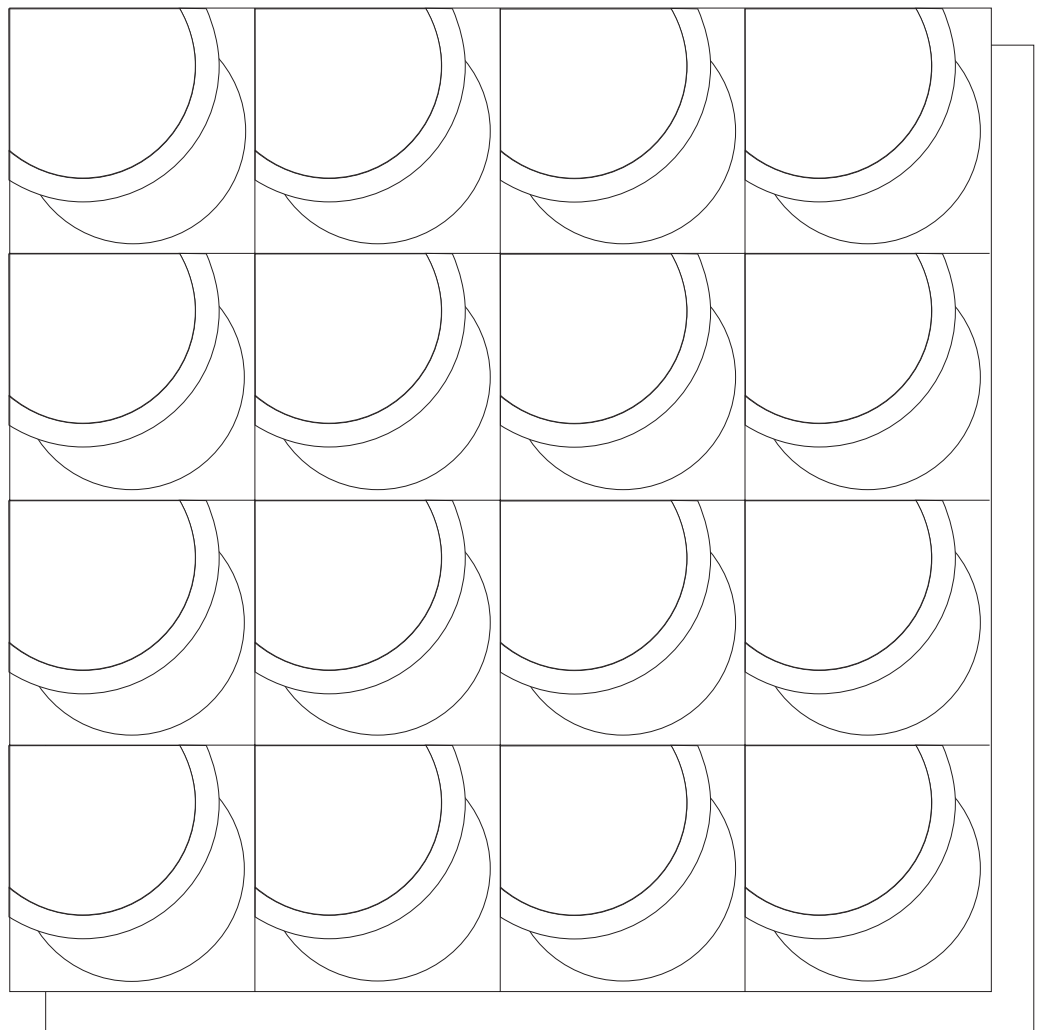


# IBM Configuration Management Version Control Concepts



**Note:** Before using this information and the product it supports, be sure to read the general information under "Notices" on page ix.

### **First Edition (June 1993)**

This edition applies to Version 2 Release1, Modification Level 0, of IBM Configuration Management Version Control/6000 (Program 5765–207), IBM Configuration Management Version Control for HP systems (Program 5765–202), IBM Configuration Management Version Control for Sun systems (Program 5622–063), and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory  
Information Development  
21/986/844/TOR  
844 Don Mills Road  
North York, Ontario, Canada. M3C 1V7

You can also send your comments by facsimile to (416) 448–6057 to the attention of the RCF Coordinator. If you have access to Internet, IBMLINK, or IBM/PROFS, or IBMMAIL, you can send your comments electronically to **torrcf@vnet.ibm.com**; IBMLINK, to **toribm(torrcf)**; IBM/PROFS, to **torolab4(torrcf)**; IBMMAIL, to **ibmmail(caibmwt9)**.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

If you choose to respond through Internet, please include either your entire Internet network address, or a postal address.

© **Copyright International Business Machines Corporation 1993. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

IBM is a registered trademark of International Business Machines Corporation, Armonk, N.Y.

---

# Contents

<b>Figure List</b> .....	<b>vii</b>
<b>Notices</b> .....	<b>ix</b>
Trademarks and Service Marks .....	ix
<b>About This Book</b> .....	<b>xi</b>
Who Should Read This Book .....	xi
What You Need to Know .....	xi
How to Use This Book .....	xi
Highlighting Style .....	xii
CMVC Publications .....	xii
<b>Chapter 1. The CMVC Environment</b> .....	<b>1</b>
System Configuration .....	1
Required Software .....	2
User Interfaces .....	2
CMVC Roles .....	2
<b>Chapter 2. An Introduction to CMVC</b> .....	<b>5</b>
Organizing Development Data .....	5
Configuring CMVC Processes .....	7
Reported Problems and Design Changes .....	8
Evaluating Proposed Design Changes and Reported Problems .....	8
Identifying the Work Required .....	8
Tracking Features and Defects .....	8
Tracking the Change Process .....	9
Integrating the Changes .....	9
Updating the Release .....	9
Verifying Defects and Features .....	10
Summary .....	10

## The CMVC Development Environment

<b>Chapter 3. Using CMVC Components</b> .....	<b>13</b>
What Is a Component? .....	13
Component Attributes .....	13
The Component Hierarchy .....	13
Using the Component Hierarchy to Manage Projects .....	14
Creating Components With More than One Parent .....	15
Component Ownership .....	16
Using Components to Manage Data .....	16
Controlling Access Authority .....	16
Defining and Modifying Authority Groups .....	18
Granting Access Authority .....	18

Inheritance .....	19
Controlling Notification .....	20
Defining and Modifying Interest Groups .....	21
Component Processes .....	21
Component-File Relationship .....	22
<b>Chapter 4. Using CMVC Releases .....</b>	<b>23</b>
What Is a Release? .....	23
Release Management .....	23
Creating a Release .....	24
Using Releases to Organize Files .....	25
The Release-File Relationship .....	25
Change Control and Integrated Problem Tracking .....	26
Configuring the Integrated Problem Tracking Subprocesses .....	26
Extracting a Release .....	27

## Change Control and Problem Tracking

<b>Chapter 5. Controlling File Changes .....</b>	<b>31</b>
What Is a CMVC File? .....	31
File Attributes .....	31
Versioning of Files .....	32
Getting Files from CMVC .....	32
Checking Out a File .....	32
Extracting a File .....	33
Checking in CMVC Files .....	34
Files Shared Between Releases .....	34
Common Files .....	35
Breaking the Common Link .....	36
Managing Access to Shared Files .....	37
Files in Releases with Integrated Problem Tracking .....	38
Undoing File Changes .....	39
<b>Chapter 6. Using Defects and Features .....</b>	<b>41</b>
What Are Defects and Features? .....	41
Defect and Feature Attributes .....	41
Opening Defects and Features .....	43
Analyzing Defects and Features .....	43
Designing the Resolution .....	43
Identifying the Required Resources .....	43
Reviewing the Design and Resource Estimates .....	45
Resolving Defects and Implementing Features .....	45
Verifying the Resolution of the Defect or Feature .....	45
Responsibilities of the Originator .....	46
Responsibilities of the Owner .....	46
Changing Component Processes .....	46

<b>Chapter 7. Using Tracks</b> .....	<b>47</b>
What is a Track? .....	47
Track Attributes .....	47
Configuring Your Change Control Process .....	48
Working With Tracks .....	48
The Approval Subprocess .....	49
The Fix Subprocess .....	49
Completing the Tracking Process .....	51
The Test Subprocess .....	51
After the Track Subprocess .....	52
Responsibilities of a Track Owner .....	52
The Track States .....	52
<b>Chapter 8 . Using Levels</b> .....	<b>55</b>
What is a Level? .....	55
Level Attributes .....	55
The Level States .....	55
The Level Subprocess .....	55
Creating New Levels and Adding Tracks As Level Members .....	56
Prerequisite and Corequisite Checks .....	56
Making Changes to Files Included In a Level .....	57
Committing and Completing a Level .....	58
Extracting a Level .....	58
Extracting File Trees .....	58
Combining File Trees .....	59
Compiling a File Tree .....	59
Updating a Release with the First Level .....	60
Changing Release Processes .....	60
<b>Appendix A. The States of CMVC Objects</b> .....	<b>61</b>
The States of Features and Defects .....	61
The States of a Track .....	63
The States of a Level .....	65
The Feature and Defect State Diagram .....	67
The Track and Level State Diagram .....	69
The Relationship Between Subprocesses and Track States .....	71
The CMVC State Diagram .....	73
<b>Appendix B. CMVC Entity Relationships</b> .....	<b>75</b>
<b>Glossary</b> .....	<b>77</b>
<b>Index</b> .....	<b>83</b>



---

## Figure List

Figure 1.	Example of a client-server network of CMVC .....	1
Figure 2.	Example component hierarchy .....	6
Figure 3.	Releases, files, and components .....	7
Figure 4.	Example of three levels of file changes committed within a release .....	10
Figure 5.	The component relationships .....	14
Figure 6.	Designing the initial component structure .....	14
Figure 7.	Adding to the initial component structure .....	15
Figure 8.	Components with more than one parent .....	16
Figure 9.	A user with the implicit authority to create a component. ....	17
Figure 10.	Grouping CMVC actions into authority groups. ....	18
Figure 11.	Granting access authority by using the access lists .....	19
Figure 12.	Managing access authority .....	20
Figure 13.	Grouping CMVC actions into interest groups. ....	21
Figure 14.	The release-component relationship .....	24
Figure 15.	Access and notification for releases .....	24
Figure 16.	File-component-release relationship .....	25
Figure 17.	A release grouping files from different components .....	26
Figure 18.	Checking files in and out of the CMVC server .....	33
Figure 19.	Two CMVC files .....	34
Figure 20.	Two releases sharing one file .....	35
Figure 21.	A common file between two releases .....	35
Figure 22.	Breaking the common link when checking in a file .....	36
Figure 23.	Two branches of a file .....	37
Figure 24.	Managing access to a shared file .....	38
Figure 25.	File changes in releases that include the track subprocess .....	39
Figure 26.	Feature and defect state diagram with all subprocesses configured .....	44
Figure 27.	Track and level state diagram with all subprocesses configured. . .	50
Figure 28.	CMVC State Diagram .....	53
Figure 29.	Example of prerequisite file changes in a level .....	57
Figure 30.	Delta and full file trees .....	59
Figure 31.	Feature and defect state diagram with all subprocesses configured .....	66
Figure 32.	Track and level state diagram with all subprocesses configured. . .	68
Figure 33.	Relationship between subprocesses and state transitions for tracks .....	70
Figure 34.	CMVC State Diagram .....	72
Figure 35.	Example entity relationship diagram. ....	75
Figure 36.	CMVC objects represented in an entity relationship diagram. ....	76





---

## Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577, U.S.A.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

---

## Trademarks and Service Marks

IBM, denoted by an asterisk (\*) in this publication, is a trademark of the IBM Corporation in the United States and other countries.

The following terms, denoted by a double asterisk (\*\*) in this publication, are trademarks of other companies as follows:

<b>SoftBench</b>	Hewlett-Packard Company
<b>INFORMIX, INFORMIX-SQL</b>	Informix Software, Inc.
<b>ORACLE</b>	Oracle Corporation
<b>OSF/Motif</b>	Open Software Foundation, Inc.
<b>PVCS Version Manager</b>	INTERSOLV, Inc.
<b>Sun, Network File System, NFS</b>	Sun Microsystems, Inc.
<b>SYBASE, SYBASE SQL</b>	Sybase, Inc.



---

## About This Book

This book is part of the documentation library that supports the IBM\* Configuration Management Version Control (CMVC) products. Read this book to gain an understanding of CMVC concepts and how CMVC can enhance your existing software development process.

---

## Who Should Read This Book

Anyone who wants to learn about CMVC should read this book. This includes administrators, planners, managers, project leaders, testers, application developers, and technical writers. It discusses the underlying concepts of CMVC and how these concepts relate to development processes. Examples of how CMVC can be used by a software development organization are included.

After reading this book refer to the *IBM CMVC Server Administration and Installation* and *IBM CMVC Client Installation and Configuration* for complete planning, installation, and administration details. The *IBM CMVC User's Guide* describes how to use CMVC with the graphical user interface (GUI), and the *IBM CMVC Commands Reference* describes CMVC commands as implemented for the command line interface.

---

## What You Need to Know

You should read this book before you read the rest of the CMVC library. To understand all concepts discussed within this book, you should be familiar with your operating system environment.

---

## How to Use This Book

The first part of this book consists of two introductory chapters.

- Chapter 1 describes the network environment, software that CMVC requires, and the different CMVC roles.
- Chapter 2 provides an overview of all parts of CMVC. Anyone who reads only parts of this book should read this chapter.

The second part of this book consists of two chapters explaining the CMVC Development Environment.

- Chapter 3 explains CMVC components, component processes, the component structure, user access, and notification of CMVC actions.
- Chapter 4 explains releases, release processes, and the relationship files have with components and releases.

The third part of this book consists of four chapters explaining change control and problem tracking.

- Chapter 5 discusses working with files, delta versioning of files, and files shared across multiple releases.
- Chapter 6 discusses working with defects and features.
- Chapter 7 explains the use of tracks in integrated problem tracking and change control.
- Chapter 8 explains the uses of levels.

---

## Highlighting Style

<b>Bold</b>	Files, directories, field names, and other items predefined by CMVC appear in bold.
<i>Italic</i>	Titles of books and the occurrences of new terms appear in italics.
Monotype	Names of authority and interest groups appear in monotype, as do examples of servers, components and releases.

---

## CMVC Publications

The following list of IBM books contain additional information on CMVC and related topics:

- *IBM CMVC Server Administration and Installation, SC09–1631*, contains detailed information needed to install, configure, customize, operate, and maintain your CMVC environment.
- *IBM CMVC Client Installation and Configuration, SC09–1596*, contains detailed information that you need to install, configure, and customize CMVC clients.
- *IBM CMVC User's Guide, SC09–1634*, describes all CMVC actions as implemented in the graphical user interface (GUI).
- *IBM CMVC User's Reference, SC09–1597*, contains the reference lists, tables, and state diagrams for CMVC, as well as a description of how the message–integrated CMVC client communicates with other integrated development environment tools.
- *IBM CMVC Commands Reference, SC09–1635*, describes all CMVC commands, their syntax, and use as implemented in the command line interface.

---

# Chapter 1. The CMVC Environment

This chapter briefly describes the Configuration Management Version Control (CMVC) system configuration, the user interfaces, and the three main user roles within a CMVC environment.

---

## System Configuration

CMVC consists of a CMVC client and a CMVC server. Designed for use in a networked environment, the CMVC products support several operating systems and are based on a client-server model.

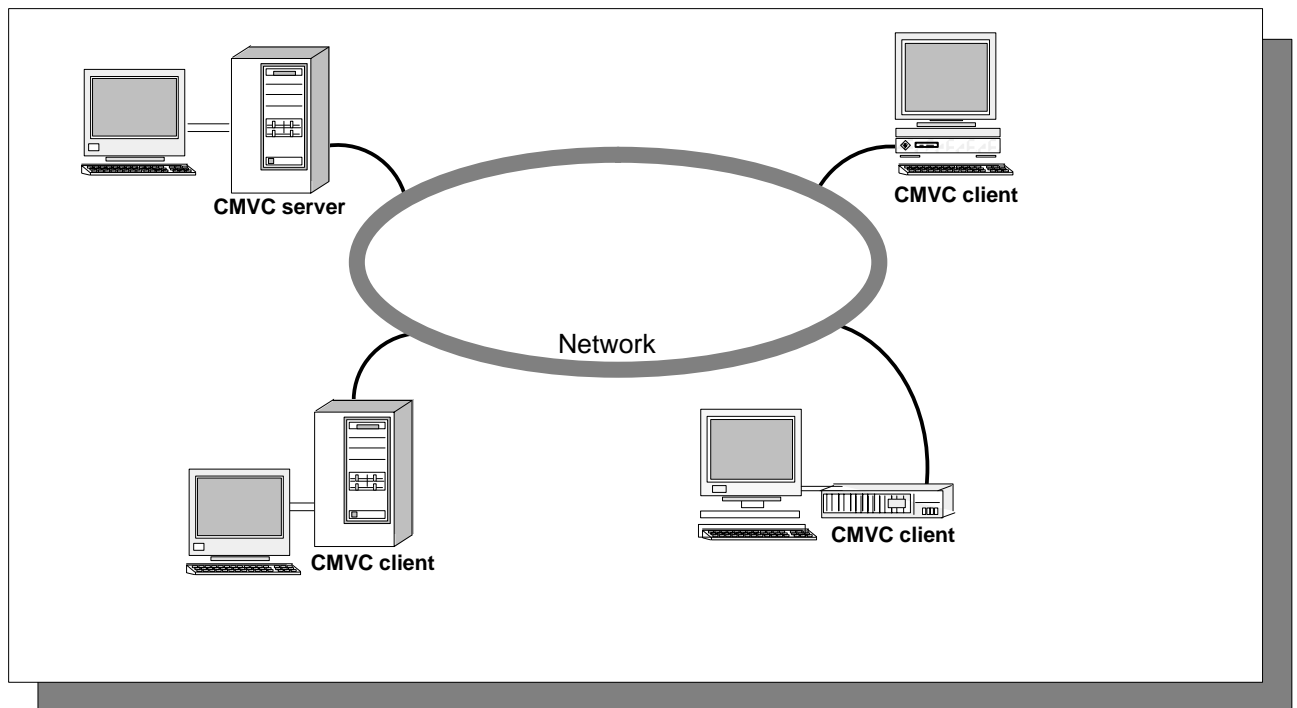


Figure 1. Example of a client-server network of CMVC

A CMVC server is a workstation that runs CMVC server software to control all data within the CMVC environment. Files are stored in a file system on the server by means of a version control system. All other development data is stored in a relational database on the CMVC server. A CMVC client is a workstation that runs the CMVC client software to access the information and files stored on a CMVC server. This client-server architecture allows users to access files and project data without having to know where the networked resources physically reside.

## Required Software

Two version control systems are available for use with your CMVC server:

- Source Code Control System (SCCS)
- INTERSOLV's Polytron Version Control System\*\* (PVCS Version Manager)

The databases available for use with CMVC servers include:

- ORACLE\*\* Relational Database Management System
- INFORMIX–SQL\*\*
- SYBASE SQL\*\*

The available databases and version control systems may be limited by the platform that the CMVC server is running on.

 For more detailed information, see *IBM CMVC Server Administration and Installation*.

## User Interfaces

The CMVC clients support the following graphical user interfaces (GUIs) and a command line interface:

- A message-integrated GUI that operates in two environments:
  - The IBM AIX Software Development Environment (SDE) WorkBench/6000 product.
  - The HP SoftBench\*\* and HP SoftBench for Sun\*\* products.

See the *IBM CMVC User's Guide* for information on these GUIs.

- A non-message-integrated GUI for operation in an environment without SDE WorkBench/6000 or HP SoftBench products. See the *IBM CMVC User's Guide* for more information about this GUI.
- A command line interface is provided for use within a shell environment. See *IBM CMVC Commands Reference* for more information about this interface.

---

## CMVC Roles

The tasks within the CMVC environment can be divided into three main categories:

- System administration
- Family administration
- End use.

For all documentation within the CMVC library, the roles are defined as follows:

### System Administrator

The system administrator is responsible for:

- Installing, maintaining, and backing up the CMVC server
- Installing, maintaining, and backing up the relational database used by CMVC
- Planning, maintaining, and configuring all client and server hardware.

The system administrator has root access to the CMVC server and database administration (dba) access to the relational database management system.

**Family Administrator**

The family administrator is responsible for:

- Planning and configuring CMVC for one or more families
- Managing user access to one or more CMVC families
- Maintaining one or more CMVC families.

The family administrator has root access to the CMVC server and dba access to the relational database management system.

**End User**

The end user uses one or more CMVC families. Most of the tasks described in this book are done by end users, such as project leaders, programmers, and technical writers.





---

## Chapter 2. An Introduction to CMVC

Software development organizations today face the challenges of planning, managing, and performing development activities. The development life cycle typically involves such tasks as planning, programming, testing, building, and documenting. These tasks all involve sharing development data. As the development process unfolds this data changes constantly. A dynamic development environment needs to systematically manage and control its work.

CMVC provides configuration management, version control, change control, and problem tracking in a distributed development environment to facilitate project-wide coordination of development activities across all phases of the product development life cycle.

Configuration management is the process of identifying, managing, and controlling software modules as they change over time. Version control is the storage of multiple versions of a single file along with information about each version.

Shared access to all development data is supported by storing all files and information on a central server and providing access control that can be configured for each component of data. CMVC provides two types of change control. The first type controls access to files and requires files to be locked while changes take place. The second type complements the first with a mechanism for tracking all file changes across multiple products and environments. You can track both problem correction and design implementation.

The integration of problem and design tracking with change control provides a systematic, configurable approach to tracking the file changes made to resolve a reported problem or to implement a proposed design. With CMVC, you can organize your development data for effective development tracking.

---

### Organizing Development Data

Data contained in CMVC is divided into one or more families. A CMVC *family* is a logical unit of related development data. The data in one family cannot be shared with other CMVC families.

#### A Component Hierarchy

Within each family, data is organized into groups called *components*. Components are arranged in a hierarchical structure with a single top component called *root* as shown in Figure 2. This hierarchy provides a mechanism for organizing components of data into structured groups. Your family administrator can configure the components and the hierarchical structure within each CMVC family. Some common ways to group data in a component are by function, by platform, for access control, or for communication needs. The component hierarchy reflects the organizational requirements of your development efforts and can be modified over time as these requirements change.

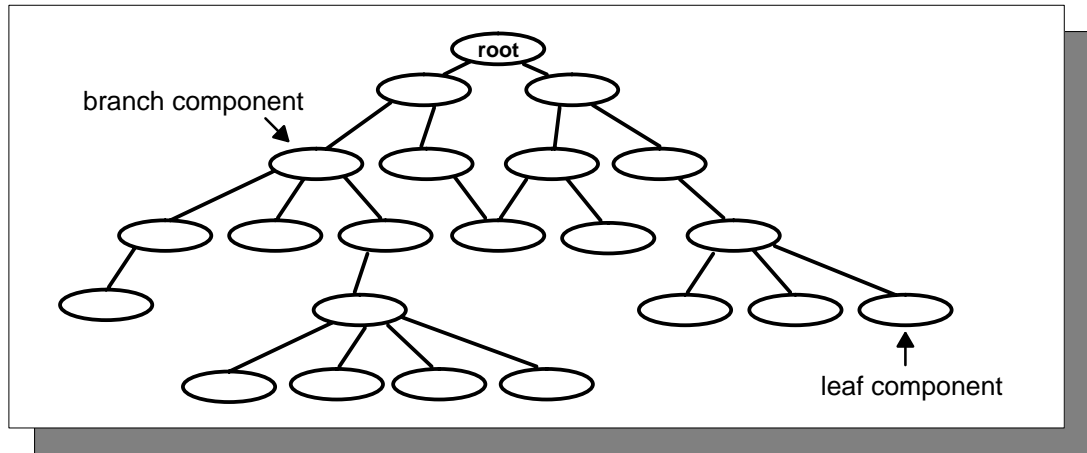


Figure 2. Example component hierarchy

Components are the building blocks of the CMVC environment. They organize data for information retrieval, access control, notification control, problem reporting, and data organization.

Each file under CMVC control is managed by a component. Components that manage a set of files are usually the leaf components of your hierarchy, while the branch components are used for organization, access control, and problem reporting.

### Component Ownership

Ownership of each component is assigned to a user. That user is responsible for managing data related to that component, including any problems reported to the component and any files managed by it.

In addition to defining ownership of data, the component hierarchy is a structure defining access and communication control as appropriate for specific groups of users and specific groups of data.

### A Release of a Product

CMVC files are also grouped into categories called *releases*. All files that make up a single version of a product are grouped in a release. Releases are defined separately from components to ease the maintenance of multiple versions of a product. Often two different versions of one file are used in two versions of a product; each release of that product can provide a link to a different version of the file. One release can group files that are managed by many components. An example of three releases grouping files is shown in Figure 3.

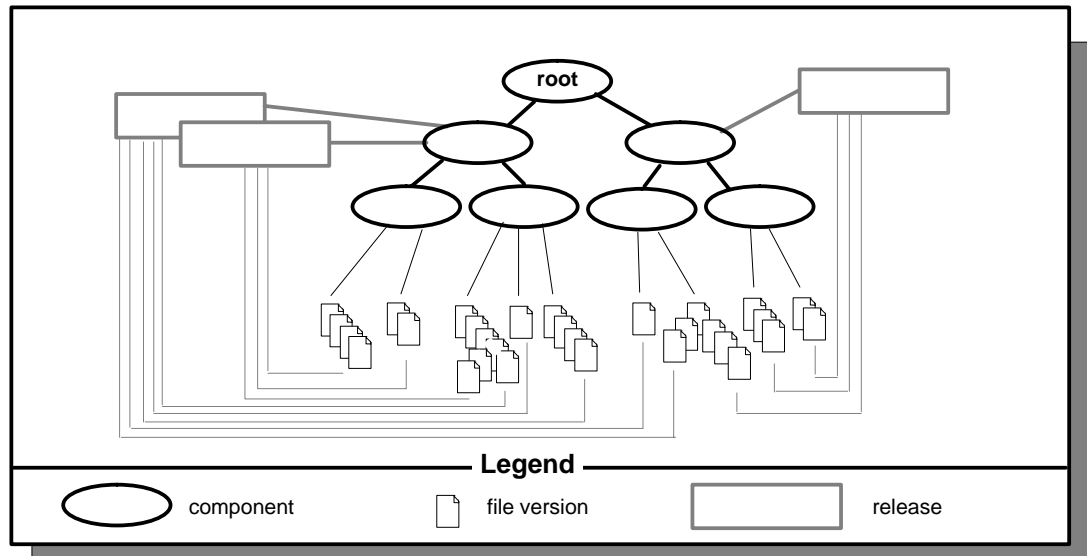


Figure 3. Releases, files, and components

Components organize files and other development data for management purposes. Releases organize files for product related activities. Each file must be managed by at least one component and contained in at least one release.

For example, one release contains many files. A group of users working on the release only needs access to a subset of the files contained in the release. A single component can be created to manage this set of files. Access to the files for this group of users is defined through that component.

Each time a development cycle begins for the next version of a product, you can define a separate release. Each subsequent release of a product will reference many of the same files as its predecessor; however, each release links to particular versions of individual files so that maintenance of an older release can progress at the same time as development of a newer release.

## Configuring CMVC Processes

CMVC monitors changes with *defects*, *features*, and *integrated problem tracking*. Each of these restricts file changes so that they are made in a systematic manner. CMVC can require users to analyze the time and resources required to make changes, verify changes, select files to be changed, approve work to be done, and test the changes. The requirements for changes are controlled by *processes*. Family administrators can create processes for components and releases to use, configuring them from CMVC *subprocesses*.

The subprocesses affecting defects and features are configured in the process defined by the component that is associated with the defect or feature. Each release must also use a process. The release process configures the subprocesses that affect integrated problem tracking.

☞ For information about how to configure processes, refer to *IBM CMVC Server Administration and Installation*.

☞ Information about specific subprocesses can be found in Chapter 6, “Using Defects and Features”, Chapter 7, “Using Tracks”, and Chapter 8, “Using Levels”. The relationship between components and processes is described in detail in Chapter 3, “Using CMVC

Components”. The relationship between releases and processes is described in detail in Chapter 4, “Using CMVC Releases”.

---

## Reported Problems and Design Changes

CMVC regulates reported problems and design changes and retains information about the life cycle of each within the database on the CMVC server. A CMVC defect records each reported problem. A CMVC feature records each proposed design change.

Each defect and feature is reported to a specific component within the component hierarchy. The person who reported the defect or feature is known as the originator. Initially, a defect or feature is owned by the owner of the component that it was opened against. The defect or feature owner is responsible for evaluating the suggested changes. If necessary, the problem can be reassigned to new owner or a more appropriate component.

## Evaluating Proposed Design Changes and Reported Problems

When a feature is proposed or a defect is reported, its owner must assess the change and can then return it to the originator, reassign it to another user or component, or accept it for Design, Size and Review (the DSR subprocess). The DSR subprocess involves three stages: design, size, and review. In the design stage, plan the implementation of the feature or the resolution of the defect. In the size stage, identify the resources that will be required. In the review stage, review the required resources and the feasibility of the planned implementation or resolution. Size and review may indicate a need for additional design work. At any of these stages, you can return the feature or defect.

A feature will not use the DSR subprocess if the component that it is assigned to uses a process that does not include the feature DSR. Similarly, a defect will not use the DSR subprocess if the defect DSR subprocess is not included. DSR records cannot be filled in if the DSR subprocess is not being used. Note that assessment and choosing between returning, reassigning, or accepting a feature or defect are always required.

Once the evaluation stages are complete you can accept the feature or defect for implementation.

## Identifying the Work Required

Defects and features record information about any problem or suggested design change. Reported proposals and problems do not need to be related to the files under CMVC control. The uses of defects include recording information about process problems, hardware problems, and plan production problems. You can use features to record proposals for process improvements, hardware enhancements, and plan changes.

---

## Tracking Features and Defects

Releases using a process that includes the track subprocess record information about progress of changes to a single feature or defect in that release. The level, approval, test, and fix subprocesses can only be configured if tracking is also configured as part of the change control process.

One defect may require changes in more than one release and one feature may be implemented in more than one release. You must identify these releases before you make the file changes needed to resolve the defect or implement the feature. When you identify the releases a separate tracking mechanism is created for each release. This tracking mechanism is called a CMVC *track*. Each track will monitor the resolution of one defect or feature in one release.

Identify affected releases during the size stage of the DSR subprocess. The tracks for these releases are created automatically when the feature or defect is accepted. Additional tracks can be created manually if they are needed.

You must manually create all tracks required for defects and features if the DSR subprocess does not apply to the managing components.

## Tracking the Change Process

A track provides a mechanism to control file changes and to incorporate those changes into each affected release. The track moves through successive states that both control and indicate the type of work being done.

Files must be checked out from CMVC before they can be edited. After checking out the file, make the required changes, and then check the changed file back in to CMVC. You must reference a specific track when you check files in. The track links the file changes to the defect or feature and the release that they affect.


Fixing a defect or implementing a feature may involve changing multiple files. A single track monitors all the files changed within one release for one specific defect or feature.

Your development team should review and test all file changes before you move a track to the next state for integration within the release.

## Integrating the Changes

Resolving defects and implementing features for a specific release involves integrating the files changed for those defects and features with each other and with the unchanged files in the release. This can be done in one of two ways. If the level subprocess is included in the release's process, then CMVC levels can be used to integrate defect and feature file changes with each other and unchanged files in the release. In this case, define a CMVC *level* for the release for which you wish to integrate file changes. Then add the tracks that monitor the file changes that you want to integrate within the release to the level as *level members*. In this way you include all the file changes made to resolve the selected set of problems in the level.

If the level subprocess is not included in the release's process, then files changed for the defect or feature can be integrated with the other files in the release by specifically integrating or committing the track.

 Information about levels can be found in Chapter 8, "Using Levels".

## Updating the Release

Once a level of changed files is extracted, compiled and verified with the unchanged files in a release, update the release by committing the level. Committing a level commits all tracks that were designated as level members and all files changed in reference to those tracks as stable. By committing the changes in a level, you establish a new baseline for subsequent development of the release.

Once a release has been updated by completing a level, it is ready for formal testing. This testing could involve testing the release on different platforms or in different environments. When formal testing of each release is complete, the tracks monitoring the changes are also complete.

You can recreate any committed or completed level at a later date. CMVC levels provide snapshots of the release at different points in the development life cycle. In Figure 4, you can recreate Level 1 or 2 at any time even though development is progressing based on the committed file changes defined in Level 3.



Figure 4. Example of three levels of file changes committed within a release

## Verifying Defects and Features

If the verify subprocess is included in the component subprocess, then once a track is completed the feature or defect for which the track was created must be verified by its originator. If you originated the defect or feature, indicate concurrence or non-concurrence with its resolution. If you do not concur with the resolution, open a new defect or feature. The original problem will close automatically when two conditions are met: 1) all verification record owners have indicated either concurrence or non-concurrence, and 2) all tracks created for the feature or defect are complete.

If no tracks exist for a defect or feature, then it is verified by the originator when the owner marks the it ready for verification.

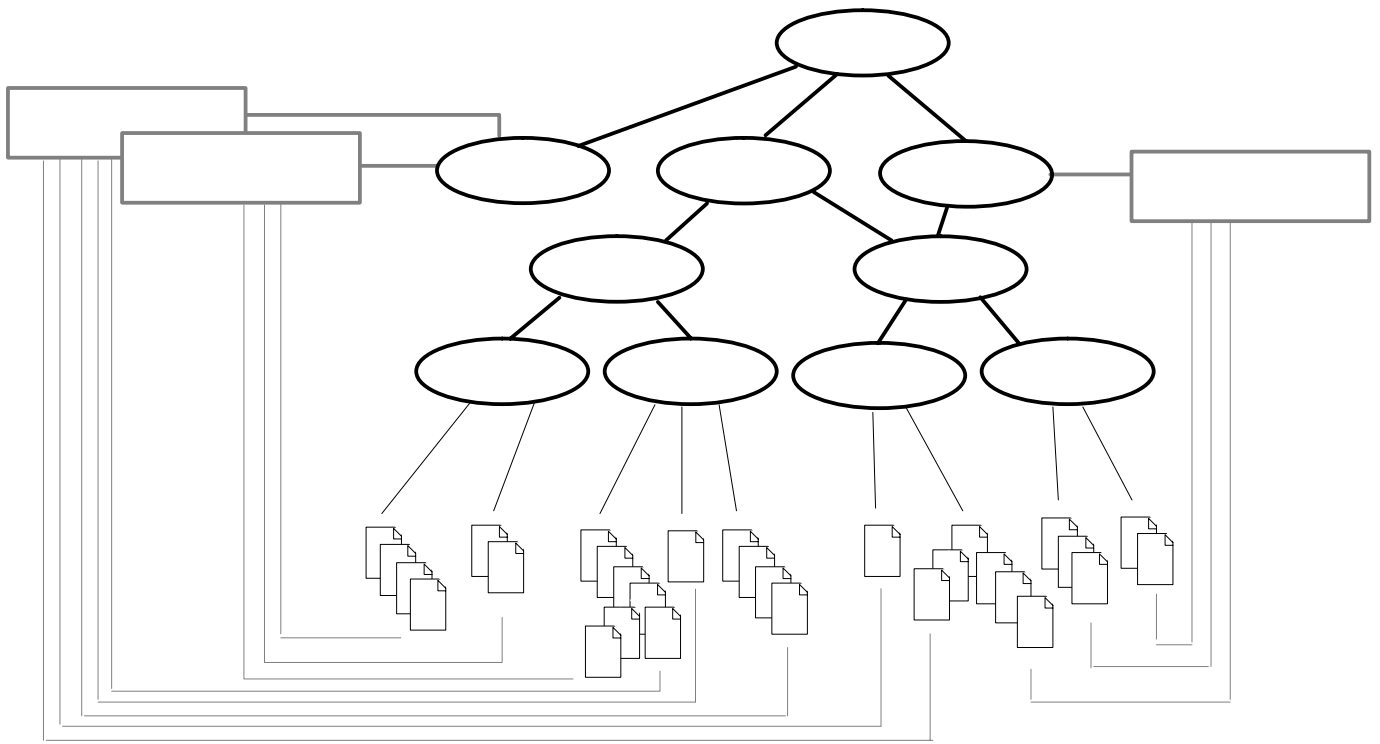
---

## Summary

A CMVC *family* is a logical organization of related development data. Within each family, data is organized into groups called *components*. Components are arranged in a hierarchical structure. Components are the focal point for information retrieval, access control, notification control, problem reporting, and data organization. Within CMVC, files are also grouped into categories called *releases*. All files that make up a single version of a product are grouped in one release. Each file must be managed by at least one component and contained in at least one release.

A CMVC *defect* records each reported problem. A CMVC *feature* records each proposed design change. You must report each defect and feature to a specific component within the component hierarchy. Defects and features record information about any problem or suggested design change that you need to monitor. A CMVC *track* monitors the resolution of one defect or feature for all affected files in one release. A CMVC *level* monitors the integration of a selected set of defects and/or features for a specific release. The tracks that monitored the defects or features that you want to integrate within the release are then added to the level as *level members*. Committing a level commits all tracks that were designated as level members and all files changed in reference to those tracks as stable. When the changes in a level are committed, they establish a new baseline for subsequent development of the release. Levels provide snapshots of the release at different points in the release development life cycle. Once a track is completed, the originator of the defect or feature for which the track was created can verify that the resolution or implementation was correct.

# The CMVC Development Environment



Within each CMVC family, data is organized into components and releases. The component structure is the basis for information retrieval, access control, problem reporting, and data organization. Releases organize files for product-related activities such as compilation and building for distribution. The following chapters describe families, components, releases and their relationship with your development data and the other objects within your CMVC development environment.





---

## Chapter 3. Using CMVC Components

The CMVC component hierarchy organizes data into manageable groups and provides a mechanism for controlling user access and notification. This chapter explains what a component is and describes the relationships between components and other CMVC objects within a family.

---

### What Is a Component?

A component is the CMVC object that provides organization and control of development data. Components are the focal point for information retrieval, access control, notification control, problem reporting, and data organization.

### Component Attributes

CMVC components have the following attributes:

- Name
  - Each component name within a family must be unique.
- Owner
  - Every component has an owner. The component owner is primarily responsible for actions relating to that component.
- Parent component
  - Every component except the root component must have at least one parent.
- Process
  - The process that the component will use. This determines whether the DSR and verify subprocesses are used for defects and features.
- Description
  - Information describing the purpose of the component.

---

### The Component Hierarchy

Four relationships exist between components within the component structure:

- Parent
- Child
- Ancestor
- Descendant.

Every component except the root component must have at least one parent. A parent component must be specified when each new component is created. Since no component may be its own descendant or ancestor, the choice of additional parent components is limited. Figure 5 illustrates the component relationships.

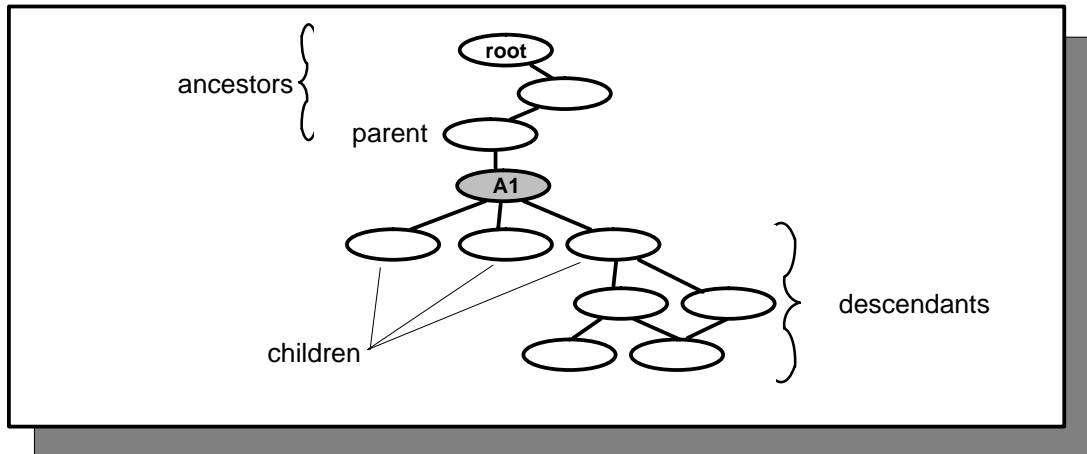


Figure 5. The component relationships

Component A1 has one parent, three ancestors, three children, and seven descendants.

Careful planning of the component hierarchy for your organization is a vital part of the configuration for CMVC, since a well-planned component hierarchy organizes your development data for effective overall development tracking. Once this structure has been created it can be modified as your organization grows, or as your needs change. Even if you are not involved in planning the component structure for your family you need to understand the uses of the CMVC component and its relationship to other CMVC objects.

## Using the Component Hierarchy to Manage Projects

You can define components to organize your development data into manageable groups. The component hierarchy subdivides your CMVC family into separate projects or work areas to meet the needs of the development teams using that family. Your family administrator may initially configure the component hierarchy but component owners and other end users will utilize this structure and may build on it as their development needs evolve.

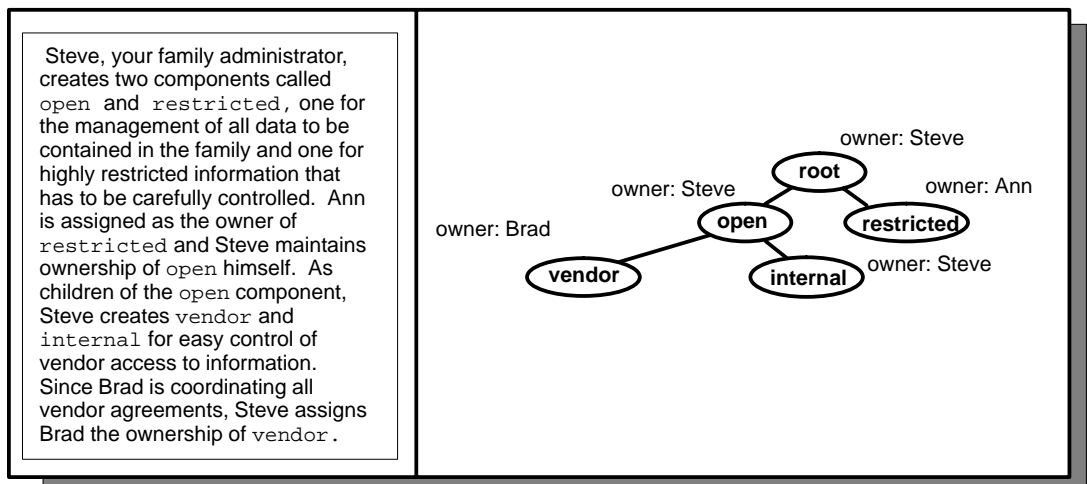


Figure 6. Designing the initial component structure

Components can be used to group data according to the needs of your organization. Useful ways to group data using the component hierarchy are by function, departmental organization, access needs, notification needs, or a combination of these. For example, in Figure 6 Steve creates a component hierarchy that addresses the access needs of his development area. In Figure 7, Steve and Brad build on the initial structure to group the data between the different projects and functions within those projects. Ownership of these new components is assigned to the individual project leaders. The creator of a component is automatically its owner unless a different owner is specified.

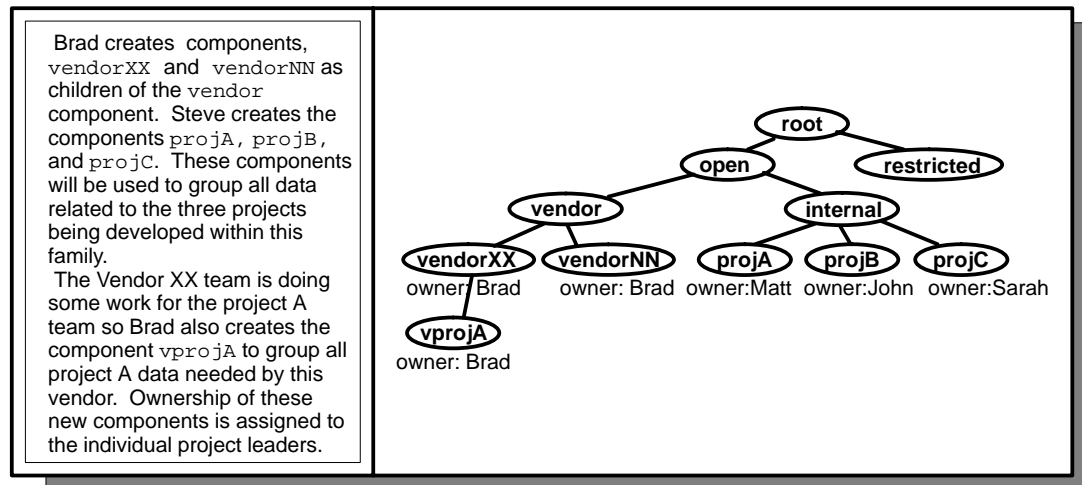


Figure 7. Adding to the initial component structure

Project leaders maintain responsibilities for their respective components. They can create components as needed to further divide management roles, responsibility, and development data.

Each additional level of the component hierarchy can be used to further distribute project management responsibilities and to organize the development data into smaller and smaller groups. Generally, the leaf components will manage a specific set of development data, such as your source code files.

## Creating Components With More than One Parent

Components can have more than one parent. Creating a second component as a parent creates a cross reference for the child component since the child component is then managed in two separate component groupings. For example, in Figure 8 the component `teamA` is created to group both the `internal` and `vendor` components for the development of project A, giving the `projA` and `vprojA` components each two parents. The `teamA` component can facilitate notification for the entire project A team. You can manage access control separately for the vendors and the internal users through the lower level components, `projA` and `vprojA`, and you can manage notification for the entire team from the component `teamA`.

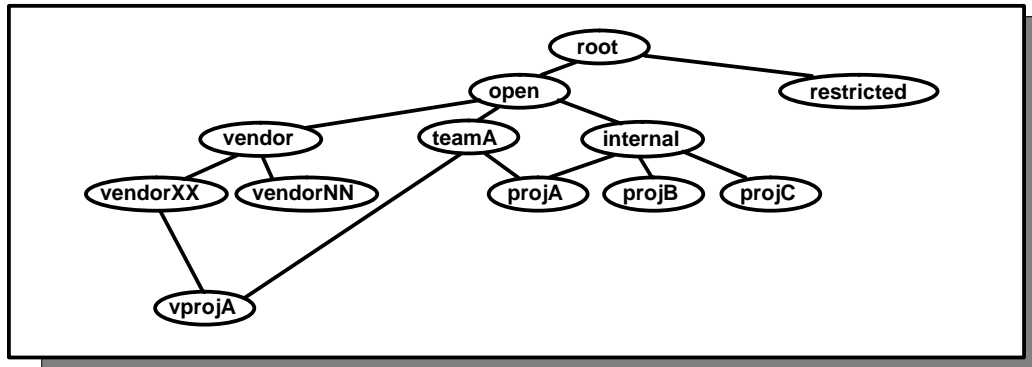


Figure 8. Components with more than one parent

---

## Component Ownership

Ownership of each component is assigned to one user. That user is responsible for managing all development data related to that component, including any problems reported against that component, any features proposed for that component, and any files managed by that component. The component owner is responsible for granting access and defining notification for all development data managed by that component.

Component ownership can be reassigned to another user or the owner can delegate aspects of management to other users by utilizing the component access list.

---

## Using Components to Manage Data

Through components, you can organize your data into manageable groups, control access to project data, and configure notification according to each user's role within the project. The component structure establishes a hierarchy that allows components to inherit access and notification properties from their ancestor components.

Each component has an *access list* and a *notification list*. The access list manages access to development data controlled by that component. The notification list manages user notification about actions performed on the development data contained in that component. Any notification interest defined for a user at one component is inherited by all descendant components. Granted access authority is inherited to all descendant components that do not have a restricted authority for the user in its access list. In this case only the component with the restricted authority will not inherit the granted authority.

There are over one hundred actions that can be performed with CMVC commands. These actions are split into various subsets to define different authority and interest groups. A user can be granted or restricted one or more authority groups on a single component access list, and one or more interest groups on a single component notification list. The authority and interest groups are maintained by your family administrator and can be configured to fit the needs of the development teams using your CMVC family.

## Controlling Access Authority

Access within CMVC is based on four types of authority: base, implicit, explicit, and restricted. The component access list defines explicit and restricted authority.

## Base Authority

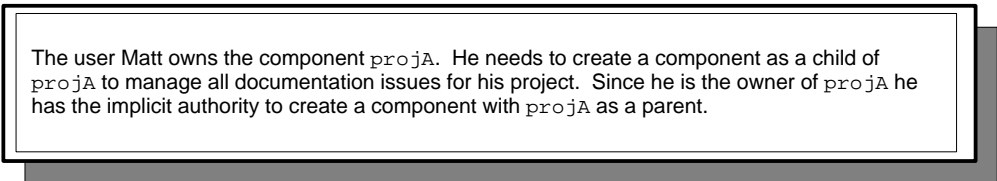
All users have the authority to perform the following actions as soon as a user ID and host list entry are created for them within CMVC:

- open defects
- open features
- comment on defects
- comment on features
- view the full path name of a file
- view user records
- search for information on CMVC objects (generate a report)

## Implicit Authority

The implicit authority to perform certain actions against a CMVC object is automatically granted to the user who owns the object.

For example,



The user Matt owns the component `projA`. He needs to create a component as a child of `projA` to manage all documentation issues for his project. Since he is the owner of `projA` he has the implicit authority to create a component with `projA` as a parent.

Figure 9. A user with the implicit authority to create a component.

☞ For a detailed list of all CMVC actions and the implicit authority required to perform them see the *IBM CMVC User's Reference*.

## Explicit Authority

Users who are not CMVC superusers and do not own the specified object need to be granted explicit authority to perform all CMVC actions except the base actions. Explicit authority to perform actions against each development object in a family is managed through the access list of the component that manages that object. The owner of each component can create entries on the access list granting users explicit authority for an action or set of actions related to the development data managed by that component.

☞ For more information about access lists see “Granting Access Authority” on page 18.

## Restricted Authority

Restricted authority is used to prevent the inheritance of granted authority to a specific component. The owner of each component can create entries on the access list to restrict users from the authority to perform an action or set of actions on that component. This does not affect the inheritance of authority to the components descending from it.

## The Superuser Privilege

A user with the CMVC superuser privilege can perform any CMVC action. A CMVC superuser is the only user who can add, delete, or recreate a user and only a CMVC superuser can grant superuser privilege to another user. The number of users who have this privilege should be minimized.

## Defining and Modifying Authority Groups

Authority groups are initially defined and can be modified over time by your family administrator. Any number of authority groups can be defined and any number of actions can be contained within one group. For example, each group may represent the actions used by a particular type of user. In Figure 10, the authority group `teamlead` is created and the `developer` and `manager` authority groups are used.

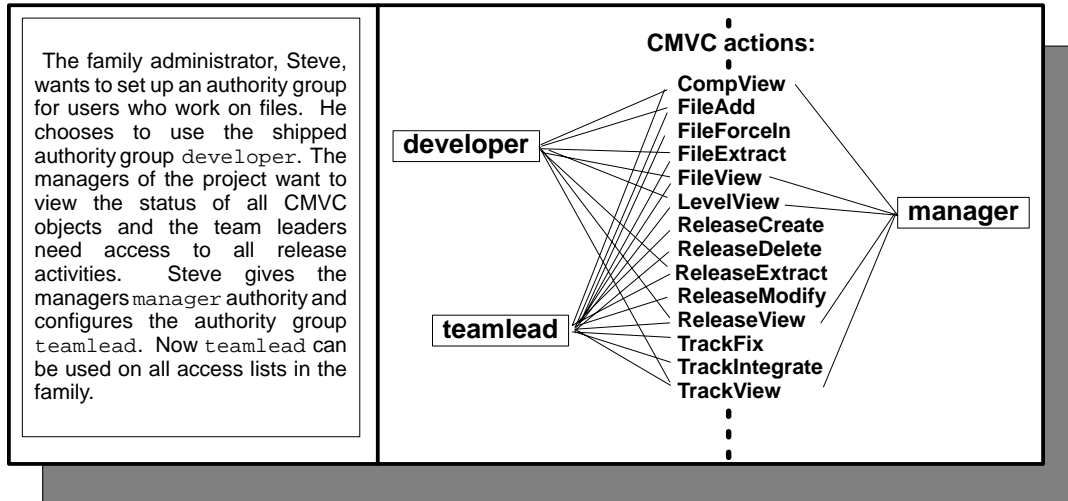


Figure 10. Grouping CMVC actions into authority groups.

Once authority groups are configured, component owners can use them to control user access to actions performed on the development data under their management.

## Granting Access Authority

Every component within your family has an access list that controls who has access to that component. Each entry on an access list maps a user ID to an authority group and an authority type. The authority group is composed of a group of CMVC actions which the designated user is allowed to perform or restricted from performing.

☞ See the *IBM CMVC User's Reference* for a complete listing of the IBM shipped authority groups.

In Figure 11 Steve and John granted different authorities to different users. Those users are shown in the access lists for each component. The authority granted on an access list has to be the name of an existing authority group. Each entry on an access list grants or restricts one user's authority to perform one set of CMVC actions against the development data managed by that component.

Only CMVC superusers, the component owner, and users with appropriate permission in their authority group can grant or restrict authority on an access list. You cannot grant authority greater than the authority that you have at that component. A superuser can grant any authority to any user on any access list. In Figure 11 Steve granted Ann `releaselead` authority at the `restricted` component. Steve is the family administrator and a CMVC superuser and therefore can grant authority to any user on any access list.

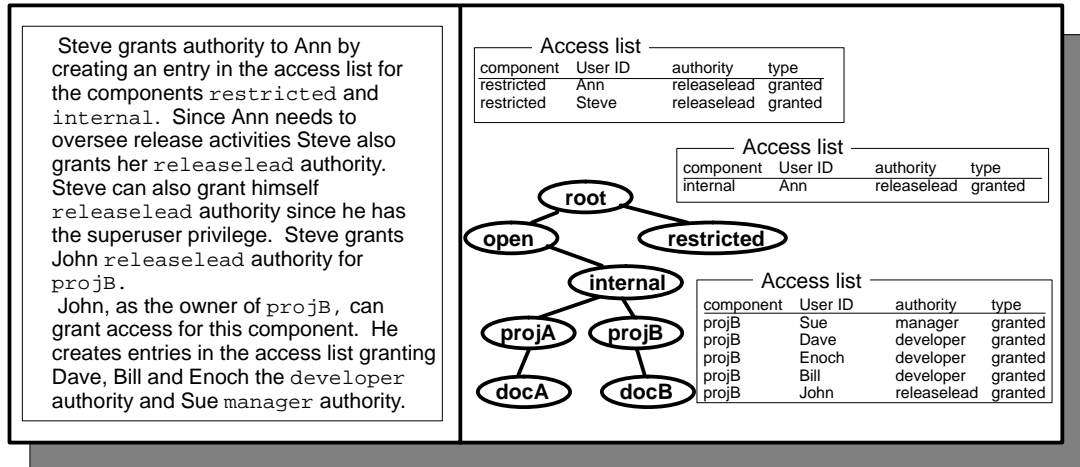


Figure 11. Granting access authority by using the access lists

Each granted entry on an access list also implicitly grants the specified authority to the user for any descendant components. For example, in Figure 11 Ann has `releaselead` authority on any objects managed by `projA`, `projB`, `docA`, and `docB` even though she is not on the access list for those components. Ann's `releaselead` authority was inherited by these components from the `internal` component.

## Inheritance

A user with authority at one component has that authority at all descendants of the original component through inheritance unless this authority is restricted at a descendant component. The parent-child relationship between components allows each child component to inherit properties from its parent components and all ancestor components. The following properties are inherited:

- Access
  - Any granted authority group defined for a user at one component is inherited by all descendant components. That is, all entries on an access list are inherited. The inheritance of authority can be restricted at a specific component, but the descendants of this component will still inherit access.
- Notification
  - Any interest group defined for a user at one component is inherited by all descendant components. That is, all entries on a notification list are inherited.

Inheritance is cumulative. At a given component, a user will have the superset of all authority groups granted at ancestor components, provided that the authorities are not restricted at the given component.

In Figure 11, all users on the access lists for `internal` and `projB` will also have authority on objects managed by the component `docB`. Any users not specified on the `internal` or `projB` access lists who need authority on objects managed by `docB` need to be entered on the access list for `docB`. If any of the users already on the access list for either `internal` or `projB` need more authority for component `docB` than the authority that they inherited, they need to be granted explicit authority on the access list for `docB` (see Figure 12).

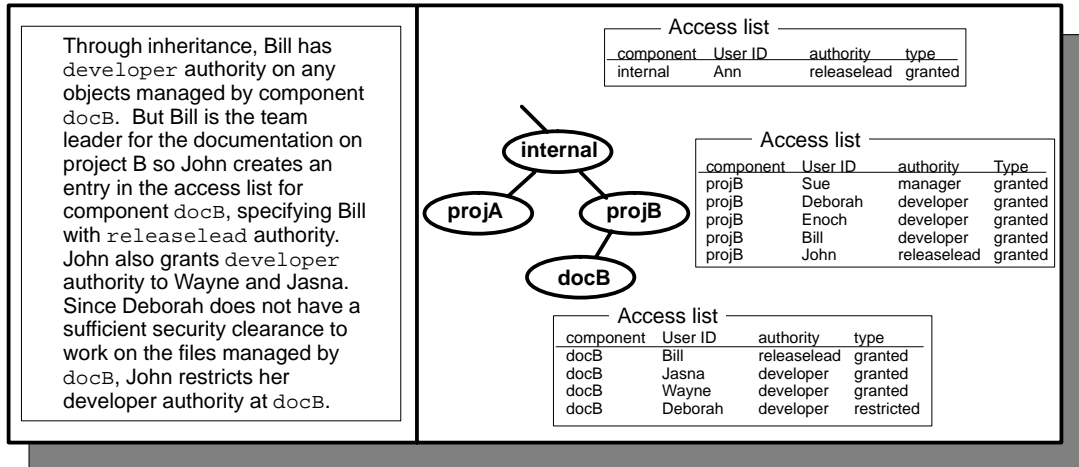


Figure 12. Managing access authority

In Figure 12, the additional entry for Bill at the component `docB` gives him more access to data managed by `docB` but does not change his access to data managed by `projB`. When granting access, you must look at both the existing inherited authority and the descendant components where any new authority will be inherited.

Inherited authority can be removed from descendant components. If it is not removed, then a component with more than one parent inherits access and notification properties cumulatively from each parent component and all ancestors.

## Controlling Notification

Notification of actions performed within CMVC are sent to the mail address specified in each user's CMVC user ID. Users are not notified of every action performed within their family. With over one hundred actions possible, even a small family would generate too many notifications. When an action is performed users get notified either automatically or explicitly.

### Automatic Notification

A user who owns a CMVC object is automatically notified of certain actions being performed against that object. In addition, users automatically receive notification if an action affects their user IDs or requires them to perform an action in return. For example, a user receives automatic notification when their user ID is added to an access list.

### Explicit Notification

Users who wish to receive additional notification need to be specified as subscribers. Explicit notification about activity affecting a CMVC object is managed through the notification list of the component that manages that object.

For a detailed list of all CMVC actions and who is notified when they are performed see the *IBM CMVC User's Reference*.

## Subscribers

A subscriber is a user who is notified when an action is performed against a development object. Users interested in receiving notification about a development object can be added to the notification list of the component that manages that object.

There is one notification list for every component within your family. Each entry on a notification list maps a user ID to an interest group composed of a set of CMVC actions. If



you subscribe to a component, you will be notified each time an action included in your interest group occurs on data managed by that component.

☞ See the *IBM CMVC User's Reference* for a complete listing of the IBM shipped interest groups.

## Defining and Modifying Interest Groups

Interest groups are initially defined by and can be modified over time by your family administrator. Any number of interest groups can be configured and any number of actions can be contained within one group. For example, each group may represent the actions a particular type of user would want to be notified about. In Figure 13, the interest group `writer` is created.

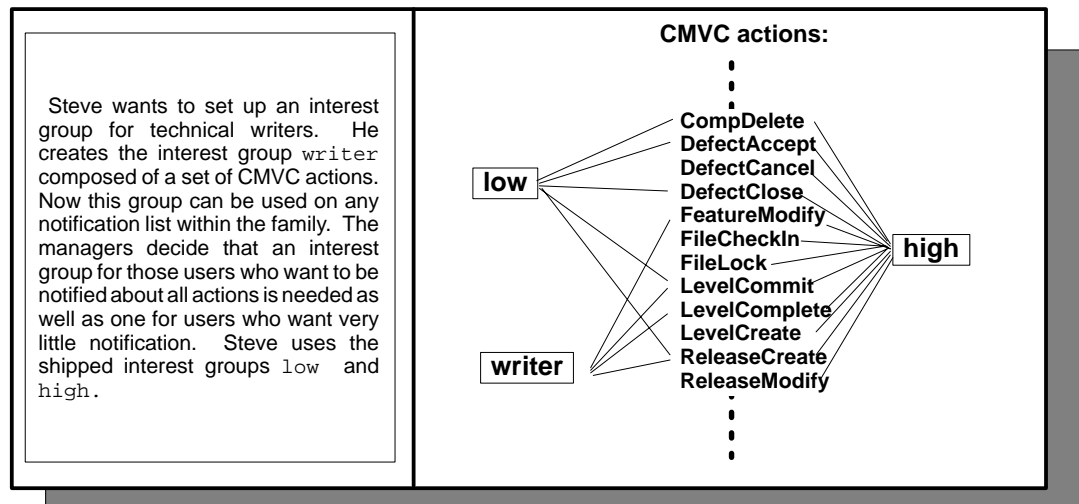


Figure 13. Grouping CMVC actions into interest groups.

Once interest groups are configured, component owners can use them to specify individual users as subscribers on notification lists.

---

## Component Processes

In addition to managing files, authority lists, and notification lists, components control the behavior of defects and features that are opened against them. This is done by selecting a process for each component. The family administrator configures the possible processes from the defect and feature DSR and verify subprocesses.

The component owner can set and change the process used by the component. Other people, such as release and project leaders, may also have sufficient authority to change the component's process. The choice of processes will change the steps that are required to resolve problems and implement design changes. Processes are chosen to reflect the current stage of development that the objects managed by the component are in. For example, components managing code may not require the defect DSR subprocess during the early stages of software development.

☞ See Chapter 6, "*Problem and Design Tracking*" for more information about the defect and feature DSR and verify subprocesses.

---

## Component-File Relationship

Components organize files into manageable groups, control access to files, and configure notification about actions performed against files. Although components organize your files into manageable groups, they do not alter the path name of the individual files. Components provide a link to the files for management and control purposes.

Each CMVC file must be managed by a component. This component must be specified at the time that the file is created within CMVC; however, it can be modified by a user with the proper authority.

Access to each file under CMVC control is controlled through the access list of the managing component. The authority group or groups that contain CMVC file actions must be specified in order to grant a user access to a file. Creating an access list entry for a user at one component grants him or her that authority over all development data, including files, managed by that component or any of its descendants that do not explicitly restrict the inheritance of the user's authority.

Notification about file activities is configured through the notification list of the managing component. The interest group or groups that contain CMVC file actions must be specified when subscribing a user for file-related notification. Creating a notification list entry for a user at one component subscribes him or her to all development data, including files, managed by that component or any of its descendants.

---

## Chapter 4. Using CMVC Releases

The CMVC release organizes files into manageable groups for development as single versions of a product. This chapter explains what a CMVC release is and describes the relationship between releases, files, and components within a CMVC family.

---

### What Is a Release?

A release is a logical organization of all files that are related to a particular version of a product. However, like components, a release does not change the physical location of any files; instead it provides a logical view of the files that must be built and distributed together in order to create a version of a product.

Releases organize files for product related activities. Each file must be managed by at least one component and grouped by at least one release.

#### Background

**Components** provide organization and control of development data.  
**Access lists** designate users with explicit authority at a specified component.

#### Release Attributes

CMVC releases have the following attributes:

- Name
  - Each release name within a family must be unique.
- Owner
  - Every release has an owner. The release owner has the implicit authority to perform most CMVC actions against that release and is primarily responsible for actions relating to that release.
- Component
  - Every release is associated with a component for the management of access control and notification.
- Process
  - The process that the release will use. This determines whether the tracking, level, approval, test, and fix subprocesses are used.
- Description
  - Information describing the purpose of the release.

---

### Release Management

Every release must have a managing component. This component is specified when the release is created. Through this component you can control access to the release and configure notification according to each user's interest in release-related actions. This is done with the access and notification lists.

While components exist in a hierarchical structure, releases do not. A release has a one-to-one relationship with a component for management purposes and a one-to-many relationship with files for product-related activities. For example, in Figure 14 the release

ToolAv1 is associated with the component teamA when teamA is specified as the managing component.

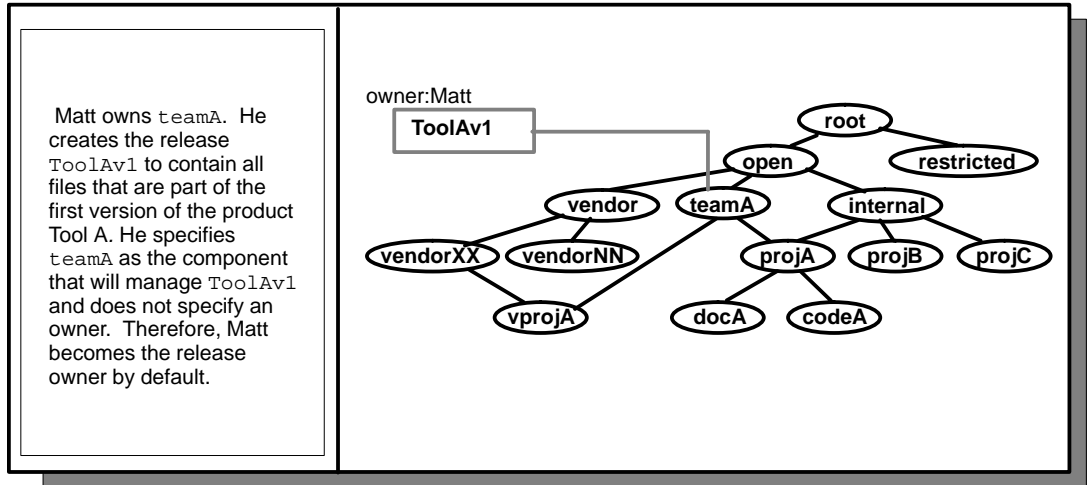


Figure 14. The release-component relationship

## Creating a Release

To create a release you must have the appropriate authority included in your authority group for the component that you specify as the managing component (or one of its ancestors).

In Figure 14, access to release ToolAv1 is managed by members of the access list of component teamA and notification about release ToolAv1 is managed by members of the notification list of component teamA. All access and notification must be controlled from the release's managing component.

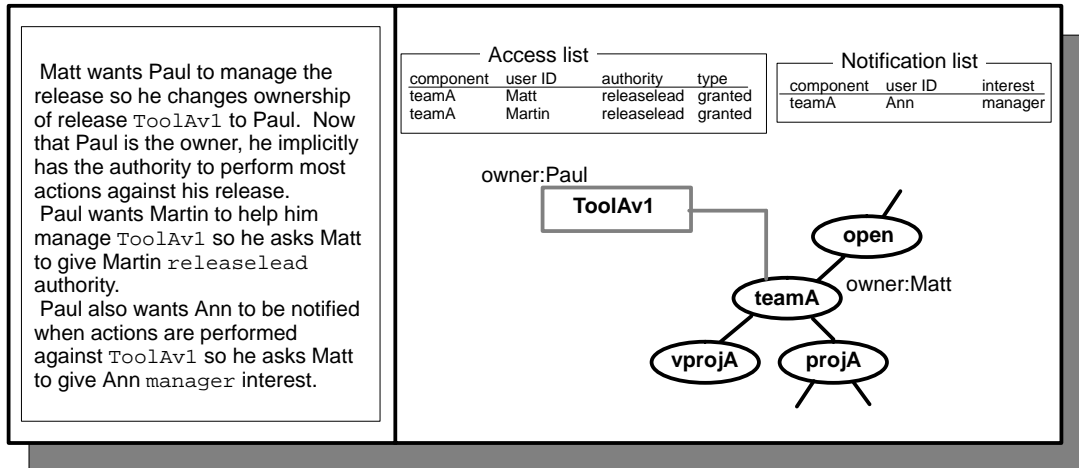


Figure 15. Access and notification for releases

In Figure 15, although ownership of release ToolAv1 was assigned to Paul, access and notification for the release are still managed by component teamA, and are therefore under the control of Matt (who owns the component teamA). In this way, Matt can maintain access and notification control of the release, even though he has delegated the management responsibilities to Paul.

If Matt wants to give Paul the ability to control the access to release ToolAv1, he can grant Paul an authority which contains the authority to add users to the access list. Both Paul and Matt would then have the authority to add entries to the access list of component teamA.

Matt can only grant other users an authority that he already has at that component. In other words, he cannot grant any authority that is greater than the `releaselead` authority that he currently has for `teamA`.

## Considering User Access When Creating a Release

When you create a release, consider the access list of the component you are specifying to manage the release. If you are in control of the access list, then you can control who can perform actions against the release. However, you must also look at the access lists of the ancestor components, since access authority is inherited. Find out what authority groups grant release-related authority and check the access lists for users who have been granted that authority. Creating a release that is managed by a component with few ancestors will give you greater control of the access to that release. You can also control access by explicitly restricting its inheritance at a given component.

---

## Using Releases to Organize Files

Releases help you organize your files into the groups needed for the development of particular versions of products. These groups could be based on products to be shipped, parts of a product that need to be built separately, documents, test cases, or anything else that requires a grouping of files for product-related activities such as extracting, building, or compiling.

## The Release-File Relationship

Releases are used to group files together along a single line of development. The files in a release may be managed by one component or by a variety of components. When a file is created in CMVC, a component must be specified to manage access and notification. A release must also be specified. For example, in Figure 16, the file `intro.i` is created under the component `docA` and the release `ToolAv1`.

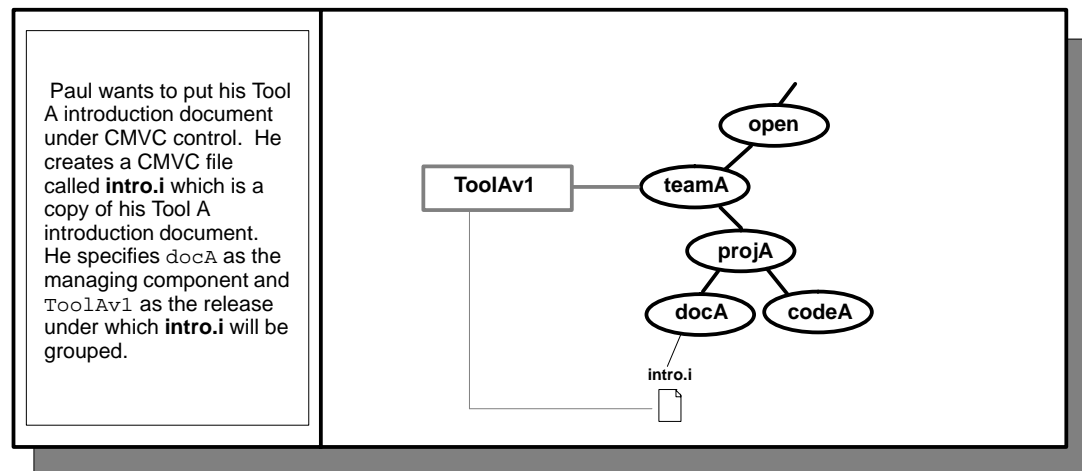


Figure 16. File-component-release relationship

A file created and placed under CMVC control is identified by its path name. This path name can be a portion of its path name on your workstation or its entire path name. This path name is recreated when the file is extracted from the CMVC server. Extracting a release copies all files grouped by that release to a specified Network File System (NFS\*\*) server, creating a file tree. This file tree can then be compiled and tested as needed.

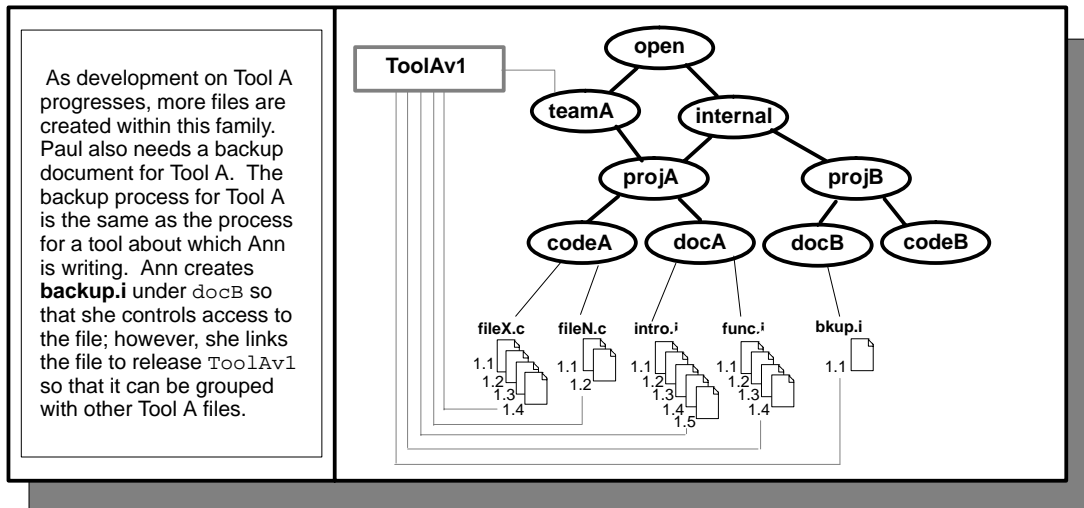


Figure 17. A release grouping files from different components

In Figure 17, `ToolAv1` contains the files `fileX.c`, `fileN.c`, `intro.i`, `func.i`, and `bkup.i`. The document files are managed by `docA` or `docB`, and the code files are managed by `codeA`. In this way, access to the document files can be managed separately from access to the code files. Note that the components managing files contained in one release can be from any part of the component tree.

## Change Control and Integrated Problem Tracking

Change control regulates files so that only one user can edit a file at a time. If a release has been configured to integrate problem tracking with change control, then all file changes are linked to features and defects. This provides a systematic, configurable approach to tracking the file changes needed to resolve a reported problem or implement a proposed design. The change control process that has been configured for a release must be followed.

To change a CMVC file you must check out the file. A CMVC file in a release can only be checked out or locked by a single user. Once the file changes are complete, the file must be checked back in to CMVC. It is then available for another user to check out.

## Configuring the Integrated Problem Tracking Subprocesses

The family administrator can configure processes, each consisting of a name for the process and the set of subprocesses that are used by the new process. Four subprocesses are available to releases that use the track subprocess:

- The approval subprocess
- The fix subprocess
- The test subprocess
- The level subprocess.

The approval subprocess requires proposed file changes to be approved by everyone on the *approver list* before the files are actually changed. An approver list is a list of users who need to approve proposed changes to any files contained in the release.

The fix subprocess requires the component owner or the user making the file changes to implement a feature or resolve a defect to mark the fix records as complete once changes

for the defect or feature are finished. This indicates that the file changes for the defect or feature are ready to be integrated with the rest of the files in the release.

If the test subprocess is included in the change control process, file changes associated with defects and features are tested in the newly updated release in certain environments. For each release, an *environment list* defines the environments and the user responsible for testing each environment.

The level subprocess provides a method of integrating files changed for defects and features with the rest of the files in the release. It also provides a method of extracting the most recent versions of files that will work together correctly. Levels can be used to facilitate the testing subprocess.

☞ Change control for tracked releases is described in detail in Chapter 7, “Using Tracks”.

---

## Extracting a Release

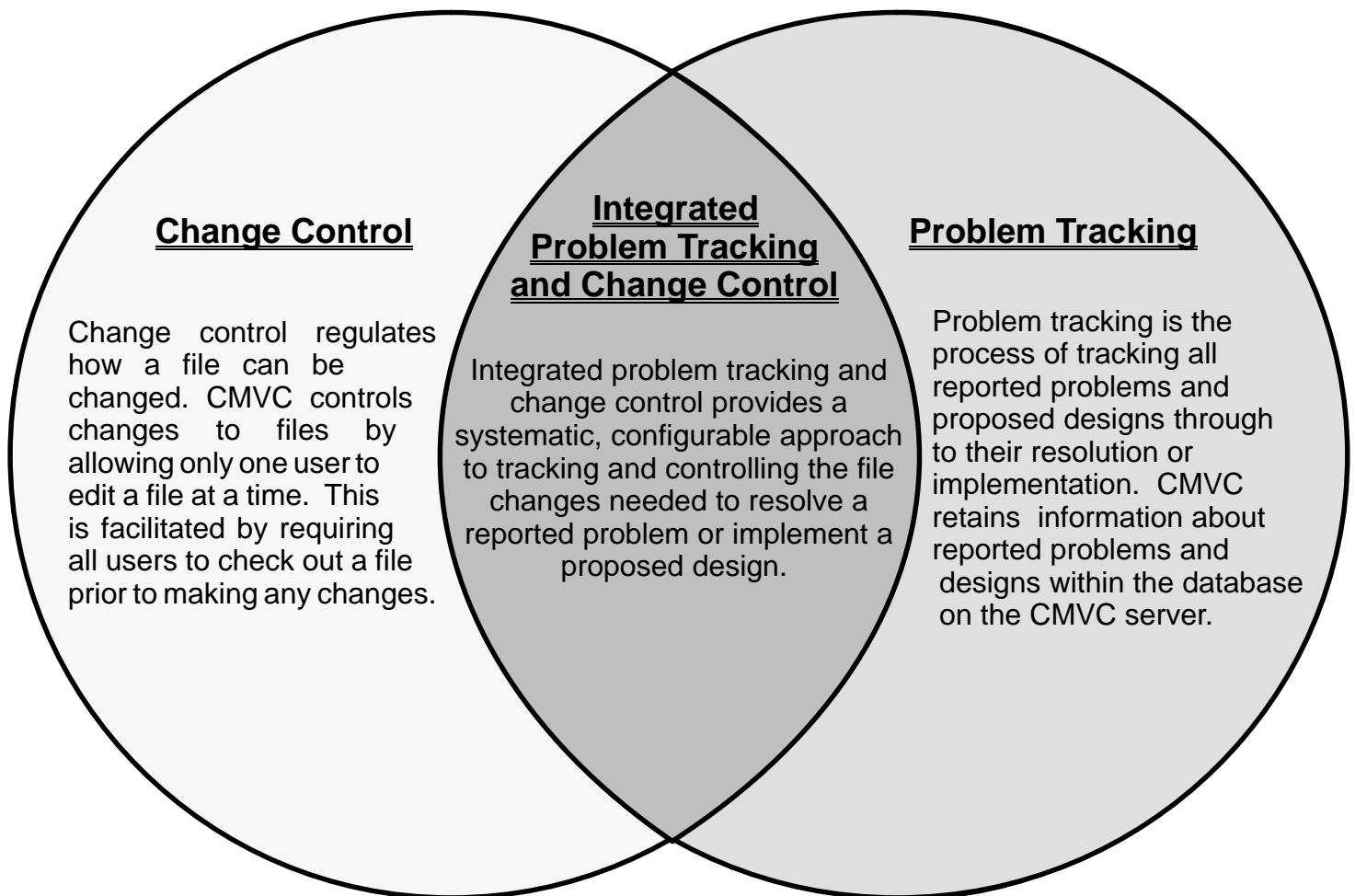
Extracting a release creates a file tree on a specified NFS server. The structure of the file tree is dictated by the path names of the files that make up the release. Any directories in a file's path name that do not already exist on the specified NFS server will be created when the release is extracted.

Several different versions of a release can be specified for extraction. The current and committed versions of files grouped by a release can be extracted. It is also possible to extract all of the files grouped by a release that were changed after a specified date.





# Change Control and Problem Tracking



Change control prevents any two users from changing the same version of the same file. Problem tracking is implemented through CMVC defects and features. Defects and features record any problems or proposals that your organization needs to monitor. Integrated problem tracking and change control are implemented by using a CMVC track to monitor the changes made to the files of one release in order to resolve one defect or feature. The following chapters outline the concepts involved in working with files, defects, features, tracks, and levels to structure and manage your existing development process.



---

## Chapter 5. Controlling File Changes

CMVC stores all versions of CMVC files on the CMVC server's file system and records information about those files in the relational database on the CMVC server. This chapter describes how to work with files in the CMVC environment and how files relate to other CMVC objects within a family.

Examples of file versioning in this book assume that SCCS is the underlying version control system on the CMVC server. Version numbers may differ when using PVCS Version Manager as the underlying version control system on the CMVC server.

---

### What Is a CMVC File?

A CMVC file is a text or binary file that is under the control of the CMVC server. Each file under CMVC control is uniquely identified by a path name and the name of the release in which it is contained. All files under CMVC control are managed by components.

When you create a CMVC file, you are placing an existing file from your workspace under CMVC control. Once a file is created under CMVC control the official copy of the file resides in the file system on the CMVC server. Changes to this file are made by checking it out to a CMVC client, making changes, and then checking the changed copy back into the CMVC server. Components control access to all files under CMVC control and releases group files for product-related activities.

CMVC stores additional information about the file each time an action is performed against it. This information is stored in the relational database on the CMVC server and can be queried at any time.

### File Attributes

A CMVC file always has the following attributes:

- Path name
  - The name of the file as known to the CMVC server. A path name can consist of a set of directory names and a base name, or just a base name. The path name must be unique within the release that contains the file.
- Base name
  - The name assigned to the file, excluding any directory names. For example, **tools/intro.i** has a base name of **intro.i**.
- File mode
  - The read, write, and execute permissions for the file, represented as an octal number. This attribute is updated each time the file is checked in.
- Component
  - The component that manages the file.
- Release
  - The release that groups the file.
- Current version
  - The version number of the last version of this file to be checked in to the CMVC server.

- Committed Version
  - The version number of the most recent version of the file that has been committed with a track or associated with a committed level. If the file is grouped by a release not under tracking then the committed version is the same as the current version.
    - ☞ Levels and the process of integrating file changes into a committed level are described in detail in Chapter 8, “Using Levels”.
    - ☞ For information about how to configure additional attributes, refer to *IBM CMVC Server Administration and Installation*.

---

## Versioning of Files

CMVC uses an underlying version control system to version files as they change. Your family administrator will have configured the CMVC server to use either Source Code Control System (SCCS) or INTERSOLV's PVCS Version Manager as the underlying version control system. The version control system installed on the CMVC server controls the versioning of all files under CMVC control.

Each CMVC file is created with an initial version number of 1.1. When a file is checked out of the CMVC server, a new version number is reserved for the version of the file which will be checked back in. When you check in a file to the CMVC server, only the differences (the delta) between the version of the file that you check in and the previous version are recorded and stored as the new version. This is called *delta versioning*. The version of a file that you check out is constructed as the sum of all deltas along a single line of development.

---

## Getting Files from CMVC

A CMVC file can be *extracted* or *checked out* by a user with the proper authority. Any version of a file may be extracted at any time, but only unlocked current versions of files may be checked out. Checking out a file implies the intention to modify it.

To check out or extract a CMVC file you must specify its path name, the release that groups the file, and the directory where you want the file to be placed on your CMVC client. A version number must also be supplied if you wish to extract a version of the file other than the current version. The authority to check out and extract files is granted to you on the access list of the component that manages the file or one of the component's ancestors.

Checking out and extracting a file both copy the file from CMVC and place it in the destination directory on a CMVC client. CMVC will avoid overwriting a file with the same name by renaming the old file. For example in Figure 18, when Ann extracted version 1.1 of **intro.i**, and then extracted version 1.5 of **intro.i**, both versions were extracted to the same directory on *jessica*. The first copy of **intro.i** was automatically renamed. The CMVC client preserves one backup copy of any file that could otherwise be overwritten by the checking out or extracting a file.

## Checking Out a File

When you check out a file, the current version is copied and placed in the specified directory on your CMVC client. The copy of the file on your CMVC client is known as the working version of the file. The working version of the file will have the permissions specified in the file mode attribute. On the CMVC server a new version number is reserved for the version of the file which will be checked in. The file on the CMVC server is locked so that no other user can check out the file until you have checked in your changes to the CMVC server. For example, in Figure 18 when Paul checks out the file **intro.i**, a copy of the current version

(1.4) is placed on his CMVC client. Version 1.4 is then locked and a new version number (1.5) is reserved for Paul's changes. Similarly, when Ann checks out the file **func.i**, a copy of the current version (1.1) is made and placed on her CMVC client.

Any locked CMVC file can be unlocked by users that have the proper authority defined for the file's managing component (or its ancestors) can perform this action. The number of users granted the authority to perform this action should be minimized.

### Extracting a File

Extracting a file copies the indicated version of the file to your workstation but does not lock the file in CMVC. By default, the file on the workstation will have the permissions in the file mode attribute, except that nobody will have write permission. A locked version of a file can be extracted by other users. For example, in Figure 18 Paul needs to look at information in a file that is already locked. Since he does not intend to make changes to the file all he needs is a copy of the most recent version. So he extracts version 1.1 of the file **func.i**. Ann needs information that she thinks was included in an earlier version of a file, so she performs a file extract to get version 1.1 of the file **intro.i**.

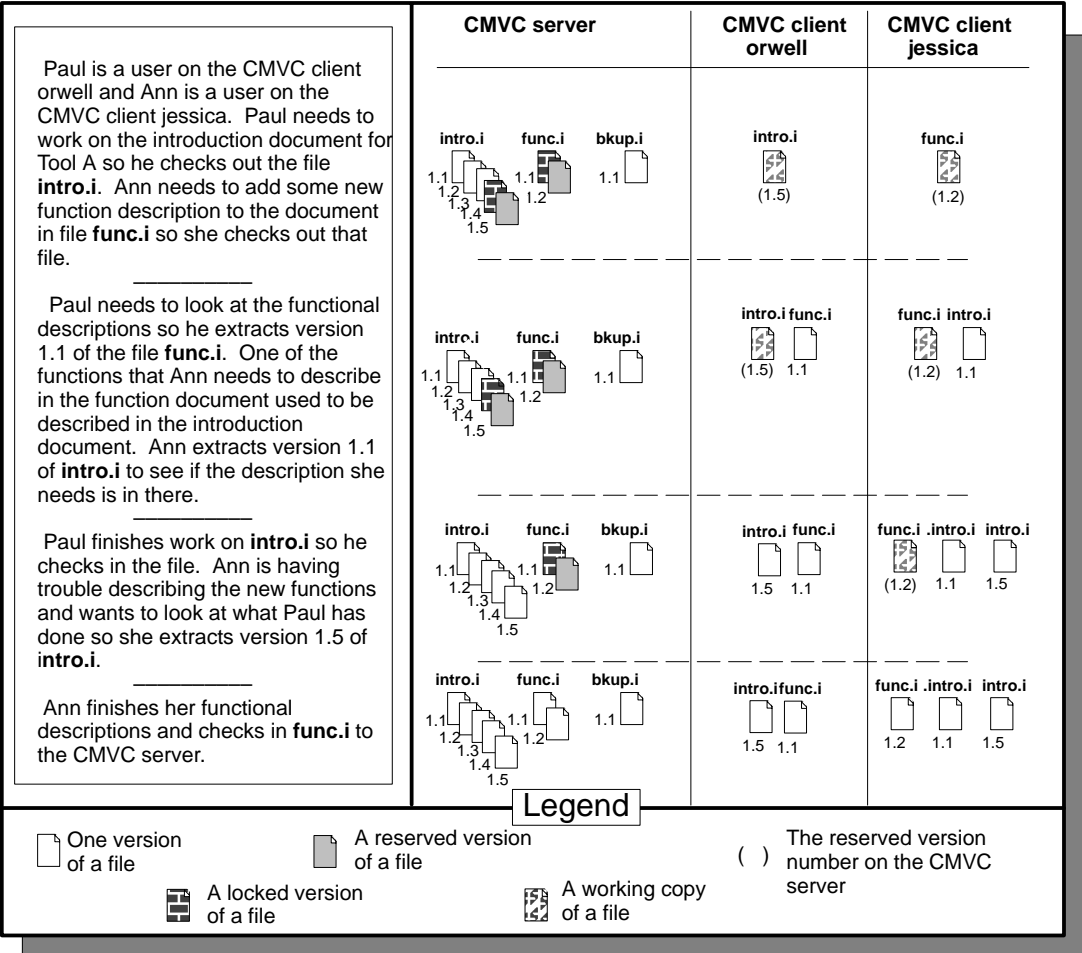


Figure 18. Checking files in and out of the CMVC server

---

## Checking in CMVC Files

If you checked out a file from CMVC, you implicitly have the authority to check in the file.

Checking in a file copies your working version of that file into the file system on the CMVC server, unlocks the current version and establishes your changes as the new current version. For example, in Figure 18 when Paul checks in **intro.i** to the CMVC server it becomes version 1.5. Note that when Ann wanted to look at this current version of **intro.i** she did not check out the file, even though it was not locked. If you are not going to make any changes to a file, extract the file instead of checking it out so that the file will still be available for other users to check out.

To check in a file to the CMVC server you must specify the path name of the file, the directory on your workstation where the file can be found, and the release that groups the file that you originally checked out. If the release process includes the track subprocess, you must also specify the track that is monitoring the file changes for that release.

---

## Files Shared Between Releases

A CMVC file is uniquely identified by the path name of the file and the name of the release in which it is contained. Both the release name and the path name must be specified whenever you perform a CMVC action on a file. Multiple releases may group the same file.

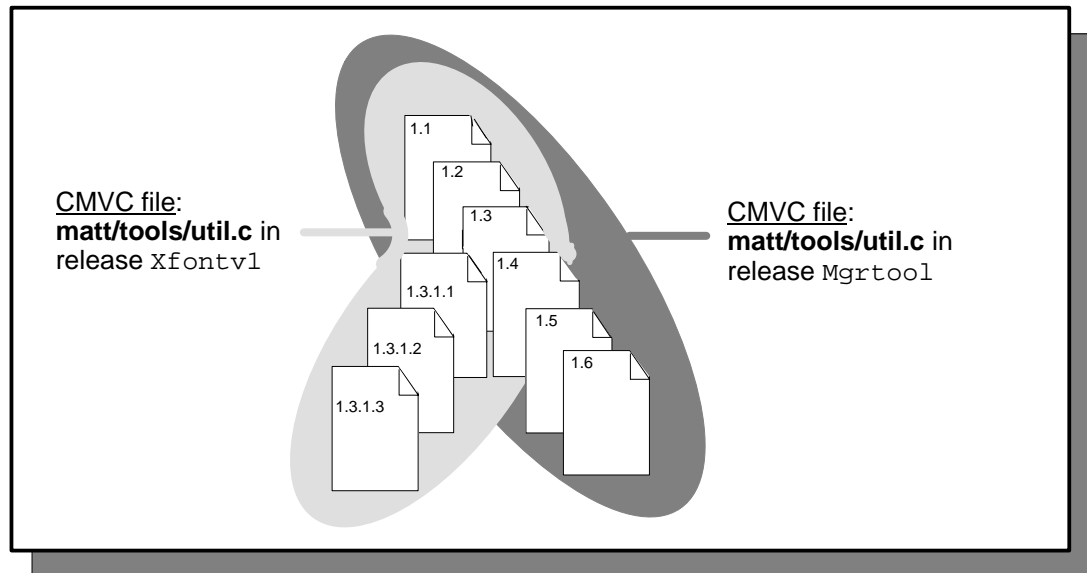


Figure 19. Two CMVC files

Any file that is contained in two or more releases is a *shared file*. A shared file consists of two CMVC files sharing information. Each path name-release pair represents a separate CMVC file, as shown in Figure 19.

If the developers of one product need to use the information that is already contained in a release for a different product then they can link the file that contains that information to their release (if they have the proper authority). Both releases would then have a link to the current version of that file. In Figure 20, the file **bkup.i** is a shared file since both `ToolAv1` and `ToolBv1` contain **bkup.i**. Since both releases are linked to the current version of **bkup.i** (version 1.2), **bkup.i** is also a common file.

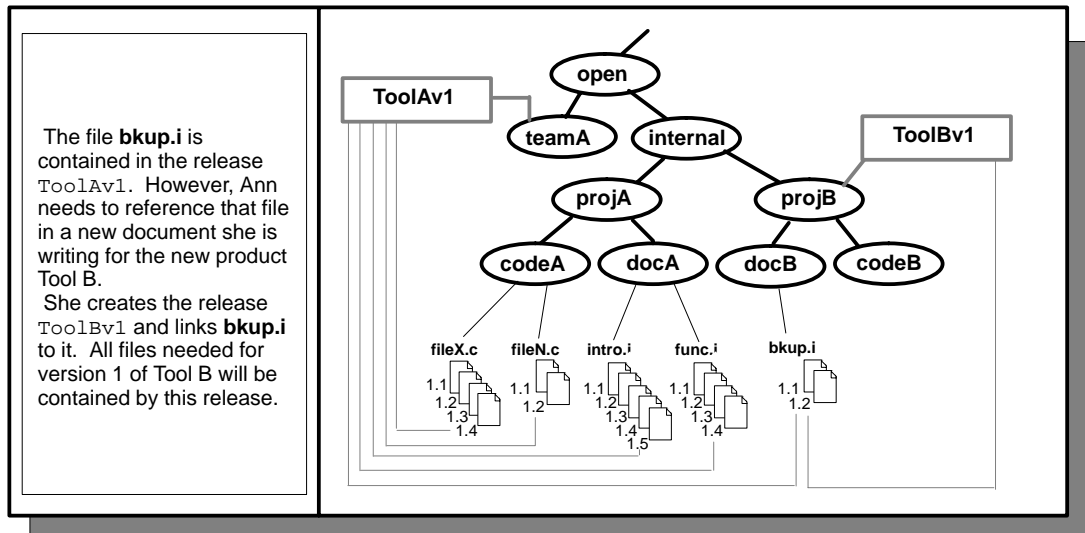


Figure 20. Two releases sharing one file

### Common Files

A *common file* is a file that is shared between multiple releases where each release references the same current version of the file. File **bkup.i** in Figure 20 is both a common and a shared file. Checking out a common file from either release locks the current version of the file. Since both releases are linked to the same current version, the file is locked in both releases. For example in Figure 21, when **bkup.i** is checked out of ToolAv1 on the CMVC server it is locked in both ToolAv1 and ToolBv1.

In releases that include the track subprocess, common files can be maintained automatically. To check in a common file to the CMVC server and maintain the common link, you must specify the release from which you checked out the file and any other releases in which it is common. In this way any changes made to a common file can be reflected in more than one release through a single check in action.

In releases that do not include the track subprocess, the common link is broken automatically as soon as the file is checked in.

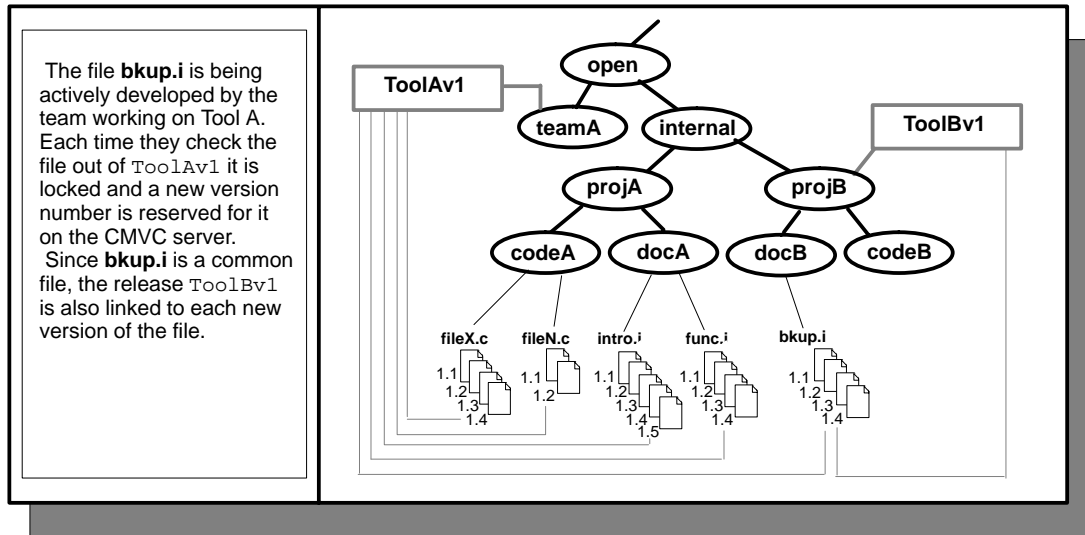


Figure 21. A common file between two releases

When development begins on a new release of a product, all the files in the current release can be linked to a new release, so that initially all the files are common between both releases. As development of the releases progresses, the common link between the files can be broken as needed to separate development of the new release from maintenance of the current release.

## Breaking the Common Link

When a file is linked to other releases, that file will remain common in all releases until the common link is broken. The common link can be broken in one of two ways:

- Breaking the common link at check in
- Breaking the common link at check out.

If you have made changes to a common file and do not want the changes to be reflected in the other releases that link to that file, then you can break the common link when you check in the file. The other releases will not be linked to the version you check in if you break the common link. This file is still a shared file but it is no longer a common file.

If a file is common between more than two releases you have the option of maintaining the common link between some of the releases while breaking the link with others. This can only be done when checking the file in to the CMVC server.

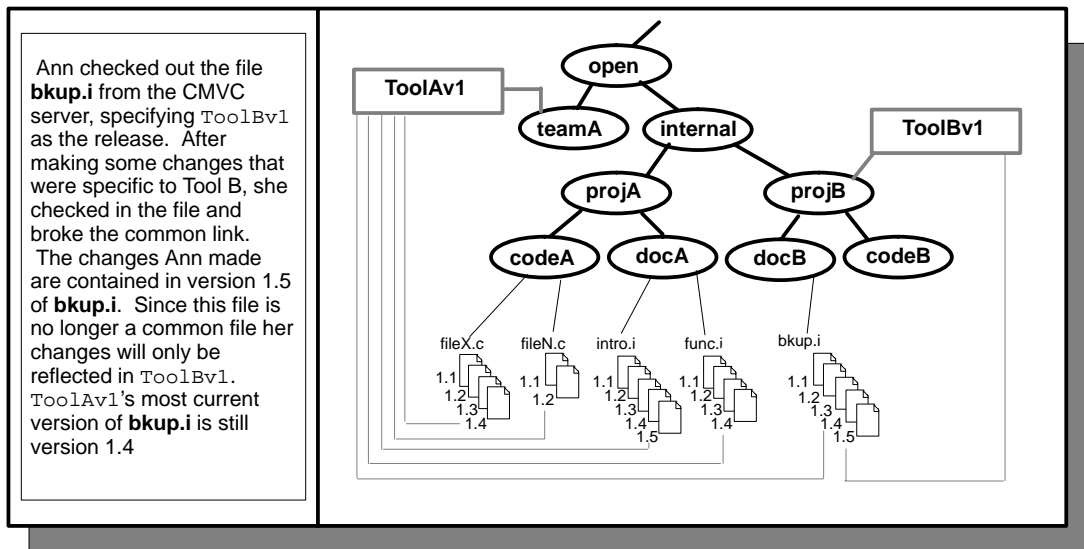


Figure 22. Breaking the common link when checking in a file

If you want to make changes to an already locked common file, you have the option of breaking the common link when you check out the file. Breaking the common link at check out can only occur if the common file is already locked in another release.

Each release contains successive versions of individual files. The versions that are contained in each release represent a single line of development. In Figure 22, the release ToolAv1 contains the versions 1.1 to 1.4 of **bkup.i** and ToolBv1 contains the versions 1.1 to 1.5. Each release contains the versions of **bkup.i** that contribute towards the development of the related product. This includes any development done while **bkup.i** was a common file.

The next time **bkup.i** is checked out of ToolAv1, the file will branch at version 1.4. Figure 23 shows a magnified view of **bkup.i** after the common link is broken.



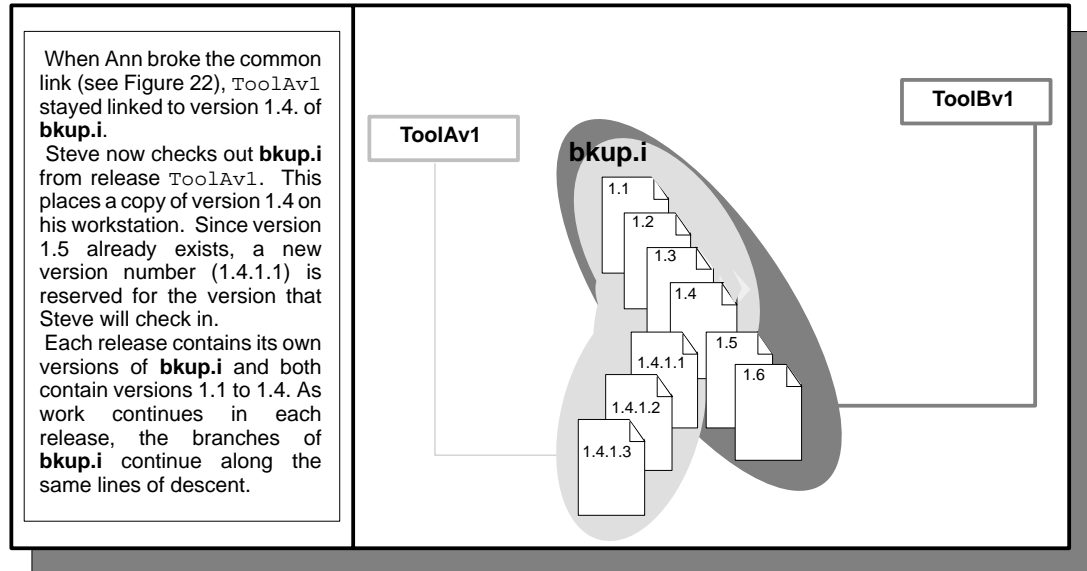


Figure 23. Two branches of a file

You can only check out the current version of a file in a release. In Figure 23, you can only check out version 1.4.1.3 of the file from the release `ToolAv1`, and version 1.6 from `ToolBv1`. In Figure 22, all versions of **bkup.i** are managed by the component `docB`. Any user who has the authority to check out **bkup.i** from `ToolAv1` can also check out **bkup.i** from release `ToolBv1`, since access is managed by the file's component. If different access control is required for different lines of development within the file, then separate managing components can be defined.

## Managing Access to Shared Files

A shared file may be managed by multiple components. One component can be specified to manage each line of development within the file. In other words, one component can be specified to manage the versions of a file contained in one release.

When a file is first created on the CMVC server, one component manages the file and one release contains the file. If the file is later linked to another release, a separate component can be specified to manage the development of the file in the new release.

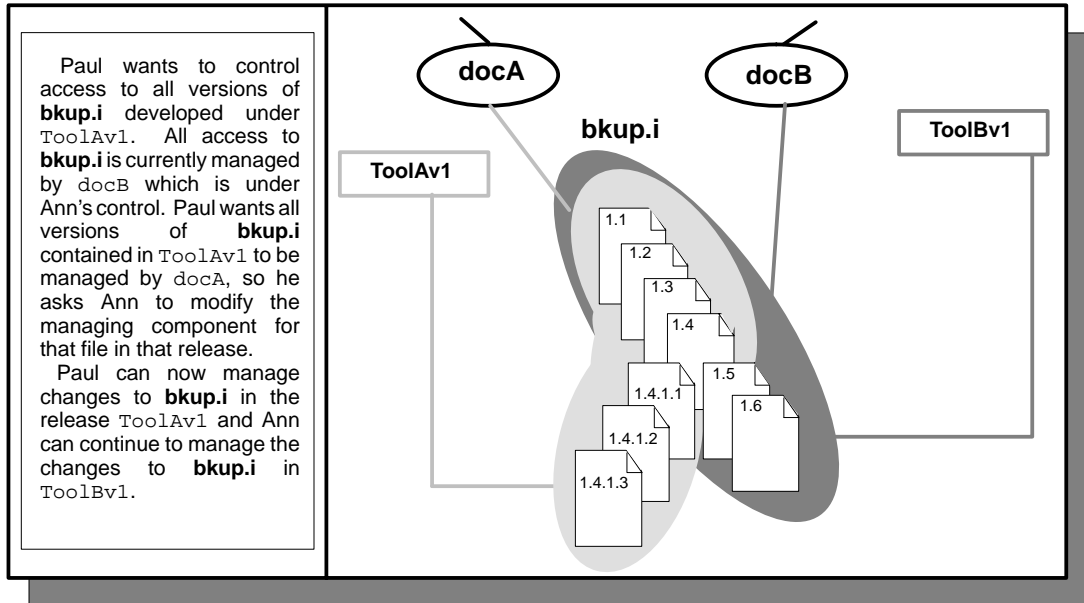


Figure 24. Managing access to a shared file

Only a user granted permission to modify file attributes in an authority group on the access list of the current managing component or one of its ancestors (provided that the authority is not restricted at the managing component) can change the component that manages the file. In Figure 24, Paul does not have the proper authority to modify the managing component for file **bkup.i** even though he owned one of the releases linked to that file (this is why he asks Ann to make the modification). Access is always controlled by the file's managing component.

## Files in Releases with Integrated Problem Tracking

CMVC *tracks* are used to monitor file changes made to resolve one defect or feature in one release. Changes to files contained in a release that includes the track subprocess must reference a track. Specify a track when performing the following actions:

- Creating a file
- Checking in a file
- Linking a file to a release
- Undoing file changes
- Recreating a file
- Renaming a file
- Deleting a file.

The track referenced must be in the appropriate state before the action is allowed. For example, a file can only be checked in if the track referenced is in the fix state. If the fix subprocess is configured as part of the component's process, the associated fix records must be in the ready or active states.

☞ For more information about tracks and their relationships to releases and file changes see Chapter 7, "Using Tracks".

## Undoing File Changes

Changes made to any file can be undone by a user with the proper authority, unless the version of the file which includes those changes has been committed. Changes to files include all actions that may be performed against files. File changes must always be undone in the reverse order that they were made. For example, suppose that you check in a file, rename it, and then realize that the file should not have been checked in yet. To undo checking in the file, the renaming must first be undone.

If the release that includes the file is not configured to include the track subprocess then any number of file changes may be undone. If the release is tracked then the committed version of a file is a permanent version of that file; changes made prior to the committed version of a file cannot be undone.

After committing a tracked file you can create, rename, delete, recreate, or link it only once before the next time it is committed. This makes the ability to undo file changes very important. For example, you cannot create a file and then rename it before it is committed; instead you must undo the creation of the file and then create a new file with the name that you wanted the old file file to have.

Figure 25 shows three files before and after their tracks were included in a committed level. In Example 1, the changes made in version 1.4 of **FileB** can be undone by a user with the proper authority. In Example 2, the changes made in version 1.4 of **FileB** have been committed and therefore can not be undone.

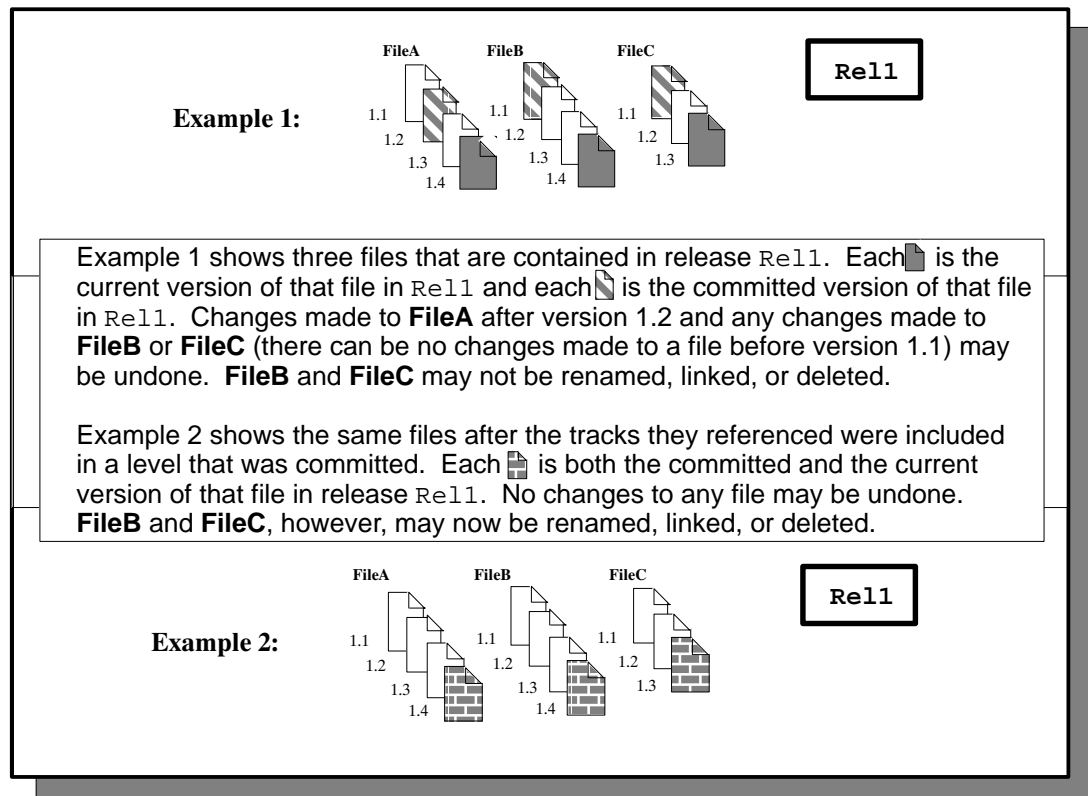


Figure 25. File changes in releases that include the track subprocess



---

## Chapter 6. Using Defects and Features

CMVC problem tracking monitors all reported problems and retains information about their life cycle in the database on the CMVC server. This chapter describes the problem tracking process as it relates to both defects and features, how configured processes affect problem tracking, and how CMVC monitors all reported defects and features.

---

### What Are Defects and Features?

#### What Is a Defect?

Defects monitor and record information about problems. Defects may refer to problems not related to the files under CMVC control. For example, defects can record information about personnel problems, hardware problems, and process problems, as well as problems that are found in products being developed under CMVC control.

#### What is a Feature?

Features monitor and record information about proposed design changes. Design changes proposed using a CMVC feature do not need to be related to files under CMVC control. For example, features can record proposals for process improvements and hardware design changes, as well as proposals for design changes in products being developed under CMVC control.

### Defect and Feature Attributes

All CMVC defects and features have the following attributes:

- Name
  - Each defect and feature has a unique name created when you open it.
- Component
  - Each defect and feature is opened against a single component for evaluation, management, and resolution or implementation.
- State
  - Defects and features move through different states during their life cycles. Valid states are open, design, size, review, working, verify, closed, returned, and canceled.
- Originator
  - The user who opened the defect or feature. This person is responsible for verifying the defect resolution or feature resolution.
- Owner
  - The user responsible for managing the defect resolution or feature resolution.
- Remarks
  - Information describing the defect or feature.
- Prefix
  - A code for specifying different types of defects or features.
- Age
  - The elapsed time that the defect or feature has been active. The aging mechanism can be configured by the family administrator.

- Reference
  - An optionally assigned value used to group related defects and features.
- Abstract
  - A short description of the defect or feature. This defaults to the first 63 characters of the remarks if no abstract is recorded.

### Attributes Specific to Defects

Since defects describe errors found in existing files (or other materials related to a family's components), defects have extra attributes to contain the additional information available.

- Answer
  - The answer to the defect. This field is used by the defect owner when accepting the defect for resolution or when returning a defect.
- Environment
  - An optional indication of the environment in which the problem was discovered.
- Release
  - Each defect can optionally have a release specified. When the track for this release moves to the complete state then the defect automatically moves to the verify state.
- Severity
  - The estimated severity of the reported problem.

### Optional Attributes


CMVC allows the family administrator to configure features and defects to have additional attributes. Several suggested attributes are provided by CMVC:

#### For Defects and Features

- Priority
  - An indication of the relative importance of the defect or feature.
- Target
  - An indication of when the defect will be resolved or the feature will be implemented.

#### For Defects Only

- Phase Found
  - The development phase in progress when the problem was discovered.
- Phase Injected
  - The development phase in progress when the defect was reported (injected).
- Symptom
  - An indication of the symptoms of the problem.

 For information about how to configure additional attributes, refer to *IBM CMVC Server Administration and Installation*.

## Opening Defects and Features

Each problem or design change is reported by opening a CMVC defect or feature. Details about the defect or feature are recorded by the user who opens it. The user who opens a defect or feature is the *originator* of that defect or feature. Each defect or feature must be reported to a component within the component hierarchy. This designates responsibility for the defect or feature to the component owner. The owner of the component becomes the defect or feature *owner* by default.

## Analyzing Defects and Features

The owner is responsible for analyzing a defect or feature once it is opened. He or she can then return it if it is not valid or feasible, reassign it to another user, or accept it for resolution. The owner can return a defect or feature for any reason. Only the originator can cancel a defect or feature. If a defect or feature is returned, then the originator may want to record more information about the problem or enhancement and then reopen it. If the defect or feature does not relate to the component to which it was reported, the owner can reassign it to a more appropriate component within the hierarchy.

This process of clarifying the defect or feature and determining who should accept it may involve reopening and returning the defect or feature many of times. Accepting a defect or feature implies the responsibility to resolve the defect or implement the feature. Not all defects will be accepted since some problems may turn out to be invalid (such as user error) and may be canceled by the originator. Not all features will be accepted since some proposals will not be feasible and may be canceled by the originator.

## Designing the Resolution

Once a defect or feature has been accepted for consideration, the actual resolution needs to be designed so that an informed evaluation can be made. This resolution needs to be designed by users who are familiar with the product or area affected by the defect or feature. Design text is recorded within the defect or feature and can be supplemented by other users until a complete resolution design exists. No resolution can be made unless design text has been recorded.

## Identifying the Required Resources

The design text identifies the resources required to resolve the defect or implement the feature. If a product being developed under CMVC control is affected, then the releases that contain affected files need to be identified, as do the components that manage those files. Sizing records are created by the owner to identify the components and releases that may be affected. Each component-release pair is identified by one sizing record.

Each owner of a component referenced in a sizing record needs to evaluate the impact of the defect or feature on the files managed by that component. If the files under the management of a component are affected by the defect or feature, then this is recorded by accepting the sizing record and adding sizing information. If the files are not affected, then this is recorded by rejecting the sizing record.

Figure 26 shows all the state transitions that features and defects with all subprocesses included in the component process can make.

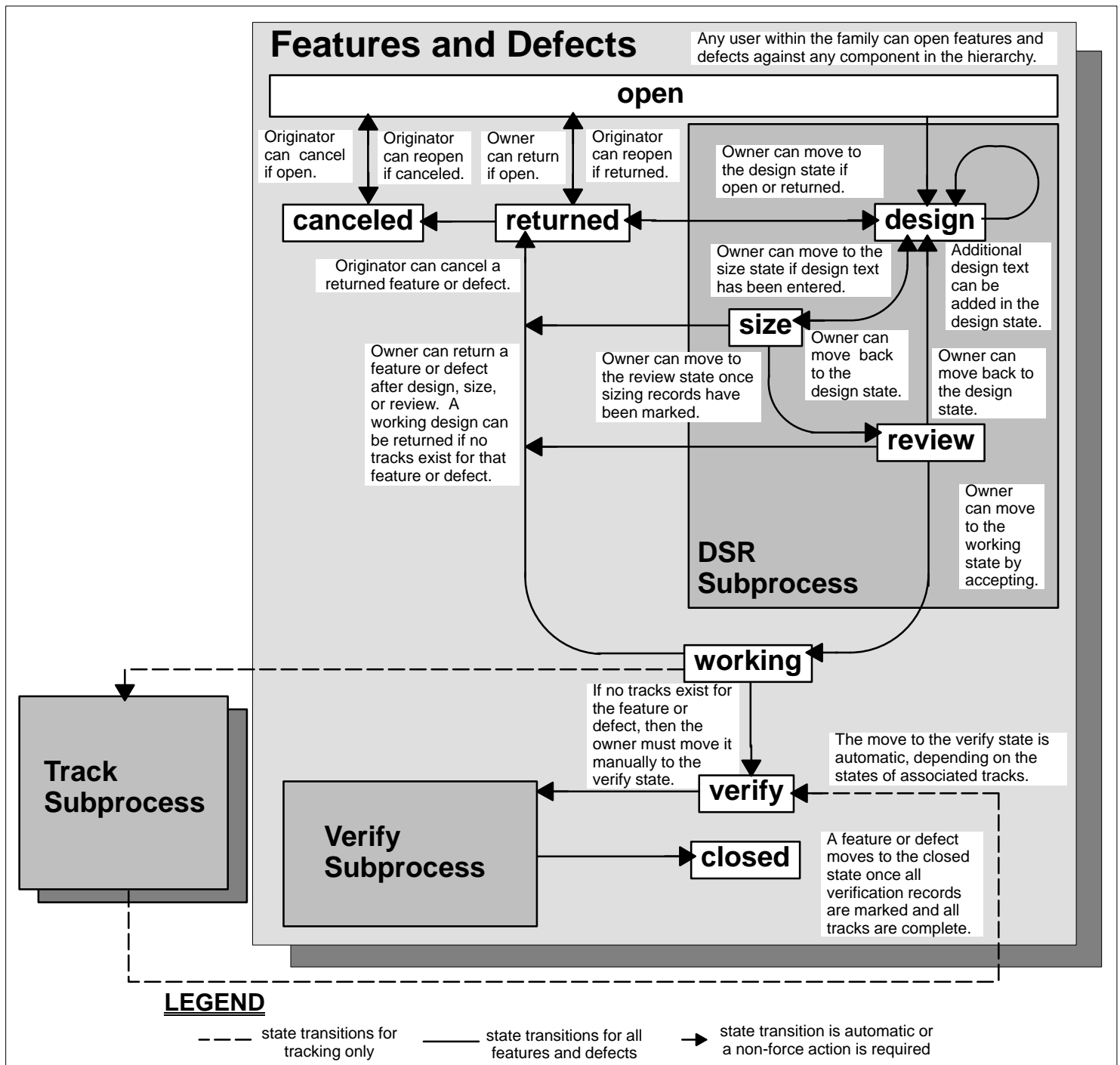


Figure 26. Feature and defect state diagram with all subprocesses configured

For a detailed description of all feature and defect state transitions see Appendix A: "The States of CMVC Objects".



## Reviewing the Design and Resource Estimates

Once the resolution has been designed and the resources have been identified, the proposal needs to be reviewed. At this time the need for additional design work may be identified. Features and defects that are not feasible can be returned to the originator. Any returned defect or feature can be reopened, if necessary, by the originator.

One defect or feature may require changes in more than one release. Before file changes are made to resolve a defect or implement a feature, the releases which are affected must be identified. These releases are identified by sizing records that have been accepted during the sizing stage. A track for each of these releases is created automatically when the defect or feature is accepted. Tracks are only created for releases that include the track subprocess.

For defects and features that do not affect files under CMVC control, you can add notes to them to indicate the work required to resolve the problem or implement the design change.


Accepting a defect or feature indicates an intention to resolve the defect or implement the feature.

## Resolving Defects and Implementing Features

A track provides a mechanism to monitor file changes required for the resolution of one defect or feature in one release. Files contained in a release under tracking can only be changed by specifying the track that is monitoring those changes. The changes made to a file in a tracked release are specifically linked to a defect or feature by this reference to a track. Once a track has been created it moves through successive states which both indicate and control the type of work being done.

Resolving one defect or implementing one feature in one release may involve one or more users changing many files. To change a file, a user must check out the file from the CMVC server, make the changes required to resolve the problem or implement the design change, and check in the file to the CMVC server. Files can be checked out at any time and are checked in with reference to the track monitoring the defect or feature. A single file may be changed many times by many users before the feature is implemented or the defect is resolved. All the file changes made for one defect or one feature within one release are monitored by a single track.

Resolving a defect or feature also involves integrating the files changed for that problem or enhancement with changes made for other defects and features in that release. These changes also must be integrated with the unchanged files within the release.

 Using a track to monitor file changes and the integration of those changes in the releases they affect is described in detail in Chapter 7, "Using Tracks".

## Verifying the Resolution of the Defect or the Feature

If the reported defect or feature does not involve tracks, then it can be verified in two steps:

1. The owner of the defect or feature indicates that the resolution or implementation is correct by moving the defect or feature to the verify state.
2. The originator indicates that the defect is resolved or the feature is implemented in the defect or feature's *verification record*. This automatically closes the defect or feature.

Once the owner of the defect or feature moves it to the verify state, it is up to the originator to record verification that the defect or feature was resolved or implemented satisfactorily.

If tracks were created, the defect or feature will move automatically to the verify state in one of two ways.

- If a release is specified for a defect, the defect will change from the working state to the verify state when the track associated with that release is completed.
- Defects with no release specified and all features will move from the working state to the verify state once the first track associated with the defect or feature is completed.

The originator uses a verification record to record satisfaction or dissatisfaction with a resolution or implementation. If a duplicate defect or feature exists for the defect or feature, then a verification record for the originator of the duplicate defect or feature is automatically created when this defect or feature is accepted.

If the originator decides that the problem has not been resolved or the enhancement has not been implemented correctly, then a new defect or feature can be opened. The original defect or feature will close automatically once the originator and the originators of any duplicates have recorded concurrence or non-concurrence and all tracks created for the defect or feature are complete.

---

## Responsibilities of the Originator

When a defect is found or a design change is proposed, a defect or feature is opened against a component. The user who opens the defect or feature is the *originator* of the defect or feature. The originator is responsible for recording information about the problem or enhancement so that the owner can analyze it completely and thoroughly plan a design. If the defect or feature is not described clearly it may be returned. A returned defect or feature is the responsibility of the originator and can be reopened against the same component, opened against a different component, or canceled. If it is reopened against the same component, more information about the nature of the problem or enhancement is probably required before the owner will accept it.

The originator is responsible for verifying the defect or feature once it is resolved. A different user can be assigned as originator if necessary. If a verification record for the defect or feature already exists, it can also be assigned to the new originator.

---

## Responsibilities of the Owner

Every defect and feature is opened against a component. The owner of that component automatically becomes the owner. The owner is responsible for managing the resolution and must therefore be familiar with the area where the problem was found.

Ownership can be reassigned. If a defect or feature is opened against a component and the owner does not feel that it pertains to that component, then ownership can be reassigned to a different component. This will cause the problem ownership to automatically switch to the owner of the new component. You can also reassign the ownership to a different user without reassigning the problem to a different component.

---

## Changing Component Processes

Users with sufficient authority, such as component owners, can modify components to use different processes under certain conditions. If the old process included the defect (or feature) DSR subprocess but the new one does not, then no defects (or features) opened against the component can be in the design, size, or review states. If the old process included the defect (or feature) verify subprocess but the new one does not, then no defects (or features) opened against the component can be in the verify state.

---

## Chapter 7. Using Tracks

The integration of problem tracking and change control provides a systematic approach to track the file changes required to resolve a reported problem or implement a proposed enhancement. This chapter describes configuring your change control process for each release, working with CMVC tracks, working with CMVC levels, and the interaction between tracks, levels and the other CMVC objects within a family.

Defects and features record information about the life cycle of reported problems and enhancements. Tracks monitor the resolution of those defects and features in releases that include the track subprocess.

---

### What is a Track?

A track is a CMVC object used to monitor the resolution of a defect or the implementation of a feature within a release that includes the track subprocess.

Tracks are created when a defect or feature is in the working state. Since a single defect or feature may affect multiple releases, a separate track must be created for each tracked release in which file changes are required. Tracks are created automatically for any releases specified by an accepted sizing record.

Every track must be created in reference to one defect or feature and one release. This pair of release and defect or feature uniquely identifies the track within your family. By default, the track is owned by its creator. If the track was created automatically then the initial track owner is the owner of the release that the track references.

### Track Attributes

A CMVC track has the following attributes:

- Name
  - Each track has a name that corresponds to the name of the defect or feature that the track is monitoring.
- Release
  - The release in which the track is monitoring a defect or feature resolution.
- Prefix
  - The track prefix is that of the respective defect or feature.
- State
  - Each track moves through different states during its life cycle. Valid states are approve, fix, integrate, commit, test, and complete.
- Owner
  - Each track has an owner.
- Abstract
  - The track abstract is that of the respective defect or feature, and is a summary of the defect or feature.

- Level
  - The level in which the track is committed. This value is updated by CMVC, and will be blank if the track is committed without a level.
- Target
  - Each track can optionally have a target for the resolution of the defect or implementation of the feature.
- Reference
  - The track reference is that of the respective defect or feature.

---

## Configuring Your Change Control Process

Tracks provide a mechanism for managing the stages involved in the development of a release. The change control requirements for each release within your family may differ. The release process can be configured by the family administrator to include any combination of the following four subprocesses:

- The approval subprocess
  - This subprocess requires all proposed file changes in a release to be approved before the files are actually changed.
- The fix subprocess
  - During the fix subprocess, files that need to be changed for a specific defect or feature are identified with fix records.
- The level subprocess
  - This subprocess integrates file changes with all unchanged files in the release. Building, compilation, and integration testing takes place during integration.
- The test subprocess
  - This subprocess requires formal testing of a newly updated release in environments specified by the environment list.

### Background

**Components** provide organization and control of development data.

**Releases** contain files for product-related activities.

**Defects** are used to record information about reported problems.

**Features** are used to record information about proposed enhancements.

---

## Working With Tracks

Tracks monitor the resolution of a defect or feature in a specific release. The track moves through successive states which both control and indicate the type of work being done.

A track provides a mechanism to control file changes and the incorporation of those changes into each affected release. When a track is first created, it starts in one of two initial states: approve or fix. If a track's release has the approval subprocess configured then the initial state of the track is the approve state and the approval subprocess begins. If the approval subprocess is not configured, then the initial state of the track is the fix state and, if the fix subprocess is configured, the fix subprocess can begin.

## The Approval Subprocess

When a project is first being developed, tight control of all changes may not be needed; however, a development team getting close to a deadline may need a checkpoint in their change control process. The approval subprocess provides a checkpoint to control which defects and features are resolved in a release.

In a release implementing the approval subprocess, approval must be given for proposed changes before work can begin on the resolution of a defect or the implementation of a feature. The users specified in the approver list for this release need to review the information recorded in the defect or feature and evaluate the changes proposed to the release in relation to other project considerations (such as the development schedule or resources required). An *approval record* is created for each approver when the track is created. Each approver indicates his or her evaluation of the changes on their approval record. CMVC will not accept any file changes for that defect or feature within that release until all approvers accept the proposed changes.

A release that has one or more tracks in the approve state cannot be modified to use a different process if the old process includes the approval subprocess but the new one does not.

## Working with the Approver List

An approver list is a list of users who must approve changes to any files contained in the release. Each tracked release with the approval subprocess configured has an approval list. If the approval subprocess is configured as part of the current process, each user on this list will be issued an approval record for each track that is created for the release.

The approval list can be modified at any time by adding and deleting users. If the approval subprocess is configured as part of the current process then there must be at least one user on the approval list. Modifying an approver list does not affect approval records or tracks that already exist for that release.

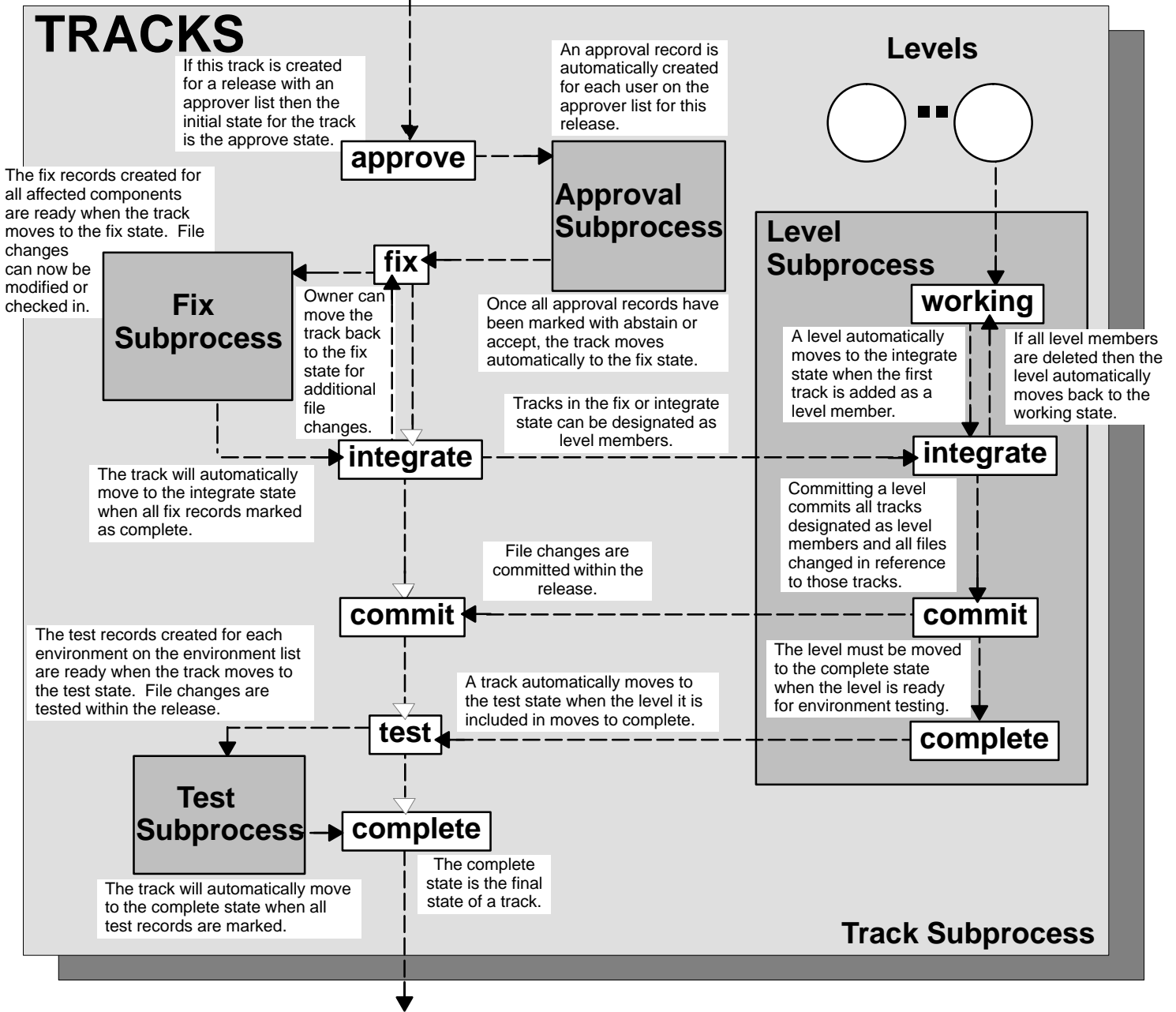
## The Fix Subprocess

During the fix state of the track, files to be changed are identified. Then changes are made and the files are tested before being checked into the CMVC server. The fix subprocess is a way of informing CMVC and users that the files changed for the defect or feature are ready for integration.

Resolving a defect or feature within a release may affect multiple files managed by more than one component. A *fix record* is used to monitor the file changes within a single component. Fix records provide a controlled mechanism for reviewing all file changes on a component basis, before allowing those changes to be integrated with changes made for other defects and features.

Fix records can be created explicitly; for example, a defect owner may create the necessary tracks and fix records for the defect at the same time. Fix records are created automatically for features or defects according to the accepted sizing records. If a fix record does not already exist for the component then one will be created automatically when a file managed by that component is modified or checked in to the CMVC server. Each fix record is initially owned by the owner of the component; this ownership can be reassigned if necessary.

A track is created automatically for every release referenced by an accepted sizing record or can be created manually to track the resolution of a defect or feature in a specific release.



When a track associated with a defect or feature is complete, the defect or feature moves to the next state after the working state.

**LEGEND**

--- state transitions for tracking

→ force action is required

→ state transition is automatic or a non-force action is required

Figure 27. Track and level state diagram with all subprocesses configured.

For a detailed description of track and level state transitions see Appendix A: “The States of CMVC Objects”.

A file can be checked out at any time. However, if the fix subprocess is included in the release process, then before a file can be modified or checked in to CMVC, the following conditions must be met:

- A defect or feature requiring the file changes must exist.
- The track (for that defect or feature and the release that contains the file you have changed) must be in the fix state.
- If a fix record exists for the component that manages this file, it must be in the ready or active state.

When all necessary file changes within the specified component have been made, these changes can be reviewed or inspected. The fix record owner is responsible for this review. When the fix record owner is satisfied that the file changes made within that component are complete and ready for integration with other files in the release, the fix record is marked complete. When all existing fix records for a track are complete, the track is ready for integration. If the fix subprocess is not included, fix records will not be created and the track must be moved to the integrate state explicitly.

A release that has one or more tracks with fix records marked ready, not ready, or active cannot be modified to use a different process if the old process includes the fix subprocess but the new one does not.

---

## Completing the Tracking Process

Once a level or track is committed, the formal environment test subprocess can begin. After a committed level has been distributed to the appropriate testers, the level subprocess is ready to be completed. Completing the level makes the committed level available for formal testing by activating the test subprocess.

If the release associated with the track does not include the level subprocess, then committing the tracks and distributing the collection of file changes for the release makes the track available for formal testing by activating the test subprocess.

If the test subprocess is not included in the release's process, then the tracks which were included in the level automatically move to the complete state. Tracks committed without levels will automatically move to the complete state.

## The Test Subprocess

Formal testing of a release may not be required for all stages of your development process. For defects and features that require file changes in only one release, the verification record for each defect or feature may provide an adequate testing focus. However, when tracking a problem resolution or feature implementation across multiple releases and release environments, an additional formal test process is often needed. The test process provides each release with a configurable set of environments and testers for formal testing.

Once a level has been completed, or a track has been committed, the release is ready to be tested formally against the environments specified in the environment list.

In a release implementing the test subprocess, testing results must be recorded by the specified tester for the related environment. A *test record* is automatically created for each tester when a track is created for this release. Each tester must test the newly updated release against the specified environment and record the results. If the defect or feature related to this track was not resolved in the test environment then the test record is rejected and a new defect or feature should be opened outlining the remaining problems. If the defect or feature was resolved in this environment then the tester should accept the test record. An abstain option is also available.

Once all test records created for a track have been marked with test results, the track automatically moves to the complete state.

A release that has one or more tracks in the test state cannot be modified to use a different process if the old process includes the test subprocess but the new one does not.

### **Working with the Environment List**

An environment list contains an entry for each environment in which a specified release needs to be tested and can be defined as needed. Environment lists can include such aspects as hardware, operating system, and related products. Each list entry must include the environment and the user who is responsible for testing that environment. Each user specified as a tester on the release environment list will be issued a test record for each track that is created for the release if the test subprocess is included in the current process.

The environment list can be modified by adding or deleting environments associated with the release. If the test subprocess is configured for that release then there must be at least one user on the environment list. Modifying an environment list does not affect test records or tracks that already exist for that release.

### **After the Track Subprocess**

If no release is specified for a defect or feature, it will move from the defect or feature working state to the verify state once the first track associated with it is completed. The defect or feature can then be verified. However, if a release is specified for a defect, the defect will change from the working state to the verify state when the track associated with that release is completed. The defect can then be verified. For more information about verifying defects and features see “Verifying the Resolution of the Defect or the Feature” on page 45.

---

## **Responsibilities of a Track Owner**

Each track is created when the associated defect or feature is in the working state. The track owner defaults to the track creator if no owner is specified when the track is created. Track ownership can be reassigned if necessary.

The track owner can move the track from one state to another if the automatic state changes need to be circumvented. If additional file changes are needed after a track has moved to the integrate state, the track owner can move the track back to the fix state. The track owner may also cancel the track if there are no active file changes associated with it.

---

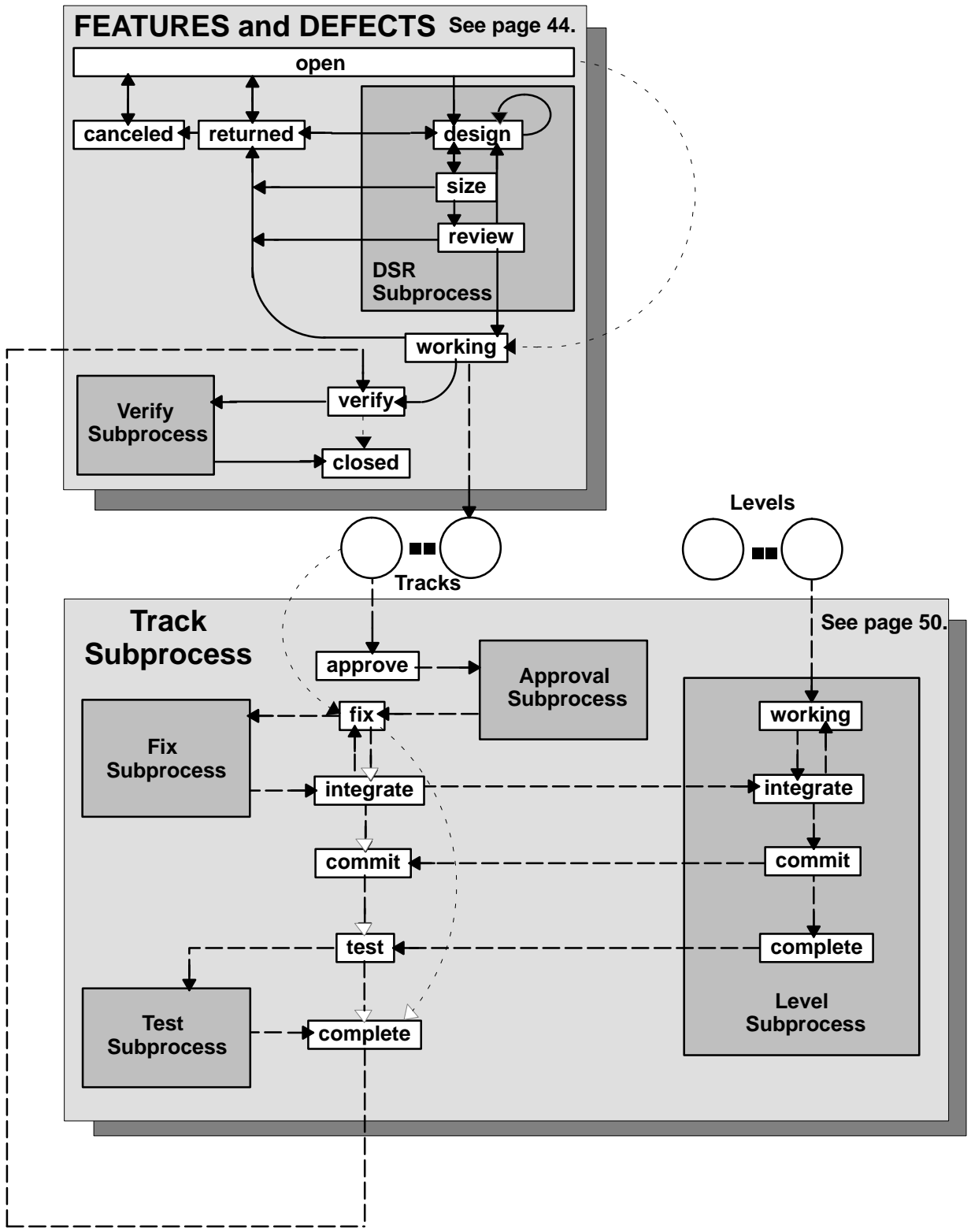
## **The Track States**

Each track moves through different states during its life cycle. Different CMVC actions can be performed against a track depending on its state. Different users will be involved in the defect or feature resolution at separate points in the track life cycle. The state diagram in Figure 27 shows the interaction of the different states and describes the movement from one state to another. A detailed description of the track states and state transitions is provided in Appendix A.

Integrating problem tracking and change control involves all stages of your development life cycle. Monitoring all stages of your development process requires the interaction of many of the CMVC objects. Figure 28 shows the main interactions between the states of defects, features, tracks, and levels.

Defects and features are used to record information about the life cycle of reported problems and enhancements. Tracks are used to monitor the resolution of those problems and enhancements in releases under tracking and levels are used to integrate those changes within the release.





**LEGEND**

- state transitions for tracked releases only
- state transitions for all changes
- - - - state transitions when subprocesses are not configured
- ⇨ force action is required
- ➔ state transition is automatic or a non-force action is required

Figure 28. CMVC State Diagram



---

## Chapter 8. Using Levels

Resolving a selected set of problems for a specific release involves integrating the files changed for those problems with each other and with the unchanged files in the release. CMVC provides *levels* to facilitate this. The tracks that monitor the file changes that you wish to integrate within the release are added as members to the level. This allows all file changes made to resolve a selected set of problems to be represented in one level. Levels also allow you to reconstruct sets of files as they were at the time that the level was committed.

---

### What is a Level?

A level is a CMVC object used to monitor and implement the integration of file changes within a release. The level provides a method for systematically integrating changes into a release under development. Levels can only be created for tracked releases using the level subprocess.


### Level Attributes

A CMVC level has the following attributes:

- Level name
  - Each level name must be unique within the release.
- Release
  - The name of the release for which this level is created.
- Owner
  - The level owner is responsible for managing the level subprocess. The level owner is automatically the user who creates the level. This ownership can be re-assigned.
- Type
  - Level types can be configured for your development process. The available types are defined by your family administrator.

### The Level States

Each level moves through four states during its life cycle: working, integrate, commit, and complete. CMVC actions can be performed against a level depending on its state. A detailed description of the level state transitions is provided in Appendix A: “The States of CMVC Objects”.

 Refer to the track and level state diagram (Figure 27 on page 50) to see how level states interact with track states.

---

### The Level Subprocess

At regular intervals during the development of a release, the release can be updated by integrating all changed files with the remaining unchanged files in the release. CMVC levels provide a controlled mechanism which defines and extracts the set of files to be integrated, and then makes the file changes permanent after integration testing is complete. CMVC

does not actually process or compile the file changes. Compilation and integration testing must be done using additional tools after the files are extracted from the CMVC server.

Once a level is committed it can be reproduced at any time by extracting the full file tree of the level. This extracts all of the file changes which were committed in the level as well as the base set of files which were unchanged in the release.

This section discusses the process by which levels are defined, extracted, and committed.

### **Steps in the Level Subprocess**

The level subprocess for a release requires the following steps:

1. Create a new level and add at least one track as a level member
2. Check the level for any existing prerequisite or corequisite track relationships
3. If necessary, extract the level to verify the compilation and the correctness of the file changes.
4. Commit and complete the level

## **Creating New Levels and Adding Tracks As Level Members**

A new level is created by assigning a new level name to the release. Level names must be unique within a release. Tracks are added as level members once a level is created. Adding a track as a level member includes the file changes monitored by that track in the level. A track in the integrate or fix state can be added as a level member; however, a level can not be committed unless all of its member tracks are in the integrate state. This allows a level to include a partial fix for testing purposes without allowing a partial fix to be committed in the level.

Once you have defined the new level, check for any existing prerequisite or corequisite track relationships. If any are found, you will need to either remove some of the level members or add tracks to the level in order to resolve the prerequisite and corequisite requirements. When there are no remaining prerequisites or corequisites, the files can be extracted in preparation for compilation.

## **Prerequisite and Corequisite Checks**

Before extracting and compiling all the files in a level, you should perform *prerequisite* and *corequisite* checks on the level. You will probably want to resolve any existing prerequisite and corequisite relationships amongst the files before beginning compilation. The CMVC server automatically checks the level for prerequisites and corequisites between all level members before committing the level, and will not permit the commit action if any are found.

Prerequisite relationships are monitored automatically by the CMVC server to support file integrity when a given set of files are changed for multiple defects and features. If a file has been changed to resolve more than one defect or feature then the track referenced by the first change is a prerequisite of any tracks referenced by the later changes. A track is a prerequisite to another track if:

1. file changes have been checked in, but not committed, in reference to the first track.
2. one or more of those same files is then checked out, changed, and checked in again in reference to the second track.

For example,

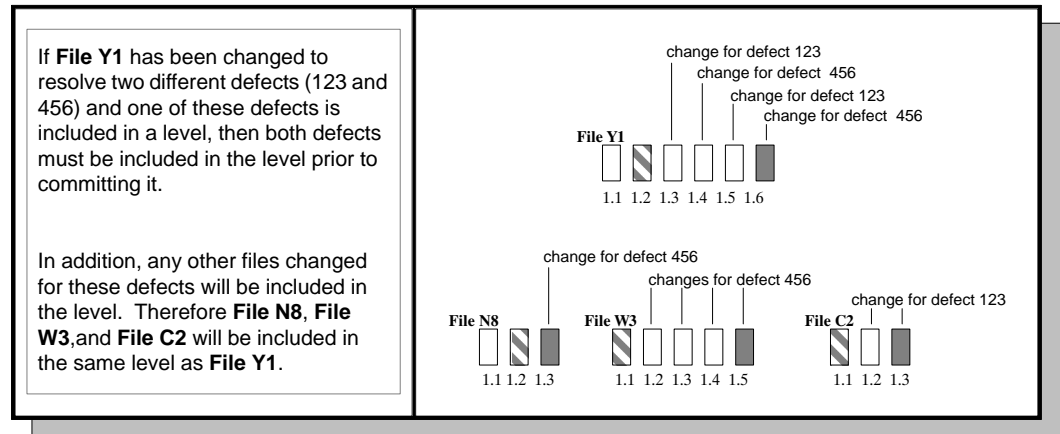


Figure 29. Example of prerequisite file changes in a level

Two or more tracks can be explicitly designated as corequisites so that all tracks in the corequisite group must be included as members in the same level. If a track is added to a level then all tracks that have a corequisite relationship with that track must also be included in the same level before the level is committed.

Checking a level for existing prerequisites and corequisites returns a list of tracks which have not been defined as members of the level but are either a prerequisite or a corequisite for one or more tracks in the level.

Corequisite and prerequisite checking are not made in releases that do not include the level subprocess. This means that in the above example, the changes to **File Y1** for defect 456 can be committed (at version 1.6) before the changes for defect 123 are committed (at version 1.5). If the changes for defect 123 are committed after the changes for defect 456, **File Y1** will still be committed at version 1.6.

## Making Changes to Files Included In a Level

If the fix subprocess is configured, then changes can only be made to files when the associated track is in the fix state and the associated fix record is in the ready or active state. If additional changes are identified when the track is still in the fix state but the required fix record is completed, the fix record needs to be moved back to the active state. If additional changes are identified when the track is in the integrate state, the track owner needs to move the track back to the fix state and the fix records for each component in which changes are needed must then be moved back to the active state.

If the track has already been included in a level when additional changes are identified, the track owner must:

1. Remove the level member associated with the track from the level.
2. Change the track from the integrate state to the fix state.
3. Change the affected fix records back to the active state. Respective fix owners can accomplish this.

The track owner can then make the required file changes and check them in to the CMVC server.

## Committing and Completing a Level

### Committing a Level

When integration testing of a new level is complete, the release can be updated by committing the level. Committing a level commits all tracks that were designated as level members, and all file changes included in those tracks. When the changes in a level are committed, they establish a new baseline for subsequent development of the release.

Commit a level when you are ready to finalize all the file changes included in the level. If any outstanding prerequisite or corequisite track relationships exist for the level, it cannot be committed. When a level is committed, all tracks which are level members change from the integrate state to the commit state, and the file changes represented by those tracks become permanent.

A single track can be a member of more than one level. If one of the levels including this track is committed, the state of the track will change to *commit*. Other levels which include this track will ignore the track.

The only alterations that can be made to a committed level are changing its type and changing its owner. If you discover a defect in a committed level, you must open a new defect to make additional file changes to a new level.

The committed level can be extracted to produce a new base file tree, against which the next set of changes will be applied. The full file tree can be extracted at any time for any previously committed level.

### Completing a Level

The final step in the level subprocess is to complete the committed level. Completing a level activates the formal test subprocess by changing all the tracks in the level from the commit state to the test state. You should complete a committed level after the updated release has been distributed to the appropriate testers or test groups.

---

## Extracting a Level

Activities associated with extracting levels include:

- Extracting the most recent committed version of all files in the release and new levels
- Combining extracted file trees to produce an updated full file tree for the release
- Compiling and testing the full file system using additional tools outside CMVC

## Extracting File Trees

Extracting files from the CMVC server copies the files from the server to a specified NFS server. When extracting a level the destination of the file tree can be specified.

When using levels, there are two steps required to update the release:

1. Extract the most recently committed version of all files in the release either by extracting the release, or by extracting the last committed level for the release.
  - Extracting this set of files copies the committed versions of all files contained in the release to the designated host's file system. The resulting file tree is the *full file tree* for that release.

2. Extract the level representing the set of changed files to the same NFS server.
  - Extracting the level copies all the files changed for the tracks that are included in the level to the designated host's file system. The resulting file tree is the *delta file tree* for that release.

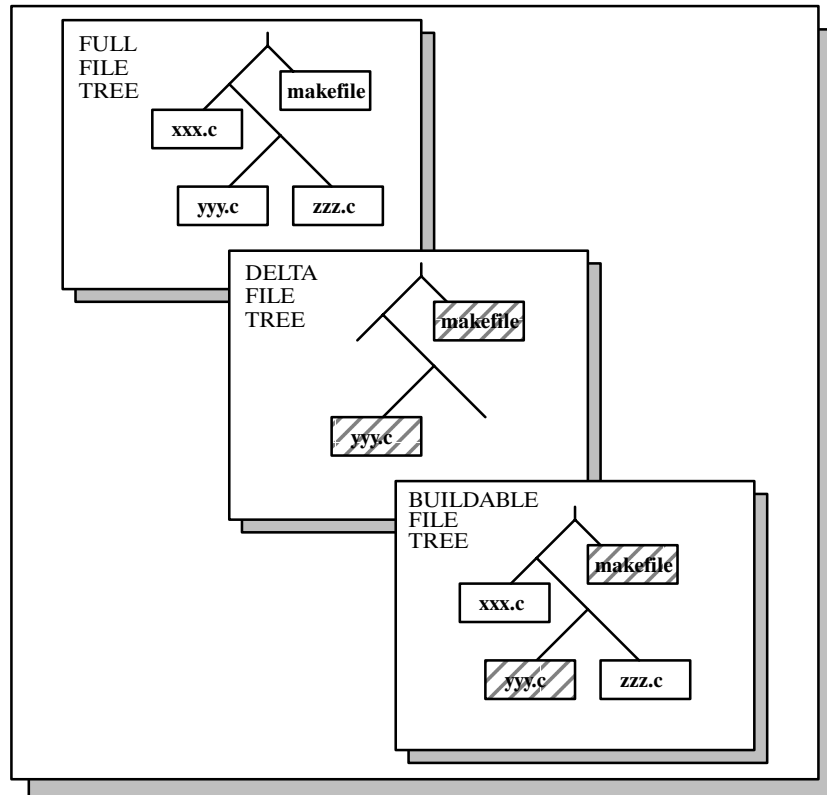


Figure 30. Delta and full file trees

## Combining File Trees

If you extract both the full file tree and the delta file tree to the same directory, this will result in an updated full file tree for the release. Alternatively, if you extract the full file tree and the delta file tree to different directories, you can combine the two extracted file trees by copying the delta file tree onto the full file tree. This will result in an updated full file tree for the release. Note that in either case, if the new level includes renamed or deleted files, then the updated file tree will have some files that need to be removed, since any deleted files will be part of the full file tree, and any renamed files will have been included by their old path name. These files can be deleted from the updated file tree. The CMVC server includes an extra file in a delta file tree extraction that lists the full path names of all files that were deleted or renamed in the level.

Multiple extractions to the same location will overwrite the previously extracted file tree.

## Compiling a File Tree

Compiling the updated file tree for the release must be done using tools outside of CMVC. Once the file tree is successfully compiled, and functional testing of the updated release produces satisfactory results, you can commit the level.

Once the updated file tree is compiled, all integration testing is performed. If errors are discovered during integration testing, selected tracks can be deleted from the level and moved back to the fix state in order to make additional file changes.

---

## Updating a Release with the First Level

When a release is first created, there are two ways of initializing, or creating, all the files for the release. The first option is:

1. Create the release with a process that does not include the track subprocess
2. Create all the files for the release
3. Modify the release to use a process that includes the track subprocess for the release.

The second option is:

1. Create the release with a process that includes the track subprocess
2. Create one feature (or defect) and one track for the release
3. Create all the files for the release in reference to this track.

If you use the second option, you will probably want to create an initial level for the release and include in it the single track associated with creating all the files. In this case, extracting the delta file tree for the level will actually produce a full file tree for the release, which can be compiled for verification before committing the level. Subsequent development can follow the integration method described above.

If you use the first option, then your first level for the release will include a set of tracks associated with changes to the files in the release. These changes may include new versions of the files, renaming some files, creating additional files, and so on. In this case, in order to create the full base file tree, you will need to extract the release using the *committed* option, which extracts the committed version of all files in the release. Although the initial version of all the files in the release were not explicitly committed in a level, the first version of the file is considered the committed version in this case, and will be extracted. The resulting file tree can be updated with the delta tree extracted for the level, and then compiled.

CMVC levels provide snapshots of the release at different points in the release development life cycle. Each committed level can be recreated at a later date by extracting either the full file tree or the delta file tree represented by the level.

Extracting the full file tree will recreate the release as it existed at the time that the level was committed. Extracting the delta file tree will recreate only the files that were changed for that level.

---

## Changing Release Processes

Users with sufficient authority, such as release leaders, can modify releases to use different processes. If one of the processes includes the level subprocess and the other does not, then certain conditions must be met before the release process can be changed.

If the old process includes the level subprocess, but the new one does not, then all of the levels in the release must be in the working state (and therefore have no level members) or in the complete state, and no tracks in the release can be in the integrate state.

If the new process includes the level subprocess, but the old one does not, then none of the tracks in the release can be in the fix state.



---

## Appendix A. The States of CMVC Objects

---

### The States of Features and Defects

Features and defects record information about the life cycle of a problem. Each feature or defect moves through different states during its life cycle. The CMVC actions you can perform against a feature or defect depend on its current state.

☞ The feature and defect state diagram is on page 66.

### The Open State

Any user within a family can open a feature or defect against any component within the family's hierarchy. That user is recorded as the originator. When a feature or defect is first opened it is in the open state and is given a unique name (which may be specified by the creator).

A feature or defect must be created in reference to a component. The owner of this component becomes the feature or defect owner and is responsible for managing the resolution. The component you open a feature or defect against should be one that manages files affected by the enhancement or problem. Deciding which component to open a feature or defect against depends on the component hierarchy created by your organization. Component descriptions and the structure of the hierarchy will help you find the most appropriate component for management of the feature or defect. If a problem is opened against an inappropriate component, the component owner can reassign it.

**Note:** The owner of a feature or defect is the user responsible for its implementation. The originator is the user responsible for verifying the resolution.

### The Returned State

A feature or defect can be returned from the open, design, size, or review state if the owner decides that it is not feasible or not valid. A feature or defect in the working state can only be returned if it is not tracked. Returning a feature or defect moves it to the returned state where the originator can either cancel or reopen it. When you return a feature or defect you should add your reason for returning it so that the originator and any other users can evaluate why you believed it infeasible or invalid.

#### Reopening Features and Defects

An originator can reopen a feature or defect in the returned or canceled state. It cannot be reopened if it is in the closed state. When you reopen a feature or defect you should add your reason for reopening it and clarify the description of the design change or problem. A reopened feature or defect retains both the original name and the managing component it was originally opened against.

### The Canceled State

A feature or defect can be canceled by the originator if it is in the returned or open state. Canceling a feature or defect moves it to the canceled state. When canceled, it is no longer active until it is reopened by the originator.

## APPENDIX A

### The Design State

A feature or defect in the open or returned state can be moved to the design state by the owner if the DSR subprocess is part of the process used by the managing component. In this state, the proposed change is designed and the design text is entered. Design text must be entered before it can move to the size state. Once all design specifications have been documented, the owner moves the feature or defect to the size state.

### The Size State

When a feature or defect is in the size state you can create sizing records for each release that contains files affected by the design, and the components that manage those files. Each component-release pair is identified by one sizing record.

#### Sizing Records

Sizing records identify the work required for and the resources affected by the feature or defect. The owner of the component referenced in the sizing record is automatically the owner of the sizing record. The sizing record owner is responsible for entering information about the approximate amount of work required to implement the feature or resolve the defect in the corresponding component.

The sizing record owner should enter the required information and change the sizing record to the accept state if the feature or defect affects that component. If it does not affect that component then move the sizing record to the reject state.

When all sizing records have been marked, then the feature or defect can be moved to the review state or back to the design state if more design information is needed.

### The Review State

The owner of a feature or defect in the size state can move it to the review state, where the design text and size records are reviewed to determine the feasibility of the proposal. If more information is needed then the feature or defect can be moved back to the design state. After reviewing the recorded information the owner can accept the feature or defect for resolution or return it.

### The Working State

A feature or defect can be accepted by the owner if it is in the review state. Accepting the feature or defect moves it to the working state and implies an intention to resolve it. If the managing component does not use the DSR subprocess then the owner can move the feature or defect to the working state if it is in the returned or open state.

Once you accept a feature or defect you need to identify what releases are affected by it. One feature or defect may require changes in more than one release. These releases were identified during sizing and a track is automatically created for each identified release that uses tracking when the feature or defect is accepted. Each track will monitor the progress of the resolution within one release and follow the change control process configured for the release that it is monitoring.

The feature or defect will automatically move to the verify state (closed state if the feature or defect verify subprocess is not included in the component process) when the first track moves to the complete state. If it has no tracks, then the owner can force the feature or defect to the verify or closed state. Note that a defect associated with a release requires that the track associated with that release is in the complete state before it moves from the working state.

## The Verify State

When a feature or defect is accepted and the verify subprocess is configured, a verification record is created for the originator. The originator cannot use the verification record to verify the implementation until the feature or defect moves to the verify state.

A feature or defect in the working state can be moved to the verify state by the owner if no tracks were created for it. If tracks were created, the feature or defect will automatically move to the verify state when a track is completed. Note that a defect associated with a release requires that the track associated with that release is in the complete state before it moves to the verify state.

### Verification Record

To ensure that the feature or defect is resolved to the originator's satisfaction, it cannot be closed until the originator moves a verification record to the accept, reject, or abstain state. As the originator, if you are not satisfied with the resolution, move the verification record to the reject state. If you are satisfied with the resolution, move the verification record to the accept state. If you are indifferent or are unable to assess the resolution, move the verification record to the abstain state.

Once in the verify state a feature or defect cannot return to the working state. If you believe the resolution to be incorrect, record your dissatisfaction by moving the verification record to reject and open a feature or defect to propose the changes needed. In this new feature or defect you can reference the feature or defect originally opened to resolve the problem or enhancement.


## The Closed State

Once all verification records have been moved to the accept, reject, or abstain state and all tracks have moved to the complete state then the corresponding feature or defect is automatically closed. A feature or defect in the closed state cannot be reopened. If it was not resolved correctly then a new feature or defect must be opened to address the changes needed.

---

## The States of a Track

Features and defects record information about the life cycle of design changes and reported problems. Tracks monitor the resolution of those features and defects in releases that include the track subprocess.

 The track state diagrams are on pages 68 and 70.

## The Approve State

When a track is created its initial state is approve if the approval subprocess is part of the process used by the track's release. When the track enters the approve state, an approval record is issued for each user on the approver list of the release associated with the track. The track will stay in this state until all approval records are marked with accept or abstain.

## The Fix State

While the track is in the fix state, file changes for the resolution of the feature or defect are made and checked in to the CMVC server. Any existing fix records for the track move to the ready state when the track moves to the fix state. A fix record will move to the active state when a file managed by the associated component is checked in or modified with reference to this track. A fix record will be created automatically if a file for which there is no

## APPENDIX A

existing fix record is checked in or modified. When the file changes managed by each component are completed and tested, the associated fix record is moved to the complete state by the fix record owner. The track will automatically move to the integrate state when all fix records are complete and if the level subprocess is included in the release's process. Otherwise, the track will automatically move to the next state governed by the release's process. When the fix subprocess is not included, fix records will not be created. When all file changes are complete, the track must be moved to the integrate state explicitly.

A track can be explicitly moved from the integrate state back to the fix state. Additional file changes can then be made if the necessary fix records are also moved back to the active state. A track that is a member of a level cannot be moved back to the fix state until it is removed from the level.

### The Integrate State

If the fix and level subprocesses are configured, the track moves automatically to the integrate state when all fix records are complete. The track owner can force a track to the integrate state if necessary, provided no file changes are associated with the track. Tracks in the integrate state can be added to an existing level as level members if the level subprocess is configured. All tracks in the integrate state do not have to be added to the same level.

If the level subprocess is configured, then levels can be created at any time. Each level will automatically move to the integrate state when the first track is added as a level member. If all tracks are removed from the level then the level will automatically move back to the working state. Committing a level will commit all tracks included as level members and all files changed in reference to those tracks.

The track will stay in the integrate state until the level in which it is a member is committed. The track owner can force a track to the commit state, provided that no file changes are associated with the track. If the release process does not include the level subprocess, the integrated track moves to the test state if the test subprocess is included, or the complete state if it is not included.

### The Commit State

The track moves automatically to the commit state when the level to which it belongs is committed. At this point all files changed for the resolution of the feature or defect in this release have been committed. The track stays in the commit state until the level to which it belongs is completed. If the level subprocess is not configured, the track can be committed explicitly.

If the level process is configured then the level is moved explicitly to the complete state when it is ready for environment testing.

### The Test State

When the associated level is moved to the complete state or when a track is committed without a level, the track will move to the test state if the release being monitored by this track has the test subprocess configured. The level is ready for formal testing in the specified environments. Any existing test records for the track will move to the ready state when the track moves to the test state. The track will stay in the test state until all test records are moved from the ready state. Each tester can mark his or her test record with accept, reject or abstain.

The track will automatically move to the complete state if the test subprocess is not configured or when all test records are marked.

## The Complete State

The complete state is the final state of a track. If the test subprocess is not included in the release process, the track will move directly to the complete state when the associated level is completed or when the track is forced to be committed.

When a track is complete the feature or defect it was monitoring moves automatically to the verify or complete state. If a defect is associated with a release then it will not leave the working state until the track for that release is complete. The feature or defect is ready to be verified by the originator.

---

## The States of a Level

Levels monitor and implement the integration of file changes within a release. Those file changes are included in a level by adding the tracks referenced by the changed files to the level as level members.

 The level state diagram is on page 68.

## The Working State

The initial state of a level is the working state. While the level is in the working state it is not associated with any tracks and therefore contains no file changes. Creating a level assigns a name and owner to the level.

## The Integrate State

Tracks can be added to levels as level members if the level is in the working or integrate state and the track is in the fix or integrate state. As soon as the first level member is added, the level automatically moves to the integrate state.

The level can be extracted when it is in the integrate state. This will copy all files changed in reference to any level members, to a designated NFS server. When the level is in the integrate state only a *delta file tree* can be extracted. A delta file tree is the file structure of the changed files within the release that are associated with level members for this level.

## The Commit State

Committing a level changes the state of the level to commit as well as the state of all tracks included as level members. A committed level can be extracted for a *full file tree* as well as a delta file tree. A full file tree is the file structure of all the files within the release. The full file tree can then be tested, distributed, or compiled as required.

When a level moves to the commit state all tracks that are included as level members move to the commit state. When a track is in the commit state all file changes associated with the track become permanent and can be extracted with the other files in the release by extracting the committed version of the release. All files within the release including the changed files can be extracted once the level has been committed.

## The Complete State

Once your level has been committed you are ready for formal environment testing. Move the level to the complete state. This automatically moves all tracks included as level members to the test state.

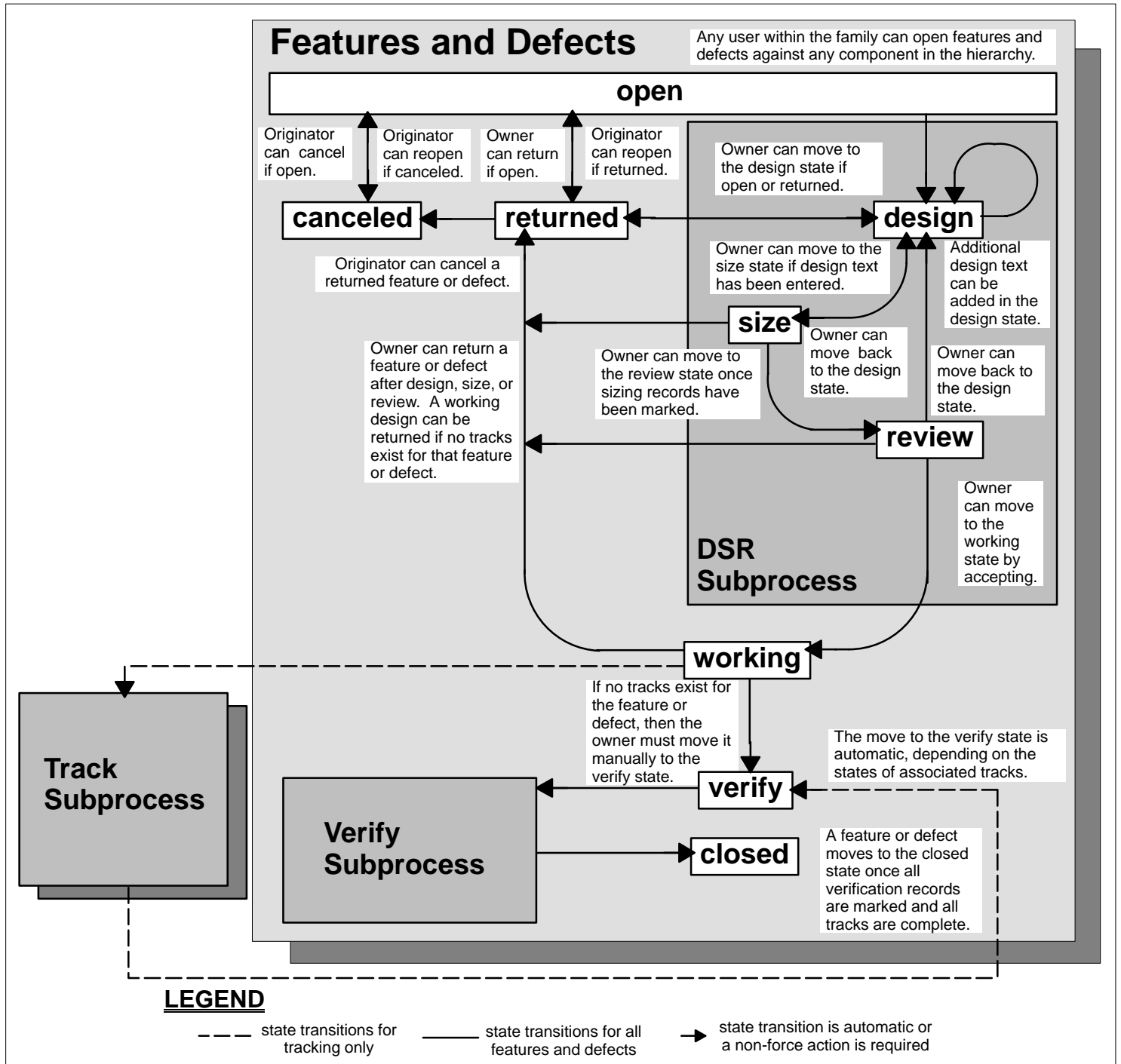


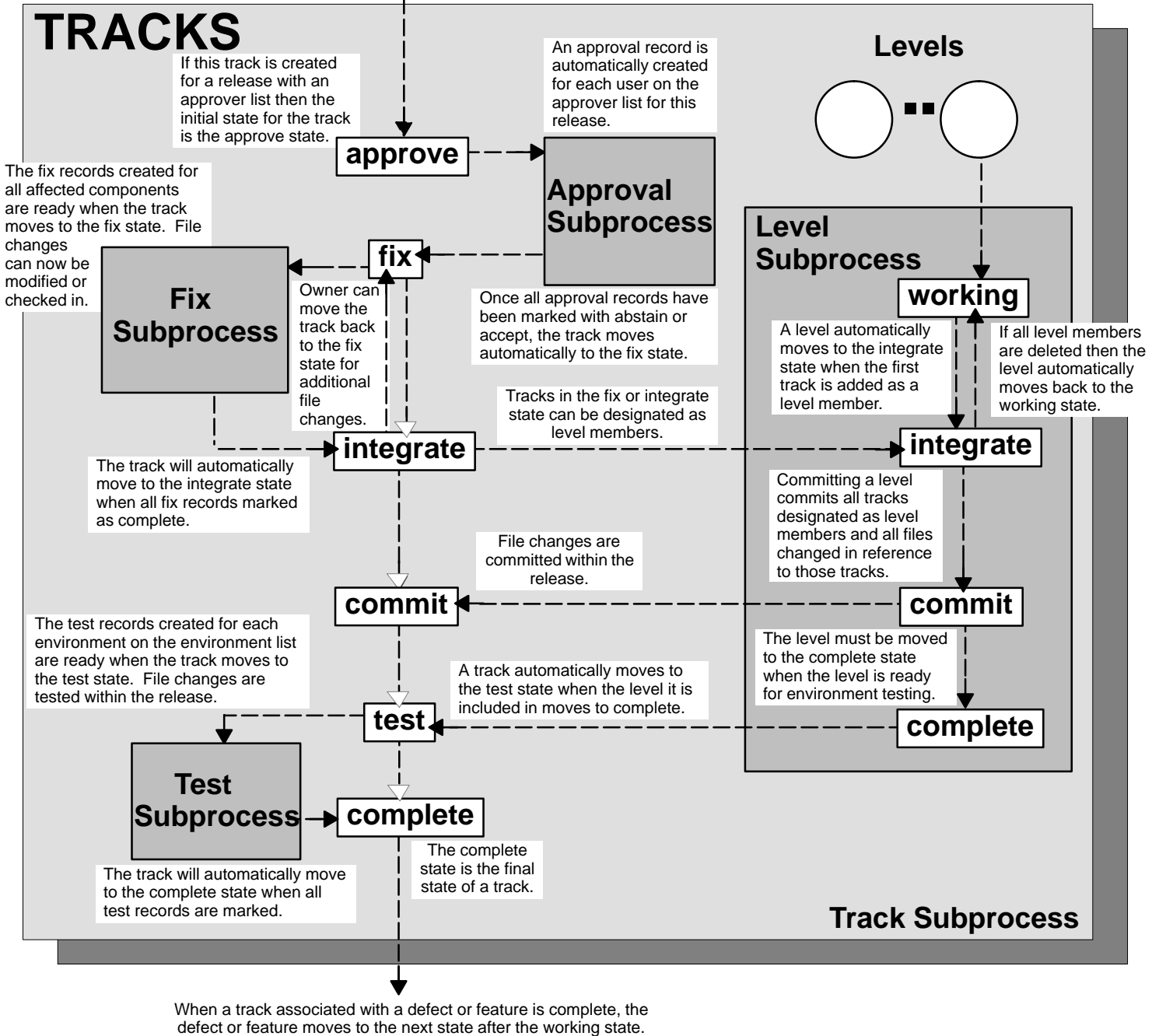
Figure 31. Feature and defect state diagram with all subprocesses configured

---

## The Feature and Defect State Diagram

# APPENDIX A

A track is created automatically for every release referenced by an accepted sizing record or can be created manually to track the resolution of a defect or feature in a specific release.



## LEGEND

--- state transitions for tracking

→ force action is required

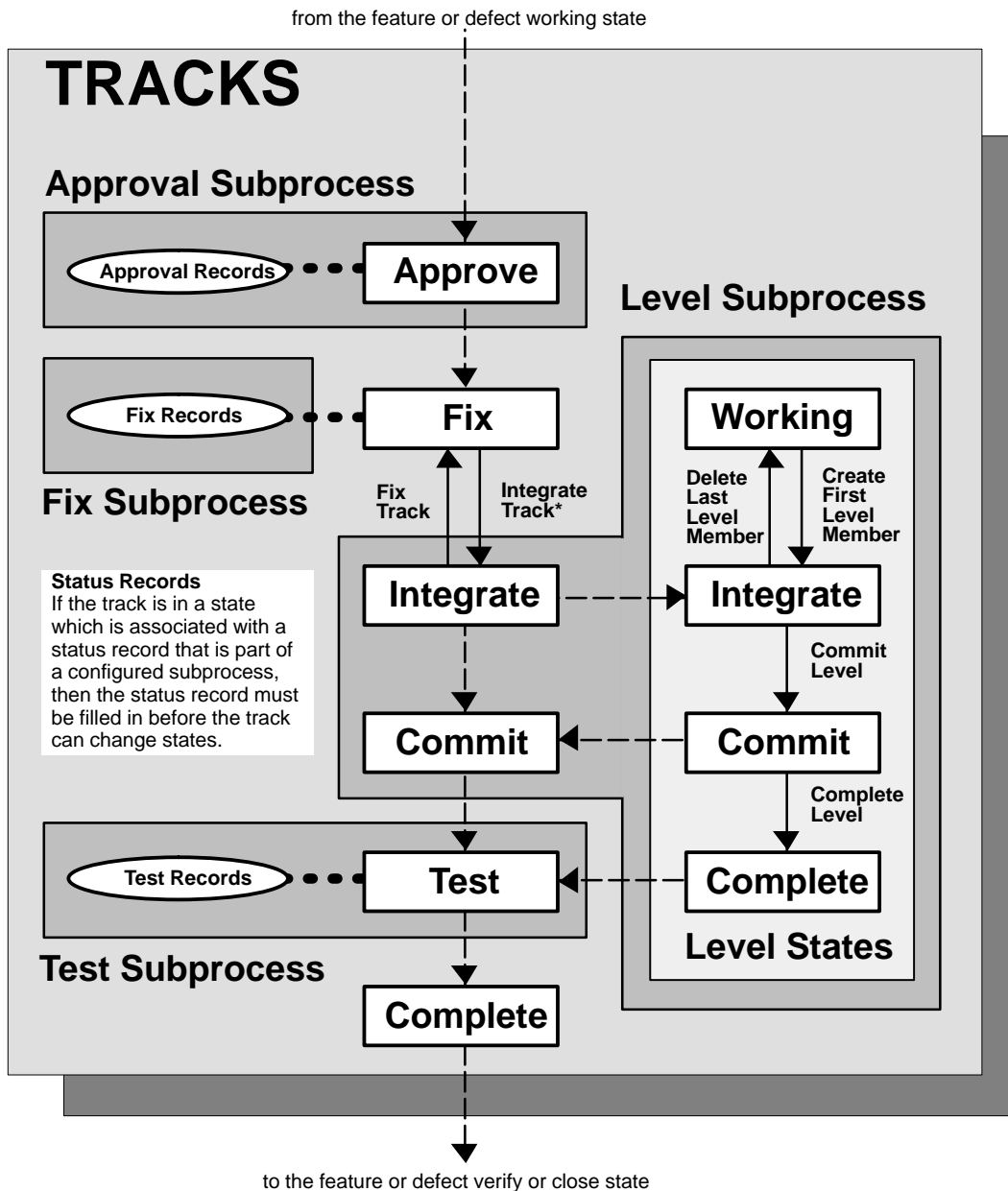
→ state transition is automatic or a non-force action is required

Figure 32. Track and level state diagram with all subprocesses configured.



---

## The Track and Level State Diagram



A track moves only to states corresponding to the subprocesses configured in the release process. For example, if your release process is configured to include only the track and approval subprocesses, then a track moves from the Approve state to the Fix state, and then to the Complete state.

**LEGEND**

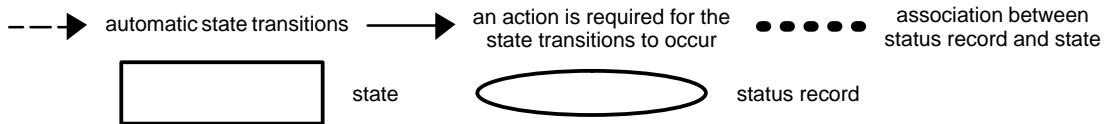
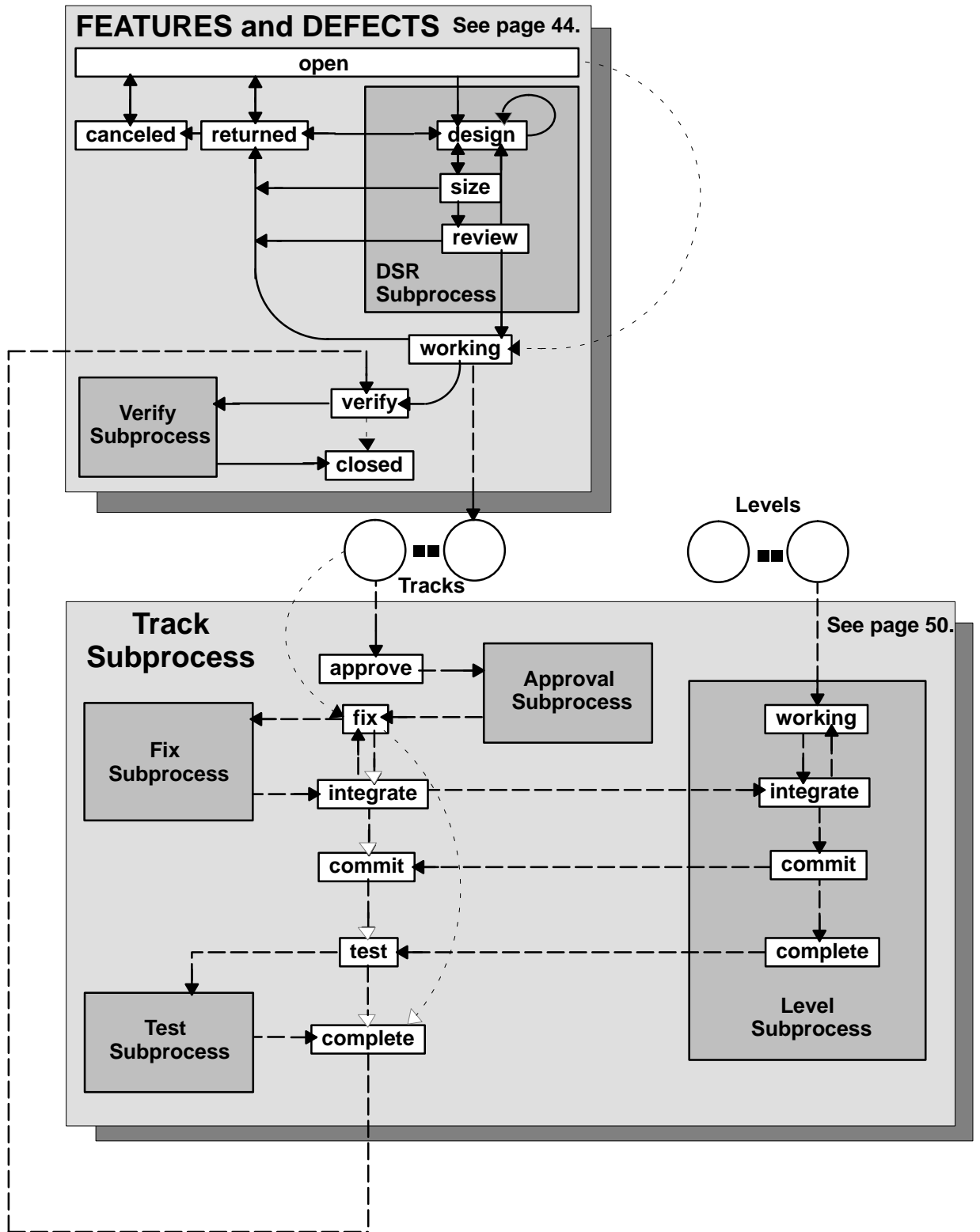


Figure 33. Relationship between subprocesses and state transitions for tracks

\* This transition is automatic if the fix subprocess is included in the release subprocess.

---

## Relationship Between Subprocesses and Track States



**LEGEND**

- state transitions for tracked releases only
- state transitions for all changes
- state transitions when subprocesses are not configured
- ⇨ force action is required
- state transition is automatic or a non-force action is required

Figure 34. CMVC State Diagram

---

## The CMVC State Diagram

## APPENDIX A

## Appendix B. CMVC Entity Relationships

### Entity Relationships Between CMVC Objects in a Family

Figure 36 shows the different entity relationships between each of the CMVC objects. Each box represents a CMVC object. Every possible relationship between one object and another is represented by a connecting line with an arrow at one end. For example, in Figure 35 the relationship between users and access lists is represented.

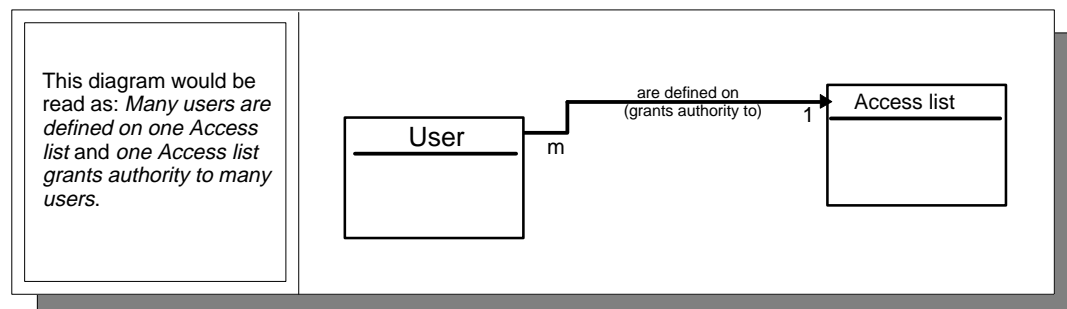


Figure 35. Example entity relationship diagram.

The arrow indicates which way to read the corresponding descriptions of the relationship. The description above the line describes the relationship in the direction of the arrow. The description below the line (in parentheses) describes the relationship in the opposite direction of the arrow. An *m* at one end of the line and a *1* at the other represents a many-to-one relationship in one direction and a one-to-many relationship in the opposite direction. Two *1*'s represent a one-to-one relationship. (An *m* in parentheses refers to a possible one-to-many relationship, however the normal case is a one-to-one relationship.) If there are two *m*'s, then the relationship is many-to-many. This relationship should be interpreted as one-to-many in both directions.

# APPENDIX B

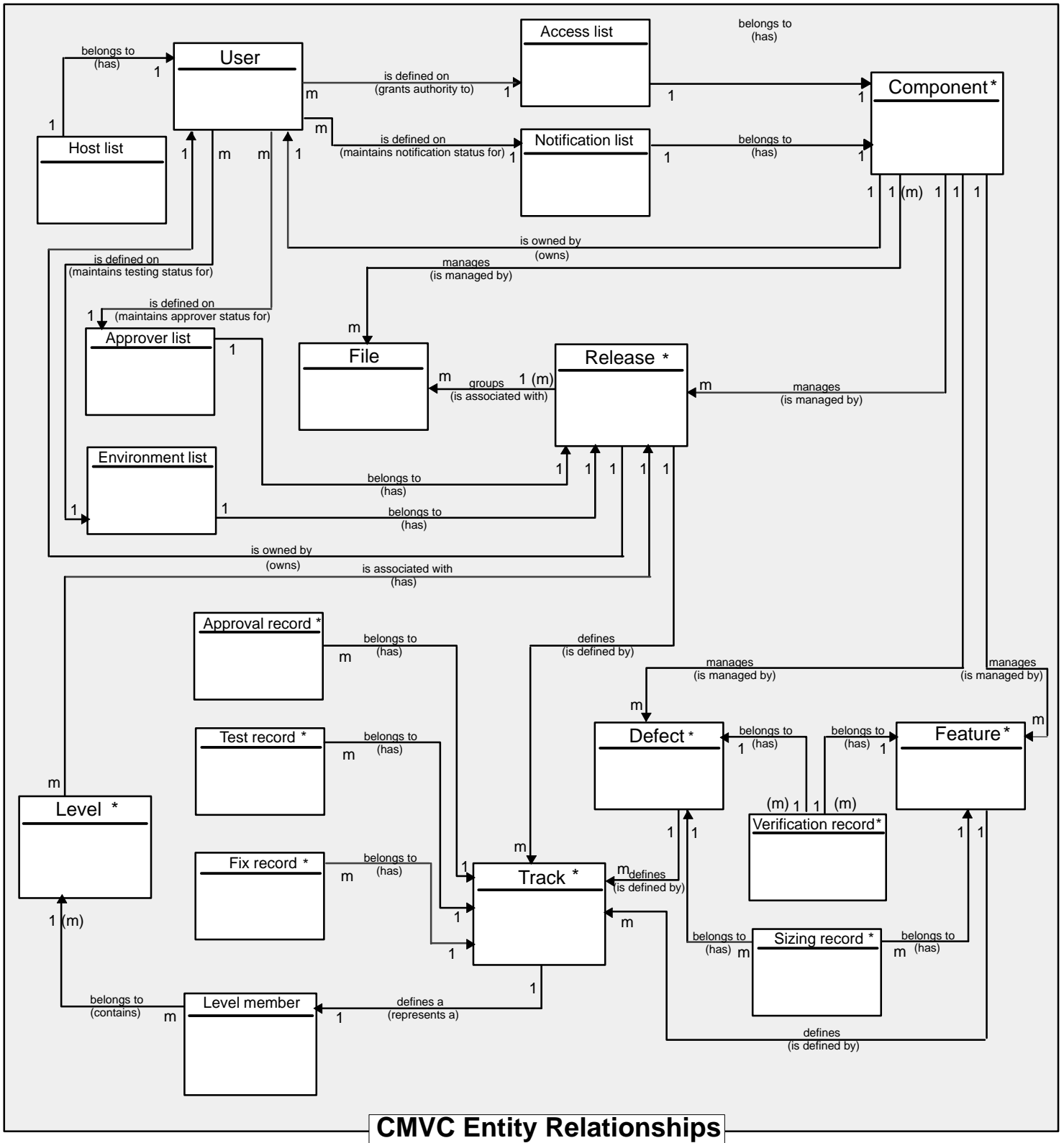


Figure 36. CMVC objects represented in a entity relationship diagram.

Any objects with a \* also have a one-to-one relationship with a user: one user owns one object and one object is owned by one user.



---

# Glossary

## Access List

A CMVC object that controls access to development data. A list of user ID-authority group pairs attached to a component, designating users and the corresponding authority access they are granted or restricted from using. *See also* Authority, Granted Authority, *and* Restricted Authority.

## Action

A task performed by the CMVC server and requested by a CMVC client. A CMVC action corresponds to issuing one CMVC command.

## Approval Record

A status record on which an approver must give an opinion of the proposed file changes required to resolve a defect or implement a feature in a release. *See also* Status Record.

## Approver

A user who approves changes within a specific release.

## Approver List

A list of user IDs attached to a release representing the users who must approve file changes required to resolve a defect or implement a feature in that release.

## Attribute

Attributes in CMVC are records containing information accessible to the user. CMVC allows family administrators to customize defect, feature, user, and file records by adding new attributes.

## Authority

The right to access development objects and perform CMVC commands. *See also* Access List, Base Authority, Explicit Authority, Granted Authority, Implicit Authority, Restricted Authority, *and* Superuser Privilege.

## Base Authority

The set of actions granted to a user whenever a user ID is created within a CMVC family. *See also* Authority.

## Base File Name

The name assigned to a file outside of the CMVC server environment, excluding any directory names.

## Base File Tree

The base set of files, associated with a release, to which changes are applied over time. Each committed level for a release updates the base file tree for that release.

## Change Control

The process of limiting and auditing changes to files through the mechanism of checking files in and out of a central, controlled, storage location. Change control for individual releases can be integrated with problem tracking by specifying a process for the release that includes the track subprocess.

## Child Component

All components in each CMVC family, except the root component, are created in reference to an existing component. The existing component is the parent component, and the new component is the child component. A parent component can have more than one child component. *See also* Component, *and* Parent Component.

## Client

A workstation that requests services from another workstation. *Contrast with* Server.

## CMVC Client

A workstation with the CMVC client software installed.

### **CMVC File**

A file that is stored by the CMVC server and retrieved by a path name. *See also* File, Common File, *and* Shared File.

### **CMVC Server**

A workstation with the CMVC server software installed.

### **Common File**

A file that is shared by two or more releases and the same version of the file is the current version for those releases. *See also* Shared File.

### **Component**

A CMVC object that organizes project data into structured groups, and controls configuration management properties. Component owners can control access to development data (*see* Access List) and configure notification about CMVC actions (*see* Notification List). Components exist in a parent-child hierarchy, with descendent components inheriting access and notification information from ancestor components.

### **Configuration Management**

The process of identifying, managing, and controlling software modules as they change over time.

### **Corequisite Tracks**

Two or more tracks designated as corequisites by a user so that all tracks in the corequisite group must be included as members in the same level. If a track is added to a level then all tracks that have a corequisite relationship with that track must also be included in the same level before the level is committed. *See also* Prerequisite Tracks.

### **Database**

A systemized collection of data that can be accessed and operated upon by a data processing system for a specific purpose.

### **Default**

A value that is used when an alternative is not specified by the user.

### **Defect**

A CMVC object used to formally report and record information about a problem. The user who opens a defect is the defect originator.

### **Delete**

Deleting a development object, such as a file or a user ID. Certain objects can be deleted only if certain criteria are met. Most objects that are deleted can be re-created.

### **Delta File Tree**

A directory structure representing only those files that have been changed and included in a specified level.

### **Destroy**

The only CMVC development object that can be destroyed in CMVC is a file. Destroying a file removes the file record from the database on the CMVC server. The file still exists in the file system on the CMVC server for access when extracting levels which refer to the file. A destroyed file cannot be re-created.

### **End User**

*See* User.

### **Environment**

A user-defined testing domain for a particular release. Testing domains might include operating systems, hardware configurations, and related software products.

### **Environment List**

A CMVC object used to specify environments in which a release should be tested. A list of environment-user ID pairs attached to a release, representing the user responsible for testing each environment. Only one tester can be identified per environment.

### **Explicit Authority**

The ability to perform an action against a CMVC object because you have been granted the authority to perform that action. *Contrast with* Implicit Authority *and* Base Authority.

**Extract**

A CMVC action you can perform on a file, level, or release. A file extraction results in the specified file being copied to the client workstation. A level extraction and release extraction result in copying the files associated with the level or release to a designated NFS server.

**Family**

A logical organization of related development data. A single installation of the CMVC can support multiple families. There is no way to access data within one family from another family.

**Family Administrator**

A user who is responsible for all non-system related tasks for one or more CMVC families such as planning, configuring, and maintaining the CMVC environment and managing user access to those families.

**Feature**

A CMVC object used to formally request and record information about a functional addition or enhancement. The user who opens a feature is the feature originator.

**File**

A collection of data that is stored by the CMVC server and retrieved by a path name. Any text or binary file used in a development project can be created as a CMVC file. Examples include source code, executables, documentation, and test cases. *See also* Common File *and* Shared File.

**Fix Record**

A status record that is associated with a track and is used to monitor the phases of change within a component that is affected by a defect or feature for a specific release.

**Full File Tree**

A directory structure representing a complete set of active files associated with a release.

**Granted Authority**

If an authority is granted on an access list, then it applies for all objects managed by this component and any of its descendants for which the authority

is not restricted. *See also* Access List, Authority, *and* Inheritance. *Contrast with* Restricted Authority.

**GUI**

The OSF/Motif\*\* based CMVC graphical user interface program.

**Host List**

A list associated with each CMVC user ID which indicates the client hosts that can access CMVC and act on behalf of the CMVC user. The list is used by the CMVC Server to authenticate the identity of a CMVC client upon receipt of a CMVC command. Each entry consists of a login and a host name.

**Implicit Authority**

The ability to perform an action against a CMVC object without being granted explicit authority. This authority is implicitly granted through inheritance or object ownership. *See also* Access List, *and* Authority. *Contrast with* Explicit Authority *and* Base Authority.

**Inheritance**

The passing of configuration management properties from parent component to child component. The configuration management properties that are inherited are access and notification. Inheritance within each CMVC family is cumulative although the inheritance of access at a given component can be restricted explicitly.

**Integrated Problem Tracking**

The process of integrating problem tracking with change control to track all reported defects, all proposed features, and all subsequent changes to files. *See also* Change Control.

**Level**

A collection of tracks which represent a set of changed files within a release. Levels are only associated with releases whose processes include the track and level subprocesses.

**Level Member**

A track that has been added to a level.

## Lock

An action that prevents editing access to a file stored within the CMVC development environment so that only one user can make changes to a given file at one time.

## Login

Operating system user identification.

## Network File System (NFS)

A program that allows you to share files with other computers in one or more networks over a variety of machine types and operating systems.

## Notification List

A CMVC object allowing component owners to configure notification. A list of user ID-interest group pairs attached to a component, designating users and their corresponding interest in receiving notification for all objects managed by this component or any of its descendants.

## Originator

The user who opens a defect or feature and who is responsible for verifying the outcome of the defect or feature on a verification record. The responsibility can be reassigned.

## Owner

The user who is responsible for a CMVC object within a CMVC family, either because they created the object or because they were assigned ownership of that object.

## Parent Component

All components in each CMVC family, except the root component, are created in reference to an existing component. The existing component is the parent component. *See also* Child Component, *and* Component.

## Path Name

The name of a file under CMVC control. A path name can be a set of directory names and a base name or just a base name. It must be unique within the release that groups the files.

## Prerequisite Tracks

If a file has been changed to resolve more than one defect or feature, the track referenced by the first

change is a prerequisite of the track referenced by the later changes. A track is a prerequisite to another track if:

- File changes have been checked in, but not committed, in reference to the first track, and
- One or more of those same files is then checked out, changed, and checked in again in reference to the second track.

*See also* Corequisite Track.

## Problem Tracking

The process of tracking all reported defects through to resolution and proposed features through to implementation.

## Process

A combination of CMVC subprocesses configured by the family administrator for a component or release, that controls the general movement of CMVC objects (defects, features, tracks, and levels) from state. *See also* State *and* Subprocess.

## Release

A CMVC object defined by the user that groups all the files that must be built, tested, and distributed as a single entity.

## Restricted Authority

The restriction of a user's ability to perform certain actions at a specified component. *See also* Access List, Authority, *and* Inheritance. *Contrast with* Granted Authority.

## Root Component

The initial component that is created when a CMVC family is configured. All components in a CMVC family are descendants of the root component. Only the root component has no parent component.

## Shared File

A file that is shared between two or more releases. *See also* Common File.

## Sizing Record

A status record created for each component-release pair possibly affected by a defect or feature. The sizing record owner must indicate whether the defect or feature affects the

specified component-release pair and the approximate amount of work needed to resolve the defect or implement the feature within the specified component-release pair. *See also* Status Record.

### **State**

Tracks, levels, features, and defects move through various states during their life cycles. An object's current state determines which actions may be performed against it.

### **Status Record**

A status record records a decision made by the owner of the status record. *See also* Approval Record, Fix Record, Test Record, and Verification Record.

### **Subprocess**

CMVC subprocesses govern the state changes for CMVC objects. The design, size, review (DSR) and verify subprocesses are configured for component processes. The track, approve, fix, level, and test subprocesses are configured for release processes. *See also* Process.

### **Superuser Privilege**

Superuser privilege allows a user to perform any action available in the CMVC family.

**Note:** Superuser privilege is internal to the CMVC tool and not related to operating system superuser authority.

### **System Administrator**

A user who is responsible for all system-related tasks involving CMVC such as installing, maintaining, and backing up CMVC and the relational database used by CMVC.

### **Test Record**

A status record used to record the outcome of an environment test performed for each defect and feature in a specific level of a release. *See also* Status Record.

### **Tester**

A user responsible for testing the resolution of a defect or the implementation of a feature for a specific level of a release and recording the results on a test record.

### **Track**

A CMVC object created to monitor the progress of changes within a release to resolve a specific defect or implement a specific feature.

### **User**

A person with an active CMVC user ID and access to one or more CMVC families.

### **User ID**

Unique identification for a login on a specific host (provided by the operating system). CMVC uses the user ID to refer to and communicate with a user.

### **Verification Record**

A status record which must be marked by the originator of a defect or a feature before the defect or feature can move to the closed state. This allows the originator of the defect or feature to verify the action's resolution or implementation. *See also* Status Record.

### **Version Control**

The storage of multiple versions of a single file along with information about each version.



---

# Index

## A

- access, 7, 16
  - controlling, 16
  - file, 5
  - inheritance, 16, 17, 19
  - shared files, 37
- access list, 16
  - definition, 77
- action, 16
  - definition, 77
- ancestor component, 13
- approval record, 49
  - definition, 77
- approval subprocess, 48, 49
- approve state, 63
- approver, definition, 77
- approver list, 49
  - definition, 77
- attribute
  - defect, 41, 42
  - definition, 77
  - features, 41, 42
  - file, 31
  - release, 23
  - track, 47
- authority, 16
  - base. *See* base authority
  - definition, 77
  - explicit. *See* explicit authority
  - granted. *See* granted authority
  - implicit. *See* implicit authority
  - restricted. *See* restricted authority
- authority group, 16, 18
- automatic notification, 20

## B

- base authority, 17
  - definition, 77
- base file tree, 60
  - definition, 77
- book audience, xi
- buildable file tree, 59

## C

- canceled state, 61
- change control, 5
  - definition, 77
  - integrated, 26
  - process, configuring, 48
- child component, 13
  - definition, 77
- client-server model, 1
- closed state, 63

- CMVC, introduction, 5
- CMVC actions, 16
- CMVC client, 1
- CMVC files, 31
  - example, 34
- CMVC roles, 2
- CMVC server, 1
- CMVC state diagram, 53, 72
- CMVC system configuration, 1
- CMVC tasks, 2
- CMVC user interface, 2
- combining file trees, 59
- command line interface, 2
- commit state
  - level, 65
  - track, 64
- common file, 35
  - definition, 78
- common link, breaking, 36
  - example, 36
- compiling, 56, 59
- complete state
  - level, 65
  - track, 65
- component, 5, 13
  - access list, 16
  - ancestor, 13
  - child, 13
  - definition, 13, 78
  - descendant, 13
  - inheritance, 19
  - multiple parents, 15
  - notification list, 16
  - parent, 13
  - relationship with features and defects, 21
  - relationship with files, 7, 22, 31, 32, 37
  - relationship with other components, 13
  - relationship with processes, 21
  - relationship with releases, 7, 23
  - root, 14
    - definition, 80
- component attributes, 13
- component hierarchy, 5, 13, 16
- component organization, 14
- component owner, 5, 13, 16
- configuration management, 5
  - definition, 78
- corequisite check, 56
- corequisite tracks, 56
  - definition, 78

## D

- data management, 5, 16
- data ownership, 6

- database, 31
  - definition, 78
  - supported, 2
- defect, 29, 41
  - accepting, 62
  - analyzing, 8, 43
  - canceling, 61
  - closing, 63
  - definition, 41, 78
  - designing, 43, 62
  - duplicate, 46
  - fixing, 9
  - opening, 61
  - relationship with components, 21, 41, 43, 61
  - relationship with processes, 21
  - relationship with tracks, 9, 45, 62
  - reopening, 61
  - reporting, 8
  - resolving, 45
  - returning, 61
  - reviewing, 45, 62
  - sizing, 43, 62
  - sizing record. *See* sizing record
  - verification record. *See* verification record
  - verifying, 10, 45, 63
- defect attributes, 41, 42
- defect life cycle, 8
- defect originator, 41, 43, 46
- defect owner, 41, 43, 46
- defect state diagram, 44, 66
- defect states, 41, 61
- delete, 39
  - definition, 78
- delta file tree, 58
  - definition, 78
- descendant component, 13
- design change. *See* feature
- design state, 62
- DSR Subprocess, 8

## E

- end user, 3
  - definition, 78
- entity relationships, 75
  - diagram, 76
- environment, definition, 78
- environment list, 52
  - definition, 78
- explicit authority, 17
  - definition, 78
- explicit notification, 20
- extracting file trees, 58
- extracting files, 32, 33
  - definition, 79

## F

- family, 5
  - definition, 79
- family administration, 2

- family administrator, 3, 16, 18
  - definition, 79
- feature, 29, 41
  - accepting, 62
  - canceling, 61
  - closing, 63
  - definition, 41, 79
  - designing, 43, 62
  - duplicate, 46
  - evaluating, 8, 43
  - fixing, 9
  - implementing, 45, 62
  - opening, 61
  - proposing, 8
  - relationship with components, 21, 41, 43, 61
  - relationship with processes, 21
  - relationship with tracks, 9, 45, 62
  - reopening, 61
  - returning, 61
  - reviewing, 45, 62
  - sizing, 43, 62
  - sizing record. *See* sizing record
  - verification record. *See* verification record
  - verifying, 10, 45, 63
- feature attributes, 41, 42
- feature life cycle, 8
- feature originator, 41, 43, 46
- feature owner, 41, 43, 46
- feature state diagram, 44, 66
- feature states, 41, 61
- file, 31
  - base name, 31
    - definition, 77
  - checking in, 34
  - checking out, 32
  - committed version, 32, 39
  - common, 35
    - definition, 78
    - example, 35
  - current version, 31, 32
  - definition, 79
  - delete, 39
  - destroy, definition, 78
  - extracting. *See* extracting files
  - lock, 32, 33, 34
    - definition, 80
  - path name, 31
    - definition, 80
  - relationship with components, 7, 22, 31
  - relationship with releases, 7, 25, 34
  - relationship with tracks, 38, 55, 57, 58
  - shared, 34
    - access, 37
    - definition, 80
    - example, 35
    - versioning, 32, 34
- file access authority, 32
- file attributes, 31



file changes, 9, 32, 51, 57  
  example, 33  
  tracking, 47  
  undoing, 39  
    example, 39

file integration, 9, 45  
file mode, 31, 32, 33  
file organization, 25  
file tree, 56, 58

  base, 60  
    definition, 77  
  buildable, 59  
  combining, 59  
  delta, 58  
    definition, 78  
  diagram, 59  
  extracting, 58  
  full, 58  
    definition, 79

fix  
  record, definition, 79  
  state, 63

fix record, 49  
fix subprocess, 48, 49  
full file tree, 58  
  definition, 79

## G

granted authority, 16, 18  
  definition, 79  
GUI (graphical user interface), 2  
  definition, 79

## H

hierarchy, component, 5, 13, 16  
highlighting style, xii  
host list, definition, 79

## I

implicit authority, 17  
  definition, 79  
inheritance, 19  
  access, 16, 17, 19  
  definition, 79  
  notification, 16, 19  
  restricting, 16, 17

integrate state  
  level, 65  
  track, 64

interest group, 16, 21  
Intersolv's PVCS Version Manager. *See* PVCS

## L

level, 55  
  committing, 9, 57  
  completing, 57  
  definition, 55, 79  
  introduction, 9

  level attributes, 55  
  level member, 9  
    definition, 79  
  level state diagram, 50, 68  
  level states, 65  
  level subprocess, 48, 56  
  login, definition, 80

## M

mode, 32, 33

## N

network overview, 1  
notification, 20  
  controlling, 20  
  explicit, 20  
  inheritance, 16, 19  
notification list, 16  
  definition, 80

## O

open state, 61  
originator, definition, 80  
ownership, 6  
  definition, 80

## P

parent component, 13  
  definition, 80  
prerequisite check, 56  
prerequisite tracks, 56  
  definition, 80  
problem tracking, 5  
  definition, 80  
  integrated, 5, 47  
  definition, 79  
  introduction, 8  
process, 7  
  definition, 80  
  relationship with components, 13, 21  
  relationship with releases, 26  
publications, related, xii  
PVCS, 2, 32

## R

related publications, xii  
release, 6  
  definition, 23, 80  
  relationship with change control, 26  
  relationship with components, 7, 23  
  relationship with files, 6, 7, 25, 31, 34  
  relationship with processes, 23  
  relationship with tracks, 26, 45  
  updating, 9, 60  
release attributes, 23  
release management, 23, 25  
release organization, examples, 24

- release owner, 23
- release-file, example, 25, 26
- restricted authority, 16, 17, 19
  - definition, 80
- returned state, 61
- review state, 62
- root component, 13, 14

## S

- shared file, 34
  - definition, 80
  - example, 35
- size state, 62
- sizing records, 62
  - definition, 80
- Source Code Control System (SCCS), 2, 32
- state, definition, 81
- status record, definition, 81
- style, highlighting, xii
- subprocess, 7
  - approval. *See* approval subprocess
  - definition, 81
  - DSR. *See* DSR subprocess
  - fix. *See* fix subprocess
  - level. *See* level subprocess
  - test. *See* test subprocess
  - track. *See* track subprocess
  - verify. *See* verify subprocess
- subscribers, 20
- superuser privilege, 17
  - definition, 81
- system administration, 2
- system administrator, 2
  - definition, 81
- system configuration, 1

## T

- test record, 51, 52
  - definition, 81

- test state, 64
- test subprocess, 48, 51
- tester, 51
  - definition, 81
- track, 29, 62
  - definition, 47, 81
  - introduction, 8
  - overview, 48
  - relationship with features and defects, 9, 45, 62
  - relationship with files, 9, 55, 57, 58
  - relationship with releases, 45
- track attributes, 47
- track owner, 52
- track state diagram, 50, 68, 70
- track states, 52, 63

## U

- user, 3
  - definition, 81
- User ID, definition, 81
- user interface, 2

## V

- verification record, 63
  - definition, 81
- verify state, 63
- verify subprocess, 52
- version control, 5
  - definition, 81

## W

- working state, level, 65
- working state, features and defects, 62