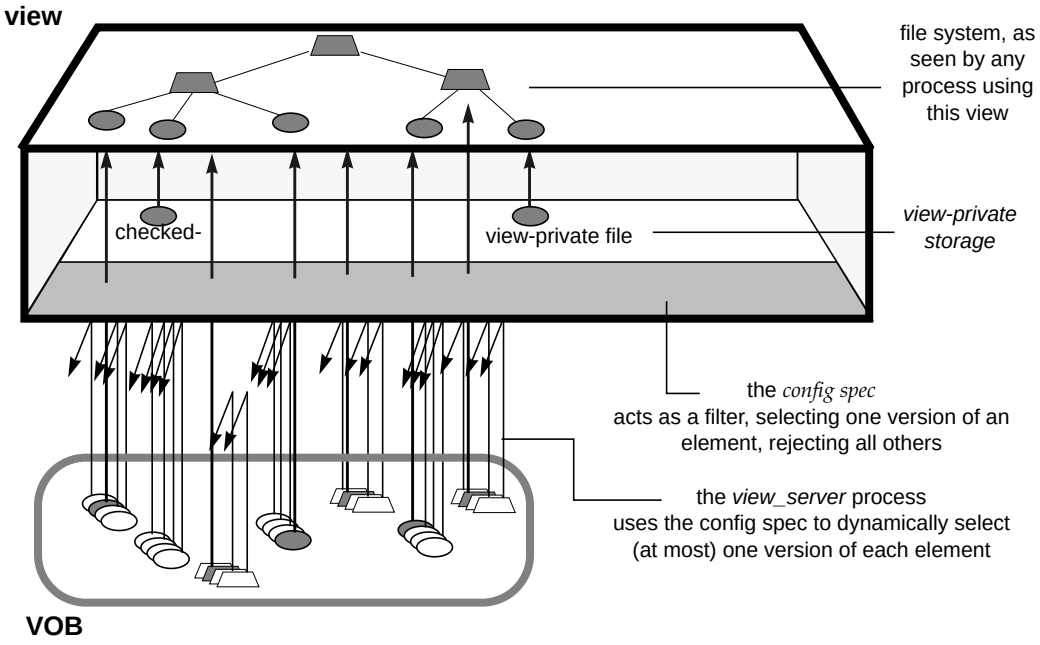


Figure 4-5 Dynamic Version Selection by a View

The Config Spec and the View Server Process

Each view has a user-defined *config spec* (short for "configuration specification"), a set of rules for selecting versions of elements. Following are typical config spec rules, along with English translations:

```
element * .../maintenance/LATEST
```

For all file and directory elements, use the most recent version on the branch named *maintenance*.

```
element -eltype c_file *.[ch] BETA_TEST
```

For all elements of element type *c_file* whose file name suffix is .c or .h, use the version labeled *beta_test*.

Each view also has an associated server process, its *view_server*. Using the config spec rules, the *view_server* automatically resolves (converts) each reference to an *element* (foo.c) into a reference to a particular *version* of the element (foo.c@@/main/4). The following steps summarize the *view_server*'s procedure for resolving element names to versions (Figure 4-6):

1. User-level software (for example, an invocation of the C compiler) references a pathname. The ClearCase MVFS,[1] which processes all pathnames within VOBs, passes the pathname on to the appropriate *view_server* process.
2. The *view_server* attempts to locate a version of the element that matches the first rule in the config spec. If this fails, it proceeds to the next rule and, if necessary, to succeeding rules until it locates a matching version.
3. When it finds a matching version, the *view_server* "selects" it and has the MVFS pass a handle to that version back to the user-level software.

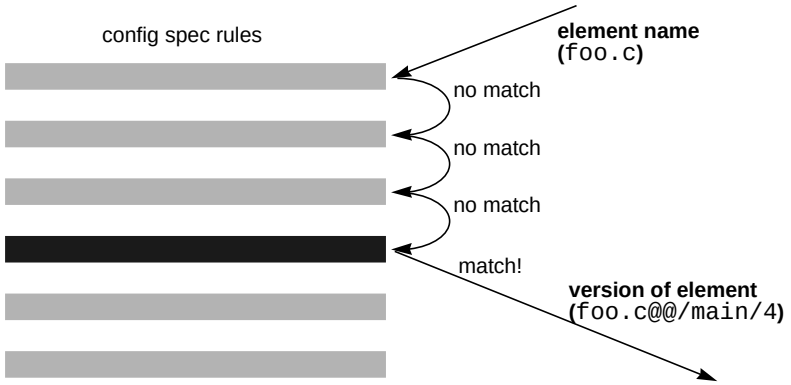


Figure 4-6 Using a Config Spec to Select a Version of an Element

The *view_server* and the MVFS use caching techniques to reduce the computational requirements of this scheme.

[1] The *multiversion file system* on a ClearCase client host is linked (statically or dynamically) with the operating system kernel.

Flexibility of Rule-Based Version Selection

This rule-based scheme for defining source configurations provides power and flexibility, supporting both "broad-brush" and "fine-tuned" approaches. The entire source environment can be configured with just a few rules, applied to all file and directory elements. When required, fine tuning can be implemented at the individual file or directory level.

For example, a user might discover that someone has created a new version of a critical header file (say, *base.h@@/main/17*) — a change that causes the user's compilations to fail. A config spec rule can "roll back" the critical file (in that user's view) to a known "good" version:

```
element base.h /main/16
  ... or ...
element base.h /main/LATEST -time 11:30
```

The second alternative above illustrates use of a *time rule*, which can "roll back" one, many, or all source files to a known "good" state. This is very useful when an application compiles correctly before lunch break, but not after.

If *no* version of an element matches any of the config spec's rules, the element is suppressed from the view. This feature can be used to configure views that are restricted to accessing a particular subset of sources.

Open-Endedness of a View

Since the rules in a config spec typically involve pathname patterns (wildcards) that apply to many files and/or directories, a view is open-ended — new directories or even entire newly-mounted VOBs are automatically incorporated into a view. There is no need to "add files to a view" explicitly.

Storage Efficiency of a View

Since access to shared versions is accomplished by mapping, rather than by copying, a view is storage-efficient. For example, all views might share version */main/3* of a rarely-changed source file. The total storage cost to the views for accessing this version is *zero* — the single version they all share is accessed from a VOB storage pool. In addition, the views can share access to the object module built from the shared source file, through ClearCase's derived object sharing feature.

View-Private Storage / The Virtual Workspace

In addition to providing access to source versions, a view provides private (isolated) storage for the files generated during software development:

- working ("checked-out") versions of source files
- object modules and executables built with ClearCase tools
- other *view-private* objects — files, directories, and links created with standard commands and tools. Examples of view-private objects include text-editor backup files, source code excerpts, test subdirectories, and test data files.

A view's private storage area is typically located within a user's home directory on a workstation. (A view shared by a group of users might have its private storage area located on a central file server host.) But the actual location of the private storage area is largely irrelevant — from the user's standpoint, the view's private storage is fully integrated with the selected versions of elements, forming a *virtual workspace*, in which all the files appear to be co-located in the VOB's directories (Figure 4-7).

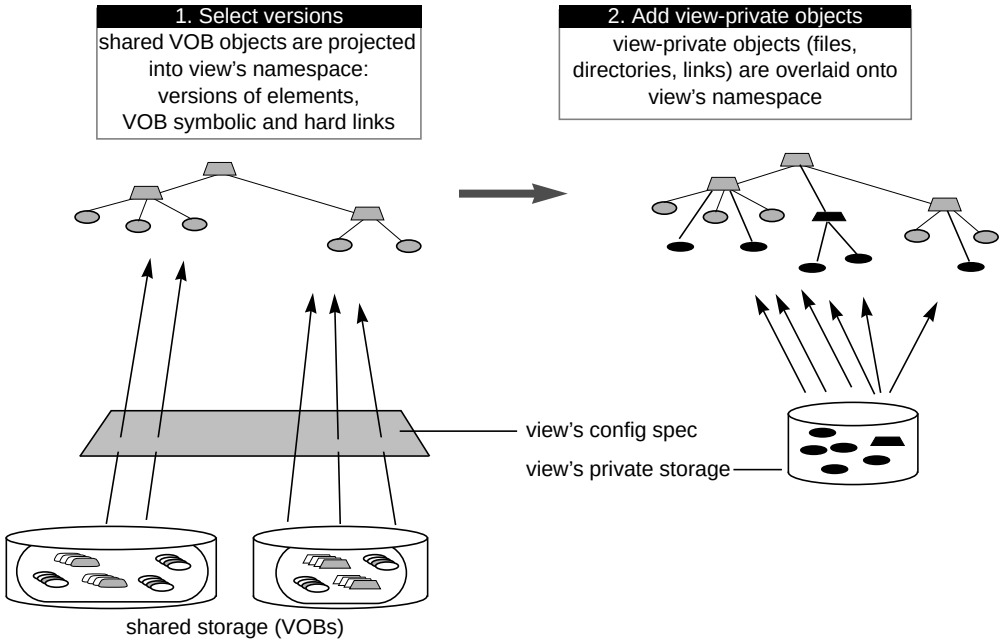


Figure 4-7 Virtual Workspace: View-Private Objects in VOB Directory

By extension (and as suggested by "View-Extended Pathnames" on page 71), a set of views is a set of "parallel" virtual workspaces — each view instantiates a variant of the complete development environment:

- Users working in different views can create files with different names or with the same names (for example, by building the same software system), without interfering with each other.
- Users can monitor the work taking place in other views, and can easily examine or copy it (subject to standard security mechanisms).
- A software build in one view can reuse the object modules and executables previously built in another view. ClearCase performs this linking operation (termed *wink-in*) automatically, but only if reusing the existing object is equivalent to rebuilding it in the current view. For more on this topic, see Chapter 5, "Building Software with ClearCase."

View Usage Scenarios

The following sections present typical scenarios of using a view for software development. These scenarios illustrate how views and VOBs interrelate, and provide opportunities to see config specs "in action", performing version-selection.

Revising a Source File / Checkout-Edit-Checkin

The simplest ClearCase development scenario involves revising a source file element on its *main* branch. (The element might also have subbranches, being used concurrently for other development tasks.) ClearCase uses a checkout-edit-checkin scheme to control the creation of new versions of elements. This scheme is similar to that used by many other version-control systems, including sccs and rcs. We track the use of this scheme to create a new version of element *util.c*.

Setting a View

"Mainline" development work requires a view configured with the default ClearCase config spec (Figure 4-8).

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

Figure 4-8 Default Config Spec

Accordingly, the user creates a shell process that is set to such a view (or starts the *xclearcase* GUI application with the view set).

Changing to the VOB Directory

Once a view context is established, the user can work directly with any VOB, for example, by *cd*'ing to the directory in which element *util.c* resides.

Before the Checkout

The user's view selects a version of *util.c* as follows:

1. The *view_server* process for the view attempts to find a version of element *util.c* that matches the first config spec rule. The pattern "*" matches the file name, but the element is not checked-out. Thus, the first rule fails to select any version.
2. The *view_server* attempts to find a version of *util.c* that matches the second rule. Once again, the pattern "*" matches; and this time, the version selector /main/LATEST matches, too; the most recent version on a branch is always matched by the special version label *latest*. Thus, this version of element *util.c* is selected to appear in the view. Any command that processes the contents of *util.c* will access the version's read-only data container in a VOB storage pool (Figure 4-9).

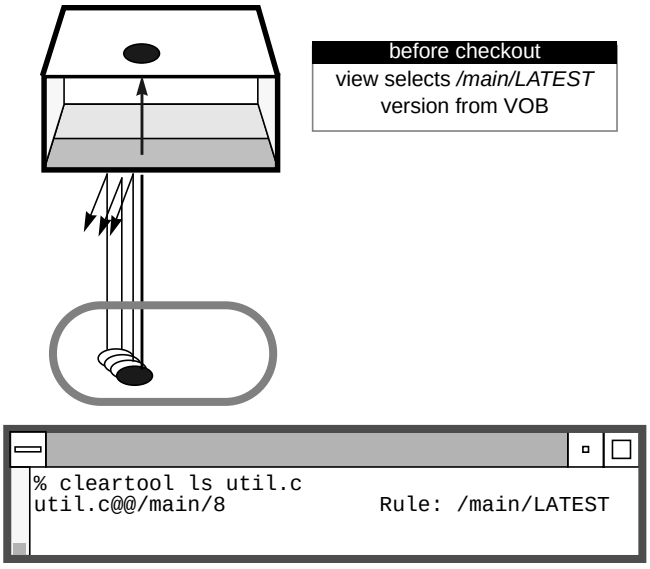


Figure 4-9 Before a Checkout

Checking Out the File

The user enters a *checkout* command on file *util.c*. This creates an editable view-private file that is a copy of the */main/LATEST* version. It also creates a *checkedout* "placeholder" object in the VOB database. (Figure 4-10). With this change, element *util.c* now matches the first config spec rule, both the pattern "*" and the keyword *checkedout*. Thus, the editable, checked-out version appears in the view.

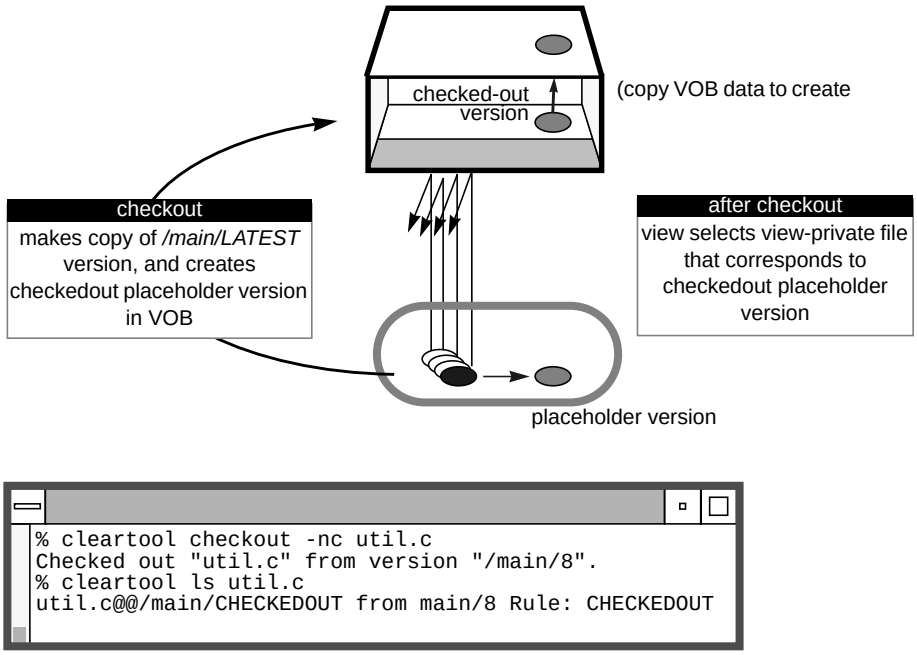


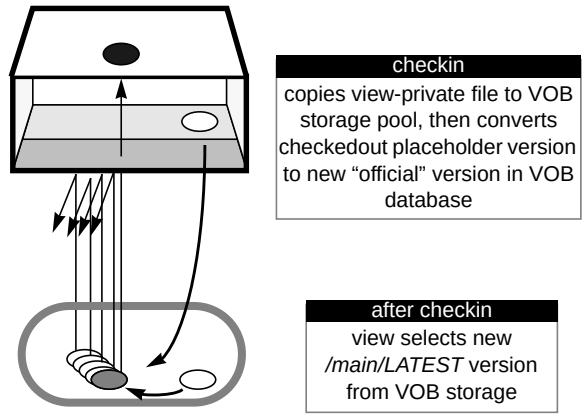
Figure 4-10 After a Checkout

Working With the File

The user works with the checked-out version of *util.c* using any available commands, programs, and scripts. All incidental files (text-editor backup files, cut-and-paste excerpts, and so on) are view-private files; such files automatically appear in the view, without needing to be selected through the config spec mechanism.

Checking In the File

The user enters a *checkin* command on file *util.c*. This creates a new, read-only version in the VOB by copying the contents of the editable, view-private file (and then deleting it). Now, the view reverts to using the /main/LATEST rule to select a version of *util.c* (Figure 4-11); it selects the newly-created version in the same way that it selected the predecessor version in "Before the Checkout" on page 79.




```
% cleartool checkin -nc util.c
Checked in "util.c" version "/main/9".
% cleartool ls util.c
util.c@@/main/9            Rule: /main/LATEST
```

Figure 4-11 After a Checkin

Parallel Development / Working on a Branch

ClearCase views play a pivotal role in ClearCase's support for *parallel development*. This topic was approached from a version-tree perspective in "Requirements for a Development Workspace" on page 67. Now, we can concentrate on how the view mechanisms work to support a *baselevel-plus-changes* model.

Different Views for Different Branches

Any number of branches can be created in an element's version tree. In general, development projects maintain their mutual independence by creating versions on different branches. Thus, a user's view must be configured to see the branches for his own project. A user working on a new port of an application might use a view that selects versions on branches named *gopher_port*; a user fixing bugs might use a view that selects versions on branches named *r2_fix*.

Figure 4-12 shows the config spec for a view used by a user working on the "gopher" port, which is based on the "R2" release of the application.

```
element * CHECKEDOUT (1)
element * .../gopher_port/LATEST (2)
element * R2 -mkbranch gopher_port (3)
```

Figure 4-12 Config Spec for Parallel Development

As the following analyses show, this config spec both imposes and follows the required branching structure:

- **Elements not modified for the port** — Many elements will not need to be modified at all to port the application. For such elements, Rules 1 and 2 fail to match, so the version-selection mechanism "falls through" to Rule 3. This rule establishes the view's *baselevel* set of versions: the versions that are labeled *R2*.
- **Beginning porting work on an element** — Prior to the first time an element is modified for the porting project, the user's view uses Rule 3 to select the version labeled *R2*, as described in the preceding paragraph. When the user enters a *checkout* command to modify the element, the -mkbranch clause is invoked, creating the required *gopher_port* branch at the version labeled *R2*, and checking out the new branch.

Now, Rule 1 applies, enabling the user to edit the checked-out version. When the user enters a *checkin* command, version 1 is created on the new branch.

- **Continuing porting work on an element** — During the porting project, the user can modify an element any number of times. All new versions go onto the element's *gopher_port* branch:
  - Before *checkout*, Rule 2 selects the most recent version on the branch.
  - After *checkout*, Rule 1 selects the checked-out, editable version.
  - After *checkin*, Rule 2 applies once again, selecting the newly-created version on the branch.

Users can *checkout* two or more branches at the same time, as long as they are working in different views.

Chapter 5

Building Software with ClearCase

ClearCase provides a software build environment closely resembling that of the "traditional" *make*(1) program. *make* was originally developed for UNIX systems, and has since been ported to other operating systems. When building software systems with ClearCase-controlled data, developers can use their hosts' native *make* programs, their organization's home-grown shell scripts, or third-party build utilities.

ClearCase includes its own build utility, *clearmake*, with many features not supported by other build programs:

- *build auditing*, with automatic detection of source dependencies, including header file dependencies
- automatic creation of permanent "bill-of-materials" documentation of the build process and its results
- sophisticated build-avoidance algorithms, guaranteeing correct results when building in a parallel development environment
- sharing of binaries among users' views, saving both time and disk storage
- parallel and distributed building, applying the resources of multiple processors and/or multiple hosts to builds of large software systems

Overview

Users perform builds (along with all other ClearCase-related work) in *views*. Typically, users work in separate, private views; sometimes, a group shares a single view — for example, during a software integration period.

As described in Chapter 4, "ClearCase Views," each view provides a complete environment for building software, including:

- a particular configuration of source versions
- a private workspace in which the user(s) can modify source files, and can use build tools to create object modules, executables, and so on

As a build environment, each view is "semi-isolated" from other views. Building software in one view can never *disturb* the work in another view — even another build of the same program taking place at the same time. But a user can examine and *benefit* from work previously performed in another view. A new build automatically shares files created by past builds, when appropriate. This saves the time and disk space involved in building new objects that duplicate existing ones.

The user can (but need not) determine what other builds have taken place in a directory, across all views. ClearCase includes tools for listing and comparing past builds.

The key to this scheme is the fact that the development team's VOBs constitute a globally-accessible repository for files created by builds — the same role they play for the source files that go into builds. A sharable file produced by a software build is termed a *derived object* (*DO*). Associated with each derived object is a *configuration record* (*CR*), which *clearmake* can use during subsequent builds to decide whether or not the DO can be reused or shared.

Figure 5-1 illustrates the ClearCase software build scheme.

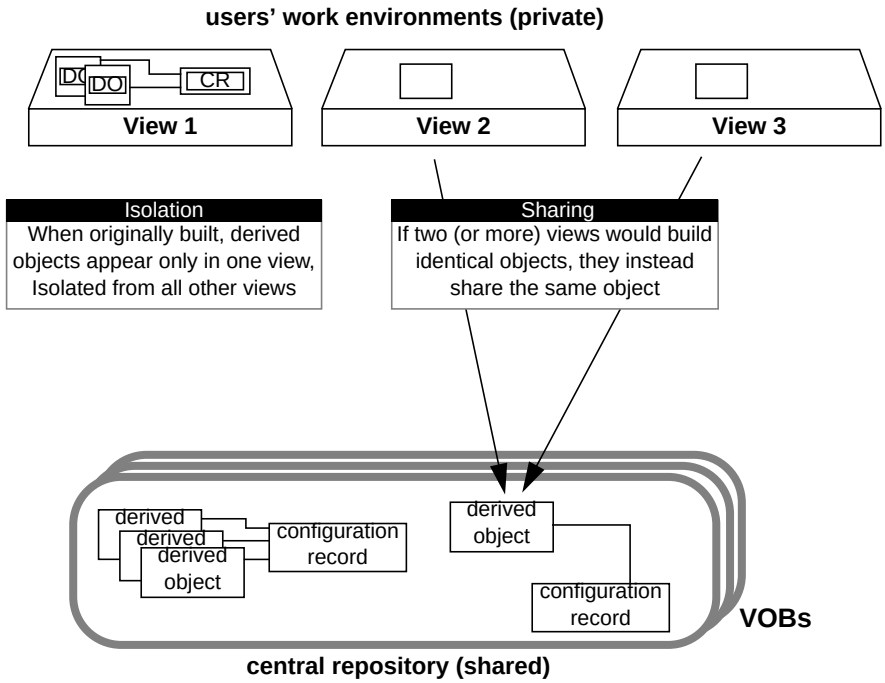


Figure 5-1 Building Software with ClearCase: Isolation and Sharing

The next section describes the way in which ClearCase keeps track of the objects produced by software builds. Later, in "Build Avoidance" on page 94, we describe the mechanism that enables such objects to be shared among views.

Dependency Tracking - MVFS and Non-MVFS Files

During build script execution, a host's *MVFS*[1] (*multiversion file system*) automatically *audits* low-level system calls performed on ClearCase data: *create*, *open*, *read*, and so on. Calls involving these objects are monitored:

- versions of elements used as build input
- view-private files used as build input — for example, the checked-out version of a file element
- files created within VOB directories during the build

Note that some of these objects are actually located in VOB storage, and others in view storage. The view combines them into a *virtual workspace*, where they all *appear* to be located in VOB directories. They are termed *MVFS files* and *MVFS directories*, because they are accessed through the MVFS.

Automatic Detection of MVFS Dependencies

Since auditing of MVFS files is completely automatic, users need not keep track of exactly which files are being used in builds — that's ClearCase's job. For example, ClearCase determines exactly which C-language source files referenced with `#include` directives are used in a build. This eliminates the need to declare such files in the makefile, and eliminates the need for dependency-detection tools, such as *makedepend*.

[1] Code that implements ClearCase's multiversion file system is linked with the operating system kernel, either statically (kernel is rebuilt when ClearCase is installed) or dynamically (at system startup time).

Tracking of Non-MVFS Files

A build can also involve files that are *not* accessed through VOB directories. Such *non-MVFS files* are not audited automatically, but are tracked "on request" (by being declared as dependencies in a makefile). This enables auditing of build tools that are not stored as ClearCase elements (for example, a C-language compiler), flag files in the user's home directory, and so on. Tracking information on a non-MVFS file includes its time-modified stamp, size, and checksum.

Derived Objects and Configuration Records

When it finishes executing a build script, ClearCase records the results, including build audit information, in the form of derived objects and configuration records.

Derived Objects (DOs)

For each new MVFS file (pathname within a VOB) created during a build, ClearCase creates a corresponding object in the VOB database. Together, the file and the database object constitute a *derived object*, or *DO*.

Sibling Derived Objects

All the derived objects created by execution of a build script have equal status, even though some of them might be explicit build targets, and others might be created as "side effects" (for example, compiler listing files). The term *siblings* describes a group of DOs created by the same script. Siblings "travel together" — whenever a DO becomes shared among views, all its siblings become shared, too.

Derived Object Identifiers (DO-IDs)

A derived object is automatically assigned a unique identifier, its *DO-ID*; this enables users to access any derived object with an extended pathname (Figure 5-2).

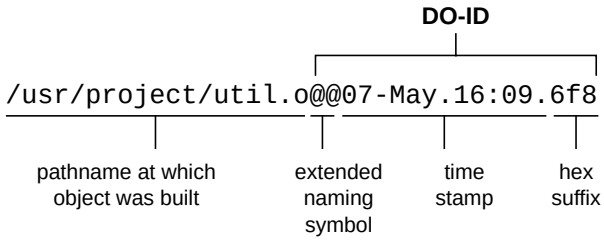


Figure 5-2 Extended Pathname of a Derived Object — DO-ID

A derived object's *DO-ID* is analogous to a version's *version-ID*: it is unique, globally accessible, and view-independent; it distinguishes a particular object in the VOB database from others that share the same file system pathname.

As development progresses, many DOs will be created at the same pathname, by builds performed in different views, and by rebuilds within a given view. But no name collisions occur, because each derived object has its own DO-ID.

For the most part, users need not be concerned with DO-IDs; they can use a derived object's standard pathname when working in the view where it was built (or any other view that shares it):

- To standard software — for example, linkers and debuggers — the standard pathname of a derived object (*util.o*) references the data file.
- To ClearCase commands, the same standard pathname references the VOB database object — for example, to access the contents of the associated configuration record.[1]

This is another instance of ClearCase's transparency feature (see "Transparency and Its Alternatives" on page 70): a standard pathname accesses one of many different variants of a file system object. Note this distinction, however:

[1] Unlike an element's *version-ID*, a derived object's *DO-ID* cannot be used to access file system data.

- A version of an element appears in a view by virtue of having been selected by a config spec rule.
- A particular derived object appears in a view as the result of a ClearCase build.

Configuration Records (CRs)

For each build script it executes, *clearmake* creates a *configuration record*, or *CR*, that is permanently associated with the newly-built derived objects. Each CR documents both the "bill of materials" and the "assembly procedure" for a set of sibling DOs.

Users can examine CRs to see a wealth of information:

- **Input files** — Identifiers for objects used as build input: *version-IDs* (for versions of elements); *DO-IDs* (for derived objects created as subtargets); timestamps (for view-private files and for each non-MVFS file explicitly declared as a dependency in the makefile)
- **Output files** — The pathname and DO-ID of each derived object created during the build (all the *siblings*)
- **Build procedure** — The text of the *makefile*'s build script, including expansions of all make macros
- **General data** — The user who performed the build; the host used to execute the build script; the view in which the build took place; the date and time at which build script execution started

Operations Involving DOs and CRs

ClearCase includes commands for working with derived objects and configuration records:

- **DO listing** — A user can list all the DOs created at a particular pathname, across all views:

```
% cleartool lsdo -long util.o
14-Mar-94.17:50:49     Allison K. Pak (akp.users@neptune)
   create derived object "util.o@@14-Mar.17:50.331"
   references: 1  => neptune:/usr/akp/tut/tut.vws
14-Mar-94.17:48:40     Allison K. Pak (akp.users@neptune)
   create derived object "util.o@@14-Mar.17:48.260"
   references: 1  => neptune:/usr/akp/tut/fix.vws
14-Mar-94.17:45:42     Allison K. Pak (akp.users@neptune)
   create derived object "util.o@@14-Mar.17:45.215"
   references: 1  => neptune:/usr/akp/tut/old.vws
```

- **CR display** — A user can display the contents of an individual CR:

```
% cleartool catcr util.o@@14-Mar.17:48.260
Target util.o built by akp.users
Host "neptune" running HP-UX A.09.01 (9000/715)
Reference Time 14-Mar-94.17:48:35, this audit started
14-Mar-94.17:48:37
View was neptune:/usr/akp/tut/fix.vws
Initial working directory was /tmp/akp_neptune_hw/src
----------------------------
MVFS objects:
----------------------------
/tmp/akp_neptune_hw/src/hello.h@@/main/1
<20-May-93.14:46:00>
/tmp/akp_neptune_hw/src/util.c@@/main/rel2_bugfix/1
<14-Mar-94.17:48:32>
/tmp/akp_neptune_hw/src/util.o@@14-Mar.17:48.260
----------------------------
Variables and Options:
----------------------------
MKTUT_CC=cc
----------------------------
Build Script:
----------------------------
c -c util.c
----------------------------
```

ClearCase can also display an entire *CR hierarchy*, which mirrors the multiple-level structure of a *makefile*-based build (building a top-level target requires building of several second-level targets, which requires building of some third-level targets, and so on).

- **CR comparison** — ClearCase can use CRs to compare different builds of the same target, showing differences in how the objects were built:

```
% cleartool diffcr util.o@@14-Mar.17:48.260
util.o@@14-Mar.17:45.215
< Target util.o built by akp.users
> Target util.o built by akp.users
< Reference Time 14-Mar-94.17:48:35, this audit started
14-Mar-94.17:48:37
> Reference Time 14-Mar-94.17:45:33, this audit started
14-Mar-94.17:45:40
< View was neptune:/usr/akp/tut/fix.vws
[uuid 7033a691.3f8011cd.ae42.08:00:09:41:e6:03]
> View was neptune:/usr/akp/tut/old.vws
[uuid d4a3a27f.3f7f11cd.adce.08:00:09:41:e6:03]
----------------------------
MVFS objects:
----------------------------
< /tmp/akp_neptune_hw/src/util.c@@/main/rel2_bugfix/1

<14-Mar-94.17:48:32>
> /tmp/akp_neptune_hw/src/util.c@@/main/1
<20-May-93.17:05:00>
----------------------------
< /tmp/akp_neptune_hw/src/util.o@@14-Mar.17:48.260
> /tmp/akp_neptune_hw/src/util.o@@14-Mar.17:45.215
```

- **Version annotation** — A user can annotate some or all the versions listed in a CR (or an entire CR hierarchy) with a particular version label or attribute.
- **View configuration** — A user can configure a view to select exactly the versions that were used to build a particular DO, or set of DOs.
- **Build management** — ClearCase uses DOs and CRs to implement its build-avoidance and derived-object-sharing capabilities.

Build Avoidance

An essential aspect of *makefile*-based software building is avoiding unnecessary builds. *clearmake*'s build-avoidance algorithm, based on the CRs produced by build audits, is more sophisticated than that of other *make* variants. The standard *make* timestamp-based algorithm cannot guarantee correct results in a parallel-development environment. An object module's being newer than a particular version of its source file does not guarantee that it was built using that version. In fact, reusing recently-built object modules and executables is likely to be incorrect when a previous release of an application is to be rebuilt from "old" sources.

clearmake's build-avoidance algorithm does guarantee correct results. Moreover, *clearmake* can avoid building a new derived object by reusing an existing one built in any view, not just the user's own view. Thus, a derived object can be *shared* by any number of views.

Configuration Lookup and Wink-In

When requested to update a build target, *clearmake* first determines whether an existing derived object in the current view qualifies for reuse. If not, *clearmake* evaluates other existing derived objects that were built at the same pathname.

The process of "qualifying" a candidate DO is termed *configuration lookup*. It involves matching the DO's configuration record against the user's current *build configuration* (Figure 5-3):

- **Files** — The version of an element listed in the CR must match the version selected by the user's view; any view-private files or non-MVFS files listed in the CR must also match.
- **Build procedure** — The build script listed in the CR must match the script that will be executed if the target is rebuilt. The scripts are compared with all make macros expanded; thus, a match occurs only if the same build options apply (for example, "compile for debugging").

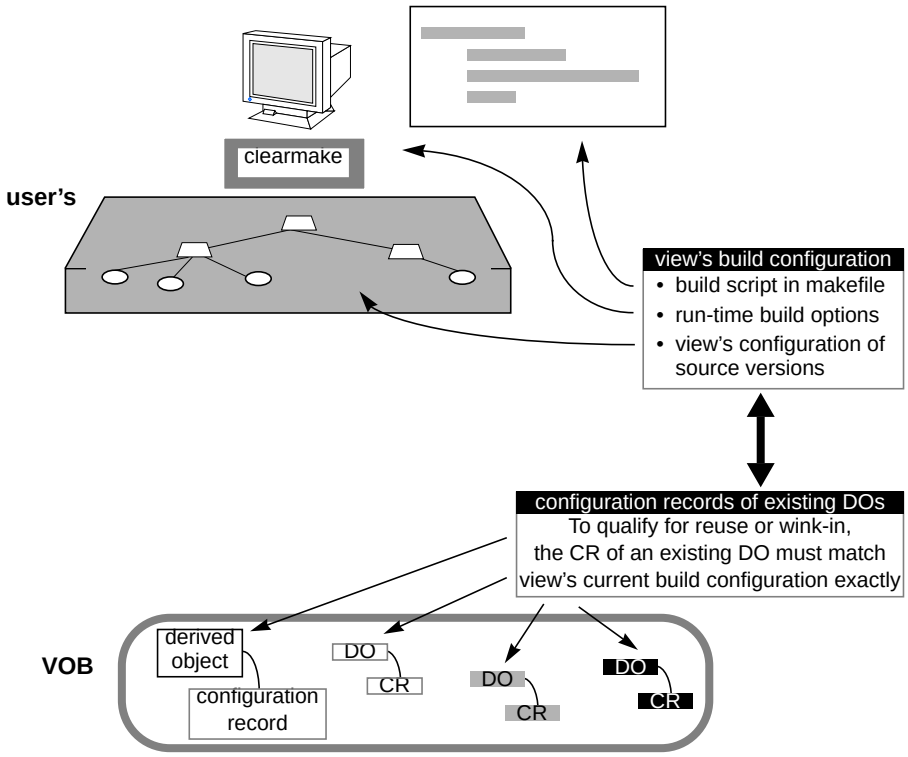


Figure 5-3 Configuration Lookup

The search ends as soon as *clearmake* finds a DO whose configuration exactly matches the user's current build configuration. In general, a configuration lookup can have these outcomes:

- **Reuse** — If the DO (and its siblings) already in the user's view match the build configuration, *clearmake* simply keeps them.
- **Wink-in** — If another, previously-built DO matches the build configuration, *clearmake* causes the DO and its siblings to appear in this view. This operation is termed *wink-in*.
- **Rebuild** — If configuration lookup fails to find any DO that matches the build configuration, *clearmake* executes the target's build script. This creates one or more new DOs, and a new CR.

Reuse and wink-in take place only if *clearmake* determines that a newly-built derived object would be identical to the existing derived object. In general, wink-in takes place when two or more views select some or all of the same versions of source elements. For example, a user might create a "clone" view, with exactly the same configuration as an existing view. Initially, the new view sees all the sources, but contains no derived objects. Executing *clearmake* causes a wink-in of many derived objects from the existing view.

Hierarchical Builds

In a hierarchical build, some objects are built and then used to build others. *clearmake* performs configuration lookup separately for each target. To ensure a consistent result, it also applies this principle: if a new object is created, then all targets that depend on it must be rebuilt.

Build Auditing with *clearaudit*

Some organizations, or some individual users, may want to use ClearCase build auditing without using the *clearmake* program. Others may want to audit development activities that do not involve *makefiles* at all. These users can do their work in an *audited shell*, a standard shell with build auditing enabled.

For example, a technical writer might produce formatted manual page files by running a shell script that invokes *nroff*(1). When the script is executed in a shell created by *clearaudit*, ClearCase creates a single configuration record, recording all the source versions used. All MVFS files read during execution of the audited shell are listed as inputs to the "build". All MVFS files created become derived objects, associated with the single configuration record.

Storage of DOs and CRs

This section discusses the physical storage of derived objects (DOs) and configuration records (CRs). As first discussed in "Derived Objects and Configuration Records" on page 89:

- A derived object is a compound entity, consisting of both a data file (its *data container*) and a corresponding VOB database object (which has a unique *DO-ID*).
- Each derived object has an associated configuration record.

When a DO is first built, its data container and CR are placed in the private storage area of the user's view (Figure 5-4). When the DO becomes shared, through *wink-in*, its data container and CR migrate to VOB storage. This implements a "localize the cost" disk-space allocation scheme: the cost of a derived object used by just one view is borne by that view; the cost of a shared derived object is borne by the VOB, which is a shared resource.

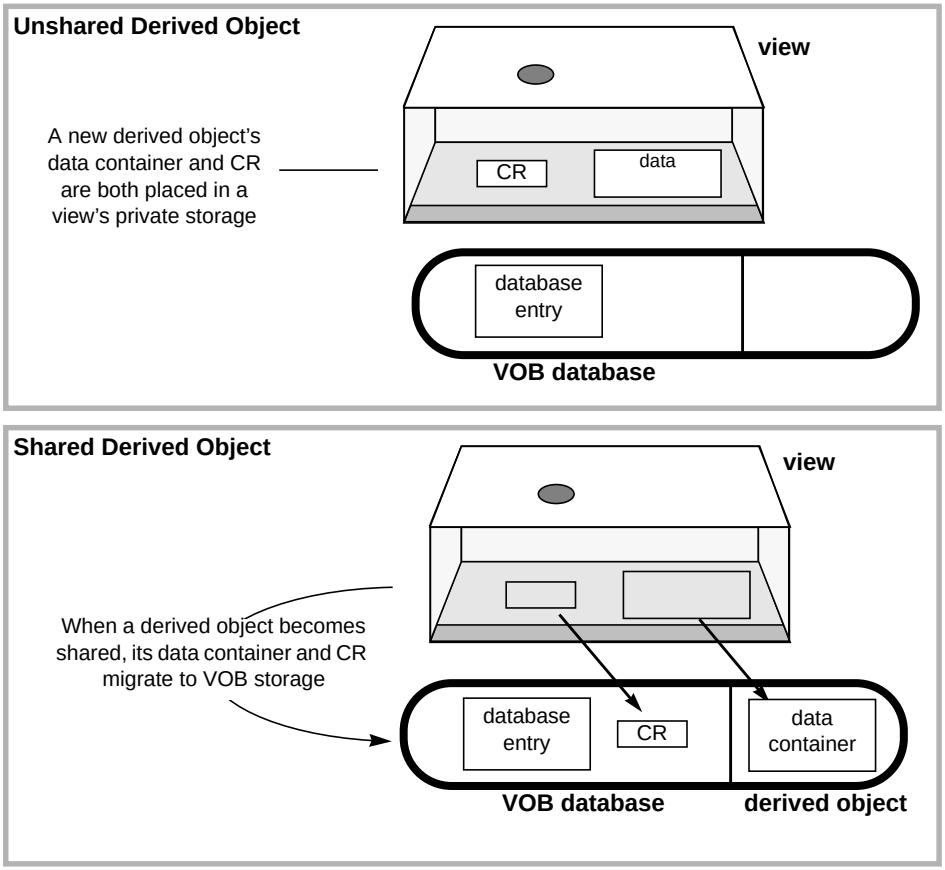


Figure 5-4 Storage of Derived Objects and Configuration Records

clearmake Compatibility with Other *make* Programs

Many *make* utilities are available in the multiple-architecture, multiple-vendor world of open systems. ClearCase's *clearmake* utility shares features with many of them, and has some unique features of its own.

Users can adjust *clearmake*'s level of compatibility with other *make* programs:

- **Suppressing *clearmake*'s special features** — Command options can selectively turn off such features as wink-in, comparison of build scripts, comparison of automatically-detected dependencies, and creation of DOs and CRs. Configuration lookup can be turned off altogether, so that the standard timestamp-based algorithm is used for build avoidance.
- **Enabling features of other *make* programs** — *clearmake* has several compatibility modes, which provide for partial emulations of popular *make* programs, such as Gnu Make and Sun Make.

Using Another *make* Instead of *clearmake*

Users can achieve absolute compatibility with other *make* programs by actually using those programs to perform builds. But in a build with a "standard" *make*, there is no build auditing, no configuration lookup, and no sharing of DOs. The MVFS files created by the build are simply view-private files, not derived objects (unless the *make* program is executed in a *clearaudit* shell).

Parallel and Distributed Building

clearmake includes support for *parallel* building (concurrent execution of a set of build scripts), and for *distributed* building (use of other hosts to execute build scripts). A command option specifies the number of hosts to use; hostnames to use are read from a *build hosts* file (Figure 5-5).

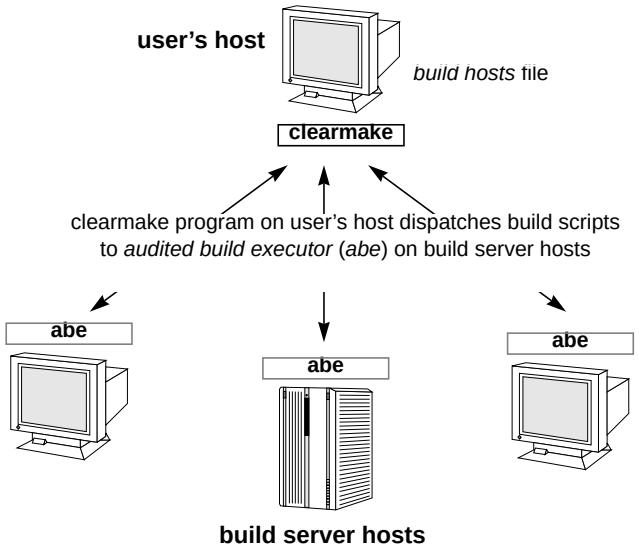


Figure 5-5 Parallel and Distributed Building

For example, a user might perform a three-way build, all of whose processes execute on a single multiprocessor compute server; an overnight build might be distributed across all the workstations in the local network.

The Parallel Build Procedure

Before starting a parallel build, *clearmake* considers each target, determining whether an existing DO can be reused, or an actual build is required. Then, it organizes all the actual build work into a hierarchical master schedule.

clearmake then dispatches the build scripts to some or all the hosts listed in the build hosts file. It uses a dynamic load-balancing algorithm that makes best use of the available hosts, preferring lightly-loaded hosts to ones that are relatively busy. Access to a "build server" host can be restricted to particular times, to particular users working on particular remote hosts, and so on.

Execution of build scripts is managed on each remote host by the *ClearCase audited build executor* (*abe*), which is invoked through standard remote-shell facilities.

Building on a Non-ClearCase Host

Many organizations develop multiple variants of their products, targeted at different platforms. But ClearCase may not currently be available for some of the platforms. How, then, can users build all the required product variants?

One solution is *cross-development* — for example, perform a build on a SunOS host that produces executables for the unsupported host. ClearCase provides another solution — hosts on which ClearCase is not installed can still access ClearCase data, using standard network file-sharing services (such as NFS). This capability is termed *non-ClearCase access*.

Non-ClearCase access takes advantage of view *transparency*. Through automatic version-selection, a view makes any VOB appear to be a standard directory tree. Any such "VOB image" can be "exported" to a non-ClearCase host (Figure 5-6).

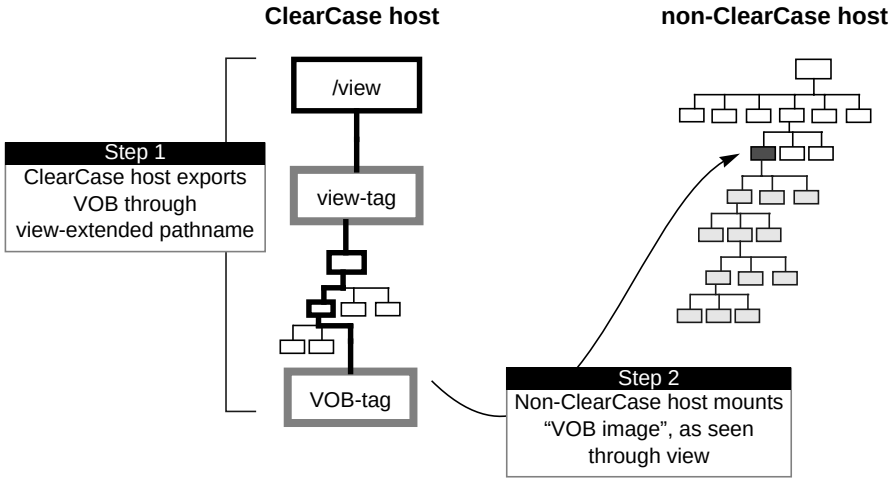


Figure 5-6 Non-ClearCase Access

Users on the non-ClearCase host can perform builds within these "VOB image" directory trees, using native *make* utilities or other local build tools. The object modules and executables produced by such builds are stored in the view that instantiates the VOB image(s).

Limitations of Non-ClearCase Access

Since ClearCase software is not running on a non-ClearCase host, non-ClearCase access has some limitations. In many cases, there are simple workarounds to the limitations.

- ClearCase build auditing, configuration lookup, and wink-in are not performed during builds on such hosts.
- Files produced by such builds are not derived objects, and have no configuration records. (Users can employ remote-shell techniques to overcome this limitation, so that the files built on a non-ClearCase host *do* become derived objects.)
- Users cannot execute ClearCase development commands, such as *checkout* and *checkin*, on non-ClearCase hosts. (But they can probably remote-login to a ClearCase host for such purposes.)
- Users cannot change the VOB image, by reconfiguring the view, from non-ClearCase hosts. (Again, a remote-login capability addresses this issue.)

Derived Objects as Versions of Elements

A typical software system is constructed hierarchically. For example, executables are built only after programming libraries have been built. With ClearCase, stable versions of libraries and other subtargets can be stored as versions of elements. At build time, the subtarget can be used essentially as a source file, instead of (potentially) being rebuilt. Derived objects that have become versions of elements are termed *DO versions*.

A development group can use DO versions to make its "product" (for example, a library) available to other groups. Typically, the group establishes a "release area" in a separate VOB. For example:

- A library is built by its development group in one location — perhaps */vobs/monet/lib/libmonet.a*.
- The group periodically "releases" the library by creating a new version of a publicly-accessible element — perhaps */vobs/publib/libmonet.a*.

Product Releases

It is easy to generalize the idea of maintaining a *development* release area to maintaining a *product* release area. For example, a Release Engineering group might maintain one or more "release tree" VOBs. The directory structure of the tree(s) mirrors the hierarchy of files to be created on the release medium. (Since a release tree involves directory elements, it is easy to change its structure from release to release.) A release tree might be used to organize "Release 2.4.3" as follows:

- When an executable or other file is ready to be released, a release engineer checks it in as a version of an element in the release tree.
- An appropriate version label (for example, *REL2.4.3*) is attached to that version, either manually by the engineer or automatically with a trigger.
- When all files to be shipped have been released in this way, a release engineer configures a view to select *only* versions with that version label. As seen through this view, the release tree contains exactly the set of files to be released.
- To cut a release tape, the engineer issues a command to copy the appropriately configured release tree.

Chapter 6

ClearCase Process Control

Software development policies and procedures differ widely from organization to organization, but share a common goal: improving the quality and time-to-market of the software under development. This chapter discusses ClearCase's process and policy control mechanisms, which are useful in activities that parallel the development process, such as quality assurance and ECO ("engineering change order") administration.

ClearCase process control tools enable:

- monitoring of the entire development process
- enforcement of development policies — controlling who can make changes; controlling how and where changes can be made
- automated communications, targeted as required at individual users, development groups, and managers
- organization and cross-referencing of all development data: source code, memos, design proposals, technical manuals, and so on

Information Capture and Retrieval

Reliable logging of information concerning development activities plays a fundamental role in any process-control scheme. ClearCase provides both automatic and user-controlled facilities for capturing such information. It also provides simple commands for retrieving the information, along with extensions for creating sophisticated reports (Figure 6-1).

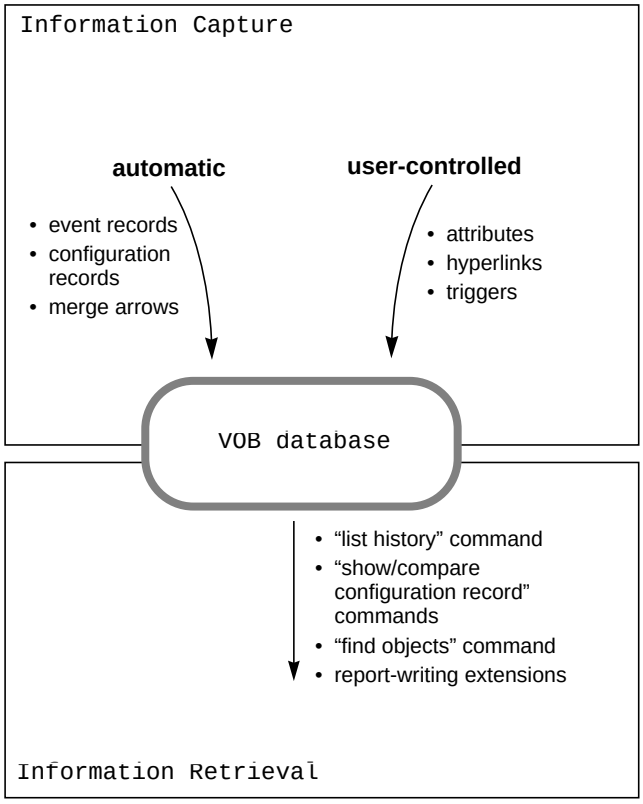


Figure 6-1 Information Capture and Retrieval

Automatic Information Capture

When ClearCase client programs make changes to the development environment, they automatically store *records* in VOB databases. For example:

- When a user checks out a file element in order to edit it (and ultimately, create a new version of it), *cleartool* automatically stores a `checkout version` event record in the element's VOB. This event record is accessible to all users (and managers), through the "list checkouts" or "list history" command.
- When a release engineer revises a source configuration by moving the version label *R1.3_FINAL* from one version of an element to another, two event records are automatically stored, and can be viewed with the "list history" command:

```
27-Oct.11:30   drp       remove label "R1.3_FINAL" from version "f2@@/main/1"
"Moved label "R1.3_FINAL" to version "/main/34"."

27-Oct.11:30   drp       make label "R1.3_FINAL" on version "f2@@/main/2"
"Moved label "R1.3_FINAL" from version "/main/33"."
```

- The ClearCase build utility, *clearmake*, automatically creates *configuration records* (*CRs*) that document software builds, as described in Chapter 5, "Building Software with ClearCase." The "show configuration record" command retrieves and lists a CR; it has many filtering and selection options to enable precise information retrieval. For example, a command can list the versions of all header files used in the compilation of object module *util.o* in a particular build of executable *hello*.
- Management of parallel development environments is made possible by the automatic recording of *merges*. Whenever a user integrates the work stored on a subbranch of an element back into the *main* branch (or performs a merge from the *main* branch "outward"), a *merge arrow* is created in the element's VOB database, logically connecting the merged versions. The presence (or absence) of merge arrows enables a project leader to determine exactly what work is required to integrate all of a project's branch-resident work back into the main line of development.

User-Controlled Information Capture

The meta-data annotations introduced in Chapter 3, "ClearCase Meta-Data" can be used to record process-control information in VOB databases. Annotations can be attached to objects manually, with commands such as "make attribute". Alternatively, scripts that include such commands can be invoked automatically when certain ClearCase operations take place. For more on automatically-invoked procedures, see "Triggers" on page 113.

Meta-data annotations include:

- **Version labels** — Version labels record development milestones: baselevels, major releases, and so on. In addition to playing a pivotal role in organizing development activities, version labels make it easy to answer fundamental process-control questions, such as "What version of that file went into the last release?".
- **Attributes** — Attributes (*name/value* pairs) can be applied to elements, branches, and versions. Attributes can record process-control metrics commonly applied to source code modules, such as code quality and comment density.

Attributes can also be used to define groups of versions involved in the same task. For example, ClearCase's integrations with third-party bug-tracking systems place attributes on versions created to fix bugs; the attribute's value indicates the associated bug report.

- **Hyperlinks** — Objects can be connected with hyperlinks to implement *requirements tracing*. For a relatively loose implementation, an element containing source code might be hyperlinked to the element containing its functional specification document (Figure 6-2).

For a tighter implementation, individual versions of the source module might be hyperlinked to the corresponding versions of the functional specification, as they both evolve. *Hyperlink inheritance* makes this a practical alternative to establishing links at the element level: as one or both elements evolve, the newly-created versions effectively "inherit" the closest hyperlink involving their direct ancestors (Figure 6-2).

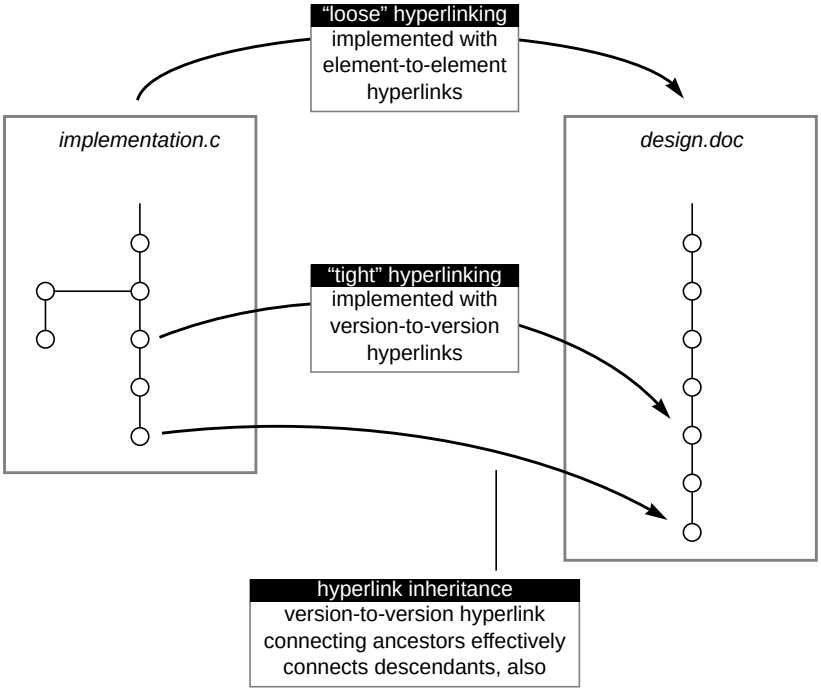


Figure 6-2 Hyperlinks / Hyperlink Inheritance

(Merge arrows, discussed in "Automatic Information Capture" on page 107, are implemented as version-to-version hyperlinks.)

Information Retrieval

The preceding sections have mentioned ClearCase commands that retrieve event records and configuration records information from VOB databases. There is also a *describe* command, which lists the version labels, attributes, and/or hyperlinks attached to a particular object. For example, this command reveals inherited hyperlinks of a particular version:

```
% cleartool describe -ihlink DesignDoc hello.c@@/main/2
hello.c@@/main/2
Inherited hyperlinks: DesignDoc@366@/usr/hw
/usr/hw/src/hello.c@@/main/1 ->
/usr/hw/src/hello_dsn.doc@@/main/1
```

ClearCase also includes a *query language*, which locates objects using their meta-data. Following are some applications of the query facility to process control:

- A quality-assurance engineer uses a query on the *CodeQuality* attribute to determine which source versions fail to achieve a desired level.
- A development project leader determines which elements have already had their *gopher_port* version merged into the *rel2_bugfix* branch.
- A group leader generates a list of all versions created by user *akp*, or perhaps all versions created by *akp* in the last month.

The *query_language* manual page provides a complete description of this facility.

Access Control

Any process-control scheme requires an *access control* component: limiting the use of individual data objects to particular users or groups, and restricting the use of specific commands to authorized individuals. This section discusses ClearCase mechanisms for controlling users' access to file system data and to configuration-management data structures (Figure 6-3). See "Triggers" on page 113 for a discussion of the mechanisms that restrict command usage.

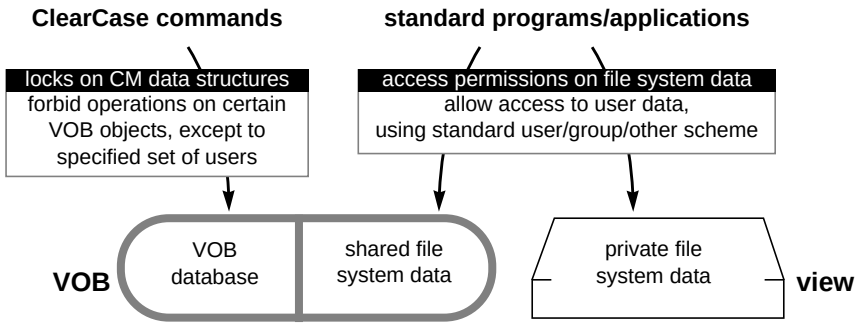


Figure 6-3 Access Control Schemes

Access Permissions

Access to file system objects is governed by a scheme that closely resembles the standard UNIX *user/group/others* permissions model. Administrators can design development environments that are both secure and flexible. For example, a VOB can be "opened up" to users in several groups, without having to grant access rights to the "entire world" (*others*).

All file system objects managed by ClearCase have access permissions, which can be listed with a standard operating system command:

```
% ls -l util.c
-r--r--r--   1 drp      dvt          511 Feb 28 12:28 util.c
```

The standard UNIX meanings of these permissions apply: the object is owned by a particular *user*, and belongs to a particular *group*; three kinds of access ("read", "write", and "execute") are either granted or denied to the user/owner, to members of the object's group; and to all others. An element, its branches, and all its versions have the same access permissions.

ClearCase uses an object's permissions to control certain operations at the configuration management level — for example, to determine who can *checkout* an element.

Access to Physical VOB Storage

Given a VOB's role as a component of the permanent data repository, access to physical VOB data storage is strictly controlled:

- Standard commands and programs cannot modify VOB storage. Users must enter ClearCase commands to modify VOBs.
- Physical storage for a VOB is protected against "back-door" access. For example, all files in a VOB's storage pools belong to a privileged user, the *VOB owner*, and are maintained in a read-only state.

Locks on VOB Objects

Configuration management data structures are implemented by various *objects* stored in VOB databases. Whereas *access permissions* work primarily at the file-system level, *locks* protect all kinds of objects at the VOB-database level — elements, branches, branch types, even entire VOBs. A lock on an individual VOB objects renders that object unchangeable by any user (except those included on an optional *exception list*).

The effect of a lock can be small or large. For example, one lock might prevent any new development on a particular branch of a particular element; another lock might apply throughout a VOB, preventing creation of any new element of type *compressed_file*, or usage of the version label *RLS_1.3*.

Locks are useful for implementing temporary restrictions. For example, during an integration period, a lock on a single object — the *main* branch type — prevents all users who are not on the integration team from making any changes.

Obsolete Objects

Locks can also be used to "retire" old names and data structures that are no longer used. For this purpose, the locked objects can be tagged as *obsolete*, effectively making them invisible to most commands.

Triggers

Automated user-defined procedures play an important role in process control. Such procedures can increase the degree to which development activities are guided, monitored, and/or controlled. For example, automated procedures can:

- disallow an operation, in order to enforce an organization's development policies ("all checkins must have a comment of at least 50 characters")
- notify users and/or managers of significant changes in the development environment
- supplement the information automatically captured by ClearCase commands with additional data, storing it as attributes

Automated procedures are implemented by *triggers*, which monitor ClearCase operations. (Access permissions and locks "guard" objects; triggers "guard" operations.) Performing a certain ClearCase-level operation on a certain object causes a trigger to *fire*, executing one or more user-defined procedures. The context in which a trigger fires can be defined either broadly or narrowly, in terms of certain elements, meta-data, and ClearCase operations. For example, the following are very broad and very narrow contexts, respectively:

- ... whenever *any* element in a VOB is modified in *any* way
- ... when a *particular* attribute is used on a *particular* branch of a *particular* element

The trigger procedure is typically a script. It can perform any analysis or test, based on file system data, on meta-data, on other factors (such as the user's identity), or on any combination thereof.

Pre-Operation and Post-Operation Triggers

Perhaps the most important aspect of a trigger definition is whether it is to fire before the ClearCase operation takes place, or after. *Pre-operation* triggers control ClearCase processing; *post-operation* triggers supplement it (Figure 6-4).

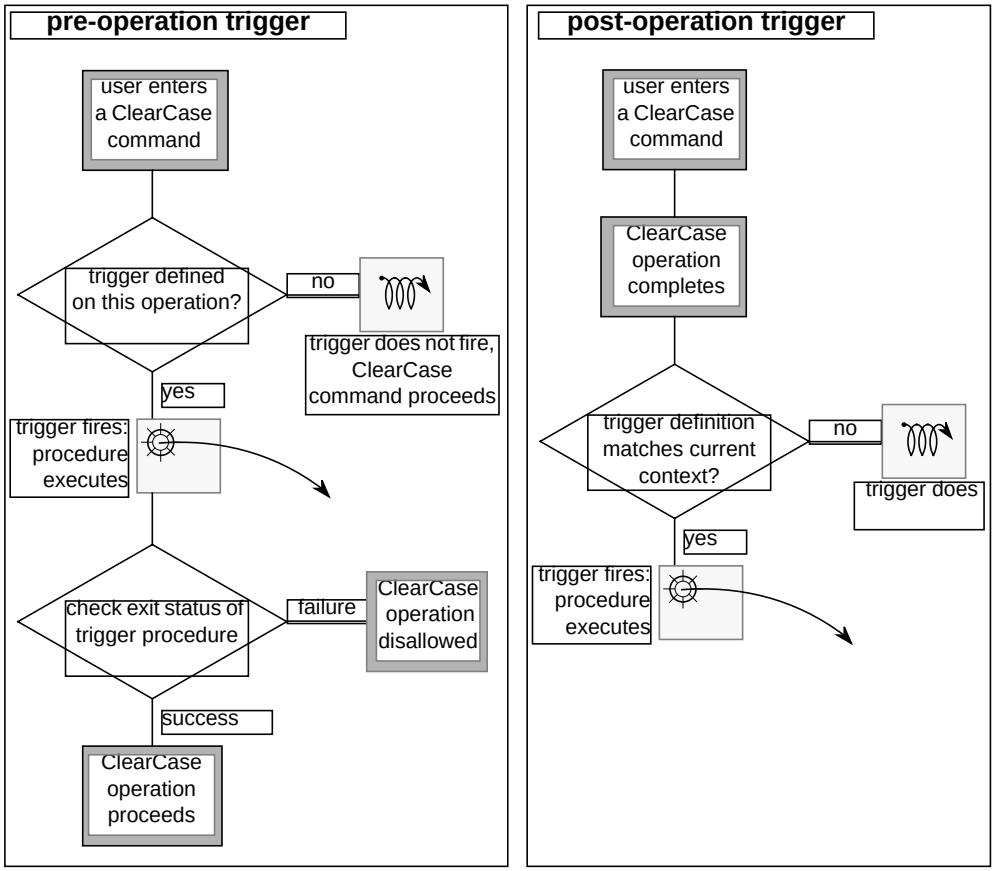


Figure 6-4 Pre-Operation and Post-Operation Triggers

Policy Enforcement - An Example

The mechanisms described in the preceding sections constitute a powerful "toolset", which administrators and project leaders can use to implement their organization's policies. To illustrate how the mechanisms work together, we present an example of implementing a "state transition" process model.

Scenario

A “final integration” task for a product release is to progress from the *active* state to the *frozen* state to the *released* state:

- During the initial *active* state, all users will be able to modify elements, but only on branches named *rls2_integ*.
- After an element enters the *frozen* state, only “priority 1” bugfixes will be permitted, and only users *david* and *sakai* will be permitted to make the fixes.
- When the task is finished, it will enter the *released* state. To record the final source configuration, the version label *RLS2* will be applied to the entire source tree. To prevent possible confusion, all further work on *rls2_integ* branches will be forbidden, as will further activity involving the *RLS2* label.

Implementation

ClearCase triggers, attributes, and locks all play a role in defining the state transition model (Figure 6-5). The three states are modeled as the permitted values of a string-valued attribute, *IntegStatus*, which is applied to each *rls2_integ* branch created during the final integration task.

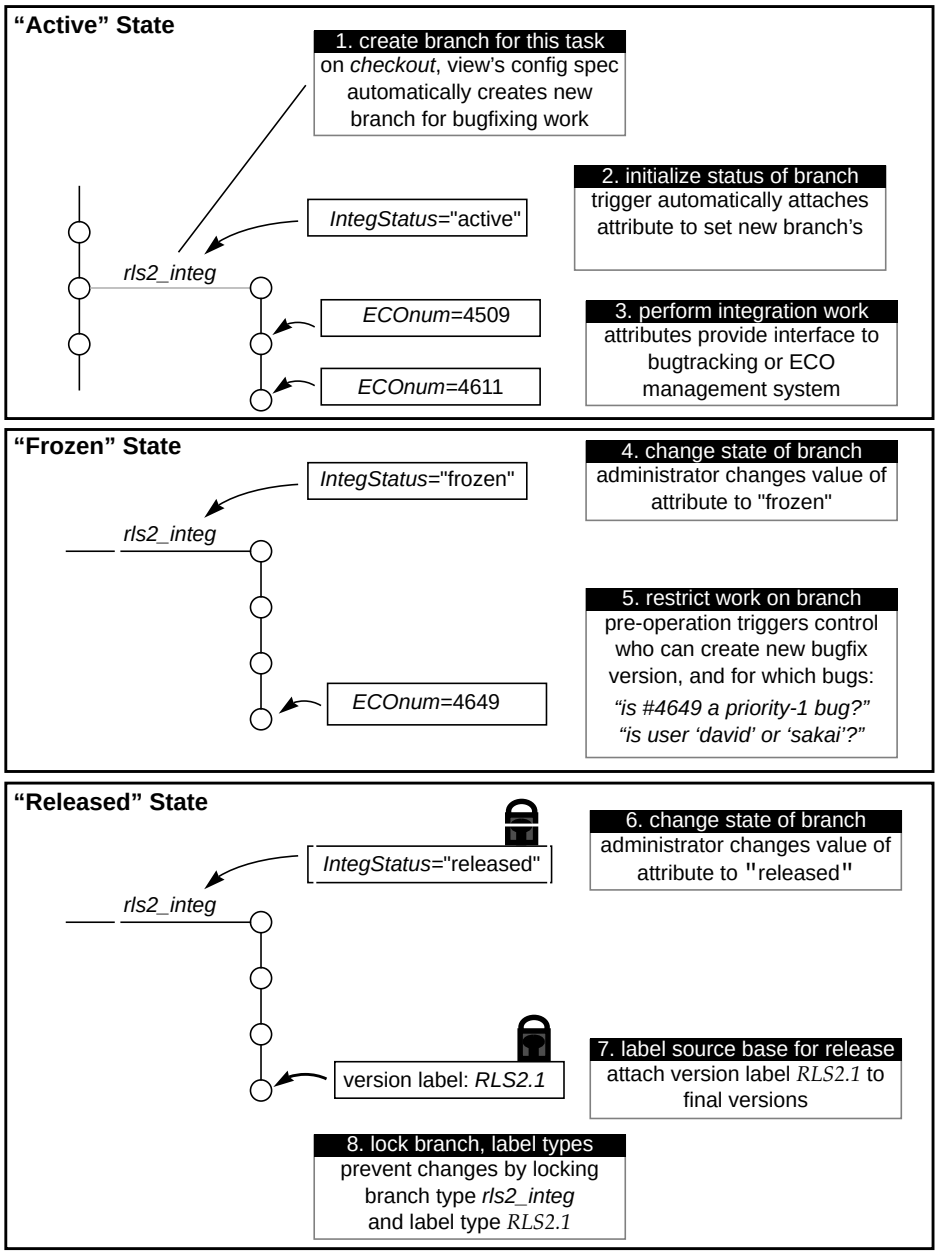


Figure 6-5 State Transition Model

Glossary

abe

See audited build executor.

active

A VOB becomes active on a host when it is mounted as a file system of type MVFS. A view becomes active on a host when it is started, which establishes a connection between the host's MVFS file system and the view's view_server process.

annotation box

Part of the *xcleardiff* display, showing how lines of one file differ from lines of other files, with which it is being compared or merged.

argument

A word or quoted set of words processed by a command.

attached list

See *trigger inheritance*.

attribute

A *meta-data* annotation attached to an *object*, in the form of a name/value pair. Names of attributes are specified by user-defined *attribute types*; values of these attributes can be set by users.

Example: a project administrator creates an attribute type whose name is *QAed*. A user then attaches the attribute *QAed* with the value "*Yes*" to versions of several file elements.

attribute name

See *attribute*, *attribute type*.

attribute type

An *object* that defines an *attribute name* for use within a *VOB*. It constrains the attribute values that can be paired with the attribute name (for example, integer in the range 1–10).

attribute value

See *attribute type*.

audited build executor

A process invoked through the UNIX remote-shell facility, in order to execute one or more build scripts on behalf of a remote *clearmake*.

audit, audited shell

See *build audit*.

auto-make-branch

ClearCase's facility, specified in a config spec rule, for automatically creating one or more branches when a *checkout* is performed.

base contributor

See *contributor*.

bidirectional

See *hyperlink*.

bitmap files

Files that store bitmaps for the icons displayed by ClearCase GUI programs.

BOS file

See *build options specification*.

branch

An *object* that specifies a linear sequence of *versions* of an *element*. The entire set of versions of an element is called a *version tree*; it always has a single *main* branch, and may also have *subbranches*. Each branch is an instance of a *branch type* object.

branch name

See *branch type*.

branch pathname

A sequence of branch names, starting with */main* (the name of an element's starting branch). Example: */main/motif*, */main/maintenance/bug459*.

branch type

An *object* that defines a *branch name* for use within a *VOB*.

Broadcast Message Server

An H-P SoftBench program that passes messages among SoftBench applications.

build

The process of invoking *clearmake* or *clearaudit* to produce one or more *derived objects*. This may or may not involve actual translation of source files and construction of binary files by compilers, linkers, text formatters, and so on. A system build consists of a combination of actual *target rebuilds* and *build avoidance* (reuse of existing derived objects)

In a *target rebuild*, *clearmake* executes the *build script* associated with a particular *target* in a *makefile*. Each target rebuild produces derived objects along with a *configuration record*, which includes an *audit* of the files involved in the actual target rebuild.

In *build avoidance*, *clearmake* "produces" a derived object either by *reusing* the one currently in the view, or by *winking-in* one that exists in another view. The process by which *clearmake* decides how to produce a derived object is called *configuration lookup*.

build audit

The process of recording which files and directories (and which versions of them) are accessed (read or written) by the operating system during the execution of one or more programs. A client host's MVFS file system performs an audit during execution of a ClearCase build program: *clearmake*, *clearaudit*, or *abe*. When the build audit ends, the build program creates one or more *configuration records* (*CRs*).

An *audited shell* is a UNIX shell process in which all file system accesses are audited. Such a shell is created by *clearaudit*.

build avoidance

See *build*.

build configuration

See *configuration lookup*.

build dependency

See *dependency.*

build host

A host used to execute build scripts during a distributed build.

build hosts file

A file that lists hosts to be used in a distributed build. The file is selected at build time by macro or environment variable *clearcase_bld_host_type*.

build avoidance

The ability of *clearmake* to fulfill a build request by using an existing derived object, instead of creating a new derived object by executing a build script.

build options specification

A file containing rules that specify settings of *make macros*, which affect the way in which a *target rebuild* proceeds.

build reference time

The moment at which a top-level *build session* (invocation of *clearmake*) begins. Versions created after this moment may be "frozen out" of the build.

build script

The set of shell commands that *clearmake* (or standard UNIX *make*) reads from a *makefile* when building a particular *target*.

build server

A host used in a *clearmake* distributed build.

build server control file

A file on a build host that controls its availability as a build server.

build session

A top-level invocation of *clearmake*; during the session, recursive invocations of *clearmake* or *clearaudit* may start subsessions.

build target

A word, typically the name of an object module or program, that can be used as an argument in a *clearmake* command. The target must appear in a *makefile*, where it is associated with one or more *build scripts*.

built-in rules

Build rules defined in a system-supplied (or ClearCase-supplied) file, which supplement the explicit build rules in a user's *makefile* (*s*).

bump

The taking away of a ClearCase license from a lower-priority user by a higher-priority user.

candidate

A *derived object* that is being considered for *wink-in* or reuse during *configuration lookup*.

cataloged

Names of elements and VOB symbolic links that appear in a version of a directory element are said to be *cataloged* in the directory version. A *derived object* is said to be cataloged (and, hence, available for *reuse* and *wink-in*) in a particular VOB.

checked-out version

A "placeholder" object in a VOB database, created by the *checkout* command. This object corresponds to the view-private file that the user edits after checking out the element. The checked-out version of a directory element exists *only* in the VOB database — there is no view-private component.

checkout/checkin

The two-part process that extends a *branch* of an *element's version tree* with a new *version*. The first part of the process, *checkout*, expresses the user's intent to create a new version at the current end of a particular branch. (This is sometimes called "checking out a branch".) The second part, *checkin*, completes the process by creating the new version.

For *file elements*, the checkout process creates an editable version of the file in the *view*, with the same contents as the version at the end of the *branch*. Typically, a user edits this file, then checks it back in.

For *directory elements*, the checkout process allows file elements, (sub)directory elements, and VOB symbolic links to be created, renamed, moved, and deleted.

Performing a checkout of a branch does not necessarily guarantee the user the right to perform a subsequent checkin. Many users can checkout the same branch, as long as the users are in different views. At most one of these can be a *reserved checkout*, which guarantees the user's right to checkin a new version. An *unreserved checkout* affords no such guarantee. If several users have unreserved checkouts on the same branch in different views, the first user to perform a checkin "wins" — another user must perform a *merge* if he wishes to save his checked-out version.

checkout record

The event record created by the *checkout* command.

cleartext file

An ASCII text file that contains a whole copy of some version of an element, having been extracted from a *data container* that is in compressed format or *delta* format. A ClearCase *type manager* creates a cleartext container the first time it accesses the version. Subsequent reads of that version access the cleartext file, for faster access.

cleartext pool

A VOB storage pool, used for *data containers* that contain *cleartext*.

CLI

ClearCase's command-line interface.

client

The programs invoked by users: *cleartool*, *xclearcase*, *clearmake*, *cleardiff*, and other programs located in the ClearCase *bin* directory.

clock skew

The discrepancies among the system clocks of several hosts.

command option

In the command-line interface (CLI), a word beginning with a hyphen (–) that affects the meaning of a command; in the graphical user interface (GUI), a setting in the "Options" part of a panel.

comment default

The action taken by a *cleartool* command when the user does not specify a comment-related option.

compatibility mode

A *clearmake* execution mode, in which it emulates another *make* variant.

compressed_file

The ClearCase element type that uses data compression on individual versions.

compressed_text_file

The ClearCase element type that uses both *delta* management and data compression on individual versions.

configuration (of a derived object)

The information recorded in a derived object's CR: versions of source files used to build the object, build script, build options, and so on.

config spec

A set of rules, specifying which versions of elements are to be selected by a view. Each config spec rule selects a particular version of one or more *elements*.

The default ClearCase config spec is:

element * CHECKEDOUT

element * /main/LATEST

See *scope*, *pattern*, *version-selector*.

configuration (of a view)

The set of *versions* (one version of each *element*) selected by a view's *config spec*.

configuration lookup

The process by which *clearmake* determines whether to produce a *derived object* by performing a *target rebuild* (executing a *build script*) or by reusing an existing instance of the derived object. This involves comparing the *configuration records* of existing derived objects with the *build configuration* of the current view: its set of source versions, the current build script that would be executed, and the current build options.

configuration management

The discipline of tracking the individual objects and collections of objects (and the versions thereof) that are used to build systems.

configuration record (CR)

A listing produced by a *target rebuild*, logically included in each *derived object* created during the rebuild. A configuration record indicates exactly which file system objects (and which specific versions of those objects) were used by the rebuild as input data or as executable programs, and which files were created as output. It also contains other aspects of the *build configuration*.

Each *target rebuild* typically involves the execution of a single build script, and creates a single configuration record. If a target has *subtargets* that must be rebuilt, also, a separate configuration record is created for each subtarget rebuild.

configuration record hierarchy

A tree structure of configuration records, which mirrors the hierarchical structure of targets in the *makefile*.

configuration rule

See *config spec*.

configuration specification.

See *config spec*.

container

See *data container.*

context

See *view context*.

contributor

One of the files or versions that supplies input to a *merge*. One of them is the *base contributor* — the merge algorithm compares each other contributor with the base contributor.

conversion script

A shell script by one of the ClearCase file-conversion programs — for example, *clearcvt_rcs*.

CR

See *configuration record*.

cross-VOB hyperlink

A *hyperlink* that connects two objects in different VOBs. The hyperlink always appears in a *describe* listing of the "from" object. It also appears in a listing of the "to" object, unless it was created as a *unidirectional* hyperlink (*mkhlink -unidir)*. See *same-VOB* hyperlink.

current working directory

The context in which *relative pathnames* are resolved by the operating system. This can be a location in ClearCase's *extended namespace*.

data container

A file (or directory) that contains the data produced by a build script. A data container and a configuration record are the essential constituents of a derived object.

default config spec

See *config spec*.

degenerate derived object

A derived object that cannot be successfully processed, because it data container and/or associated configuration record are not available.

delta

The incremental difference (or set of differences) between two versions of a file element. Certain type managers (for example, *text_file_delta*), store all versions of an element in a single data container, as a series of deltas.

dependency

(same as for standard UNIX *make*) In a makefile, a word listed after the colon (:) on the same line as a target. A *source dependency* of a target is a file whose version-ID is taken into account in a configuration lookup of the target. A *build dependency* is a derived object that must be built before the target is built.

derived object (DO)

A file produced by a *clearmake* build or a *clearaudit* session. Each derived object is associated with a *configuration record* produced by *clearmake* during the build.

derived object storage pool

A *storage pool* for the *data containers* of a VOB's *derived objects*. Only those derived objects that are shared by two or more *views* are stored in these pools — data containers of unshared derived objects are stored in view-private storage.

The first time a derived object is *winked-in* to a view, the *promote_server* program copies the data from the original view, creating a data container in a derived object pool. Different pools for the same VOB can be located on different disks, or on different machines in the local area network.

derived object scrubbing

The removal of data containers from derived object pools, and of derived objects themselves from a VOB database. Periodically, ClearCase automatically scrubs derived objects that are not referenced in any view.

derived object sharing

The ClearCase feature wherein several views can simultaneously use the same *derived object*, through a mechanism resembling a UNIX hard link. See *wink-in*.

derived object version

See *DO version*.

difference pane

A subwindow in an *xcleardiff* window that shows the contents of one of the files being compared or merged.

difference section

A set of lines in a difference pane, or in *cleardiff* output, that differs among the files being compared or merged.

directory element

An *element* whose versions are like UNIX directories — they catalog the names of *file elements*, other *directory elements*, and *VOB symbolic links*.

directory version

A version of a a *directory element*.

distributed build

See *parallel build*.

DO

See derived object.

DO version

A derived object that has been checked in as a version of an element.

DO-ID

A unique identifier for a derived object, including a time stamp and a numeric suffix to guarantee uniqueness. Example: the characters beginning with "@@" in *hello.o@@12-May.19:15.232*.

DSEE

The Domain Software Engineering Environment.

eclipsed

Invisible, because another object with the same name is currently selected by the current view.

element

An *object* that encompasses a set of *versions*, organized into a *version tree*.

element type

A class of versioned objects. ClearCase supports predefined element types (for example, *file*, *text_file*). Users can define additional types (for example, *c_source_file*) that are refinements of the predefined types. When an *element* is created, it is assigned one of the currently-defined element types.

Each user-defined element type is implemented as a separate VOB object, created by a user command.

ellipsis

The *wildcard* symbol "...". In a *version-selector*, it indicates zero or more directory levels.

encapsulator

A program that packages the functionality of an external software system.

event

A ClearCase operation that is recorded by an *event record* in a VOB's event *history*.

event record

An item in a VOB database that contains information about an operation that modified that VOB.

export view

A view used to export a VOB to a *non*-ClearCase *acess* host.

extended namespace

ClearCase's extension of the standard UNIX pathname hierarchy. Each host has a *view-extended namespace*, allowing a pathname to access VOB data using any view that is active on that host. Each VOB has a *VOB-extended namespace*, allowing a pathname to access any version of any element, independently of (and overriding) version-selection by views. *Derived objects* also have extended pathnames, which include *DO-IDs*. See *namespace*.

extended naming symbol

A symbol (by default, *@@*) appended to an element name or derived object name, signaling the MVFS file system to bypass automatic version-selection by a view.

extended pathname

A *VOB-extended pathname* specifies a particular location in an element's *version tree*, or a particular derived object cataloged in that VOB. If the pathname specifies a particular version, is termed a *version-extended pathname*. Examples:

foo.c@@/main/17

/usr/myproduct/bin/xtract@@/RELEASE_1

/usr/myproduct@@/main/bug403/5

A *view-extended pathname* accesses a file system object in the context of a specified view. For an element, such a pathname specifies the version of the element selected by that view's config spec; for a view-private file or derived object, such a pathname accesses an object in the view's private storage area. Examples:

/view/akp/usr/project/foo.c

/view/archive/usr/project/foo

/view/bugfix/usr/project/to_do.list

file type

The identifier returned by ClearCase file typing subsystem, through a lookup in ClearCase-supplied and/or user-supplied *magic files*. File types are used to select an *element type* for a new element; they are also used by ClearCase GUI programs to select display icons.

file contents

See *file system data*.

file element

See *element*.

filename pattern

See *pattern*.

file system configuration

The set of element versions selected by a view.

file system data

The bytes stored in a *version* of a *file element*. A file's contents are distinguished from its *meta-data* (*attributes*, *hyperlinks*, and so on).

fire a trigger

The process by which ClearCase verifies that the conditions defined in a *trigger* are satisfied, and causes the associated *trigger action*(*s*) to be performed.

flat

A non-hierarchical listing, combining information from a collection of configuration records.

from-object

See *hyperlink*.

from text

A string-valued attribute attached to a hyperlink object, conceptually at its "from" end.

full pathname

A standard operating system pathname beginning with "/". For example, */usr/project/foo.c* is a full pathname, but */view/akp/usr/project/foo.c* is a *view-extended pathname*.

g-file

A file produced by the SCCS *get* command.

global element trigger type

A trigger type that is automatically associated with all elements in a VOB.

global pathname

A network-wide pathname for a ClearCase view storage directory or VOB storage directory. Some "global" pathnames are valid only within a particular *network region*.

Gnu make

A *make* variant distributed by the Free Software Foundation.

hard link

An additional name for a file system object, cataloged in the same directory or in a different directory. UNIX hard links are cataloged in standard UNIX directories; *VOB hard links* are cataloged in versions of *directory elements*.

header file

A source file whose contents are included in a program with an #*include* statement.

history

Meta-data in a *VOB*, consisting of *event records* involving that VOB's *objects*. The history of a file element includes the creation event of the element itself, the creation event of each

version of the file, the creation event of each branch, the assignment of attributes to the element and/or its versions, the attaching of hyperlinks to the element and/or its versions, and so on.

history mode

The state of a process whose current working directory is in the ClearCase *VOB-extended namespace* (for example, *hello.c@@/main*).

hyperlink

A logical pointer between two objects. A hyperlink is implemented as a VOB object; it derives its name by referencing another VOB object, a *hyperlink type*.

A hyperlink can have a *from-string* and/or *to-string*, which are implemented as string-valued attributes on the hyperlink object.

hyperlink browser

An *xclearcase* panel that enables a user to traverse hyperlinks.

hyperlink-ID

A system-generated identifier that, in conjunction with the name of the *hyperlink type*, uniquely identifies a *hyperlink* object. Example: *@391@/usr/hw.*

hyperlink type

An *object* that defines a hyperlink name for use within a *VOB*.

hyperlink selector

A string that specifies a particular hyperlink. It consists of the name of a hyperlink type object, followed by a (possibly abbreviated) hyperlink-ID. Examples:

DesignFor@391@/usr/hw

icon

A small picture used by ClearCase GUI programs.

icon file

A file containing rules that map ClearCase *file types* to names of *bitmap files*.

inclusion list

A list of type objects, defining the scope of a *trigger type*.

inherit, inheritance list

See *trigger inheritance*.

installation host

A host of which ClearCase has been (or is about to be) installed.

interactive mode

The mode of *cleartool* usage in which the program prompts you (with *cleartool*>) to enter a command, executes the command, then prompts you to enter another one. See *single-command mode*.

label type

A *type object* that defines a version label for use within a VOB.

leaf name

The simple file name at the end of a multiple-component pathname.

license

Permission for one user to run ClearCase programs and/or use ClearCase data, using any number of hosts in the local area network.

license database file

A file that defines a set of ClearCase licenses.

license priority

A "slot" in the scheme by which some ClearCase users can *bump* others, taking their licenses.

license server

A host whose *albd_server* process controls access to the licenses defined in its *license database file*.

link text

The text string that is the contents of a UNIX-level symbolic link or a VOB symbolic link.

load balancing

The ClearCase facility for intelligently managing a distributed build, so as not to overload individual hosts.

lock

A mechanism that prevents a VOB object from being modified (for file system objects) or unreferenceable (for *type* objects). See *obsolete*.

logical operator

A symbol that specifies a Boolean arithmetic operation.

lost+found

A subdirectory of a VOB's top-level directory, to which elements are moved if they are no longer cataloged in any version of any directory element. See *orphaned element*. There is also a *lost+found* directory at the top level of a view storage directory.

magic file

A file used by the ClearCase *file typing* subsystem to determine the type of an existing file, or for the name of a new file.

main branch

The starting branch of an element's *version tree*. The default name for this branch is *main*.

make macro

A parameter in a *makefile*, which can be assigned a string value within the makefile itself, in a *build options spec*, on the *clearmake* command line, or by assuming the value of an *environment variable*.

makefile

A text file, read by *clearmake*, that associates *build scripts*, consisting of shell commands ("executable commands"), with *targets*. Typically, executing a build script produces one or more *derived objects*.

Makefiles constructed for *clearmake* need not include source-level dependencies (for example, header file dependencies), but they must include build-order dependencies (for example, that executable program *hello* is built using object module *hello.o*.

makefile dependency

See *dependency*.

master conversion script

The script, created by one of the ClearCase conversion utilities, that is explicitly executed by the user to perform the data conversion.

merge

The combining of the contents of two or more files into a single new file. Typically, all the files involved are versions of a single file element. This process may be completely automated or may require the user to resolve conflicting changes. The merge can be recorded with a *merge arrow*, which is implemented as a hyperlink of type *Merge*.

meta-data

Data associated with an *object*, supplementing the object's *file system data*. Example: The contents of a file version is a series of text characters. User-specified *meta-data annotations* attached to the file version includes *version labels*, *attributes*, and *hyperlinks*. ClearCase automatically maintains other meta-data for some objects, in the form of *event records* and *configuration records*.

method

A program that implements one of the functions of a *type manager*.

minor event

An event whose event record is, by default, not listed by the *lshistory* command.

multiversion file system (MVFS).

A directory tree which, when activated (mounted as a file system of type MVFS) implements a ClearCase VOB. To standard UNIX commands, a VOB appears to contain a directory hierarchy; ClearCase commands can also access the VOB's meta-data.

MVFS file system also refers to the *multiversion file system*, a virtual file system extension that is linked with the UNIX kernel, providing access to VOB data.

MVFS object

A file or directory whose pathname is within a VOB (that is, whose pathname is within the directory tree beneath the top-level *VOB-tag*. A *non-MVFS object* has a pathname that is not within a VOB.

namespace

A file/directory name hierarchy. Different views can "see" different namespaces, because they can select different versions of directory elements. See *extended namespace*.

network region

A logical subset of a local area network, within which all hosts refer to VOB storage directories and view storage directories with the same network pathnames.

nobody

The username sometimes assigned to a remote process owned by the *root* user on the local host.

non-ClearCase access

Access to ClearCase data from a host on which ClearCase has not been installed.

non-MVFS object

See *MVFS object*.

notice forwarder

One of the message-passing programs in an H-P SoftBench environment.

null-ended

A *hyperlink* that is connected to only one object, not two.

object

An item stored in a versioned object base (*VOB*). This includes *elements*, *versions* of elements, *branches*, *derived objects*, *hyperlinks*, *locks*, *pools*, and *types*.

object-ID (OID)

A ClearCase-internal identifier for an object. See UUID.

obsolete object

An object that has been *locked* with the *lock –obsolete*. By default, such objects are not listed by commands such as *lstype* and *lslock.*

orphaned element

An element that is no longer cataloged in any version of any directory. Such elements are moved to the *lost+found* directory.

owner

The user who owns a VOB, a view, or an individual file system object. The user who creates an item becomes its initial owner.

parallel build

A *build* process in which multiple build scripts are executed concurrently. In a *distributed build*, execution takes place on multiple hosts in a local area network.

parallel development

The concurrent creation of versions on two or more branches of an element.

pathname

A sequence of directory names, perhaps ending with a simple filename. A pathname that begins with "/", indicating the root directory, is termed a *full pathname*. Any other pathname is termed a *relative pathname*. See *extended pathname*, *namespace*.

pattern

A character string that specifies one or more file and/or directory names. Examples:

/usr/project/.../include
*.c
lib/*.[ch]

See *scope*, *version selector*, *config spec*.

permission

Ability to perform an operation that modifies a VOB or a file system object.

pool inheritance

The feature by which a newly-created element is assigned to the same VOB storage pools as its parent directory element.

post-operation trigger

A trigger that fires after the associated operation.

predecessor version

A *version* of an *element* that immediately precedes another version in the element's *version tree*. If version *X* is the predecessor of version *Y*, then *Y* is the *successor* of *X*. If there is a chain of predecessors linking two versions, one is called an *ancestor* of the other.

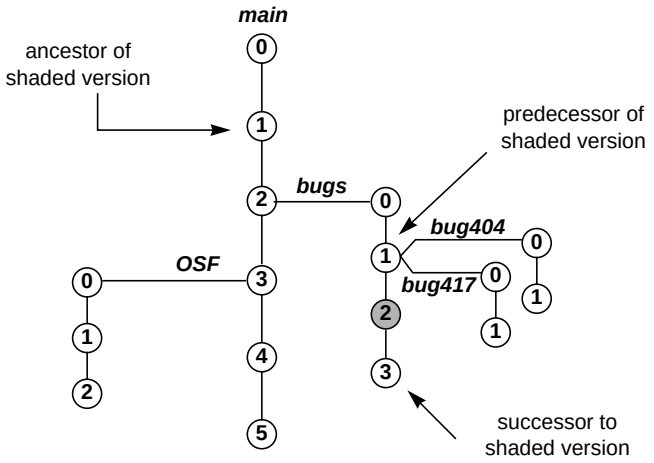


Figure 6-6 Version Predecessors, Successors, and Ancestors

pre-operation trigger

A trigger that fires before the associated operation, possibly cancelling the operation itself.

principal group

The group to which a user belongs, by virtue of being listed in the password database. A user can belong to additional groups, as well.

private storage area

The directory tree (*.s*) in which view-private files, directories, and links are stored. By default, this is a subtree of the view storage directory, but ClearCase supports creation of remote private storage areas.

private VOB-tag

See *public VOB-tag*.

promotion

Migration of a derived object from view storage (private) to VOB storage (shared).

public VOB-tag

If a VOB's VOB-tag is *public*, it is mounted automatically at system startup, and it can be mounted or unmounted by any user. If a VOB's VOB-tag is *private*, only the VOB's owner can mount the VOB. See *VOB-tag password file*.

query language

A collection of predicates and logical operators that allows selection of elements, branches, and versions based on their associated meta-data. This language can be used in *config specs*, in the ClearCase *find* command, and in the *–version* option to many commands.

RCS

The Revision Control System.

record

See *event record*, *configuration record*.

reference count

The number of references to a derived object, in multiple views and, perhaps, several times in the same view (through UNIX hard links).

reference time

See *build reference time*.

refinement

See *supertype*.

relative pathname

A pathname that does not begin with "/".

release host

The host on which the ClearCase software is unloaded from the distribution medium.

reserve state

For a checked-out version, either "reserved" or "unreserved". The *reserve* and *unreserve* commands change the reserve state of a checked-out file.

reserved checkout

See *checkout*.

restriction list

A specification of which kinds of objects are to be associated with a trigger type.

reuse

clearmake is said to reuse a *derived object* if the object already exists in a view and a build leaves it alone.

root

The privileged "superuser" on a host.

same-VOB hyperlink

A *hyperlink* that connects two objects in the same VOB.

schema

The format of a database.

scheme file

A file that contains a collection of X Window System resources, which control various aspects of GUI applications.

scope

The part of a *config spec* rule that restricts it to a particular kind of file system objects. See *config spec*, *pattern*, *version selector*.

scrubbing

Discarding of data container files from cleartext pools and derived object pools, performed by *scrubber*(1MA).

selection operator

See *selection expression*.

selection expression

An expression used by ClearCase's file typing mechanism to match a file system object (or just the name of one). The expression consists of *selection operators* and *logical operators*.

set view

(noun) The *view context* of a process, established by using the *setview* command. "Setting a view" creates a process in which all standard pathnames are resolved in the context of a particular view. See *working directory view.*

s-file

A file in which SCCS stores all versions of a versioned file.

shared derived object

A derived object that is referenced by multiple views, and whose data container has migrated to a VOB's *derived object storage pool*.

sibling

A derived object created by the same build script as another derived object. When *clearmake* reuses (or winks in) a derived object, it always reuses (or winks in) all of its siblings, too.

single-command mode

The mode of cleartool usage in which you specify the (sub)command, options, and arguments on the shell command line, along with "cleartool". After executing that one (sub)command, cleartool exits and control returns to the shell.

SoftBench

An interprocess messaging system.

source control

See *version control*.

source dependency

See *dependency.*

source pool

A *storage pool* for the *data containers* that store versions of file elements.

storage pool

A *source pool*, *derived object pool*, or *cleartext pool*.

storage registry

A network-wide database, which records the actual storage locations of all VOB storage directories and all view storage directories.

stranded derived object

A derived object that cannot be accessed, because the VOB directory (or the entire VOB) in which it was created is not currently accessible, or has been deleted.

subsession

A build session that was started while a higher-level *build session* was active.

subtarget

In a hierarchical build, a *makefile* target upon which a higher level target depends. Subtargets must be built (or reused, or *winked-in*) before higher-level targets.

subbranch

A branch of an element's version tree other than the main branch.

successor version

See *predecessor version*.

supertype

An *element type* that is used as the basis for defining another element type (said to be a *refinement* of the supertype).

tags registry

A network-wide database, which records the globally-valid access paths to all VOB storage directories (or all view storage directories), along with the *VOB-tags* (or *view-tags*) with which users access these data structures.

target, target rebuild

See *build*.

text_file

The ClearCase element type that uses *delta* management to combine all versions of an element into a single data container. The associated *type manager* is named *text_file_delta*.

text-only

A hyperlink for which there is no "to" object, only a text annotation on the "from" object.

time rule

A separate config spec rule that specifies a time to which the special version label LATEST should evaluate in all subsequent rules; or, a clause that sets the LATEST time within an individual rule,

to-object

See *hyperlink*.

from text

A string-valued attribute attached to a hyperlink object, conceptually at its "from" end.

ToolTalk

An interprocess messaging system.

transcript pad

A scrollable subwindow in which a ClearCase GUI program displays output.

transparency, transparent access

The ClearCase feature that enables standard programs to access versioned files and directories using standard pathnames.

trigger

A "monitor" that specifies one or more standard programs or built-in actions to be executed automatically whenever a certain ClearCase operation is

performed.

See *pre-operation trigger*, *post-operation trigger*, *trigger type*.

trigger inheritance

The process by which *triggers* in the *inheritance list* of a *directory element* are automatically attached to new elements created within the directory.

trigger type

An object through which triggers are defined. Instances of a "element" *trigger type* can be attach to one or more individual elements ("attached trigger"). A "global element" trigger type is implicitly attached to all elements in a VOB. A "type" trigger type is attached to a specified collection of *type* objects.

type

A *type object* defines a ClearCase data structure. Users can create instances of these structures: *meta-data annotations* are placed on objects by creating instances of label types, attribute types, and hyperlink types. The structure of the *version tree* of a file or directory is defined by creating an instance of an element type, along with instances of branch types. Triggers are defined as instances of trigger types.

type manager

A set of routines (*methods*) that stores and retrieves versions of file elements from disk storage. Some type managers include methods for other operations, such as comparison, merging, and annotation,

type trigger type

A *trigger type* that is associated with (and thus, monitors changes to) one or more type objects.

uncheckout

The act of cancelling a checkout operation.

unidirectional

See *hyperlink*.

universal unique identifier

See *UUID*.

unregister

See *storage registry.*

unreserved checkout

See *checkout*.

unshared derived object

A derived object that has never been *winked-in* to another view.

user profile

A file that stores a user's individual specifications for *cleartool* comment handling.

UUID

A universal unique identifier, which ClearCase uses to track VOBs, vies, and the objects they contain.

version

An *object* that implements a particular revision of an *element*. The versions of an element are organized into a *version tree* structure. Also: the *view-private file* that corresponds to the *checked-out version* object created in a VOB database by the *checkout* command.

version 0

The original *version* on a *branch*. It is automatically created when the branch is created, and has the same contents as the version at the branch point. Version 0 on the main branch is defined to be empty.

.

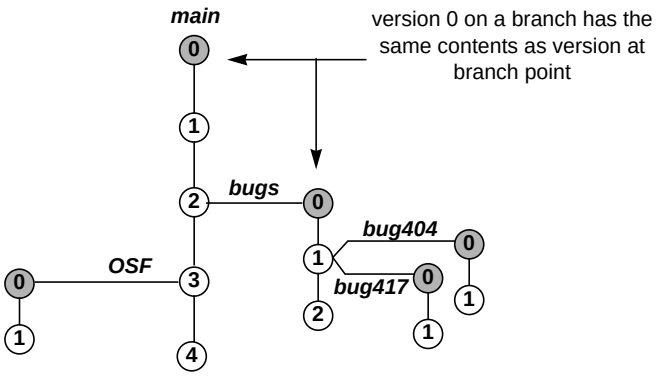


Figure 6-7 Version 0 of a Branch

version control

The discipline of tracking the version evolution of a file or directory.

version-extended namespace

See *extended namespace*.

version-extended pathname

A pathname that explicitly specifies a version of an element (or versions of several elements), rather than allowing the version-selection to be performed automatically by a view.

version-ID

A *branch pathname* and version number, indicating a version's exact location in its version tree:

```
/main/4
/main/rel2_bugfix/2
/main/bugs/bug405/9
```

version label

An instance of a label type object, supplying a user-defined name for a version.

version-number

The ClearCase-assigned integer that identifies the position of a version on its branch.

version selection

The process of choosing one version from an element's entire version tree. A ClearCase *view* performs automatic version selection, using its *config spec*. Users can select versions with *version-extended pathnames* and with the ClearCase *query language*.

version selector

A command-line option, a part of a config spec rule, or a portion of a *version-extended* pathname, specifying the version of one or more elements to be used. See *scope*, *pattern*, and *configuration specification*.

version tree

The hierarchical structure in which all the *versions* of an *element* are (logically) organized. When displaying a version tree, ClearCase also shows *merge* operations (indicated by gray arrows in the illustration)

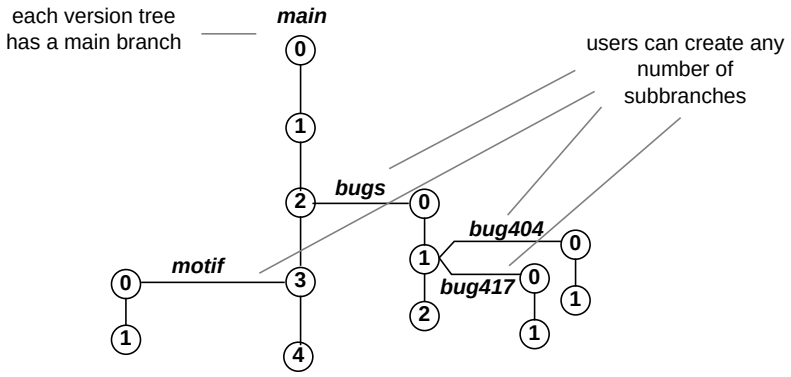


Figure 6-8 Version Tree Structure

version-extended namespace

See *VOB-extended namespace*.

versioned object base (VOB)

A repository that stores *versions* of *file elements*, *directory elements*, *derived objects*, and *meta-data* associated with these objects. A *view* makes a VOB appear to be a standard directory tree.

view

A ClearCase data structure which provides a virtual workspace for one or more users — to edit source versions, compile them into object modules, format them into documents, and so on. Users in different views can work with the same files without interfering with each other. For each *element* in a *VOB*, a view uses its *config spec* to select just one version from its *version tree*. Each view can also store *view-private* objects, which are do not appear in other views.

view context

The view (if any) which will be used to resolve a pathname to a particular *version* of an *element*.

view database

The database within a view storage directory, which the associated *view_server* process uses to track objects in the view's private storage area.

view-extended namespace

See *extended namespace*.

view-extended pathname

A pathname that begins with a *view prefix* (for example, */view/alpha*), specifying a particular view to be used for resolving element names to particular versions.

view object

An object stored in a *view*: a *checked-out* version of a file, an *unshared derived object*, or a *view-private* file, directory, or link. No historical information is retained for view-objects. See *VOB object*.

view prefix

One or more components at the beginning of a pathname that specify a particular view. For example, */view/gamma* and *../epsilon*.

view-private directory

A directory that exists only in a particular *view*, having been created with the standard UNIX *mkdir* command. A view-private directory is not *version-controlled*, except insofar as it is separate from private directories in other views.

view-private file

A file that exists only in a particular *view*. A private file is not *version-controlled*, except insofar as it is separate from private files in other views.

view-tag

The name with which users reference a view. This name appears as a subdirectory of a client host's *viewroot directory* (*/view*).

view storage directory

The directory tree in which a view's data is stored: checked-out version, view-private objects, and unshared derived objects.

view storage registry

A file on the network's *registry server host* that records that actual storage locations of all the views in the network.

viewroot directory, view-tag

A main-memory-only data structure, maintained by the MVFS file system, that behaves like a directory. Each view is accessed through a *view-tag* entry in this directory. By default, the viewroot directory is named */view*. A view whose tag is *alpha* is accessible as */view/alpha*.

view storage area, view storage directory

The directory tree in which a view's private data is stored: config spec, checked-out versions of file elements, view-private files and directories. The top-level directory of this tree is called the *view storage directory*.

viewroot directory

The directory (default name: */view*) in which view-tag entries appear, allowing views to be accessed through the UNIX file system.

view_server

The daemon process that continually interprets a view's *config spec*, mapping element names into versions.

virtual file system

An extension to the UNIX kernel, allowing alternative file systems to be implemented without revision to the kernel itself.

VOB

See *versioned object base*.

VOB database

The part of a *VOB storage area* in which ClearCase *meta-data* and VOB objects (elements, branches, versions, and so on) are stored. This area is managed automatically by ClearCase's embedded database management software. The actual file system data, by contrast, is stored in the VOB's storage pools. For data integrity, ClearCase maintains a copy of this "live" VOB database, the *shadow database*.

VOB-extended namespace

An extension to the operating system's file naming scheme, which allows any historical version of an element to be accessed directly by any program. The extension also provides access to the meta-data (but not the file system data) of all of a VOB's existing derived objects.

VOB hard link

A name, cataloged in a (version of a) directory element, for an element. Typically, the first such link is called the element's "name"; the term *VOB hard link* is used to refer to any additional names for the element.

VOB host

A host on which one or more *VOB storage directories* reside.

VOB link

A *VOB symbolic link* or *VOB hard link*.

VOB mount point

The directory on which a *VOB storage area* is mounted. All UNIX commands, and most ClearCase commands, access a VOB through its mount point.

VOB object

An object stored in a *VOB*: *element*, *version* of element, *type*, *hyperlink*, *derived object*, and so on. See *view object*.

VOB owner

Initially, the user who created a VOB with the *mkvob* command. The ownership of a VOB can be changed subsequently, with the *chown_vob* command.

VOB host

A machine on whose disk a VOB is physically stored.

VOB storage directory

The directory tree in which a VOB's data is stored: elements, versions, derived objects, CRs, event history, hyperlinks, attributes, and other meta-data. The top-level directory of this tree is called the VOB *storage directory*.

VOB storage registry

A file on the network's *registry server host* that records that actual storage locations of all the VOBs in the network.

VOB symbolic link

An *object*, cataloged in a (version of a) directory element, whose contents is a pathname. ClearCase does not maintain a version history for a VOB symbolic link.

VOB-tag

The full pathname at which users access a VOB. The VOB storage directory is activated by mounting it as a file system of type MVFS at the location specified by its VOB-tag.

VOB-tag password file

A file used to validate the password entered by a user when creating a public VOB-tag.

VPATH

A *make macro* that specifies directories that will be searched during a build for data.

wildcard

See *pattern*.

wink-in

Causing a derived object to appear in a view, even though its *file system data* is actually located in a VOB's *derived object storage pool*.

working directory view

The *view context* of a process, established by using the *cd* command to change the current working directory to a *view-extended pathname*. See *set view*.

Index

B

C

D

E

F

G

H

M

N

O

P

Q

R

S

T

U

V

W

X

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 007-1612-020.

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