A TimeSafe® Configuration Management System

AccuRev/CM Manual

Release 3.1

June, 2002

AccuRev/CM Manual
June, 2002

AccuRev/CM Concepts

AccuRev/CM Concepts

The chapters in this section describe the main concepts and facilities of the AccuRev®/CM software configuration management system:

- *The AccuRev/CM Data Repository*
- *What is a Software Configuration?*
- *AccuRev/CM Software Configurations: The Stream Hierarchy*
- *AccuRev/CM Workspaces and Reference Trees*
- *AccuRev/CM Transactions*

The AccuRev/CM Data Repository

As a data management product, AccuRev/CM's foremost job is to provide a secure data repository for long-term storage of your organization's development data. AccuRev/CM's implementation of the repository is straightforward and flexible; a repository can grow gracefully to span multiple disks, possibly on multiple machines. And key product features make it easy to protect the repository from accidental or malicious damage.

AccuRev/CM has a simple client-server architecture. A single program, the AccuRev/CM Server (**accurev_server**), is the only program that accesses the data repository directly. This "single point of entry" to the repository makes it easy to enforce tight security at the operating system level.

The data repository is built around a unique database technology, which is both transaction-based and append-only. This makes the repository extremely resistant to accidental damage. Using "atomic" transactions means that the database won't become corrupted, even if a power failure occurs while the database is being modified. The append-only feature enables "live backup" of the repository, without having to interrupt developers' work. This means that backups can be made as often as desired — even continually; and the more recent the backup, the less data is lost in the event of a catastrophic hardware failure.

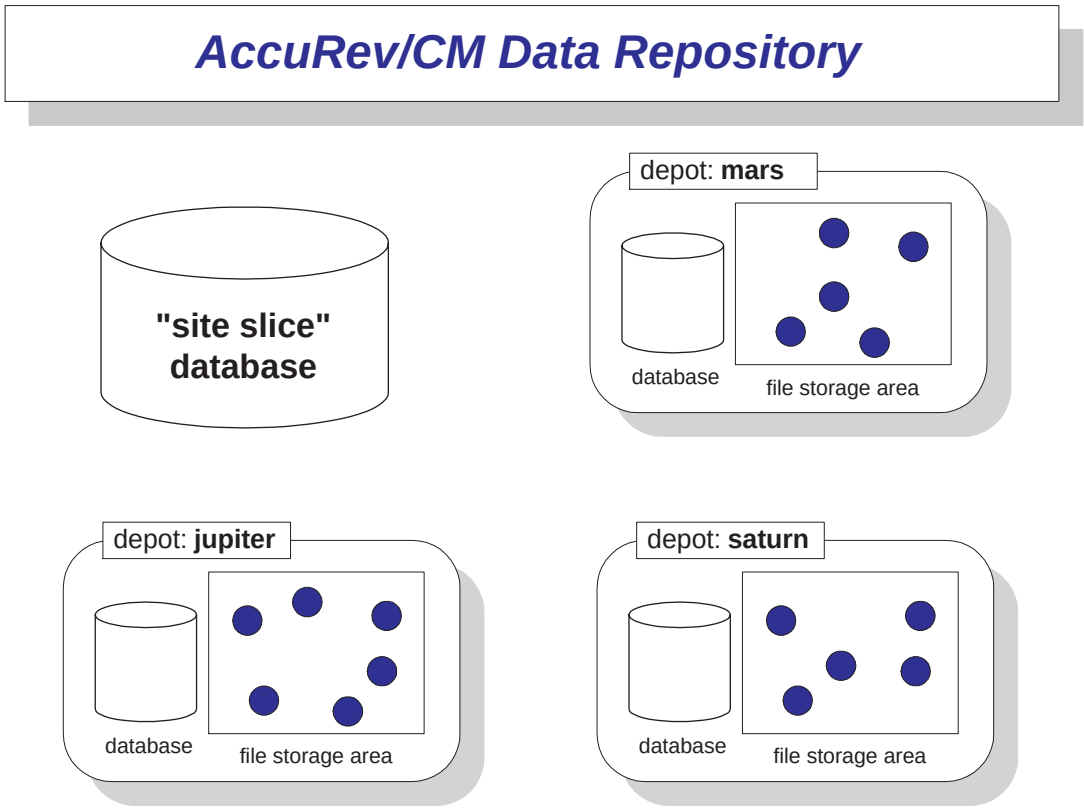
Organization of the Repository: Storage Depots

An AccuRev/CM data repository consists of:

- The site slice, a central database that contains repository-wide information. (It's the "slice" of the repository that contains data pertaining to the entire AccuRev/CM "site".) This includes a user registry and lists of global data structures.
- Any number of depots — short for "storage depots" — which contain separate sub-repositories. Each depot implements a version-controlled directory tree. It provides protected, permanent storage for all the versions of the files in the tree; it also includes a database that tracks the changes to the files themselves, their names, and their organization into directories.

  Alternatively (or additionally), a depot's database can store issue records, which are managed by the companion product AccuRev/Dispatch. Typically, a depot's issue records hold bug reports relating to the depot's files.

The illustration below shows the modular structure of the AccuRev/CM data repository. Logically, the entire repository is located in a single directory tree on the machine where the AccuRev/CM Server program runs. But only the various databases must physically reside of the server machine. The file storage areas — which typically are far larger than the databases and grow far faster — can be located elsewhere. For example, the file storage area of depot **jupiter** might be located on another disk on the AccuRev/CM server machine, and the file storage area of depot **saturn** might reside within the local area network's disk farm.



Single Depot vs. Multiple Depots

You can place all version-controlled files in a single depot, or split them among multiple depots. In general, we advise storing all files for a given project in the same depot. By “project”, we mean all the programs and other software deliverables that share the same development/test/release procedures and the same release cycle. The procedures determine how a depot’s stream hierarchy will be structured; the release cycle determines how the stream hierarchy will be used.

If Project_X and Project_Y have completely different release cycles, then put their source files in different depots. Likewise, if Project_A requires stringent in-house regression testing and two levels of beta-testing, whereas Project_B is mandated to “ship yesterday”, use different depots.

AccuRev/CM has no scalability limits, so there is no problem in storing thousands, tens of thousands, or even hundreds of thousands of files in a single depot.

Inside a Depot: Versions and Files

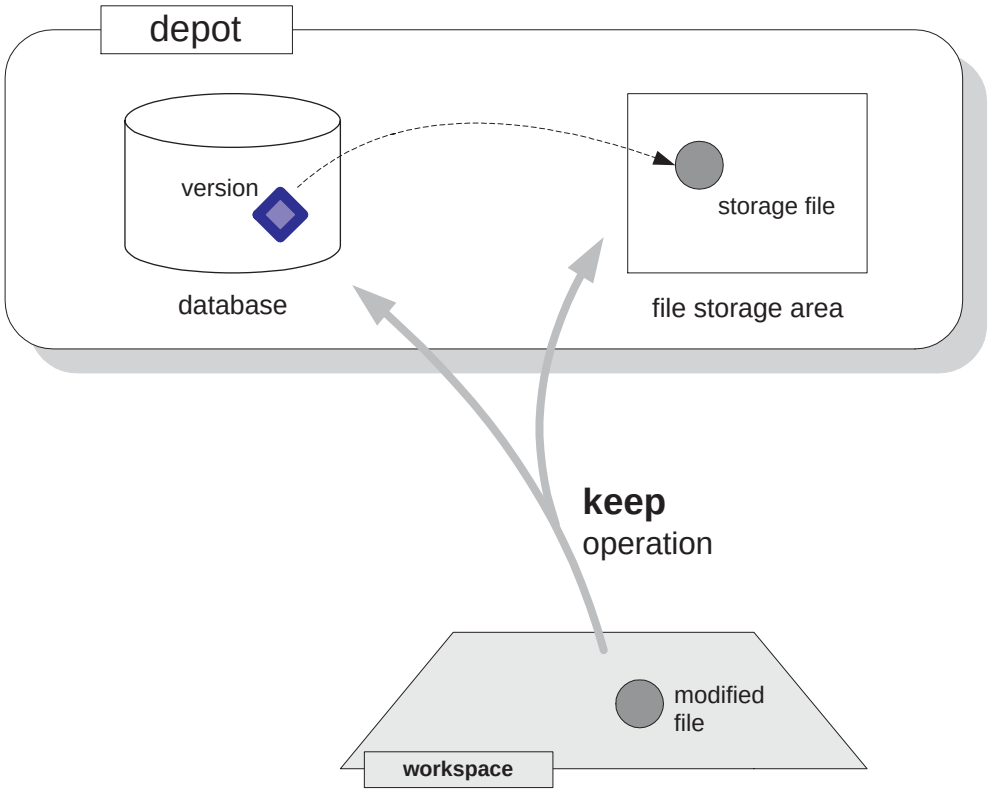
Let’s look inside a depot, to examine its database/file-storage-area architecture. This will help explain how AccuRev/CM works, and will illuminate some of its most important, and unique, features.

Developers working on their files — that’s the principal activity in any software development environment. With AccuRev/CM, a developer’s files are stored in an ordinary directory structure

— perhaps on the hard drive of a personal computer or laptop, perhaps in a designated area of a well-backed-up disk farm, etc. The only thing special about such a "developer's work area" is that AccuRev/CM keeps track of its association with a particular depot. (The work area is termed a workspace — for more information, see *AccuRev/CM Workspaces and Reference Trees* on page 16.)

A developer can use any software tools to create and edit files, compile and build modules and applications. AccuRev/CM doesn't get involved in these operations at all, so there's no performance penalty. Every so often, the developer tells AccuRev/CM to save the current contents of a file (or a group of files). This operation, called a keep, does two things:

- Copies the current contents of the file to a storage file in the depot's file storage area.
- Creates an associated version object in the depot's database.



This association is permanent: no matter what happens in the future, the contents of the file will always be available, through a reference to the version object. (For now, we'll skip the details of how to specify a version — it's just a bit more complicated than saying "version 45 of file gizmo.c".)

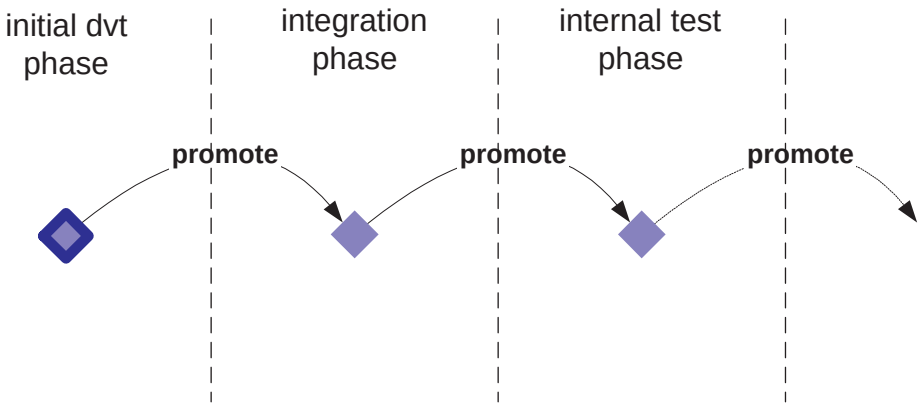
In addition to providing access to the actual file contents, the version object stores additional information relating to the "keep" operation: a timestamp, the user who performed the operation, a user-supplied comment, etc. This kind of information is often termed "metadata".

In general, version objects are much smaller than the corresponding storage files. (Developers often work with large source files; they also work with audio, image, and multimedia files, which can be *really* big.) As developers create more and more versions, the depot's file storage area may

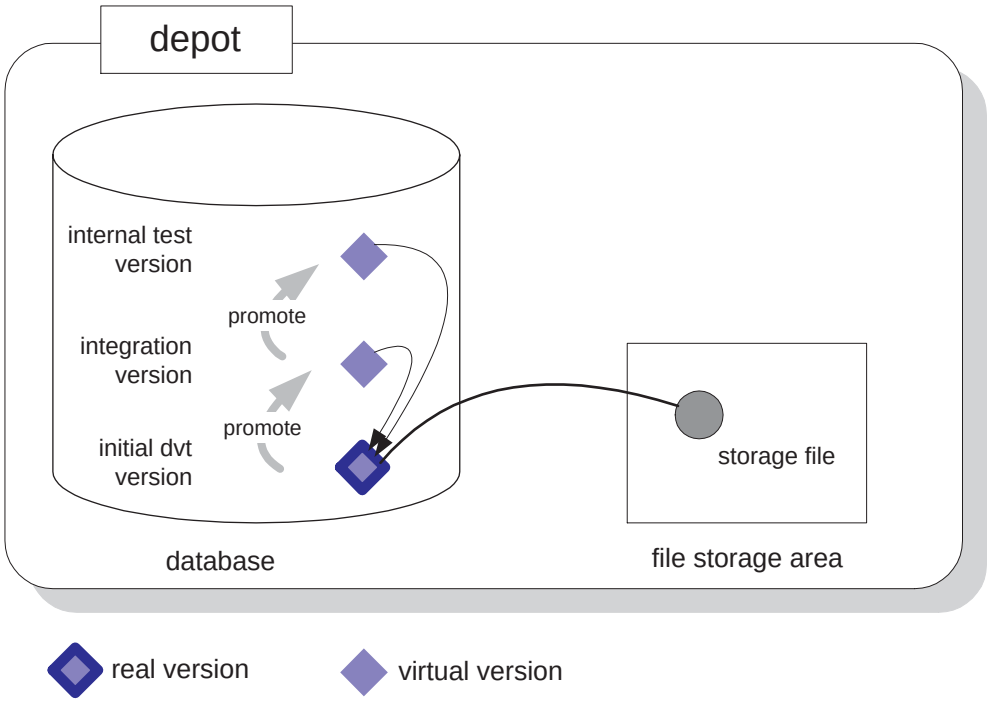
grow to many gigabytes, requiring it to be split among multiple disk drives. But since the depot's database stores the relatively small version objects, it grows much more slowly. Most likely, it will never outgrow its original storage location.

Promotion: Real Versions and Virtual Versions

Software development is much more than just creating and modifying files. A typical development project involves many phases, possibly including initial development, integration of work done independently, internal system testing, external testing, and final production. AccuRev/CM uses a "promotion model" to manage files in these multiple development phases. Files progress through the phases, one by one: when a file passes the tests (if any) mandated for a particular phase, a user working on that phase promotes it to the next phase.



AccuRev/CM keeps track of each promotion by creating a new version of the file. But promotion doesn't change the contents of a file; it only changes the file's "approval level". Thus, each new version object created by promotion is merely an additional reference to (or "alias for") the same file in the depot's file storage area.



AccuRev/CM distinguishes between the original version, created by a **keep** operation, and all the additional versions created by a **promote** operation:

- A real version is created by a **keep** (or an **add**, which places a new file in the depot). The operation creates a new version object in the depot's database, and also places a new file in the depot's file storage area.
- A virtual version is created by a **promote**. It creates a new version object in the depot's database, which provides an additional reference to an existing file in the file storage area.

What is a Software Configuration?

AccuRev/CM is a "software configuration management" (SCM) product. So what's a software configuration? Software developers (programmers, QE engineers, tech writers, etc.) work with information stored in files. The contents of the files change over time, as developers work on them. The developers save the changes in new versions of the files. The organization of the files changes, too: new files are created, old files are deleted, some files get renamed, and directory structures get reorganized.

Take a particular set of files — for example, the files required to build and deliver an application named **Gizmo**. At any given moment, this set of files is in a particular state, which can be described in terms of version numbers:

```
gizmo.c                     version 45
frammis.c                   version 39
base.h                      verion 8
release_number.txt          version 4
Gizmo_Overview.doc          version 19
Gizmo_Release_Notes.doc     version 3
```

... or in terms of time:

```
gizmo.c                     last modified 2001/11/18 14:15:03
frammis.c                   last modified 2001/11/18 14:15:19
base.h                      last modified 2001/10/08 09:09:44
release_number.txt          last modified 2001/11/17 21:59:34
Gizmo_Overview.doc          last modified 2001/11/20 17:25:00
Gizmo_Release_Notes.doc     last modified 2001/11/21 19:29:57
```

That's a software configuration: a particular set of versions of a particular set of files. (AccuRev/CM's naming scheme for versions is slightly more complicated than "version 45 of file gizmo.c".)

> Note: Unlike some other SCM products, AccuRev/CM keeps track of changes to both files and directories. In this discussion, though, we'll concentrate on files.

Suppose one of the files changes:

```
   ...
release_number.txt          last modified 2001/11/24 07:19:18 (version 5)
   ...
```

(Somebody forgot to modify the release number; we're sure that has never happened at your organization.) You can think of this change as producing a new software configuration. But in many situations, it's more useful to think of this as an incremental change to an existing, long-lived configuration — the one called "Gizmo source base" or, perhaps more precisely, "Gizmo Release 2.5 source base".

So in the end, is a software configuration just "a bunch of files"? Almost, but not quite. It's important to keep in mind that a software configuration does not contain the files themselves, but

only a *description* or listing of the files and their versions. Think of the difference between an entire book (big) and its table of contents (small). This crucial distinction makes it possible for AccuRev/CM to keep track of hundreds or thousands of software configurations, without needing an infinite amount of disk storage.

The change described above to file **release_number.txt** illustrates the distinction between files and configurations of files. The change to the contents of the file is something like this:

> replace text line “RELEASE=2.5” with text line “RELEASE=2.5.1”

The change to the software configuration is something like this:

> replace version 4 of file “release_number.txt” with version 5

For another example of the distinction, recall that a configuration takes into account filenames and directory structures, too. Consider this configuration:

src/gizmo.c	version 45
src/frammis.c	version 39
src/base.h	verion 8
src/release_number.txt	version 4
doc/Gizmo_Overview.doc	version 19
doc/Gizmo_Relnotes.doc	version 3

Boldface shows the differences from the first configuration listed above. The file contents are exactly the same; but one filename has changed, and the files have been organized into subdirectories. So this is a different software configuration, even though there has been no change to the *contents* of the files.

Software Configurations and Development Tasks

In most modern software development organizations, many tasks are under way concurrently. At the beginning of this section, we listed a few: new products, new releases of existing products, ports to different platforms, and bugfixes. In addition, consider the fact that each one of the above tasks is often several coordinated efforts: initial development, unit testing, internal system testing, external system (“beta”) testing, final production.

To enable all the tasks to progress smoothly at the same time, each person gets her own software configuration — her own set of versions of the files in the repository. (A small, close-knit team might choose to share a single configuration.)

It’s the job of the software configuration management system, such as AccuRev/CM, to help the organization:

- Keep track of the various configurations.
- Manage, preserve, and protect changes to the files.
- Detect conflicting changes that take place in different configurations (for example, two people modify the same section of the same file).
- Resolve such conflicting changes.

AccuRev/CM Software Configurations: The Stream Hierarchy

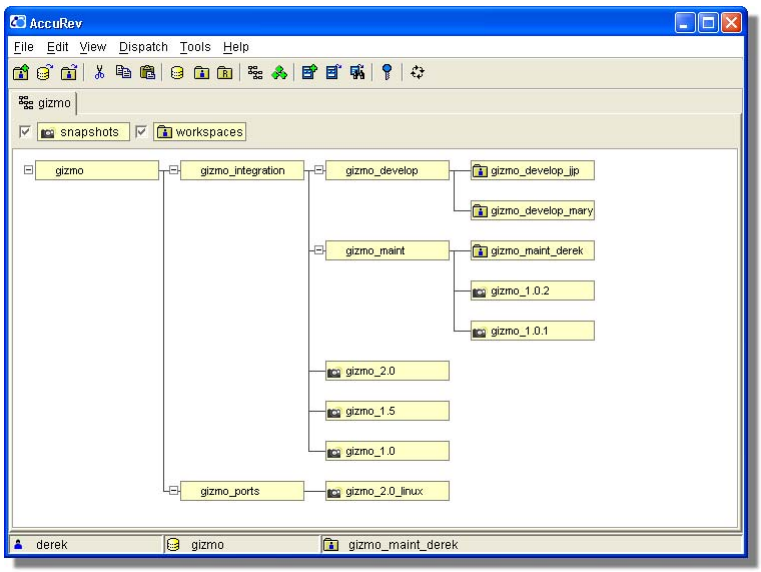
This section discusses the AccuRev/CM implementation of software configurations. Be sure to read the section "What is a Software Configuration?" before this section. First, we set the scene and introduce some necessary terminology.

AccuRev/CM's basic job is to keep track of the changes that a development team makes to a set of files. That's called version control. A file under version control is called an element; developers can create any number of versions of each element. AccuRev/CM saves all the versions permanently in a database called a depot.

> Note: we're oversimplifying here. AccuRev/CM version-controls directories as well as files; and there can be multiple depots, each one storing a separate directory tree. But the above paragraph is enough to get us into a discussion of software configurations. For more on depots and version-controlled files and directories, see section *The AccuRev/CM Data Repository* on page 2.

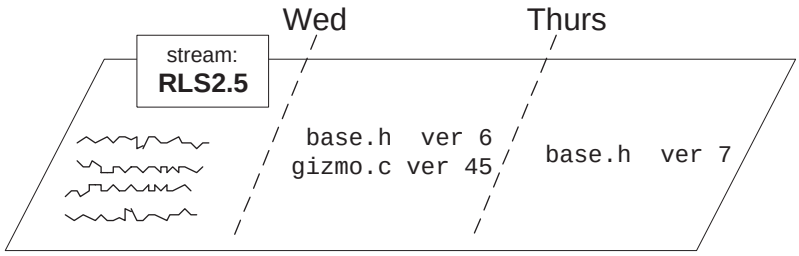
AccuRev/CM can manage any number of configurations of a depot's elements. Each configuration contains one version of every element in the depot — or perhaps, just some of the elements. Here are the basic data structures:

- A stream is a configuration of the depot that changes over time.
- A snapshot is a configuration of the depot that never changes.
- A depot's streams and snapshots are organized into a stream hierarchy: each stream or snapshot has one "parent", and can have any number of "children". The stream hierarchy can be changed at any time: move a child to a different parent, interpose a new stream (the "baby-sitter"?) between a child and its parent, etc.

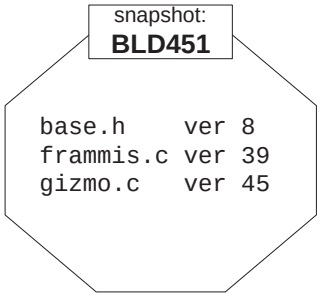


Using these structures, it's easy and intuitive to model many aspects of the software development process. The main idea is to enable multiple development tasks to take place concurrently, and to manage when (and if) work done for one task is shared with other tasks. For example:

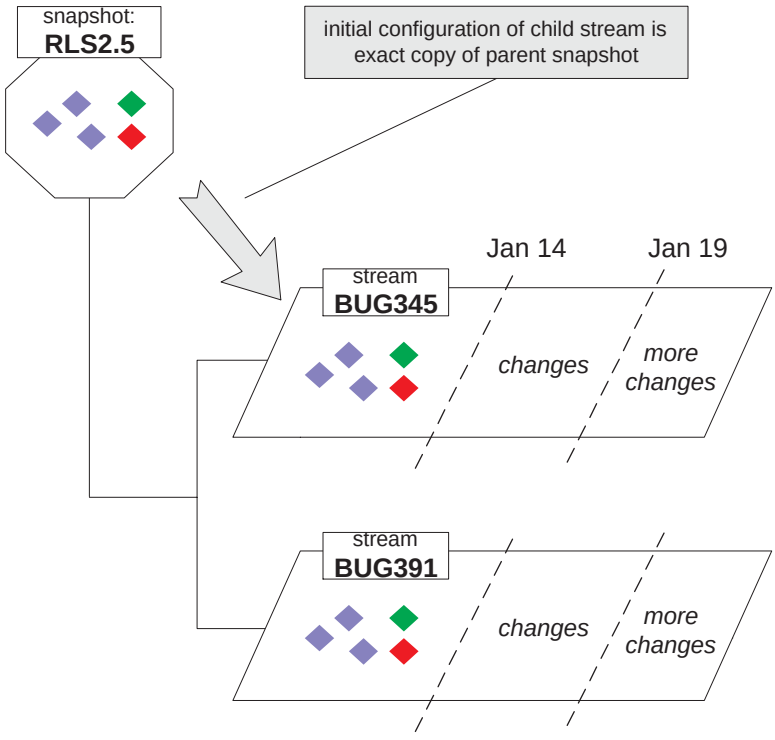
- A stream corresponds to a development task. It might be a long-lived project, such as “the Release 2.5 development effort”; or it might be a quickie, such as “fix error message ERR037”. When a developer modifies an element, the new version is recorded as a change to the configuration of a particular stream.



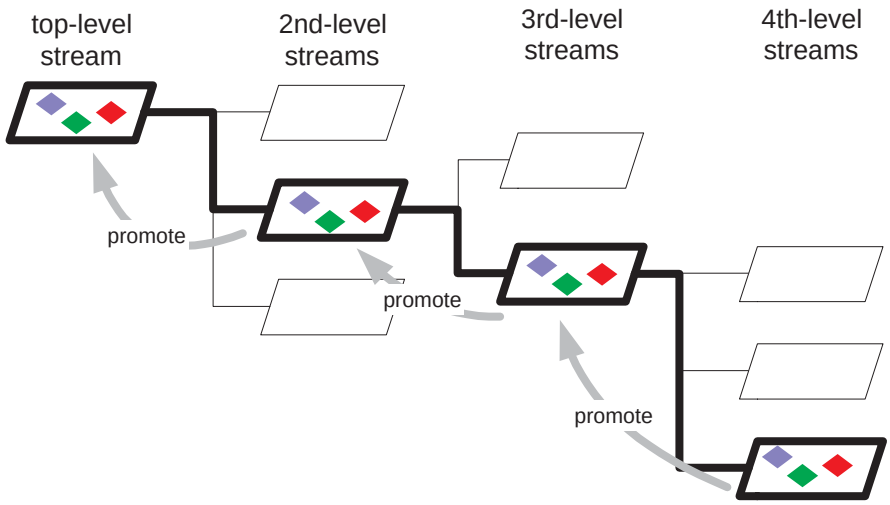
- A snapshot corresponds to a project milestone, such as “Build 451” or “Release 2.5 final build”. It’s vitally important to be able to tell exactly which versions of which files went into Build 451, no matter what changes were made subsequently. A snapshot answers this need precisely and completely reliably, because it’s a never-changing configuration.



- A “parent” snapshot acts as a stable starting point for any number of “child” streams. No matter when a new child is created, its initial configuration is an exact copy of the parent snapshot. This structure is appropriate for managing multiple bugfixes to an old release. Each bugfix stream starts with the versions that were used to build the original release — say, the versions in snapshot “Release 2.5 final build”.



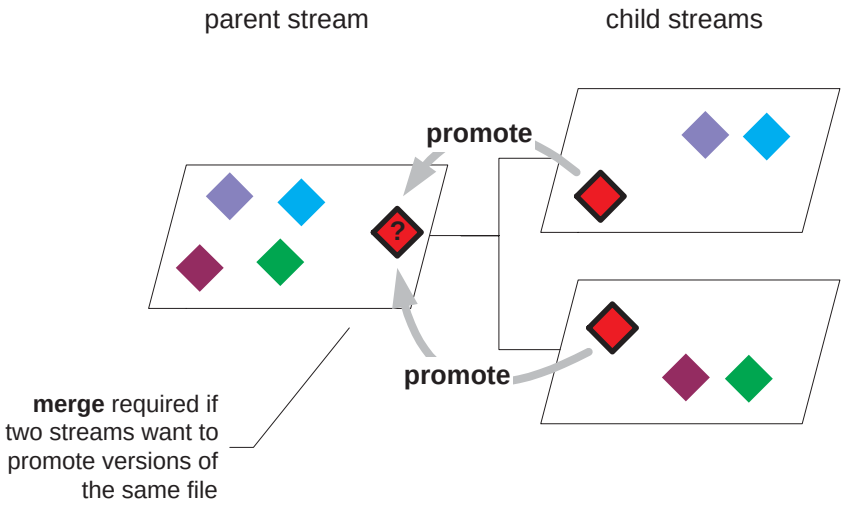
- Versions created at the bottom of the stream hierarchy rise up through the hierarchy by being promoted from stream to stream — from child to parent, then from parent to grandparent, etc. Promotion is one of AccuRev/CM’s most important operations, enabling you to intuitively model a project’s workflow.



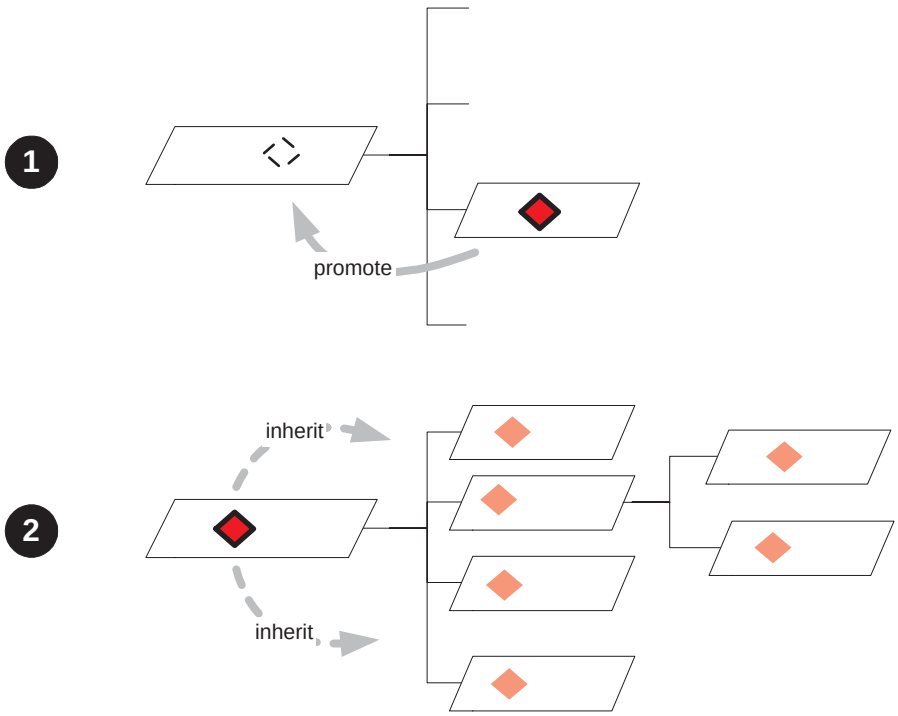
For example, after initial development work on a set of files is completed, the files are submitted to unit testing, then to internal system testing, then to external system ("beta") testing, then to final production. If this workflow is too elaborate for your organization, or not elaborate enough, just design your stream hierarchy differently. You can redesign a project's workflow at any time by changing the stream hierarchy.

- A parent stream provides an integration point for any number of child streams. This structure is appropriate for a development effort that is divided into multiple tasks, to be undertaken concurrently by different developers. As developers complete their changes, they promote the changes to the parent "integration stream".

If two or more developers happen to change the same file, AccuRev/CM makes sure that the changes are merged together. This ensures that one person's work is not overwritten accidentally by another person's.



- Each stream provides a change scope for the subhierarchy beneath it: child streams, grandchild streams, etc. Once a version has been promoted to a stream, that version becomes available to the stream's entire subhierarchy. In many cases, the newly promoted version will appear automatically in ("be inherited by") all the descendant streams. This auto-integration mechanism complements the explicit integration of merging, described in the preceding paragraph.



  For example, suppose a new corporate logo has been designed and saved in a new version of file **corp_logo.png**. Promoting this version to a high-level stream makes it appear instantly in many lower-level streams where Web pages are being developed and updated.

It may be worthwhile to study the above scenarios a bit more, and to consider how your organization might use AccuRev/CM's streams and snapshots in your own development environment. As you do so, keep these two important points in mind:

- A stream is a software configuration, a specification of particular versions of particular elements. A stream doesn't contain copies of files stored in the depot's file storage area; it just contains a "matched set" of versions, selected from all the versions recorded in the depot's database.
- A depot's files are organized into a directory tree; a depot's streams are organized into a tree-structured hierarchy. These two tree structures are different and independent of each other. In a sense, the directory tree is a "picture" of a software application, and the stream hierarchy is a "picture" of the software development process that creates and maintains the application.

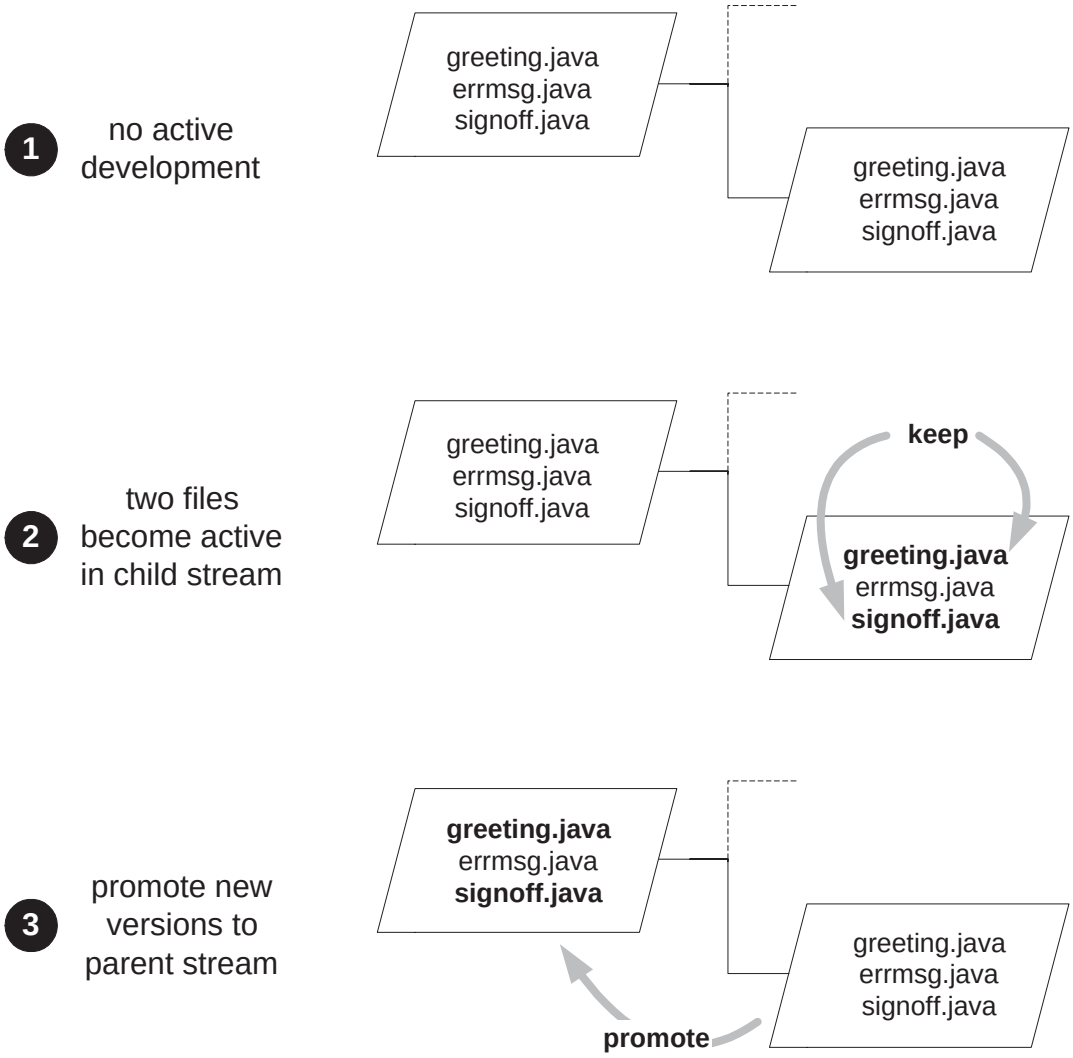
How Changes Migrate Through the Stream Hierarchy

AccuRev/CM provides configuration-management capabilities that are sophisticated and robust, without sacrificing ease of use. What's the secret? One main reason is that AccuRev/CM sees the development environment in the same way as a typical development team:

- Many development tasks are active concurrently, all using the same source base.

- Tasks are often interrelated; they must share their changes with each other ("integration") and weed out inconsistencies; some tasks cannot be completed until one or more others have been completed.
- Most tasks are accomplished by making changes to relatively few files.
- A task is completed by "delivering" a set of changes to another task. For example, a development task might deliver its changes to an integration task, or to a testing task.
- A developer's next task may involve changing a completely different set of files from the previous task.

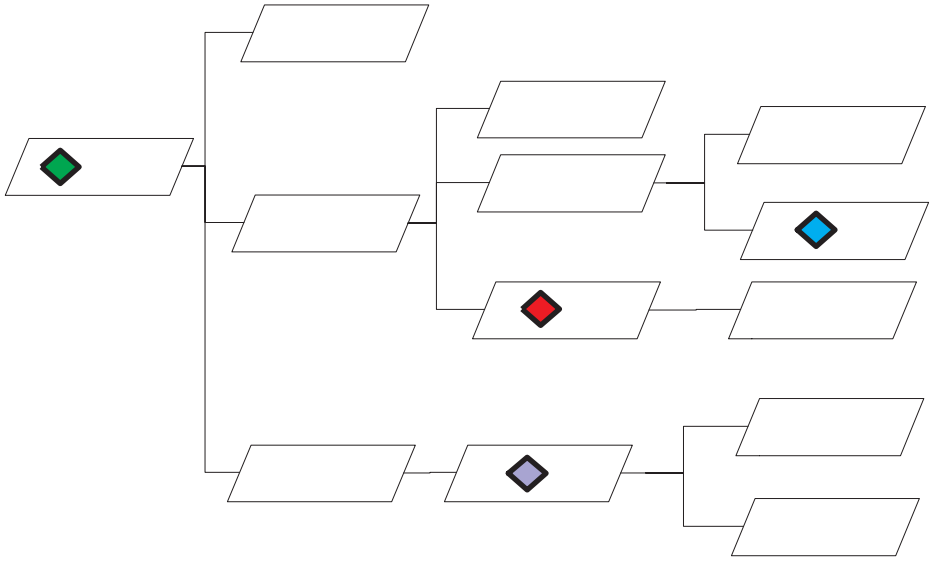
AccuRev/CM streams neatly model all these aspects of development tasks. The (relatively few) files that a developer changes for a task become active in a particular stream. Typically, this occurs when the developer records new versions of the files, using the **keep** command. To complete the task, or to mark an intermediate milestone, the developer delivers the changes to the parent stream, using the **promote** command. The files become active in the parent stream, and they revert to being inactive (not under active development) in the child stream.



In a multiple-level stream hierarchy, several promotions are required to propagate a set of changes all the way to the top level. Each promotion causes the file(s) to become active in the "to" stream, and inactive in the "from" stream.

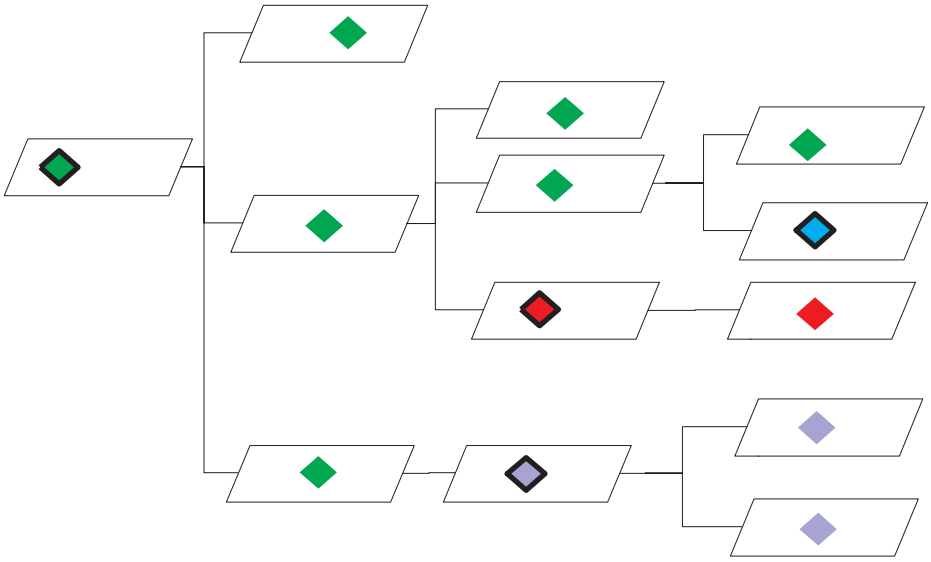
You may have gotten the impression that a given file can be active in only one stream at a time. Not so — that would mean only one development task at a time could be actively working on the file. AccuRev/CM allows each file to be active in any number of streams — even *all* of the streams at once. Typically, though, a file is active in just a few streams at any particular moment.

The diagram below uses contrasting colors to show how a particular file might be active in four different streams. That is, four different versions of the same file are in use at the same time, for various development tasks.



Inheriting Versions From Higher-Level Streams

What about the other streams? Each stream in the hierarchy contains *some* version of the file; if a file is not active in a particular stream, the stream automatically inherits an active version from a higher-level stream. The diagram below shows how the four active versions fill out the entire stream hierarchy:



This scheme makes it easy for an organization to manage many development tasks concurrently, each with its own software configuration in a separate stream. As changes are made for certain tasks, AccuRev/CM takes care of automatically applying the changes to the software configurations used by other subsidiary tasks — except for the tasks that are actively working on the same file(s). Just a few **promote** operations can effectively propagate versions to tens or even hundreds of other streams.

AccuRev/CM Workspaces and Reference Trees

As described in *AccuRev/CM Software Configurations: The Stream Hierarchy*, AccuRev/CM uses streams to organize your development data, as any number of projects are under way concurrently. But streams are not the entire story:

- A stream is just a bookkeeping device, though a very sophisticated one!. It's a database mechanism that records which versions of files are in use for a particular development task. But what about the actual files themselves, which developers edit and build software systems with?
- The **promote** command propagates an existing version of a file from a lower-level stream to a higher-level stream. But how are new versions of files created in the first place?

In other words, how do users access AccuRev/CM-controlled files, in order to perform their day-to-day development tasks? The answer: through workspaces.

A workspace is an ordinary directory tree that instantiates a stream. That is, the workspace contains files that are copies of the versions in the stream. We say that the workspace is "attached to the stream" or "based on the stream". And the stream is said to be the backing stream for the workspace; we'll explain this term in *Updating a Workspace* on page 20.

For example, suppose a stream contains these versions of the elements in a (very small) depot:

```
src/gizmo.c                    version 45
src/frammis.c                  version 39
src/base.h                     verion 8
src/release_number.txt         version 4
doc/Gizmo_Overview.doc         version 19
doc/Gizmo_Relnotes.doc         version 3
```

A workspace attached to this stream is a directory tree containing:

- a **src** subdirectory, containing four files (**gizmo.c**, **frammis.c**, **base.h**, **release_number.txt**).
- a **doc** subdirectory, containing two files (**Gizmo_Overview.doc**, **Gizmo_Relnotes.doc**).

Another stream in the depot's stream hierarchy might contain different versions of some or all the files. So, for example, the contents of files **release_number.txt** and **Gizmo_Relnotes.doc** might be different in a workspace attached to another stream.

Any number of workspaces can be attached to the same stream. A typical scenario is for all the members of a project team to maintain workspaces attached to the stream that records the project's ongoing work. Conversely, a workspace can be attached to any stream. But typically, workspaces are created only at the "leaf level" of a depot's tree-structured stream hierarchy: if a stream acts as the backing stream for one or more workspaces, it generally doesn't have child streams, too.

Using a Workspace

As the name implies, a workspace provides a location for performing development tasks: editing source files, compiling, debugging, testing, creating web sites, etc. Since a workspace is a regular directory tree in the file system, there are no special issues involved with using software development tools with AccuRev/CM data. Just do it.

Here are a few points that show how easy it is to do day-to-day work in a workspace:

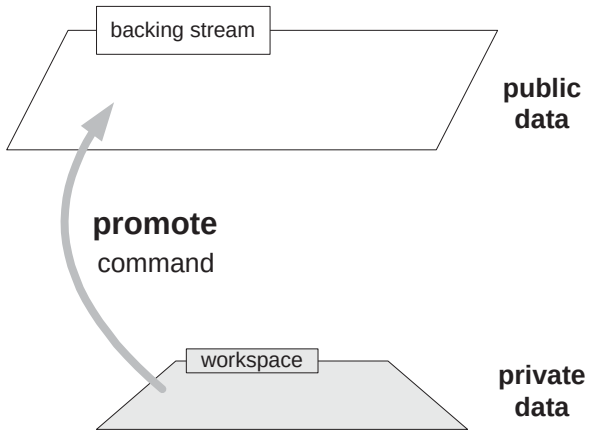
- A workspace need not be in any special file system location. Any place where you have permission to store data will do.
- If you decide you need more space, you can move a workspace to another location. And you don't have to worry about losing track of your workspaces — AccuRev/CM keeps track of every workspace's location.
- You can modify any file in a workspace at any time. Some configuration management systems require you to perform a "check out" operation before working on a file, and keep most files in a read-only state — but not AccuRev/CM.

The thing that's special about a workspace is that it provides a two-way portal to the AccuRev/CM data repository: you put your own changes into the repository, and you draw out the changes that your colleagues have previously recorded there.

Putting Data Into the Repository

A workspace enables you to create new versions of the files in a particular depot. (Each workspace is attached to a particular stream, which belongs to a particular depot.) First, you use any development tools to work with the workspace's copies of existing versions; then you use AccuRev/CM commands to store new versions in the depot. In addition to creating new versions of existing files (**keep** command), you can use the workspace to add new files and directories to the depot (**add** command), rename files and directories (**mv** command), and even rearrange the depot's directory hierarchy (**mv** command).

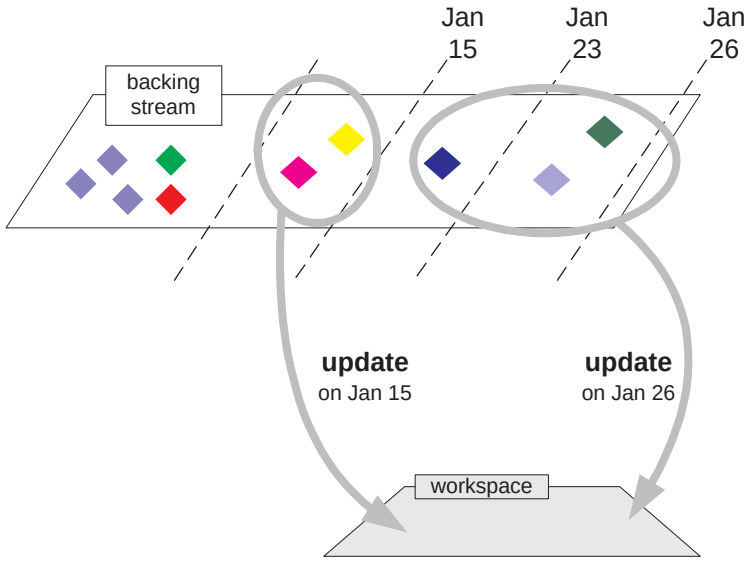
Because it's a separate directory tree, a workspace provides an isolated, private development environment. The changes you make become public only when you enter a **promote** command. This creates versions of one or more files in the attached stream. These versions are public: your changes are now available to be incorporated into other workspaces attached to the same stream. Subsequent promotions to higher-level streams will make the changes available to an even wider scope of workspaces.



Getting Data Out of the Repository

A stream is a changing software configuration of a depot. A typical stream has new versions entering it all the time. Some of the versions are promoted from the workspaces attached to them, as described just above; other versions are inherited automatically from higher-level streams. (See *Inheriting Versions From Higher-Level Streams* on page 15.)

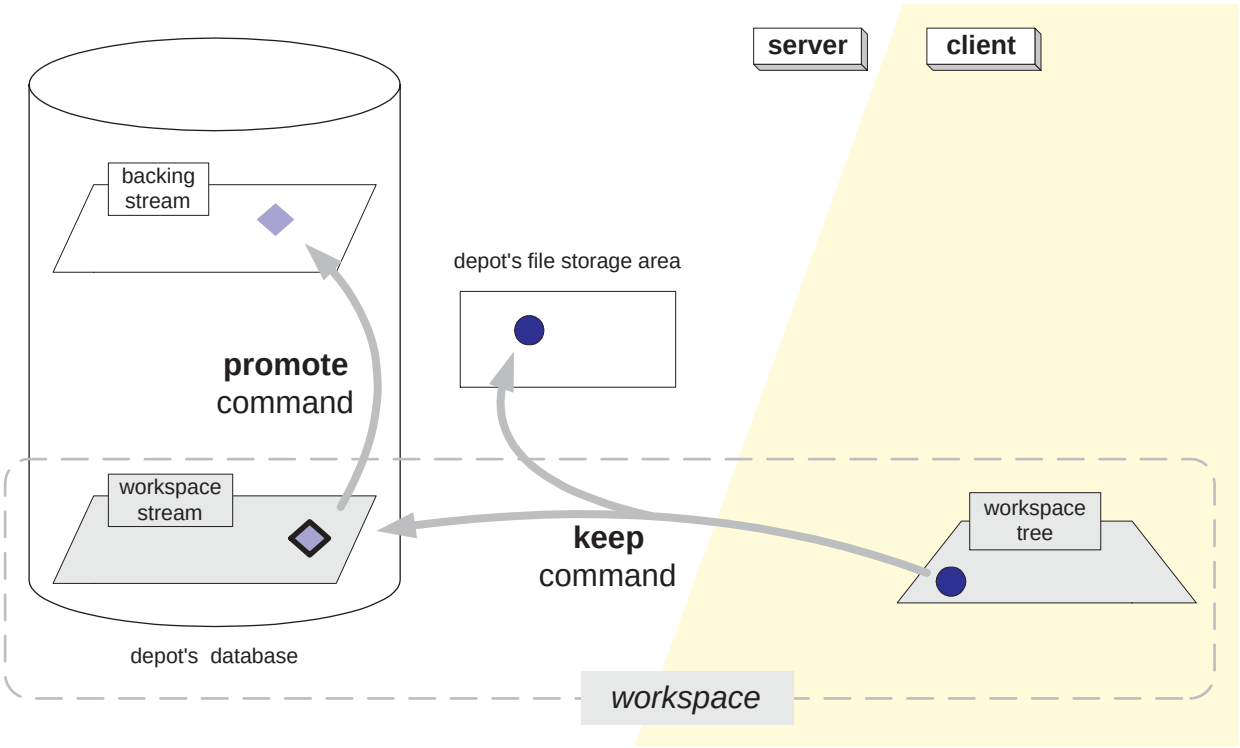
As new versions enter a stream, they become available to the workspace(s) attached to the stream. But AccuRev/CM never copies a new version of a file into your workspace automatically. Instead, you periodically use AccuRev/CM commands to "update" the workspace. This replaces existing files (or adds new ones), so that the files in the workspace accurately reflect the backing stream's current contents, including any recently-arrived versions. AccuRev/CM takes care not to "clobber" files that you're working on when it copies new versions to the workspace.



The Workspace's Builtin Stream

The diagram above, showing how data flows from a workspace into the repository, is an oversimplification. Changes that you make in your workspace don't actually go directly into the backing stream. Long experience with configuration management systems has shown that users sometimes enter changes into the repository before they're truly ready to be shared with others — for example, code that's never been tested. But a delaying strategy also has its drawbacks — for example, it increases the chances of mistakenly deleting several weeks worth of changes without ever preserving them in the repository.

Some other version control systems use "private branches" to address these issues. AccuRev/CM solves the problem by building a private stream into each workspace. This builtin stream is separate from the backing stream. Here's a more detailed diagram showing how data flows from a workspace into the repository; this one includes the workspace's builtin stream.



This diagram shows that in AccuRev/CM's client-server world, a workspace has one foot on each side of the divide:

- The ordinary directory tree that we discussed above (the workspace tree) lives on the client side. The development data you work with on a day-to-day basis lives entirely in the workspace tree; it's "just a bunch of files".
- The builtin stream (the workspace stream) lives on the server side, in the data repository managed by the AccuRev/CM Server. It contains all of the workspace's configuration management information. And it resides, as all streams do, entirely within the database of a particular depot.

The diagram also shows that recording a new version of a file in the backing stream is a two-step process:

1. The **keep** command creates a new version in the workspace stream. Think of **keep** as moving data from the client side to the server side. This command also copies the file in your workspace tree, storing the copy in the depot's file storage area. The data stays within the workspace, remaining private.
2. The **promote** command propagates the version from the workspace's builtin stream to the backing stream. This command operates totally within the depot's database. No data file is copied to the file storage area during a **promote**.

Why the extra stream and the extra step? Isn't it redundant? No, because the workspace stream and backing stream play different roles. The whole idea of the workspace is to provide a degree of isolation from the changes that others are making concurrently. The workspace stream makes the isolation more flexible. It enables you to **keep** any number of intermediate versions of a file in

your workspace, before "going public" by **promote**'ing the most recent version. If you decide that you've headed off in the wrong direction, you can revert a file to any of those intermediate versions and promote that version instead. No one else needs to know. You can even **purge** *all* the work you've done on a file, reverting the workspace to using the version in the backing stream.

All the intermediate versions that you **keep** are stored permanently in the depot, even the versions you never **promote** to the public stream hierarchy. Thus, the **keep** command provides a data-backup capability: "save a copy of this file, just in case I ever want to restore it to its current state". It also means you can change your mind as many times as you like about which version of a file should be shared with the rest of the world.

Real Versions and Virtual Versions

The difference between **keep** and **promote** highlights an important aspect of the way that AccuRev/CM organizes and manages development data. It also highlights the difference between backing streams and workspace streams.

All "real" development takes place in workspaces, because that's where the files are. The **keep** command preserves the changes you've made to a file (Java source file, Perl script, MPEG audio file, etc.) Accordingly, versions created by the **keep** command are called real versions. Real versions live in workspaces — more precisely, every real version is created in the builtin workspace stream of some workspace.

By contrast, the **promote** command does not record a new change to any file. Rather, it changes the approval level and availability of a change that was previously recorded with **keep**. The version that **promote** creates in a higher-level stream is called a virtual version; each virtual version just a pointer to, or alias for, an existing real version in some workspace stream.

'Active' Files and the Default Group

AccuRev/CM keeps track of which files you're actively working on in your workspace. This set of files is called the workspace's default group. It includes all the files for which you've recorded changes in the repository. Typically, most of the changes are new versions, created with **keep**. The default group also includes renamed or relocated files (**mv** command) and deleted files (**defunct** command).

When you **promote** a file's changes from your workspace to the backing stream, the file is removed from the workspace's default group. This reflects that fact that you're done working on that file — at least for now! Similarly, a **purge** of your work on a file removes the file from the workspace's default group.

Updating a Workspace

The two-part structure of a workspace — workspace tree on the client side, workspace stream on the server side — plays an important role in how AccuRev/CM keeps a workspace synchronized with the stream to which it's attached.

At any given time, a workspace should contain:

- the files you're actively working on (that is, the members of the workspace's default group)

- for each other version-controlled file in the depot, a copy of the backing stream's version

(You can think of the active files as being in the "foreground" of the workspace, and the non-active files as being in the "background". Those "background files" are copies of versions in the stream to which the workspace is attached. That's why it's officially called the workspace's backing stream.)

But a workspace often gets out of date with respect to its backing stream. Typically, each member of a development team has his own workspace, and all the workspaces are based on the same backing stream. For files that you're not working on, your workspace continues to have copies of old backing stream versions, even as your colleagues are promoting new versions of those files to the backing stream.

It's the job of the **update** command to synchronize the workspace and its backing stream in this way. To determine which files you're actively working on, **update** looks in the workspace stream; it considers a file to be active if you've created one or more new versions of it in the workspace stream. Then, **update** makes sure that the workspace tree contains a copy of the backing-stream version of each non-active file. Typically, this involves replacing old files with new files. But it can also involve renaming, relocating, and removing files — if those kinds of changes have recently been recorded in the backing stream.

Variation #1: Workspace Based on a Snapshot

A workspace can be based on a snapshot, instead of a stream. Initially, this might not seem to make sense; after all, a snapshot is an *unchanging* software configuration, and a workspace is a tool for getting changes in and out of the data repository (a "two-way portal"). But a snapshot-based workspace is quite useful — for example, for performing maintenance work on a previous product release.

When you create a snapshot-based workspace, AccuRev/CM copies the versions in the snapshot to the new workspace tree. (This step is just like the creation of a stream-based workspace.) For example, you might create a workspace containing exactly the source versions that were used to build Release 6.1 of your product. That's the only time development data flows from the repository to the workspace. It doesn't make sense to **update** the workspace, because there's guaranteed to be nothing new in the snapshot. It's a configuration that never changes.

You can make changes to the files in a snapshot-based workspace, saving the changes in the workspace stream with the **keep** command. You can't **promote** the changes to the snapshot, though, because — once again — the snapshot is a configuration that never changes. In some cases, there won't be any need for such promotions. For example, some of the bugfixes to a previous product release never need to be propagated elsewhere. You can just build the maintenance release(s) in the maintenance workspace where you've fixed the bugs.

In other cases, you'll want to incorporate bugfixes into ongoing development work — perhaps Release 6.2 or 7.0 of your product. AccuRev/CM has special facilities, including the Change Palette, which enable you to propagate changes from a maintenance workspace (or any snapshot-based workspace) to any stream.

Variation #2: Reference Tree

Let's go back to our original definition of a workspace: an ordinary directory tree that instantiates a stream (or a snapshot). We expanded that definition, showing that a workspace also includes mechanisms for creating new versions in the stream. Sometimes, though, you don't need to create any new versions — you just need the files. For example, you might want a complete set of your product's source files in order to test the speed of a new C++ compiler.

For such "just the files" purposes, you can create a reference tree instead of a workspace. A reference tree instantiates a stream or snapshot, but doesn't provide any mechanism for creating new versions. Thus, you can't use the **keep** or **promote** commands when working in a reference tree. You can use the **update** command, though. Here's a typical scenario:

- Create a reference tree named **nightly**, based on stream **gizmo_dvt**.
- Each night, perform an **update** of the reference tree. This retrieves new copies of the files for which new versions appeared in the **gizmo_dvt** stream that day.
- After the update is complete, build the Gizmo software application using the updated sources.

You can think of a reference tree as a 1-way portal to the AccuRev/CM data repository (in contrast to a workspace, which is 2-way).

Parallel and Serial Development

Like other advanced configuration management systems, AccuRev/CM supports parallel development:

- **Edit Stage.** Two or more users start with the same data: a particular backing-stream version of a file. Each user works on a copy of the file in his own workspace. He can keep as many (private, intermediate) versions as he wishes in his workspace stream.
- **Merge Stage.** The merge stage begins when one of the developers promotes his private version of the file to the backing stream. After that, each other developer must merge the current version in the backing stream into his own work, then promote this merged, private version. In the end, all users' changes are incorporated into the backing stream; conflicting changes to the file, if any, are both detected and resolved.

If two developers work on a file concurrently, a single merge-and-promote is required. If *N* developers work on a file concurrently, then *N–1* merge-and-promotes are required.

Serial Development through Exclusive File Locking

Parallel development is flexible and powerful, but it's not appropriate for every situation. Some organizations don't like the extra steps involved in merging, even though merging is largely automated. Some files cannot be merged, because they are in binary format. (The merge algorithm handles text files only, not binary files such as bitmap images and office-automation documents.)

Accordingly, AccuRev/CM supports serial development through its exclusive file locking feature. Each workspace is in parallel-development mode (exclusive file locking disabled) or is in serial-

development mode (exclusive file locking enabled). You can switch a workspace back and forth between these modes, using the **chws –k** command.

If a depot is created with **mkdepot –ke**, all of its workspaces use serial-development mode (exclusive file locking enabled). You cannot switch these workspaces to parallel development mode with the **chws –k** command.

The serial development model places more restrictions on users in the edit stage, but it eliminates the merge stage altogether. Here's the standard scenario, in which all the workspaces are in serial-development mode:

1. A user starts working on a file by specifying it in a **co** (“checkout”) or **anchor** command. The file changes from being read-only to writable.
2. AccuRev/CM places an exclusive file lock on the file. This prevents the file from being processed with **co**, **anchor**, or **keep** in other workspaces.
3. The user can edit and **keep** any number of private versions of the file in his workspace. Then, the user **promote**s his most recently kept version to the backing stream. The exclusive file lock guarantees that no merge will be required before this promotion.
4. After **promote** records the new version in the backing stream, things return to the initial state: AccuRev/CM releases the exclusive file lock, and the file returns to read-only status in the user’s workspace.
5. A user in any workspace can now **co** or **anchor** the file, which starts the exclusive-file-locking cycle again.

More generally, an exclusive file lock is placed on an element when “active development” commences on the element in some workspace:

- An **anchor** or **co** commands declares your intention to modify the current version of an element.
- A **co –v** or **revert** declares your intention to use (and possibly modify) an old version of an element.
- A **keep** command creates a new version of an element in the workspace stream.
- A **move**, **defunct**, or **undefunct** command creates a new version of an element in the workspace stream. The new version records a naming-level change to the element, rather than change to its contents.

And the exclusive file lock is released when active development on the file ends in that workspace:

- A **promote** command sends your private changes to an element from your workspace stream to the backing stream.
- A **purge** command discards your private changes to an element.

Either way, the workspace returns to using a backing-stream version of the element.

The Limited Effect of an Exclusive File Lock

Exclusive file locking does not freeze an element completely:

- The lock applies only within the scope of a particular backing stream. It doesn't affect other backing streams and the workspaces based on them.
- The lock applies only to workspaces in serial-development mode. Users in parallel-development-mode workspaces can make changes and promote the changes to the backing stream.
- The lock doesn’t prevent the current version in the backing stream from being promoted to higher-level streams.

Exclusive file locking does not prevent any user from modifying any file with a text editor or IDE. AccuRev/CM encourages users in serial-development-mode workspaces to “ask permission first”: it maintains files in a read-only state, and makes a file writable when a user executes a **co** or **anchor** command on it. But users can modify a file “without asking permission”, by changing the access mode (Unix: **chmod** command, Windows: **attrib** command or **Properties** window) and then editing it. Such “unauthorized” changes can’t be sent to the AccuRev/CM depot, though: the exclusive file lock disallows a **co**, **anchor**, or **keep**.

AccuRev/CM Transactions

The AccuRev/CM data repository is organized into a set of depots, each of which stores the complete revision history of a particular directory tree. Each depot has its own database. Changes to a depot's database are structured as a series of transactions, each of which captures all the information involved in a particular change to the database. Thus, the entire story of how a depot's directory tree has evolved is contained in its transaction history.

Transactions are a well-established database technology, helping to guarantee that the database is always in a self-consistent state. But for AccuRev/CM, transactions are not just a low-level mechanism for achieving database integrity. They play an essential role in organizing the user environment. Two aspects of AccuRev/CM transactions make this possible: atomicity and immutability.

Transactions are Atomic

A user command that modifies elements is recorded as a single transaction in the depot's database, no matter how many elements are involved. For example, if a user enters a **keep** command to create new versions of 12 files, a single transaction records all 12 versions. What if something goes wrong (for example, a network failure) while AccuRev/CM is processing those 12 files? The entire transaction is cancelled, and no new version is created of any file. We use the term atomic to describe this "all or nothing" aspect of AccuRev/CM transactions.

The atomicity of transactions makes life simpler for the user. He never needs to worry about how to finish up the work of a partially-successful command. If a command fails, he just fixes the problem that caused the failure and enters exactly the same command again. Atomicity also means that AccuRev/CM's view of the various changes applied to the repository matches the user's view.

> Note: AccuRev/CM does not record *every* change in a transaction, only changes to your development data. Thus, **keep**ing a new version is recorded in a transaction, as is **promote**'ing an existing version to a higher-level stream. But no transaction is recorded when you create a new stream or change the location of a workspace.

Transactions are Immutable

Once a transaction is recorded in a depot's database, it's there permanently. There is no way to revise or delete an existing transaction — we describe the transaction as immutable. (And we describe the depot's database as being "append-only".) This property is essential to successful configuration management. Users must be able to recreate previous configurations with absolute reliability. The immutability of transactions means that users can reproduce *any* previous configuration, not just a few configurations that they happened to designate with a "save" or "label" command.

AccuRev/CM does make it easy to undo the *effect* of a transaction. For example, the **revert** command reinstates an old version of one or more files. But this is accomplished by recording an additional transaction, not by removing any existing transaction.

Transactions and Workspaces

This section describes how AccuRev/CM uses a depot's transaction history to efficiently manage the contents of the depot's workspaces.

Over time, the version-controlled files in a workspace change in two ways: you modify certain files yourself, using text editors and other development tools; and you periodically use the **update** command to get copies of the files that your colleagues have modified. Accordingly, at any given moment the version-controlled files in a workspace fall into two categories:

- **Files placed in the workspace by the 'update' command.** All of these files are unmodified copies of the versions in the workspace's backing stream at the time of the most recent **update**. (Some of them may have been placed in the workspace during previous updates. Typically, some files are copied into the workspace when it is originally created and are never touched thereafter, because no new versions of the files are ever created in the backing stream.)

  AccuRev/CM records the fact that the workspace is up-to-date as of the transaction that most recently precedes the time of the update. (This is completely accurate — by definition, no new versions were created between that transaction and the update.) This transaction is called the current update level of the workspace.

- **Files that you've worked on in the workspace.** These are files that you've modified (or newly created), and whose changes you've preserved with the **keep** (or **add**) command. You may also have **promote**d the latest version you created to the workspace's backing stream.

AccuRev/CM can quickly fulfill a request to **update** the workspace, because it doesn't need to consider every file in the depot. Instead, it needs to process only the files for which new versions have been created since the workspace's last update. It accesses these versions by examining the set of transactions between the workspace's current update level and the most recent transaction. When the update is complete, the most recent transaction becomes the workspace's new update level.

The update algorithm is further optimized in that it only needs to consider the workspace's unmodified files, not the files that you're currently working on. AccuRev/CM takes care not to "clobber" files you've just edited with copies of old versions. There's one all-important exception: files that you've modified and then promoted to the backing stream revert to being "inactive" in the workspace, and thus become candidates for updating. (They'll be updated only if someone has subsequently promoted an even newer version to the backing stream.)

Transactions and Issue Tracking

The atomicity of transactions makes it efficient to implement the integration between AccuRev/CM's basic version-control facility and the Dispatch issue-tracking facility. Suppose a particular Dispatch issue record contains a bug report. When you fix the bug by modifying 5 files, you'll want to annotate the issue record accordingly. Instead of noting the 5 individual files in the issue record, you simply note the single **promote** transaction that placed the fixed versions of the 5 files in the backing stream.

Virtual Versions

The AccuRev/CM stream hierarchy is made possible by virtual versions. To understand virtual versions, it is necessary to understand conventional versions. In AccuRev/CM terminology, conventional versions are called real versions and are also the most basic building block in the system.

Real Versions

A real version is created every time a user performs a **keep** (similar to the "check-in" operation in other version-control systems). The first **keep** creates version 1, the second **keep** creates version 2 and so on.

Creating versions in this manner works fine until you want to do something more complicated such as allowing more than one person to change a file. Conventional CM systems handle this with branches and labels. Slightly more sophisticated systems also do some simple ancestry tracking to handle merges.

AccuRev/CM uses hierarchical streams, complete ancestry tracking, and virtual versions to handle all versioning tasks. For now, let's concentrate on virtual versions.

Streams and Versions

Most CM systems, especially those based on RCS either directly or conceptually, start out with two-part version numbers. The first part is the branch number, and the second part is the version along that branch. The branch number can vary from file to file, so there is no correspondence between a branch number and a particular branch repository-wide. If you want to determine the branch number in a particular file, you must examine the symbolic name associated with the branch to get the branch number.

In AccuRev/CM, each stream has a number that remains constant for every file and directory in the whole repository. You can use the stream name and stream number interchangeably.

Each version, whether it is a real version or a virtual version, is associated with a stream. Thus, every version consists of two parts: a stream and a version. There are always just these two pieces. In AccuRev/CM, creating a "branch of a branch" translates to making a new stream which is based upon or backed by another stream. This creates a stream hierarchy which is well known to AccuRev/CM. Therefore, you only ever need two pieces of information to refer to a particular version: its stream and its version number in that stream. Together, these constitute a version-ID.

The convention is to write the version-ID as **<stream>/<version #>**. For instance, the 3rd version in stream 5 would be 5/3. Real version-IDs are written in parentheses: (5/3) to distinguish them from virtual version-IDs.

Virtual Versions

Virtual versions are aliases for real versions. Every virtual version has a single real version associated with it. To show the association between a virtual version and a real version, the virtual

version is written first followed by the real version. For instance, if virtual version 5/3 is associated with real version (4/2), you would write 5/3 (4/2).

Every time a real version is created, a virtual version with the same stream and version number is also created. Creating real version (4/2) also creates a virtual version 4/2 automatically.

Any number of virtual versions can be created, all of which correspond to a real version. More than one virtual version can correspond to the same real version. You might have versions 7/1 (4/2), 6/8 (6/8), 5/3 (4/2), and 4/2 (4/2). Thus, there can be more virtual versions than real versions, but never more real versions than virtual versions.

Benefits of Virtual Versions

Virtual versions allow you to do everything you can do with branches and labels without the problems associated with branches and labels. Consider the problem of keeping track of where a label has been. This is impossible to do unless you create another label that points to the original location.

Pretty soon you will have more labels than you know what to do with and no systematic way that is built into the CM system to utilize them. AccuRev/CM uses virtual versions to handle the jobs that labels do in other systems.

If you want to apply the label **RLS4.02** to a set of versions, you can create a new stream named **RLS4.02**. Every time you want to apply the label to a real version, you simply promote the real version to the label stream. This creates a virtual version in the **RLS4.02** stream that points to the real version. If you subsequently decide that a different version of some file should have the **RLS4.02** label, you promote that version. Each time you "move the label" by promoting another version, you create a new virtual version in the stream. The sequence of virtual versions for a given element in the **RLS4.02** stream provide a record of "the moving label". In other CM systems, it's difficult or impossible to capture this kind of historical detail.

Permanent Record

One of the central ideas of AccuRev/CM is that you can never change the past. You can change things by renaming them or removing them, but these operations can only affect the present and future. The name, status, and content of things in the past remain the same.

Accordingly, virtual versions can never be changed or destroyed once they are created. A reference to a particular version today will be the same tomorrow and forever. You never need to worry that using an old reference will produce something different than when it was originally created.

Ancestry Tracking and the Version Browser

AccuRev/CM maintains complete ancestry information for each element, keeping track of how each version of the element was created. There are four possibilities:

ancestor

Modifying an existing version, then keeping the results to create a new real version.

alias

Promoting an existing version, to create a new virtual version.

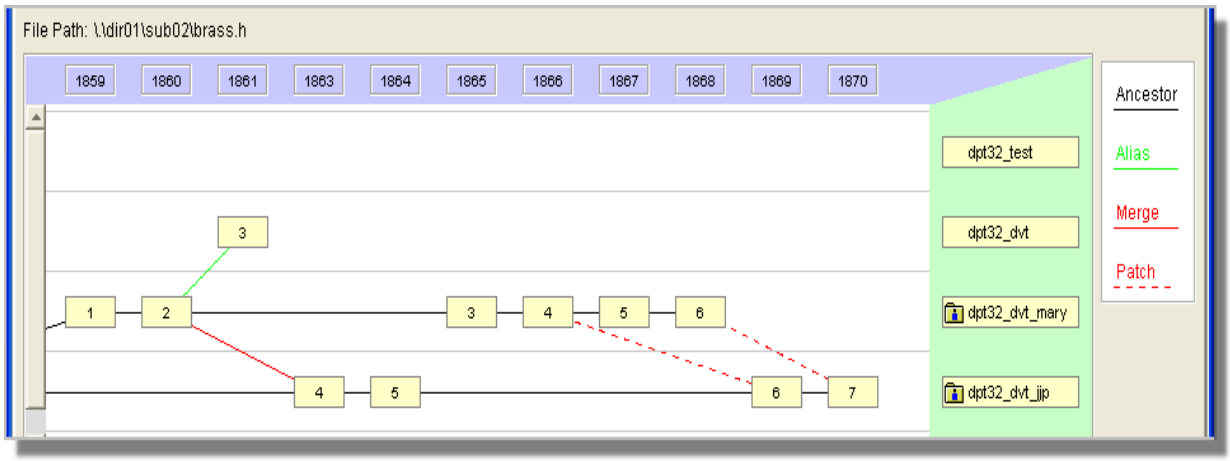
merge

Merging two versions, then keeping the results to create a new real version.

patch

Incorporating a subset of the changes made in one version into another version, then keeping the results to create a new real version.

The **Version Browser** can display some or all of an element's versions, using color-coded lines to indicate the way in which each version was created.

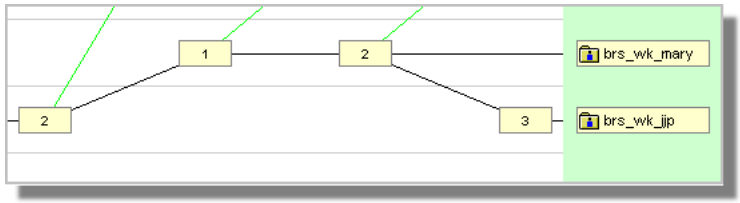


The following sections discuss the four kinds of ancestry in more detail, along with the important concept of closest common ancestor.

Ancestor — Modification of an Existing Version

Probably the most common operation in AccuRev/CM is starting with an existing version (created by you or by someone else), making changes, and then **keep**ing the changes. This creates a new real version, whose direct ancestor is the real version you started with. In the Version Browser, a black line connects the two real versions.

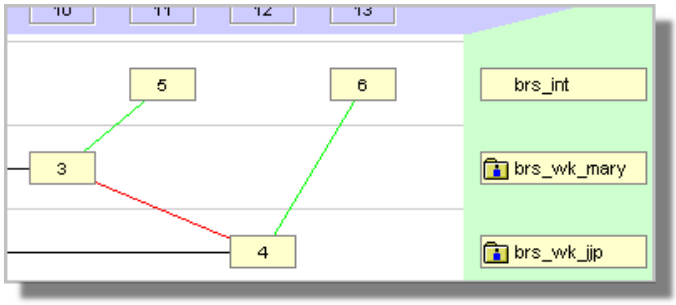
In the figure at right:



- Version 2 in the **brs_wk_jjp** stream was edited to create version 1 in the **brs_wk_mary** stream.
- Version 1 in the **brs_wk_mary** stream was edited to create version 2 in the same stream.
- Version 2 in the **brs_wk_mary** stream was edited to create version 3 in the **brs_wk_jjp** stream.

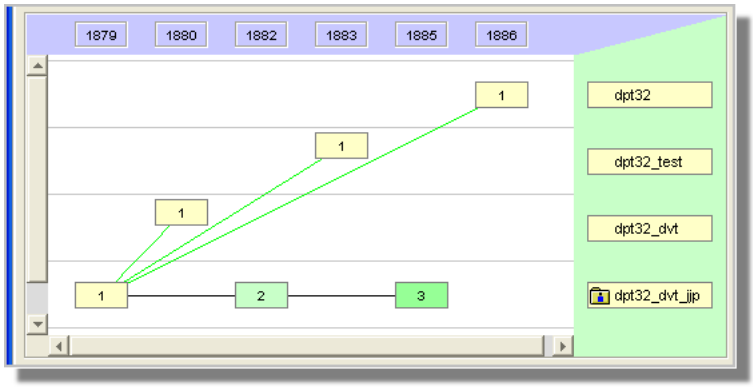
Alias — Virtual Version Ancestry

Virtual versions are created principally with the **promote** command. (A few other commands, such as **co** and **mv**, also create virtual versions.) Each virtual version is an alias for — that is, another name for — some real version. In the Version Browser, a green line connects a virtual version to the corresponding real version.



In the figure above, version 5 in the **brs_int** stream is an alias for (was promoted from) version 3 in the **brs_wk_mary** stream. Similarly, version 6 in the **brs_int** stream is an alias for version 4 in the **brs_wk_jjp** stream.

In a depot with a deep stream hierarchy, it's common to successively promote a particular version to the parent stream, then to the grandparent stream, then to the great-grandparent stream, etc. All the versions created by this series of **promote**s are aliases for the same real version. The Version Browser shows how all the virtual versions relate back to the original real version. In the figure at right, the versions in streams **dpt32_dvt**, **dpt32_test**, and **dpt32** are all aliases for the real version in workspace stream **dpt32_dvt_jjp**. (The display does not indicate the fact that the version was promoted from **dpt32_dvt** to **dpt32_test**, and from **dpt32_test** to **dpt32**.)



Merge — Merging of Two Versions

A standard **merge** operation combines the contents of these two versions of a file:

- The most recently kept version in your workspace stream. This version contains the changes that you have made to the file in your workspace.
- The most recent version in the backing stream.

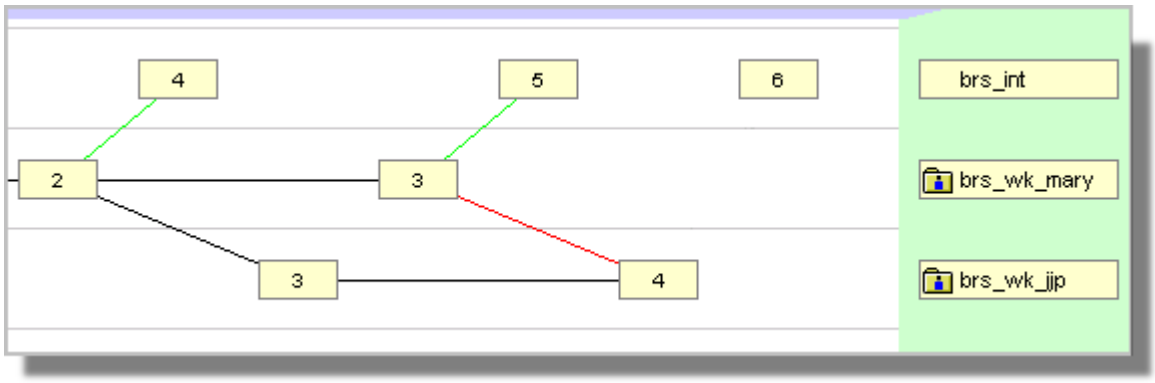
The result file of the merge operation is kept as a new version in the workspace stream. (You can think of merging as a fancy text-editing operation; as with any edit to a file, you preserve the results with **keep**.) This new, merged version has two ancestors: the two versions listed above.

This is all simple enough. There's a twist, though, which shows up in the Version Browser display: AccuRev/CM always records real versions, not virtual versions, as the two ancestors of a new, merged version. Thus, the ancestors in the standard merge scenario described above are:

- The most recently kept version in your workspace stream.
- The version in some other workspace stream that was promoted to the backing stream, causing the overlap that necessitated the merge.

The screen shot below shows a merge from the backing stream **brs_int** to the workspace stream **brs_wk_jjp**. The new, merged version is **brs_wk_jjp/4**. Its ancestors are:

- Real version **brs_wk_jjp/3**.
- Real version **brs_wk_mary/3**, which was promoted to become virtual version **brs_int/5** in the backing stream.



A solid red line shows the merging of data from one stream, **brs_wk_mary** to a different stream, **brs_wk_jjp**. The black line ("direct ancestor") between versions 3 and 4 in the **brs_wk_jjp** stream reflects the viewpoint that merging is just a fancy text-editing operation, automating the creation of the next version of a file.

Closest Common Ancestor

It's instructive to follow all the black and solid-red lines in an element's Version Browser display. This traces the entire ancestry of real versions of an element. In particular, you can use the real-version ancestry to determine the closest common ancestor of any two versions. This is the most recent version upon which the two versions are both based, by some combination of ancestor and merge connections.

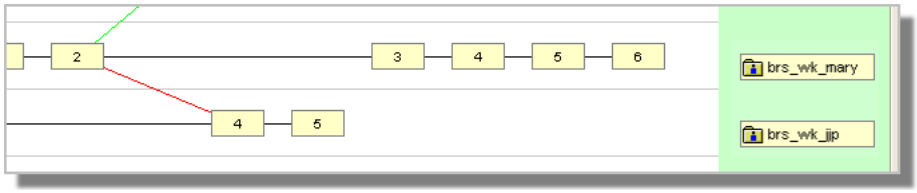
(When considering a virtual version in a closest-common-ancestor analysis, first follow the green line back to the corresponding real version.)

The **merge** command determines the closest common ancestor of the two versions to be merged, and uses this version to perform a 3-way merge. This merge algorithm evaluates each difference between the two versions as a *change* — in one or both versions — from the closest common ancestor.

Patch — Selective Merging of Two Versions

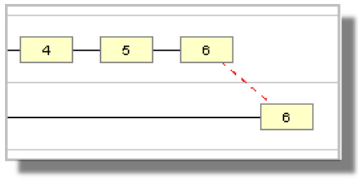
A patch operation is similar to a merge operation. In both, text from another version (the "from" version) is incorporated into your workspace's version. Here's the difference:

- A merge operation considers the entire contents of the "from" version.
- A patch operation considers only the parts of the "from" version that are changes from its immediate ancestor version

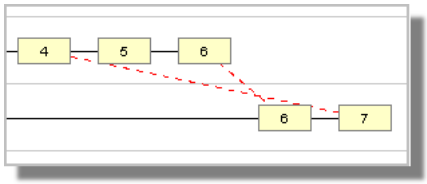


An example will clarify the distinction. Keep in mind that the merge algorithm considers differences to be changes from the closest common ancestor. Suppose user **jjp** wants to create a new version of a particular file in his workspace stream, **brs_wk_jjp**, by incorporating text from version 6 in stream **brs_wk_mary**.

Performing a merge would consider the entire contents of version **brs_wk_mary/6**. That is, it would consider the entire series of changes **mary** has made since version **brs_wk_mary/2**: the changes in all the versions 3, 4, 5, and 6.



By contrast, performing a patch from version **brs_wk_mary/6** incorporates only the changes made in that single version. It ignores the changes made in versions 3, 4, and 5. The Version Browser uses a dashed red line to indicate a patch. (Recall that a solid red line indicates a merge.)



How can **jjp** incorporate the rest of **mary**'s changes? He can simply perform a merge, part of which harmlessly duplicates the patch operation. Or he can perform additional patch operations, incorporating **mary**'s change in any order. Here's what the Version Browser would display after patching from version **brs_wk_mary/4**.

> Note: AccuRev/CM tracks patch ancestry separately from merge ancestry. In determining the closest common ancestor of two versions for a merge operation, AccuRev/CM takes into account previous merge operations (solid red), but not previous patch operations (dashed red). The patchlist

System Clock Synchronization

Time plays a fundamentally important role in AccuRev/CM's architecture and in its day-to-day operations. Some examples: each transaction is logged in as depot database at a particular time; a snapshot reconstructs the state of a stream at an arbitrary time; the **stat** command and the AccuRev/CM GUI use timestamps to optimize the lookup of modified files within a workspace.

AccuRev/CM is a networked product: programs execute on one server machine and (typically) multiple client machines. In a perfect world, the system clocks on all these machines would always be perfectly synchronized. This would ensure that data items on the server machine (say, versions created by **keep** commands) and corresponding data items on a client machine (the files that were kept) have timestamps that are consistent with each other.

Software systems do exist that keep all the machines in a network synchronized to within milliseconds. If your organization has deployed such a system, then you don't need to read any further in this chapter!

Most software development organizations don't have — and don't need — synchronization at the millisecond level. AccuRev/CM defines a 5-second tolerance as "good enough for software configuration management". This chapter describes AccuRev/CM's own facilities for detecting and fixing system-clock discrepancies, along with other facilities commonly available on Windows and Unix systems.

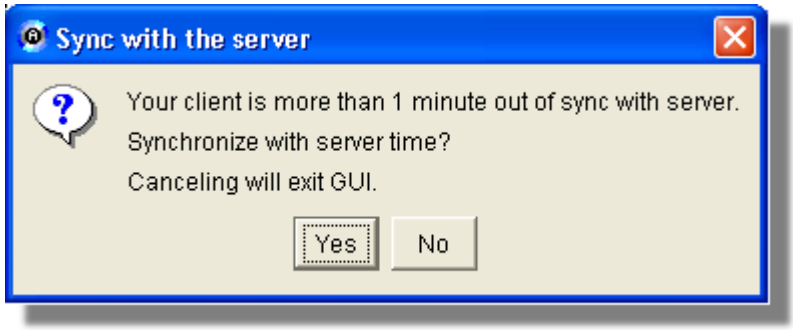
Detecting System Clock Discrepancies — Timewarp

A timewarp (clock skew) occurs when the discrepancy between a client machine's system clock and the server machine's system clock exceeds the allowable tolerance — currently, hard-coded to be 5 seconds. Timewarp problems typically occur during initial system setup and during time zone adjustments. For example, the change from Eastern Standard Time to Eastern Daylight Time can cause a timewarp on a machine that is not configured correctly to handle the time zone adjustment.

For most AccuRev/CM operations, a timewarp check is performed when the client contacts the server. CLI commands report large timewarps like this:

```
client_time:    2001/10/23 20:00:03 Eastern Daylight Time (1003881603)
server_time:    2001/10/23 21:17:02 Eastern Daylight Time (1003886222)
timewarp:       4619 seconds
The time on this machine is more than 5 seconds
different than the time on the server.
Please fix this and try again.
You may have a problem with your system clock.
You can force the time on your system to match
the server time using the accurev synctime command.
AccuRev Error: 13
```

The GUI displays a dialog that reports the timewarp condition and offers to fix it:



Fixing System Clock Discrepancies

The sections below describe various schemes for dealing with discrepancies between system clocks in the AccuRev/CM client-server environment. We begin with the most desirable scheme: automatic, smooth clock adjustment. We end with the least desirable scheme: manual, sudden clock adjustment.

Automatic, Gradual Convergence of System Clocks

An optimal scheme for synchronizing machines' system clocks has these attributes:

- All machines in the network participate in the scheme, so the entire network is kept synchronized.
- Each machine's system clock is adjusted automatically (perhaps requiring some initial installation or configuration task).
- System clock adjustments can be made "smoothly": for example, a discrepancy of 10 seconds can be gradually eliminated over the span of a few minutes by a minor speed-up or slow-down of a machine's clock. Presumably, such adjustments are imperceptible to human users and won't cause any "surprises" in time-sensitive applications.

Synchronization systems fitting this "gradual convergence" description are typically based on the standard Network Time Protocol (NTP) or its variant, the Simple Network Time Protocol (SNTP). One example, available on recent versions of Windows, is the Windows Time service (http://www.microsoft.com/WINDOWS2000/techinfo/howitworks/security/wintimeserv.asp). This provides a complete solution if all machines in your network are running Windows.

For a more general, multi-platform solution, see http://www.ntp.org. AccuRev has gotten good results from one particular SNTP client, **Automachron** (http://www.oneguycoding.com).

AccuRev/CM-Related Guidelines

Here are guidelines for using a "smooth convergence" system in an AccuRev/CM network:

- Configure the system so that a single machine in the network acts as the "time source" that other machines synchronize with.
- Ideally, have all AccuRev/CM machines participate in the synchronization system.

- If this isn't possible, make sure that the AccuRev/CM server machine participates in the synchronization system. (AccuRev/CM itself will take care of synchronizing its client machines to the server machine; see the next section.)

The purpose of these guidelines is to ensure that no AccuRev/CM client machine gets into a situation of synchronizing itself with two different, and possibly conflicting, machines: the AccuRev/CM server machine and the (S)NTP "time source" machine.

AccuRev/CM's Builtin Synchronization Scheme

The system clocks on all the machines running AccuRev/CM client or server software are automatically synchronized by AccuRev/CM itself. There is no way to disable or change the configuration of this synchronization system. And there is no way to include non-AccuRev/CM machines in this system.

In contrast with the "smooth" clock-adjustment scheme used by sophisticated (S)NTP-based systems, AccuRev/CM uses a simpler "sudden adjustment" scheme. For example, a 10-second discrepancy is eliminated all at once, not gradually. Moreover, adjustments are not made on a regularly scheduled basis, but only when an AccuRev/CM client program contacts the server program.

For many networks, AccuRev/CM's simpler scheme is altogether satisfactory. It has the advantage of never allowing a transaction to be completed when the system clocks of the client and server differ by more than 5 seconds. (See *Synchronization Algorithm* below.) But some organizations may be running networked applications that don't react gracefully to the "surprise" of a machine's system clock suddenly changing by a significant number of seconds. Such organizations may not be satisfied with AccuRev/CM's simpler scheme.

Synchronization Algorithm

Each time a client program contacts the server program, AccuRev/CM compares the system clocks on the two machines:

- If the discrepancy is less than 5 seconds, no clock-related change occurs.
- If the discrepancy is between 5 and 60 seconds, AccuRev/CM automatically changes the client machine's system clock to match the server machine's. (On a Unix client machine, this change occurs only if the client program is running as the **root** user; this is not advisable in most situations.)
- If the discrepancy exceeds 60 seconds:
  - A CLI client exits immediately, without performing the user's command.
  - A GUI client offers to change the client machine's system clock before executing the user's command. If the user declines this opportunity to synchronize, the GUI client exits immediately. (This is preferable to recording a transaction in a situation with a substantial timewarp.)

Manual Synchronization Tools

The least desirable scheme for keeping system clocks synchronized is to occasionally type clock-adjustment commands manually on one or more of the machines. This method can be improved a bit by using scripts and scheduling tools such as **cron** (Unix) and **at** (Windows).

Only the **root** user (Unix) or a user with administrator privileges (Windows) can set the system clock manually.

Setting the System Clock on the AccuRev/CM Server Machine

On a Unix machine, the **date** command changes the system clock. What time should you set the clock to? In many cases, you can use **rsh** or **telnet** to determine the time on another "time source" machine.

On a Windows machine, use the **net time** command to synchronize with a specified "time source" machine, or with the domain controller machine. To set the clock to a particular time, use the **date** command in a Command Prompt window, or double-click the digital clock in the Windows task bar (lower-right corner of the screen).

Setting the System Clock on AccuRev/CM Client Machines

The **accurev synctime** command changes a client machine's system clock to match the clock on the server machine. The GUI command is **Tools > Synchronize Time**. These commands should not be necessary very often, given the scheme described in section *AccuRev/CM's Builtin Synchronization Scheme* above.