Rational Environment Reference Manual

Project Management (PM)

Document Control Number: 8001A-31

Rev. 0.0, February 1986
Rev. 1.0, July 1986
Rev. 2.0, December 1987
Rev. 3.0, March 1988
Rev. 4.0, August 1988

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
3320 Scott Boulevard
Santa Clara, California 95054-3197

Contents

Reference Entries

How to Use This Book

The Project Management (PM) book of the *Rational Environment Reference Manual* contains reference information describing commands and tools provided by the Rational Environment™ that are useful primarily for partitioning a project into components, testing and releasing implemented components, tracking the history of Ada-unit versions and configurations, and coordinating multiple developers and multiple development efforts. This reference information is intended for users who are familiar with the Environment and Ada programming.

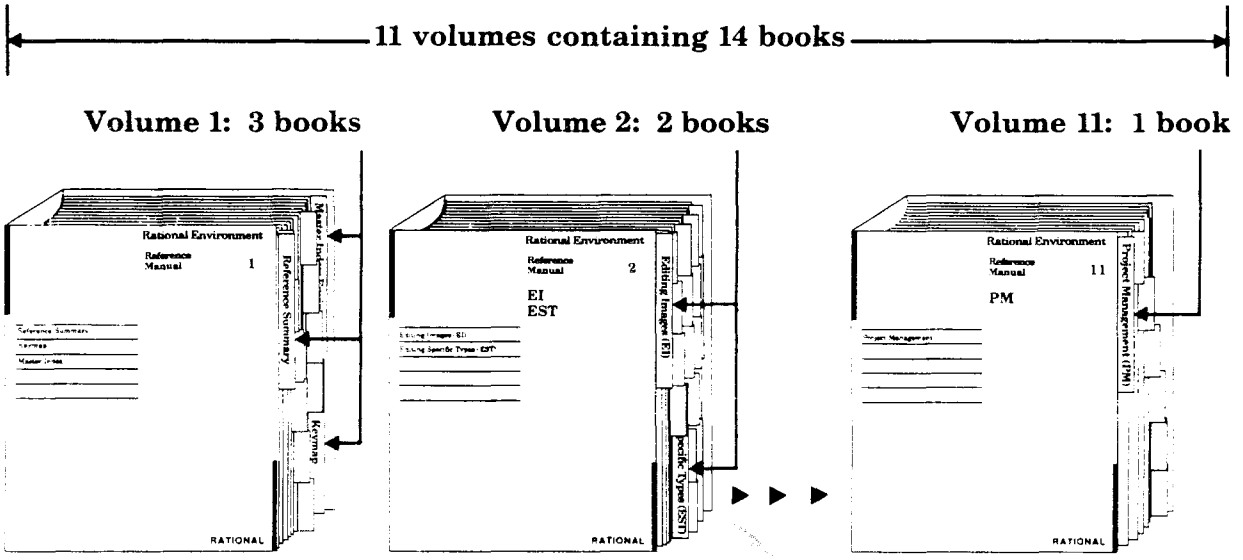
Organization of the Reference Manual

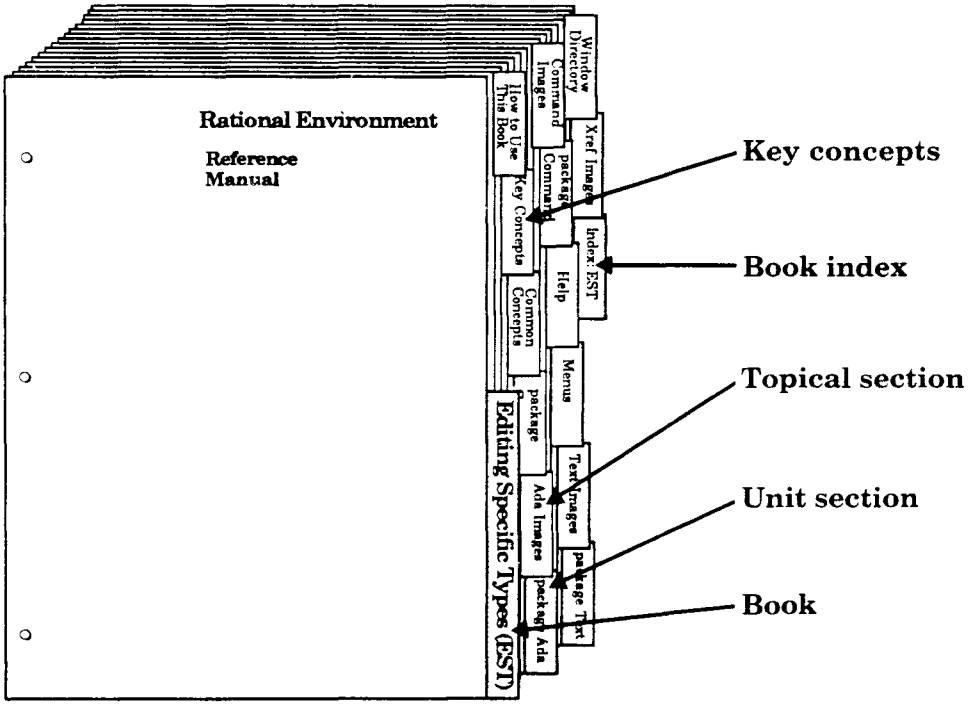
The *Rational Environment Reference Manual* (Reference Manual for brevity) includes the following volumes (see accompanying illustration):

1	Reference Summary Keymap Master Index
2	Editing Images (EI) Editing Specific Types (EST)
3	Debugging (DEB)
4	Session and Job Management (SJM)
5	Library Management (LM)
6	Text Input/Output (TIO)
7	Data and Device Input/Output (DIO)
8	String Tools (ST)
9	Programming Tools (PT)
10	System Management Utilities (SMU)
11	Project Management (PM)

Each *volume* of the Reference Manual contains one or more *books* separated by large colored tabs. Each book contains information on particular features or areas of application in the Environment. The abbreviation for the name of each book (for example, EI for Editing Images) appears on the binder cover and spine, and this abbreviation is used in page numbers and cross-references. The books grouped into one volume are not necessarily logically related.

Organization of the *Rational Environment Reference Manual*





A sample book

The Reference Manual provides reference information organized to efficiently answer specific questions about the Rational Environment. The *Rational Environment User's Guide* complements this manual, providing a user-oriented introduction to the facilities of the Environment. Products other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility) are documented in individual manuals, which are not part of the Reference Manual.

Volume 1

Volume 1, intended to be used as a quick reference to the resources provided by the Environment, contains the following books:

- **Reference Summary**: The Reference Summary contains the full Ada specification for each unit in the standard Environment. The unit specifications are organized by their pathnames. The World ! section provides a list of the units in the library system of the Environment and the manual/book in which they are documented.
- **Keymap**: The Rational Environment Keymap presents the standard Environment key bindings, organized by topic and by command name. The topical section includes both a quick reference for commonly used commands and a more detailed reference for key bindings.
- **Master Index**: The Master Index combines all of the index information for each of the books in the Reference Manual.

Volumes 2–11

Each book in Volumes 2–11 begins with a colored tab on which the name of the book appears. Each book typically contains the following sections:

- **Contents**: The table of contents provides a complete list of all the units in the book and their reference entries.
- **Key Concepts section**: Most of the books contain a section describing key concepts that pertain to all of the Environment facilities documented in that book. This section is located behind its own tab after the table of contents.
- **Unit sections**: Each of the commands, tools, and so on has a declaration within an Ada compilation unit (typically a package) in the Environment library system. For each unit, there is a section that contains reference entries for the declarations (for example, procedures, functions, and types) within that unit. Each section is preceded by a tab.

  The sections for units are alphabetized by the simple names of the units. For example, the section for package !Tools.String_Utilities is alphabetized under String_Utilities.

  For many units, introductory material and/or examples specific to the unit appear after the section tabs.

  Within the section for a given unit, the reference entries describing the unit's declarations are organized alphabetically after the section introduction. Appearing at the top of each page in a reference entry are the simple name of the given declaration and the fully qualified pathname of the enclosing unit.

- **Explanatory/topical sections**: Like the unit sections, explanatory/topical sections are preceded by tabs, and they are alphabetized with the unit sections. The topical sections, such as Help, located in Editing Specific Types (EST), discuss Environment facilities.
- **Index**: Preceded by a tab, the Index appears as the last section of each book. It contains entries for each unit or declaration, along with additional topical references. Each book index covers only the material documented in that particular book. The Master Index (in Volume 1) provides entries for the information documented in all the books within the Reference Manual.

  Italic page numbers indicate the page on which the primary reference entry for a declaration appears; nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Suggestions for Finding Information

The following suggestions may help you in finding various kinds of information in the documentation for Rational's products.

Learning about Environment Facilities

If you are a novice user starting to use the Environment, consult the *Rational Environment User's Guide.*

If you are familiar with the Environment but are interested in learning about the Environment's library-management commands, for example, you might start by scanning the specifications for these units in the Reference Summary to get an idea of the kinds of things these tools can do. You should also look at the Key Concepts for the particular book, which describes important concepts and gives examples.

It may also be useful to glance through the introductions provided for some of the units in the book. These introductions, located immediately after the tabs for the units, often contain helpful examples.

Finding Information on a Specific Item

If you know the name of the item and the book in which it is documented, consult either the table of contents or the index for that book. You can also turn through the pages of the book using the names and pathnames of the reference entries to locate the entry you want. Remember that the reference entries for a unit are organized alphabetically within the unit, and the units are organized alphabetically by simple name within the book.

If you know the simple name of the entry but do not know the book in which it is documented, look in the Master Index (in Volume 1) to find the book abbreviation and page number.

If you know the pathname of the entry but do not know the book in which it is documented, the World ! section of the Reference Summary (in Volume 1) provides a map of the units in the library system of the Environment and the books in which they are documented.

If you cannot find an item in the Master Index, the item either is not documented or is documented in the manuals for a product other than the Rational Environment (for example, Rational Networking—TCP/IP or Rational Target Build Utility). If you know the pathname, consult the World ! section of the Reference Summary to determine whether that item is documented and in which manual.

Using the Index

The index of each book contains entries for each unit and its declarations, organized alphabetically by simple name. When using the index to find a specific item, consult the italic page number for the primary reference for that item. Nonitalic page numbers indicate key concepts, defined terms, cross-references, and exceptions raised.

Viewing Specifications On-Line

If you know the pathname of a declaration and want to see its specification in a window of the Rational Environment, provide its pathname to the Common.Definition procedure—for example, `Definition ("!Commands.Library");`. If you know the simple name of the unit in which the declaration appears, in most cases you can use searchlist naming as a quick way of viewing the unit—for example, `Definition ("\Library");`.

Using On-Line Help

Most of the information contained in the reference entries for each unit is available through the on-line help facilities of the Environment. Press the Help on Help key or consult the *Rational Environment User's Guide* or the *Rational Environment Reference Manual*, EST, Help, for more information on using this on-line help facility.

Cross-Reference Conventions

The following conventions are used in cross-references to information:

- **Specific page/book**: For references to a specific place in a specific book, the book abbreviation is followed by the page number in the book (for example, LM-322). If the book abbreviation is omitted, the current book is implied (for example, the page numbers in the table of contents for a book do not include the book prefix).
- **Declaration in same unit**: References to the documentation for a declaration in the same unit are indicated by the simple name of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Library.Move procedure would be simply "procedure Move." Note that if there are nested packages in the unit, references to nested declarations use qualified pathnames.
- **Declaration in different unit, same book**: References to the documentation for a declaration in another unit are indicated by the qualified pathname of the desired declaration. For example, within the reference entry for the Library.Copy procedure, a reference to the Compilation.Delete procedure would be "procedure Compilation.Delete."

- **Declaration in different book**: References to the documentation for a declaration in another book are indicated by the addition of the abbreviation for that book. For example, within the reference entry for the Library.Copy procedure, a reference to the Editor.Region.Copy procedure in the Editing Images book would be "EI, procedure Editor.Region.Copy."

References to specific declarations in the library system of the Rational Environment (not the documentation for them) are typically indicated by fully qualified pathnames—for example, "procedure !Commands.Library.Copy." When the context is clear, however, a shorter name will be used. If the unit in which the declaration appears is undocumented, you may want to see its explanatory comments to understand what it does. To see these comments, either look at the unit's specification in the Reference Summary or view it on-line using the Rational Environment.

Feedback to Rational: Reader's Comments Form

Rational wants to make its documentation as useful and error-free as possible. Please provide us with feedback. The last page of each book contains a Reader's Comments form that you can use to send us comments or to report errors. You can also submit problem reports and make suggestions electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Key Concepts

Introduction

Managing a complex software project involves:

- Partitioning the project into components and designing the interfaces among these components
- Implementing these components
- Testing and releasing implemented components
- Tracking and reporting the history of versions and configurations of Ada units
- Coordinating multiple developers and multiple development efforts

The Rational Environment provides support for project management through its system of configuration management and version control (CMVC). Although project management on the Environment typically refers to managing software systems and applications written in Ada, CMVC resources also can support the development and maintenance of documentation, application test beds, and the like.

This Project Management book of the *Rational Environment Reference Manual* describes the Environment's CMVC resources, which are defined in the following packages:

- Activity: Defines a set of operations for creating, editing, and using *activities*. Activities enable you to combine alternative implementations of project components for test or execution.
- Check: Defines a set of operations for checking whether a software component's implementation is *compatible* with its exported interface.
- Cmvc: Defines a set of operations that support the following activities of project management:
  - — Partitioning projects into components using *subsystems* and managing the interfaces among these components
  - — Creating and releasing alternative implementations (*views*) of individual project components
  - — Placing the objects within project components under source control to record generations of change history and to coordinate the work of multiple developers

- Cmvc_Hierarchy: Defines a set of operations for grouping multiple subsystems into higher-level application components called *systems.* Inclusion in a system provides an automated means of tracking the latest release from each subsystem and performing system builds by creating activities that reference those releases.
- Cmvc_Maintenance: Defines a set of operations for checking and restoring the integrity of the various databases associated with the CMVC system. This package also provides operations for managing ***primary*** and ***secondary*** subsystems (copies of subsystems that support development on multiple R1000s).
- Work_Order: Defines a set of operations for creating, editing, and using *work orders*, *work-order lists*, and *ventures.* These objects enable you to define, assign, and track the progress of project tasks and the objects they affect.

What to Read in This Book

CMVC resources are flexible and can be used to support many different kinds of development conventions. Accordingly, what you should read in this book depends on the development conventions in effect at your installation. At most installations, the development process involves the following participants (whose roles may be combined in various ways):

- Designers who determine the fundamental components and their interfaces within the project
- Implementers (or teams of implementers) who write the source code and documentation for each component
- System integrators who build the finished product from its components
- Managers who coordinate parallel development efforts and track the project's progress

Within this Key Concepts section of this book, all project participants should read at least "CMVC Overview" and "Getting Started." These sections summarize the features of CMVC and describe in detail how to develop a program within a single subsystem (including how to put objects under CMVC control, reserve such objects for modification, and create released configurations of specific object versions).

Teams of implementers assigned to single project components also should read "Coordinating Development in a Subsystem," which describes how to reserve shared objects for exclusive use, how to propagate changes, and how to operate concurrently with shared objects.

Project managers should read "Coordinating Development in a Subsystem" to manage project teams working on shared objects. Resources for gathering project-level information and for tracking progress are documented in package Work_Order.

Project designers and integrators should read "Developing Applications Using Multiple Subsystems." This section summarizes design considerations for partitioning projects into subsystems and describes how to set up subsystem interfaces and execute a program composed of multiple subsystems.

CMVC Overview

Projects such as software applications or documents typically consist of many component objects. For example, an Ada application typically consists of a set of interdependent Ada compilation units such as procedures and packages; similarly, a document such as this book of the *Rational Environment Reference Manual* consists of a set of text files, with one file per chapter or major section. As development and maintenance proceed, individual objects change, new objects are created, and existing objects are deleted. Thus, over time, the contents of the individual objects and the overall *configuration* of objects changes.

The Rational Environment provides the following kinds of support for managing such projects:

- Project partitioning: You can break a project into a manageable number of higher-level components called *subsystems*, each containing a group of logically related objects. For Ada programs, subsystems are units of decomposition similar to, but larger than, the Ada package, which preserve on a larger scale the Ada notion of separate specification and implementation.
- Version control: You can control and track changes to individual objects within each subsystem and record what changes were made and why they were made.
- Configuration management: You can construct, release, and maintain multiple consistent sets (or configurations) of versions within each subsystem. (Each alternative configuration constitutes a *view* of the subsystem.) At a higher level, configuration management refers to combining views from each subsystem in order to create entire applications.

The Environment's resources for project management are integrated into a set of operations known as configuration management and version control (CMVC).

Issues of Project Management

Software projects that contain a large number of Ada units pose certain management problems:

- It can be difficult to reason about the application's overall design and to allocate well-defined portions of the application to individual developers or teams.
- It can be difficult to keep track of dependencies among units and to prevent the introduction of unwanted dependencies.
- Making changes can be time-consuming because the changes must be verified by recompiling the changed units and all of their direct and indirect dependents.
- Recompilation dependencies make it difficult for individuals and teams to work and test in parallel, because a change in one team's portion of the application may entail recompilation of another team's portion.
- Preserving a consistent set of previous versions of units and coordinating access to shared units can be time-consuming and error-prone.

Although using worlds or directories can make it easier to understand the high-level structure of a large project, such use cannot solve the other problems of project management listed above. Another more powerful kind of Environment library structure, the *subsystem*, can be used instead to express and enforce an application's design and to make CMVC operations available.

Subsystems

Subsystems encapsulate a program's compilation units in higher-level components, just as Ada packages encapsulate related subprograms, type declarations, and the like. Depending on its size, each subsystem can be assigned to an individual developer or to a team of developers.

Subsystems are more powerful than other libraries for the following reasons:

- Subsystems, like Ada packages, provide a means for defining and enforcing interfaces among an application's components. These interfaces provide explicit control over dependencies among units in different subsystems.
- Subsystem interfaces impose explicit bounds on the recompilation required after changes are made to the implementation. With reduced recompilation requirements, development teams can work and test in parallel.
- Subsystems provide a mechanism for developing alternative implementations of an application's components. Execution and testing of the entire application is a matter of specifying the desired combination of precompiled implementations, one from each subsystem within an application.
- CMVC operations are available within subsystems for tracking unit changes, coordinating access to shared units, and propagating changes across shared units.

Figure 2-1 represents an Ada application that has been encapsulated into subsystems. The application's compilation units are represented with both a specification and a body (dark shading), and dependencies among units are represented by arrows. The units are partitioned into three subsystems (lightly shaded areas), whose interfaces are represented by heavy arrows.

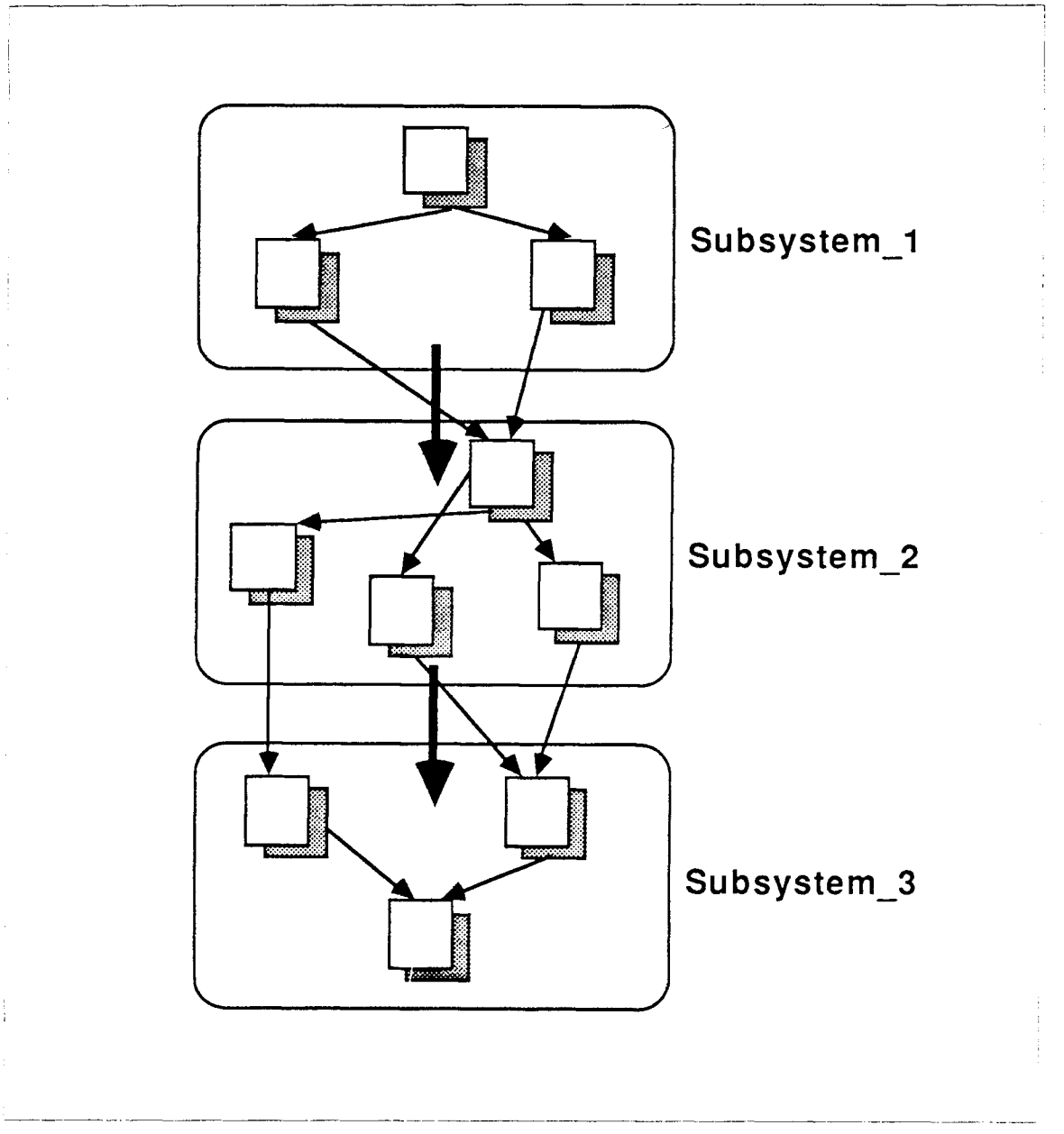


Figure 2-1. An Application Partitioned into Three Subsystems

Version Control

When a component of an application is encapsulated in a subsystem, individual objects in the component can be *controlled*—that is, made subject to version control. Controlled objects must be *checked out* to be modified; checking out an object reserves it for editing by acquiring the object's *reservation token*. When desired, the modified object can then be *checked in* and made available for other users to check out.

Every subsystem contains a *CMVC database* that records the changes made to each controlled object. Each time an object is checked out and then checked in, a new *generation* of the object is created in the CMVC database. Therefore, the CMVC database records the contents of successive generations of each controlled object within a subsystem, as indicated in Figure 2-2.

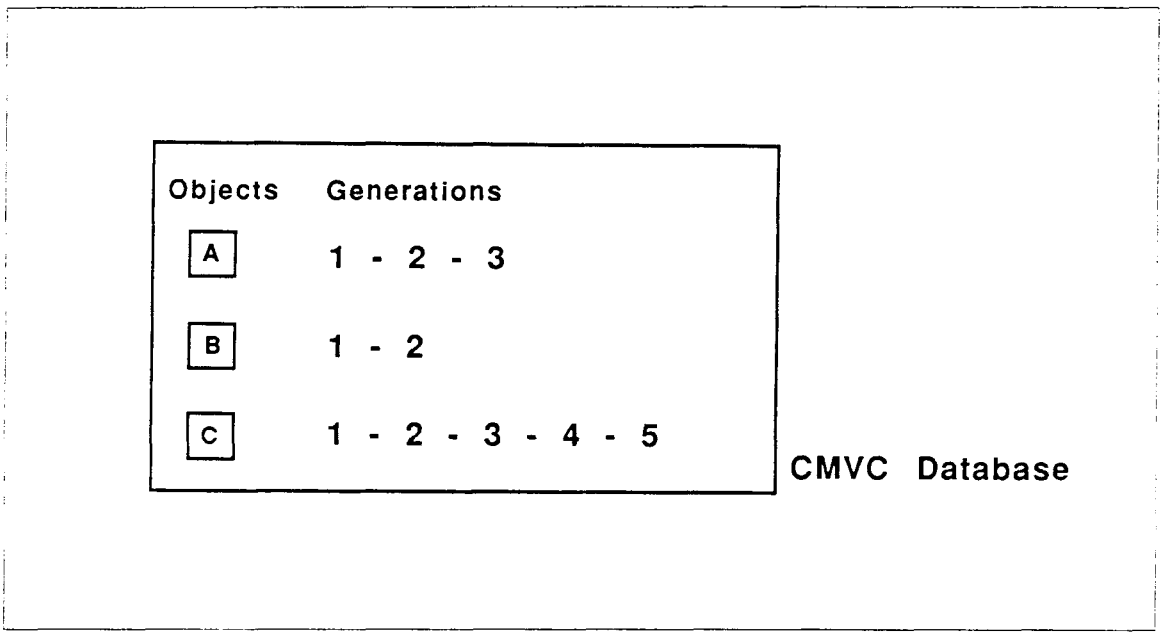


Figure 2-2. The CMVC Database

Configurations and Releases

Ada units in subsystems reside in program libraries and therefore can be compiled using the normal Environment mechanisms. Each subsystem contains at least one *working* library in which units can be checked out, modified, checked in, compiled, and tested.

At any given time, the working library contains exactly one generation of each object, usually the latest (although an object can be *reverted* to any previous generation stored in the CMVC database). A combination of generations, one per controlled object in the library, is called a *configuration*; Figure 2-3 represents the configuration for a sample working library.

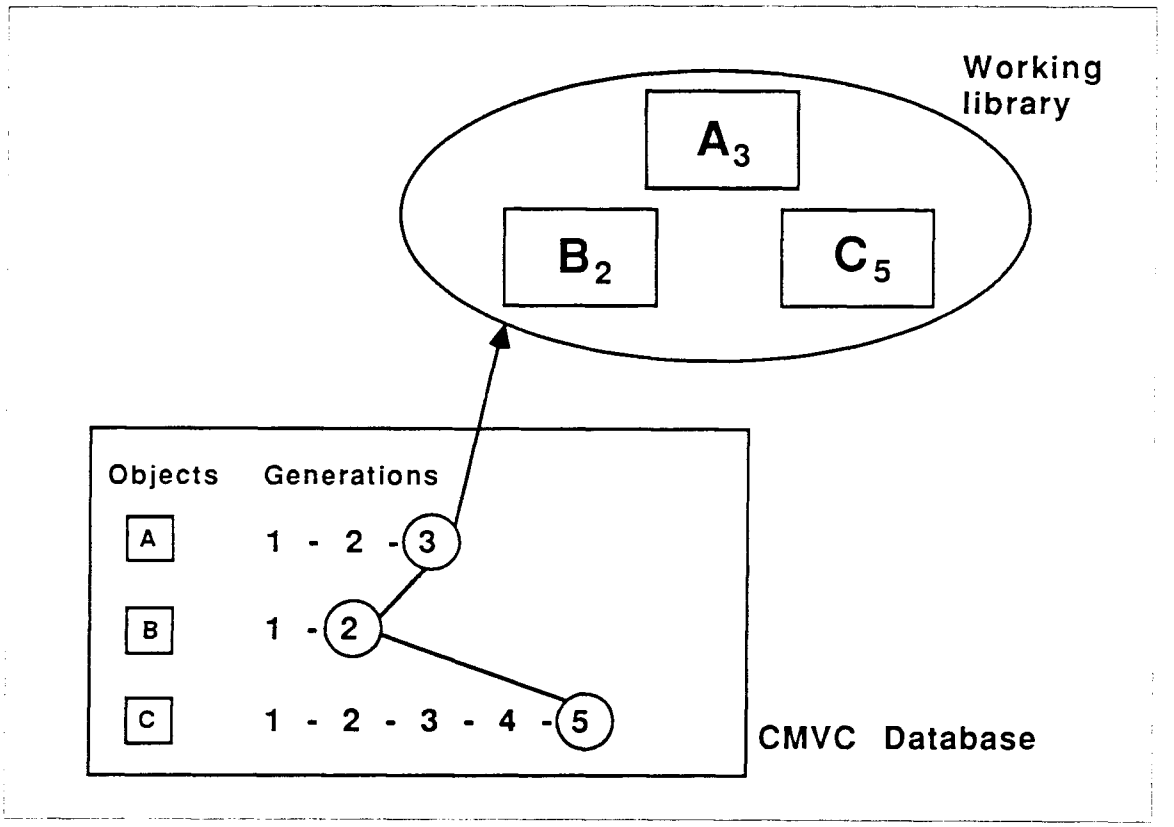


Figure 2-3. A Configuration Containing the Latest Object Generations

When a configuration of Ada units compiles satisfactorily in the working library, a *release* of that configuration can be made. Each release is a frozen copy of the working library and therefore is itself a full, compiled program library. Successive releases can be thought of as "views," where a view is a "snapshot" of the contents of a library at successive points in time. Accordingly, the released libraries and working libraries within a subsystem are called *views* (more specifically, *released views* and *working views*). A series of releases created from a single working view is called a development *path*, as shown in Figure 2-4.

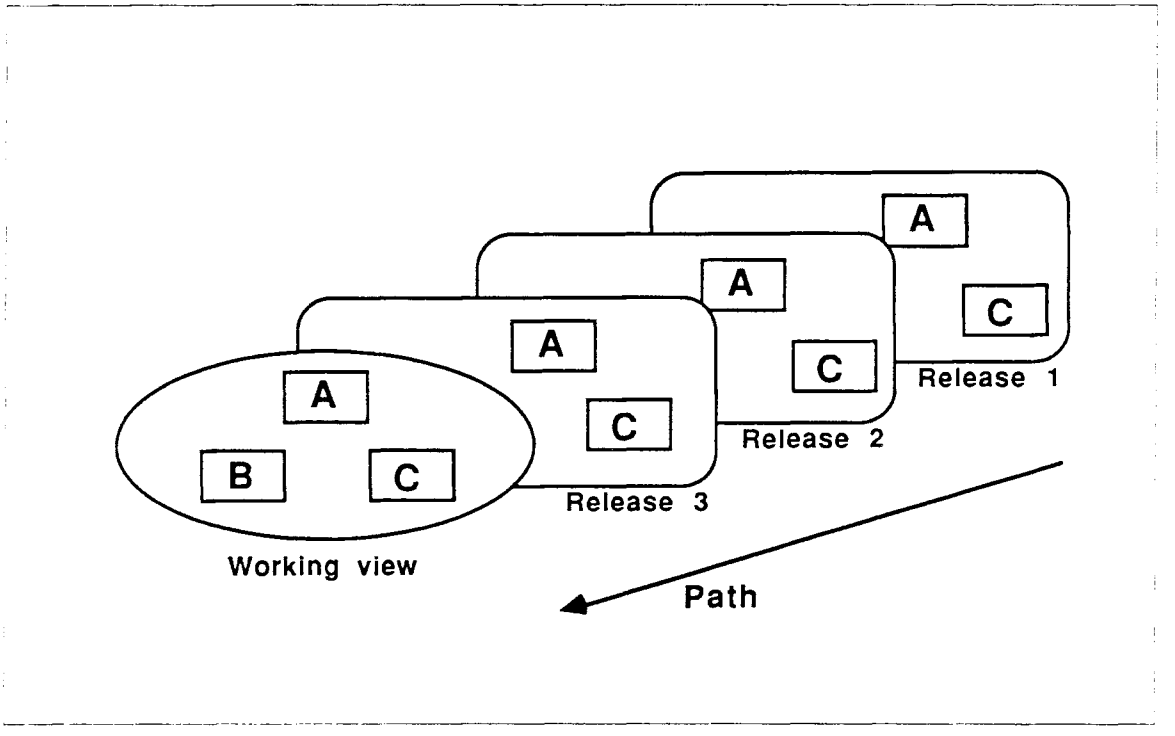


Figure 2-4. A Development Path

As shown in Figure 2-5, the CMVC database records not only individual object generations but also the configurations of generations embodied in each view. The CMVC database therefore makes it possible to revert to previous configurations or to reconstruct deleted views.

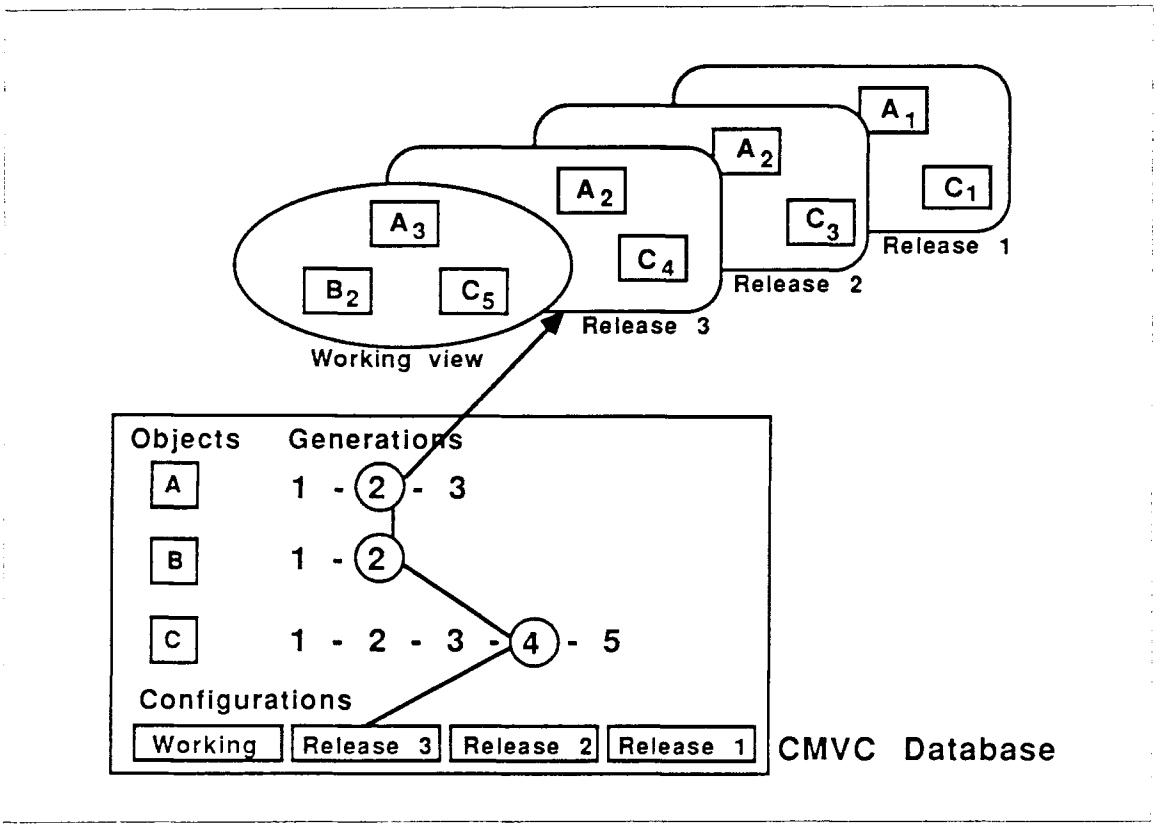


Figure 2-5. Configurations in a Development Path

It is important to bear in mind that each view is both:

- A source configuration, in that it specifies a particular generation for each object in the subsystem
- A program library, in that it enforces Ada semantic consistency among the specified generations

Thus, unlike other configuration-management facilities you may have used, CMVC operations integrate configuration management with library compilation management.

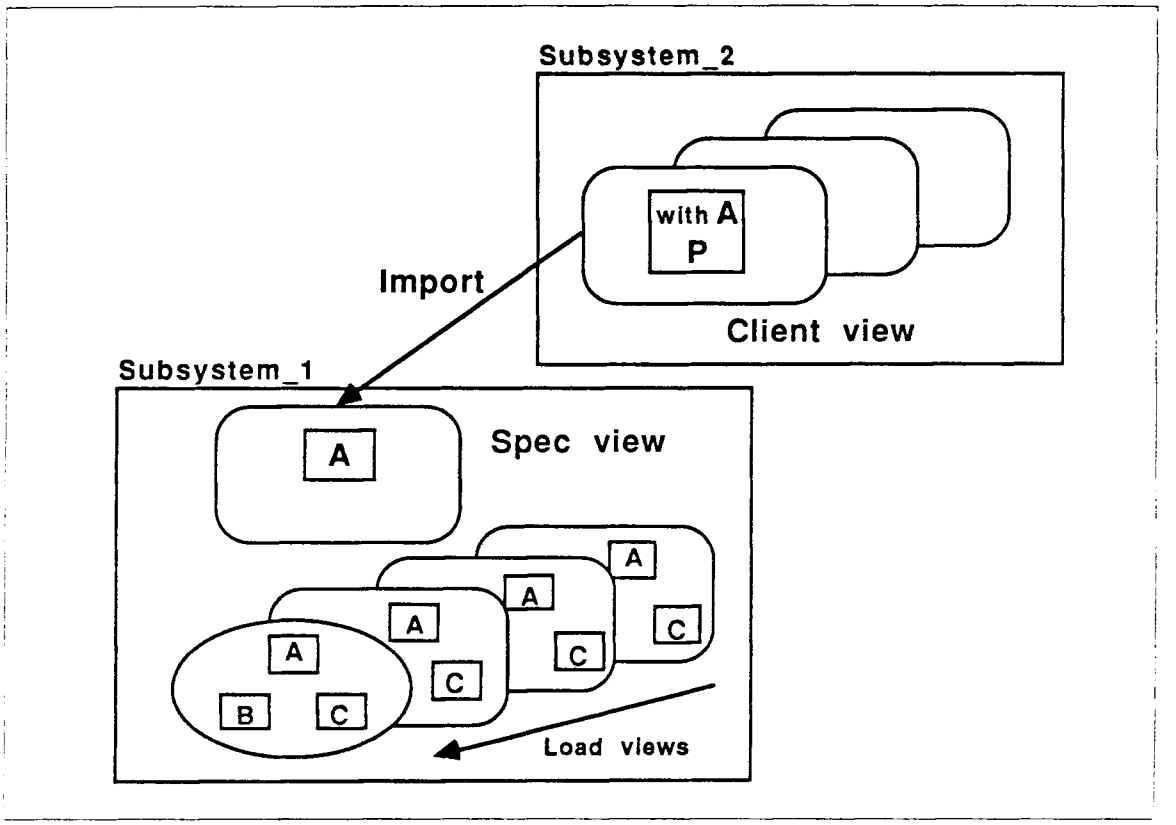


Figure 2-6. A Client View Importing a Spec View

Interfaces among Subsystems

Interfaces can be defined between subsystems using different kinds of views. The working and released views described above are *load views*; each load view contains a full implementation of the application component that is encapsulated in the subsystem.

A second kind of view, called a *spec view*, can be created to define the set of implemented units that are potentially available, or visible, to units in views of other subsystems. Spec views thus define a subsystem's *exports*; as such, spec views can be *imported* by *client* views in other subsystems. When a client view from one subsystem imports a spec view from another, dependencies can be set up among units from the two subsystems. Subsystem imports and exports thus enforce design decisions, because Ada context clauses (*with* and *use* statements) can reference nonlocal units only from imported spec views.

For example, Figure 2-6 shows that Subsystem_1 has load views implementing three units. In addition, a spec view has been created in Subsystem_1 in order to export one of those units. A client view in Subsystem_2 imports the spec view in Subsystem_1. As a result, units in the client view can *with* or *use* the exported unit.

Subsystem interfaces are analogous to Ada package interfaces:

- A spec view is analogous to an Ada package specification, which defines the resources that are available to client units.
- A load view is analogous to an Ada package body, which implements the resources promised by the specification. (However, only one Ada package body implements a package specification, whereas more than one load view can implement a given spec view.)
- At the subsystem level, the import relation is analogous to a *with* clause; importing enables a unit in a client view to actually *with* or *use* exported resources.

Because only spec views can be imported, client views compile against spec views, not load views. Therefore units in a working load view can be changed without requiring recompilation of any other views, provided that the working view remains *compatible* with the spec view that defines its exports. By definition, a load view is compatible with a spec view if it implements all of the resources made available by the spec view.

This definition of compatibility is broad enough to allow a load view to differ in certain specific ways from the spec view that represents it. For example, changes to the private part of an exported unit are one important kind of change that preserves compatibility. If a private type is changed in the load view, no change or recompilation is required of the spec view or any of its client views. In this way, subsystem interfaces make *closed private parts* possible.

By buffering recompilation for many kinds of changes, subsystem interfaces enable subsystems to be developed in parallel—a team of developers can change and test the implementation in its own subsystem without necessarily causing recompilation elsewhere. (However, design changes do not preserve compatibility and therefore require modification of both the spec and the load views; a changed spec view can affect client views in other subsystems.)

Program Execution

A subsystem typically contains at least one spec view, against which client views are compiled, and at least one load view, which contains the units that are actually executed. As releases are made from the working load view, a single subsystem typically accumulates multiple load views, each implementing the interfaces and capabilities specified in a given spec view.

To execute an application composed of such subsystems, an execution table called an *activity* must be set up to specify which of the alternative load views is to be used from each subsystem. The activity contains one entry for each subsystem that is required for execution. Activities thus specify combinations of load views for execution.

Figure 2-7 represents an activity used for executing a program containing two subsystems. In this program, each view in Subsystem_1 imports, and is compiled against, the spec view in Subsystem_2. The activity specifies that the shaded releases will be used during execution.

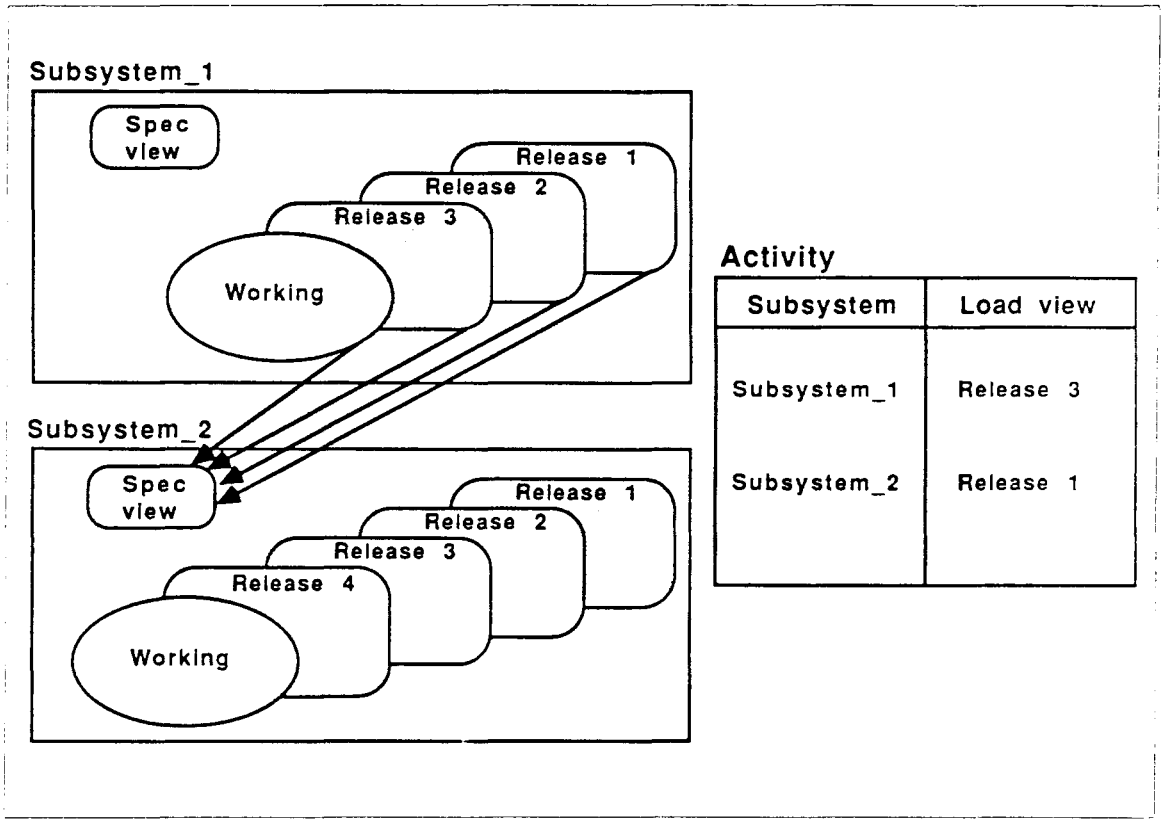


Figure 2-7. An Activity Used for Executing Two Subsystems

Activities provide a flexible means of constructing applications from a set of alternative subsystem implementations. A number of activities can be created; for example, one activity can specify all of the latest releases, and other activities can specify customer-specific releases and the like. Recombinant testing is a matter of editing an activity to specify precompiled views rather than recompiling an entire application from scratch.

By choosing appropriate views from each subsystem, you can construct system tests that include only one changed view. This will isolate the effects of specific changes for testing purposes. For example, to test a new implementation of a particular subsystem, an activity typically specifies the working view from that subsystem, along with stable, tested released views from the other subsystems in the application. The availability of both released and working views enables subsystem development and testing to proceed in parallel, because each team can continue to work in its own working view while other teams are testing against stable, frozen releases that have known characteristics.

Parallel Development within Subsystems

Parallel development is possible *within* subsystems as well as *between* them. When a team is assigned to implement a subsystem, a separate *subpath* can be created for each individual on the team. Subpaths are working views in which changes can be made and tested; they are created as full copies from the path's working view, as indicated in Figure 2-8.

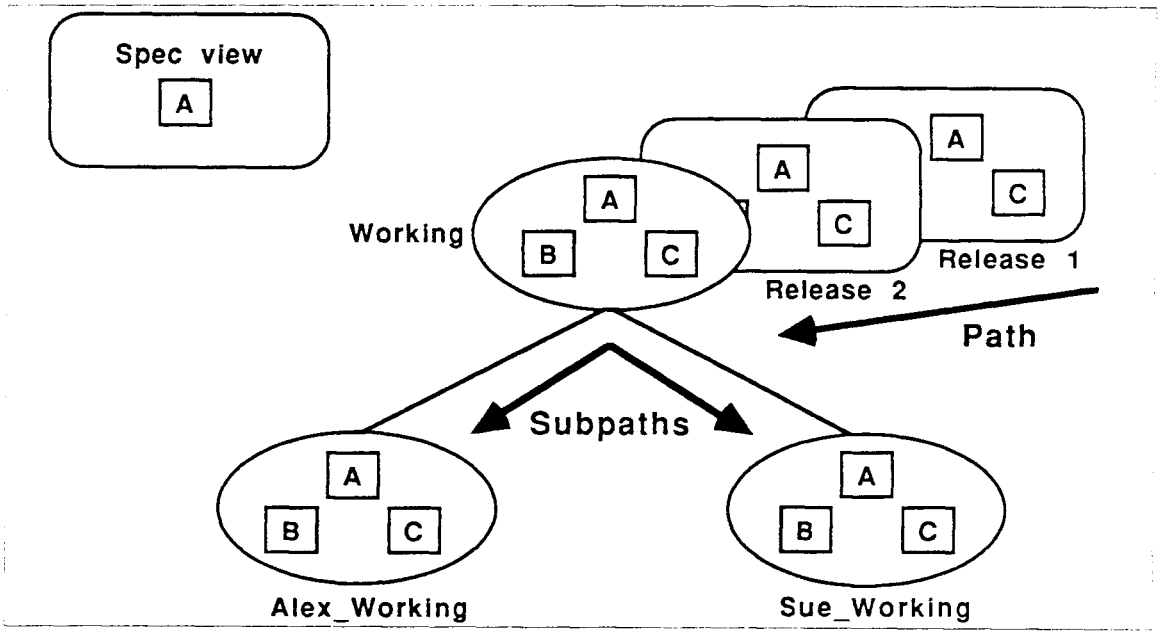


Figure 2-8. Subpaths Created from a Main Path

Editing can proceed without conflict because controlled objects are *joined* across subpaths. Each joined object shares a single reservation token with the corresponding object in the other subpaths; a given joined object can be checked out in only one subpath at a time. In this way, a single set of generations is maintained for multiple copies of an object. Figure 2-9 shows two subpaths containing joined objects.

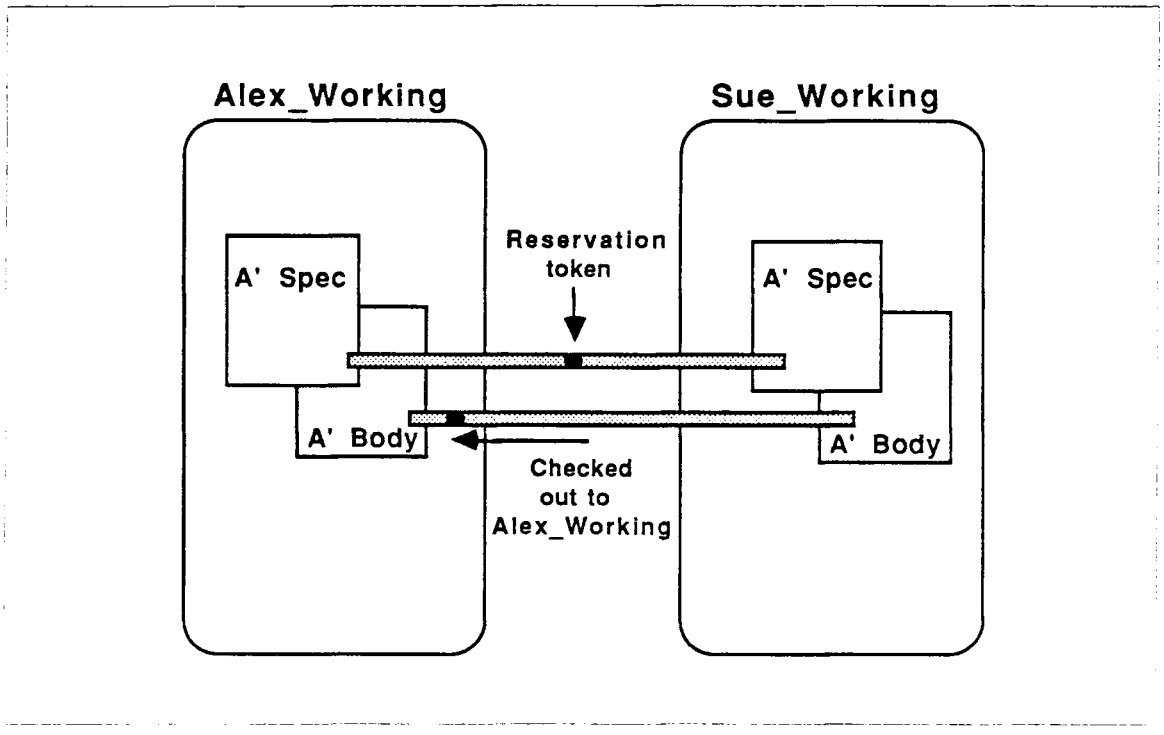


Figure 2-9. Objects Joined across Two Subpaths

A subpath can become out of date when objects are checked out and modified in other subpaths. Objects in a subpath can be brought up to date by *accepting changes*, usually from the latest generation into that subpath.

Implementers working in two subpaths can access an object concurrently if it is *severed*. Severing provides each copy of the object with its own reservation token, so that each copy can be checked out independently. Separate sets of generations are kept for severed objects. The contents of severed objects can be synchronized later by ***merging changes*** from one object to the other.

In addition to supporting parallel development among multiple developers, subsystems support the parallel development of different product variants (for example, variants of the same application that are to execute on different target processors). Separate development efforts within a single subsystem are maintained in multiple development paths. Thus, two development paths can be used to develop an application for execution on an R1000 and on an MC68020 microprocessor. Similarly,

maintenance of a previously delivered application can continue in one path while development of the next customer release proceeds along another path. Note that units shared by product variants can be joined across paths, whereas units specific to a particular variant can be severed.

Single-Library Applications and Documentation

Because subsystems are the context in which CMVC operates, they are useful even for developing documentation and applications that can be contained in a single library:

- Objects developed in a subsystem can be placed under CMVC control to save history information for objects that are checked out and checked in.
- Notes and comments about each object can be recorded (see "Project Reporting," below).
- Releases can capture configurations at important points during development.
- For single-library Ada applications, spec views and activities can be used to execute alternative implementations.

Project Reporting

Information about a CMVC-based project can be gathered in several ways. Each generation of every controlled object has *notes* associated with it, which can be used as a scratchpad for arbitrary commentary. In addition, the date, time, and comments from checkout and checkin commands are automatically logged in an object's notes. A scratchpad for notes is also associated with each release.

For more comprehensive project reporting, *work orders* can be used to define and assign units of development work, often referred to as development *tasks*. When development proceeds in response to a given work order, time-stamped comments are logged to the work order whenever any command from package Cmvc is executed. In addition, information characterizing individual tasks can be entered in user-defined fields on each work order. Work orders can be queried to find out their current status, who has done work against them, what units were affected, and the like.

Work orders can be grouped for easy reference using *work-order lists*. For example, a work-order list can reference all work orders assigned to a given developer or all work orders that have been closed.

All the work orders for a given class of tasks are created from a single template called a *venture*. Ventures are the place where user-defined work-order fields are created; ventures also specify *policies* that govern the work done in response to their work orders. For example, a policy can prevent a CMVC command from executing unless the parameter for comments is filled in.

Higher-Level Application Components

When an application consists of multiple subsystems, these subsystems optionally can be included in an Environment object called a *system*. Inclusion in a system is a way of identifying particular subsystems as components of a given application or of a major portion of an application. Inclusion in a system also provides an automated means of tracking the latest release from each subsystem and performing system builds by creating activities that reference those releases.

Subsystems that are included in a system are called *children* of the system. After a system's children are established, a *release activity* can be built in the system. A release activity automatically contains an entry for each child subsystem that specifies the latest release from that subsystem. When new releases are made in the child subsystems, the release activity can be rebuilt so that it references these new releases.

Release activities can be used to track current releases; they also can be used during execution of the application. Frozen versions of release activities can be maintained as releases within the system.

Multihost, Multisite Development

When an application is partitioned into subsystems, it can be developed on multiple R1000s, either at the same or at different geographic sites. Development on multiple hosts accommodates very large applications and is especially useful when program components are assigned to subcontractors.

When multiple R1000s are used, each one hosts a copy of every subsystem in the application. However, only one copy of a given subsystem, called the *primary subsystem*, supports ongoing development. The other copies, called *secondary subsystems*, essentially are local copies for execution and test.

Typically, each R1000 hosts a primary subsystem and some number of secondary subsystems. When a new release is made in a primary subsystem, that release can be copied via network or tape into the corresponding secondary subsystems on each of the other R1000s. On each R1000, the copied release then can be compiled with releases from the other subsystems and the application can be executed.

Note that instead of copying the source for a load view, a *code view* can be made on a primary subsystem and then copied into a secondary subsystem. A code view contains only the executable code from the compiled load view. This no-source view is especially useful when security requirements restrict visibility to portions of source code.

Getting Started

This chapter covers the fundamentals of developing a single subsystem, specifically:

- Creating and traversing a subsystem
- Using configuration management and version control (CMVC) within a subsystem

These operations apply to any given subsystem, whether or not it is part of a larger, multisubsystem program. This chapter focuses on creating one of three subsystems in a sample program.

The Sample Program

Assume that a team of developers is implementing a basic mail program that will enable users to:

- Send and receive messages
- Store messages in a mailbox
- Display and delete messages from the mailbox

Eight unit specifications have been defined, with the dependencies shown in Figure 3-1. At the highest level, the program has a main procedure called Run_Mail, which makes a series of commands available for manipulating mail messages. Run_Mail depends on Command_Utilities, which provides the interface for entering mail commands.

The basic object in the mail system is the message, which is defined in package Messages. Messages consist of lines (defined in package Lines) and have addresses to which they are sent (defined in package Destinations). Objects such as lines and destinations are strings of unknown length; resources for handling such strings are provided in package Unbounded. Package Symbolic_Display handles the display of messages. Finally, messages are stored in mailboxes, which are defined in package Mailbox.

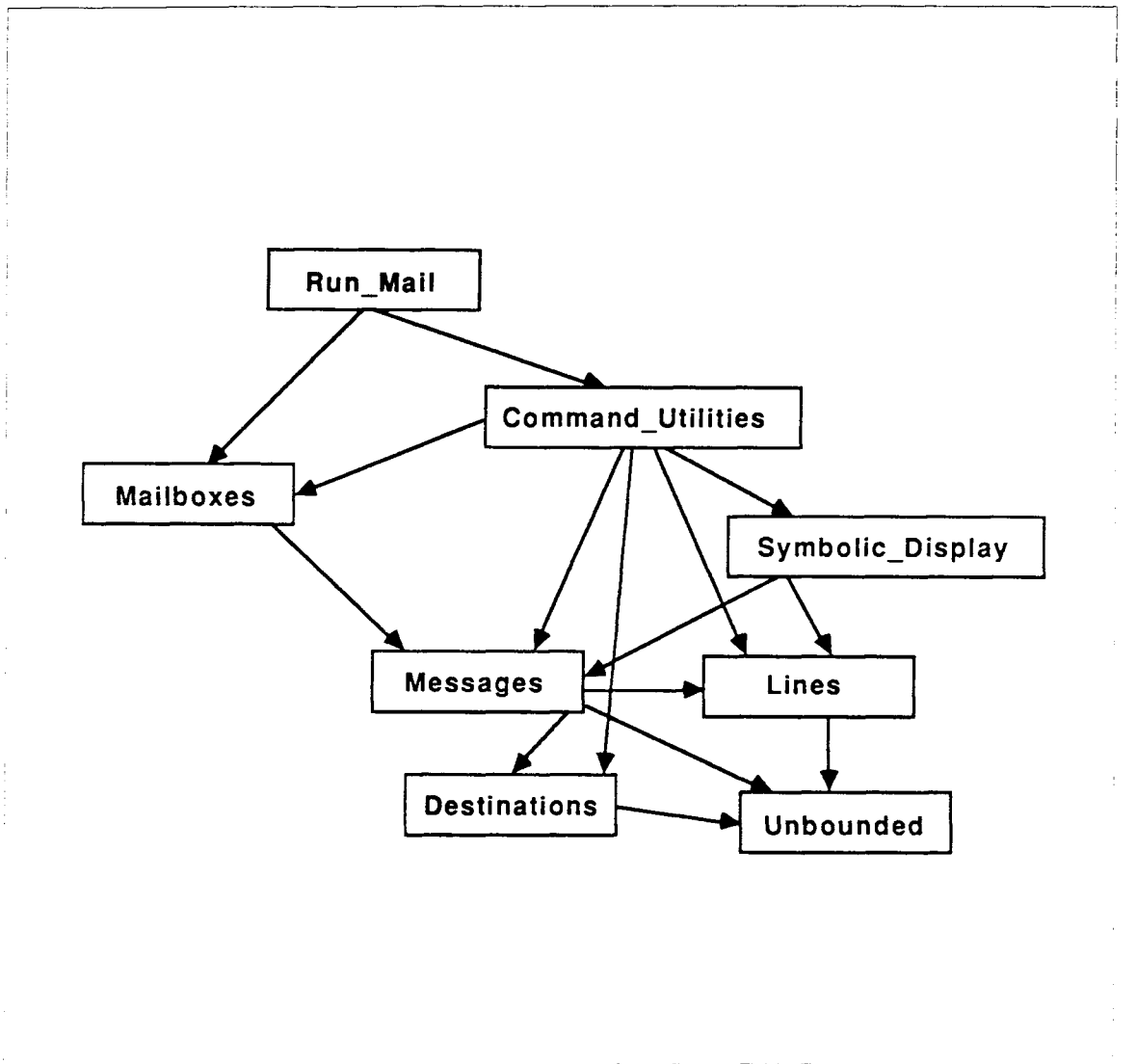


Figure 3-1. Units in the Mail Program

The project designer has decided to use three subsystems to partition the program's units into logical groupings. These subsystems are:

- Command_Interpreter, which contains units implementing the mail system's user interface
- Mailbox, which contains units implementing mailboxes for storing messages
- Mail_Utilities, which contains units implementing the mail system's basic elements

Figure 3-2 represents the proposed partitioning of the mail program. Dependencies are indicated at the subsystem level by heavy arrows. Note that the Mail_Utilities subsystem does not depend on either of the other subsystems in the program (that is, no units in the Mail_Utilities subsystem depend on units from other subsystems). In this sense, the Mail_Utilities is the bottom layer of the program and will be developed first.

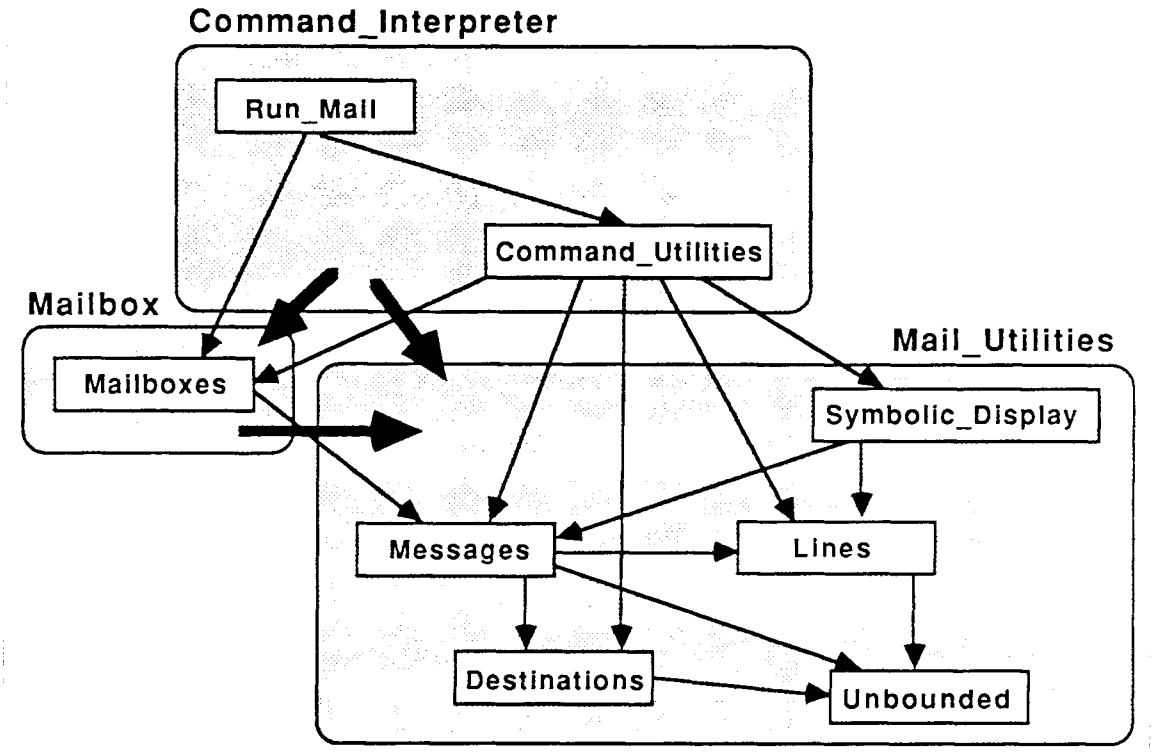


Figure 3-2. The Mail Program Partitioned into Subsystems

Creating a Subsystem

The subsystems of a program such as the mail system typically are created within a single project library. A project library is a world that eventually may contain not only subsystems, but also libraries for documentation, an optional main driver for the program, and any activities used for executing the entire program.

In this example, assume that a project library called !Programs.Mail exists and that you are responsible for creating the Mail_Utilities subsystem in it.

To create a subsystem:

1. In a Command window, enter the Cmvc.Initial command with the name of the subsystem you want to create. For example:

```
Cmvc.Initial (System_Object => "Mail_Utilities");
```

Assuming that the command was entered in the context !Programs.Mail, the subsystem !Programs.Mail.Mail_Utilities is created.

For most purposes, it is sufficient to specify only the System_Object parameter and to use the default values for the remaining parameters.

Internal Structure of a Subsystem

Subsystems are created containing various libraries, which in turn contain directories, files, and other objects. Although much of this predefined internal structure is for the Environment's use, there are several libraries and files that users need to know about.

Figure 3-3 shows the Mail_Utilities subsystem.

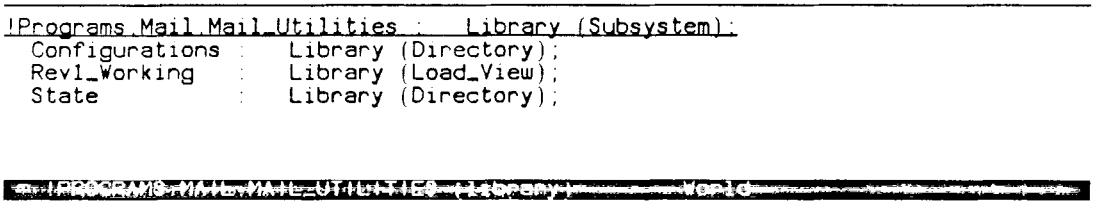


Figure 3-3. The Mail_Utilities Subsystem

As shown in Figure 3-3, the newly created subsystem contains three libraries:

- A directory called Configurations, which contains summary information about each view in the subsystem (see "Representation of Releases," later in this chapter)
- A program library called Rev1_Working, in which your ongoing work takes place (see "Working Views," below)
- A directory called State, which contains information about the underlying objects in the subsystem

Within a subsystem, users can create other directories for tests or documentation, as desired. (However, worlds must not be created within subsystems.)

Working Views

Every subsystem is automatically created containing a *working view.* A working view is the program library in which the subsystem's Ada units actually are developed. Although frozen "snapshots" of your work (called *releases*) can be made from a working view, the working view itself is never frozen, so it is always available for further development.

By convention, the working view's name ends with the string "_Working". The first portion of the working view's name is specified by the Working_View_Base_Name parameter of the Cmvc.Initial command. Because the example used the default value ("Rev1") for this parameter, the name of the subsystem's working view is Rev1_Working.

Views such as Mail_Utilities.Rev1_Working are created with predefined internal structure. As shown in Figure 3-4, Mail_Utilities.Rev1_Working contains four directories.

```
!Programs.Mail.Mail_Utilities.Rev1_Working :   Library (Load_View);
  Exports :    Library (Directory);
  Imports :    Library (Directory);
  State   :    Library (Directory);
  Units   :    Library (Directory);
```

Figure 3-4. The Working View Mail_Utilities.Rev1_Working

As in subsystems, views can contain additional user-created directories (not worlds).

A view's four predefined directories are:

- Exports, in which users can create *export restriction files* (see the chapter entitled "Developing Applications Using Multiple Subsystems")
- Imports, in which users can create *import restriction files* (see the chapter entitled "Developing Applications Using Multiple Subsystems")
- State, which contains files and other objects that provide information about this view to various CMVC commands
- Units, in which the subsystem's Ada units will be created and edited (note that other user-defined objects, such as text files, also can be kept in the Units directory)

Of these directories, it is the Units directory in which you will do your day-to-day work. The other directories are described in the chapter entitled "Developing Applications Using Multiple Subsystems."

Note that subsystems and views both contain directories called State. In the course of developing a program, you may need to edit or view one or more of the files in the view's State directory; however, you normally will not need to visit the subsystem's State directory.

Predefined Library Characteristics

By default, the working view within a subsystem is created with certain predefined library characteristics such as library switches, target keys, links to Environment commands and tools, and additional user-defined subdirectories. These library characteristics are copied from a *model world*, which is specified by the Model parameter in the Cmvc.Initial command.

The links provided by a view's model world determine the Environment resources that are visible to the units in that view. Note that a model controls visibility only to units that are not defined in other subsystems. Units defined in other subsystems are made visible through *imports*.

The Environment provides several standard model worlds that provide different sets of links:

- !Model.R1000 provides links to most common Environment commands and tools—for example, packages !Tools.String_Utilities, !Tools.Time_Utilities, and the like. !Model.R1000 is the default model world specified by the Cmvc.Initial command and was used in creating Mail_Utilities in the sample program.
- !Model.R1000_Portable provides links only for Ada-specified standard facilities—for example, packages Calendar, Text_Io, Unchecked_Conversion, and the like. As its name implies, !Model.R1000_Portable ensures a program's portability.
- !Model.R1000_Implementation provides links for system programming facilities and for many programmatic interfaces to the Environment.

When creating a subsystem, you can choose among the standard model worlds or you can specify a user-defined model world, which can be any existing Environment world with the desired links, switches, and the like. A view's model world can be replaced using the Cmvc.Replace_Model command. For more information on models, see "Setting Up Subsystems: A Second Look," in the chapter entitled "Developing Applications Using Multiple Subsystems."

Setting Up the Units Directory

After a subsystem has been created, you can prepare the Units directory in either of the following two ways:

- Create Ada units and files directly in the Units directory of the subsystem's working view
- Copy Ada units and files from other Environment libraries into the Units directory

In the Mail_Utilities example, assume that the specifications for packages Unbounded, Messages, and Lines already exist in another Environment library. In this case, you can use the Library.Copy command to put these objects into the Units directory of Rev1_Working in the Mail_Utilities subsystem. The bodies for these packages, as well as the specifications and bodies for Symbolic_Display and Destinations, can be created in the Units directory. Figure 3-5 shows the Units directory after units have been created and copied.

```
!Programs.Mail.Mail_Utilities.Rev1_Working.Units :   Library (Directory):
  Destinations       : S Ada (Pack_Spec);
  Destinations       : S Ada (Pack_Body);
  Lines              : S Ada (Pack_Spec);
  Lines              : S Ada (Pack_Body);
  Messages           : S Ada (Pack_Spec);
  Messages           : S Ada (Pack_Body);
  Symbolic_Display   : S Ada (Pack_Spec);
  Symbolic_Display   : S Ada (Pack_Body);
  Unbounded          : S Ada (Pack_Spec);
  Unbounded          : S Ada (Pack_Body);
```

```
MAIL_UTILITIES REV1_WORKING.UNITS (library)          Directory
```

Figure 3-5. The Units Directory in Mail_Utilities.Rev1_Working

Figure 3-6 represents the internal structure of the Mail_Utilities subsystem.

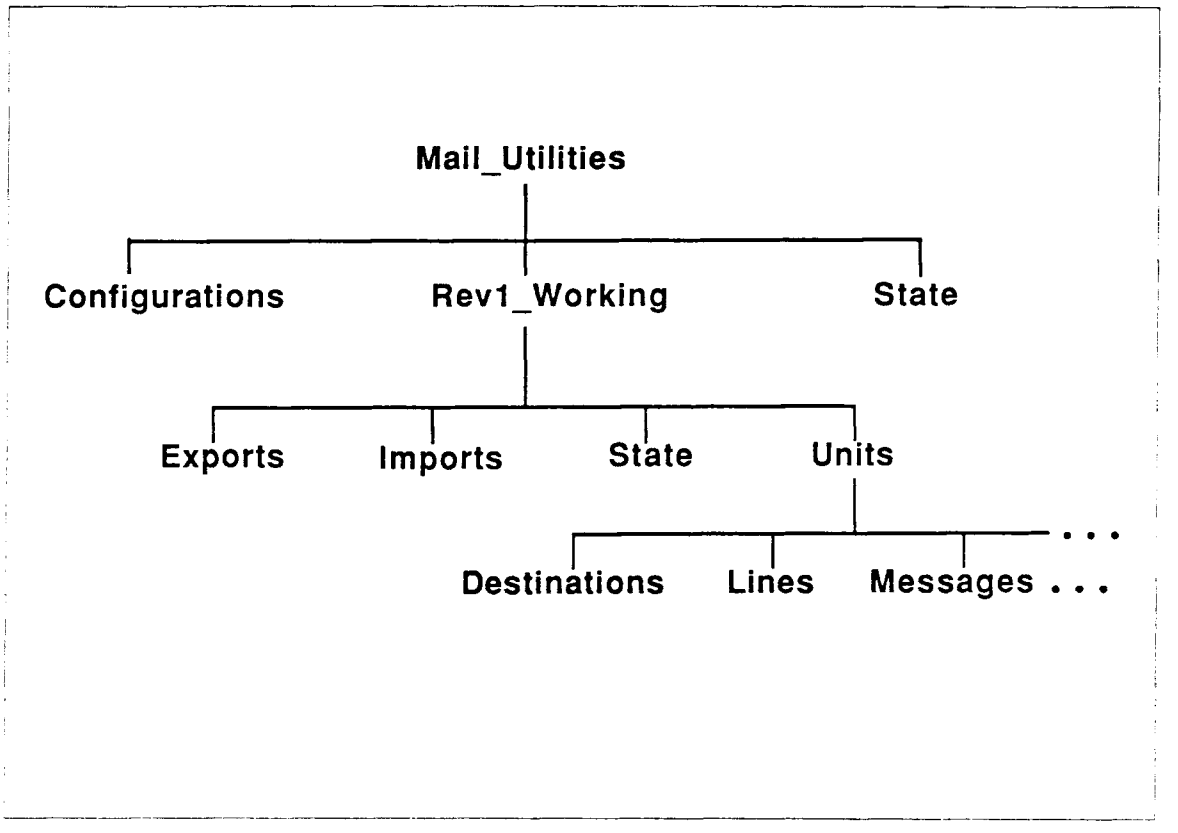


Figure 3-6. The Structure of the Mail_Utilities Subsystem

If you need to accommodate a large number of units, you can organize these units in the following ways. You can:

- Subdivide the Units directory by creating other directories within it
- Create additional directories in the view, at the same level as the Units directory

Note that directories, not worlds, should be created. Furthermore, such user-defined directory structure can be created automatically by predefining it in the model world; see "Setting Up Subsystems: A Second Look," in the chapter entitled "Developing Applications Using Multiple Subsystems."

Controlling Objects Using CMVC

Objects in a working view can be put under CMVC to track change history. When an object is made controlled using CMVC, that object is registered in the CMVC database for the enclosing subsystem. (Each subsystem has its own CMVC database.) All changes made to a controlled object are recorded in the CMVC database to permit the reconstruction of earlier versions of the object, if needed.

Controlled objects must be reserved, or *checked out*, before they can be modified. Checking out an object instructs the CMVC database that a new *generation* of the object is to be created. The CMVC database associates a *reservation token* with each object; these tokens are used by the CMVC database to keep track of checked-out objects.

Any file, Ada unit, or subunit in a working view can be made controlled, including objects in user-defined directories. (Note, however, that objects in the view's State directory cannot be made controlled, because the Environment must be able to access these objects freely.)

You can make an object controlled at any time during its development; change history is recorded from that point on. The earlier in its development that you make an object controlled, the more change history will be recorded for that object. Note that you should make an Ada unit controlled only after its Ada name (for example, Destinations) appears in the Units directory listing. Do not make a new Ada unit controlled while it has a temporary name (for example, "_Ada_8_").

To make one or more objects controlled:

1. In a Command window, enter the Make_Controlled command with the names of the objects to be controlled. You can use a naming expression to specify multiple objects.

   For example, assume that you are viewing the Units directory of Mail_Utilities-.Rev1_Working. Entering the following command from this context controls all the units in the Units directory:

   ```
   Cmvc.Make_Controlled (What_Object => "@");
   ```

Note that objects cannot be created controlled; they must be made controlled explicitly. It is not necessary to make every object in a view controlled.

Special Note: Controlling Binary Objects

By default, controlled objects are represented textually in the CMVC database; changes to such objects are recorded as changed lines of text. This is appropriate for objects such as Ada units and text files. However, such recording is not appropriate for binary objects, which may not have an ASCII representation. Therefore, when making binary objects controlled, you must request that the CMVC database not "save source" for these objects. To do this, enter the value false for the Save_Source parameter of the Cmvc.Make_Controlled command.

Objects for which source is not saved still must be checked out before they can be modified; however, earlier generations cannot be reconstructed for them. Throughout this and subsequent chapters, it is assumed that all objects have source saved for them.

Editing Controlled Objects

Before you can edit a controlled object, you must check it out using the Cmvc.Check_Out command. For example, assume that you want to edit Messages'Body. To do so:

1. Select the desired unit or put the cursor in its image.
2. Enter the Cmvc.Check_Out command, optionally filling in the Comments parameter (see "Collecting and Displaying Information about Generations," below).
3. Display the unit and open it for editing with the Common.Edit command.

You can check out multiple objects by using a naming expression with the Cmvc.Check_Out command.

Note that Cmvc.Check_Out checks out a unit to a view, not to a particular user. Consequently, if a unit in Mail_Utilities.Rev1_Working.Units is checked out, anyone with access to Mail_Utilities.Rev1_Working can modify the object.

When you have finished modifying the object and you want the changes you made to be recorded in the CMVC database, you must check in the object. For example, to check in a displayed object:

1. Select the unit you want to check in or put the cursor in its image.
2. Enter the Cmvc.Check_In command, optionally filling in the Comments parameter (see "Collecting and Displaying Information about Generations," below).

   If the object was open for editing, the Cmvc.Check_In command closes it.

You can use a naming expression with the Cmvc.Check_In command to check in multiple objects.

Generations and Versions

Each time you check out an object, a new *generation* of that object is created in the CMVC database. Editing changes are collected in the new generation and saved in the CMVC database only when you check in the object. Thus generations capture the changes made from checkout to checkout.

Each generation of an object is numbered, starting with generation 1. Generation 1 is created when you make an object controlled; initially generation 1 contains the text of the object at the time it was made controlled. Over time, the CMVC database builds up a series of numbered generations for each controlled object.

A generation differs from a *version*, which is created each time you use Common.Edit to open a unit for editing. Because you can open and close a unit for editing multiple times while it is checked out, a given generation can include the changes made in multiple versions. (The same is true for text files, except that new versions of text files are created whenever you use Common.Commit.) The relationship between versions and generations is shown in Figure 3-7.

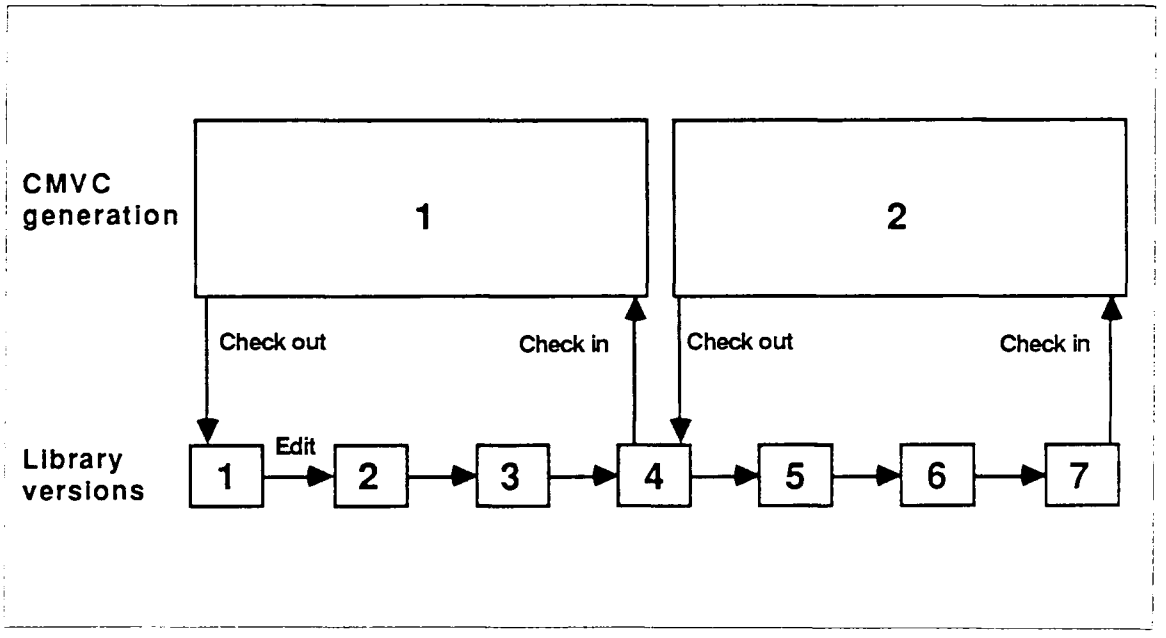


Figure 3-7. Generations and Versions

Note that versions are saved in libraries, and the number of saved versions is limited by the retention count for the library. In contrast, generations are saved in the CMVC database, which saves every generation back to generation 1. (To save space, successive generations are saved as changed lines of text from which full textual images can be reconstructed.)

Canceling a Checkout

Although checking out an object reserves a new generation, the generation is saved in the CMVC database only when the object is checked in. If you check out an object and then decide that the new generation should not be saved, you can cancel the checkout instead of checking the object back in. You can cancel a checkout by *abandoning* your reservation on the object. Abandoning a reservation discards a generation whether or not changes have been made during that generation.

For example, assume that checking out Destinations'Body creates generation 4 and that you have made changes that you want to discard. As long as you have not checked in Destinations'Body, you can cancel generation 4 and return the unit to generation 3. To abandon your reservation on an object:

1. Designate the object whose reservation you want to abandon.
2. Enter the Cmvc.Abandon_Reservation command, using default parameters.

Reverting to a Previous Generation

You can go back to any previous generation of an object using the Cmvc.Revert command. An object can be reverted on a temporary basis or it can be reverted so that subsequent development can proceed from the older generation.

Reverting an object on a temporary basis means that you can look at a previous generation, compile against it, and even make releases including it. However, if you check out the reverted object, its latest generation is retrieved from the CMVC database, so that subsequent development proceeds from the latest generation. For example, to revert Symbolic_Display'Body from generation 5 to generation 4 on a temporary basis:

1. Designate the object you want to revert. The object must not be checked out currently.
2. Enter the Cmvc.Revert command, using default parameter values. (By default, the Cmvc.Revert command goes back one generation.)

If you check out Symbolic_Display'Body at this point, generation 5 is restored and made available for editing.

Now assume that you want to revert Messages'Body from generation 8 and continue development from generation 6. To do this:

1. Designate the object you want to revert. The object must not be checked out currently.
2. Enter the Cmvc.Revert command, using nondefault values for the To_Generation and Make_Latest_Generation parameters:

```
Cmvc.Revert (To_Generation => 6,
             Make_Latest_Generation => True);
```

As a result, a new generation (generation 9) is made, which contains a copy of the contents of generation 6.

Collecting and Displaying Information about Generations

When you use the Cmvc.Check_Out and Cmvc.Check_In commands, you can provide commentary through the Comments parameter. This commentary is stored with the *notes* for the relevant generation. You can view the notes for a generation using the Cmvc.Notes command. Additional information can be entered into a generation's notes using the Common.Edit and Common.Commit commands from the window displayed by Cmvc.Notes. The Cmvc.Notes command also displays the date and time at which the generation was checked out and checked in.

You can view change history between specified generations of an object using the Cmvc.Show_History_By_Generation command. Among other things, this command displays:

- The notes for each generation
- When each generation was created
- The lines that were changed from generation to generation

An alternative method for viewing change history is to display an *expanded generation image* using the Cmvc.Edit command. See the introduction to package Cmvc for information about generation images.

Compiling Units in a Subsystem

Because a view is a program library, units in a view are compiled using the same Environment facilities that are used for compiling units in worlds and directories. Units can be promoted from the source state, through the installed stated to the coded state, and then executed. As usual, the Environment automatically determines the compilation order. Units need not be checked in to be compiled.

The units in Mail_Utilities.Rev1_Working can be compiled successfully because the model world for the view provides all of the required links to Environment resources. If the units in a view depend on units from views in other subsystems, then those views must be imported to make compilation successful.

Test programs can be developed and executed within Mail_Utilities to unit-test the resources in each package. However, an *activity* is required for execution in views that depend on imports.

Note that you can make a spec view at this point to define the units to be exported from the Mail_Utilities subsystem. However, you need do this only when another subsystem is ready to compile against the units in Mail_Utilities. Making a spec view is not required for development within a single subsystem.

Creating spec views, importing, and using activities are covered in the chapter entitled "Developing Applications Using Multiple Subsystems."

Releasing Configurations

As you develop units in a working view, you can preserve certain significant combinations (*configurations*) of generations. You can do this by making a *release* from the working view. A release typically represents a baseline configuration that has been compiled and tested, and thus is considered stable and usable for execution by other subsystems. Releases also can serve as reference points in the development history of a single subsystem.

Several kinds of releases can be made, depending on your needs:

- Released views, which preserve both the source code and the compilation information to permit execution.
- Configuration releases, which preserve enough information about configuration state to permit the construction of released views.
- Code views (also called *code-only releases*), which permit execution without making program source code available. Code views typically are made from a working view for use by the developers of other subsystems, particularly when the subsystems are developed on different R1000s. See the Cmvc.Make_Code_View command.

Released views and configuration releases are discussed below.

Released Views

A released view (also called a *full-view release*) is a complete, frozen copy of a working view. As such, a released view contains program source code, and, if the released view has been compiled, the units in the released view can be executed. You should make a released view from a compiled working view whenever you want to both preserve a configuration in a working view and be able to execute its units. Note that when a release is made from a working view that contains compiled Ada units, the release is created by copying the compiled units; no recompilation is necessary when creating releases.

Released views are created using the Cmvc.Release command with default parameter values. For example, assume that development in Mail_Utilities.Rev1_Working has reached the point at which you want to release this view. To make a release from a working view:

1. Compile the view's units, if desired. (Once the released view is created, it is frozen, so units in it cannot subsequently be compiled or otherwise changed.)
2. Designate the working view to be released. For example, select the view's entry in the subsystem image.
3. Enter the Cmvc.Release command, using default parameters.

As shown in Figure 3-8, a released view named Rev1_0_1 is created within Mail_Utilities. (Naming conventions are covered in "Release Names," below.)

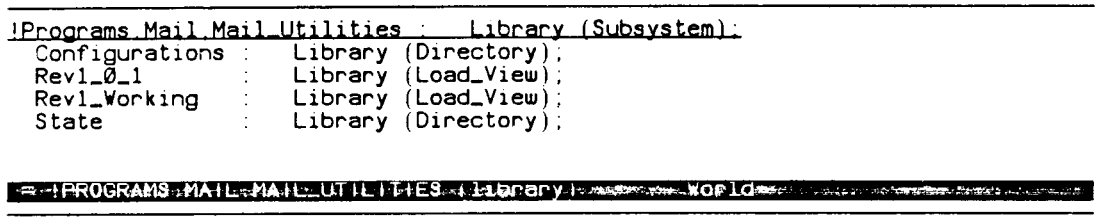


Figure 3-8. The Mail_Utilities Subsystem with Released and Working Views

A released view has the same predefined libraries and library characteristics as the working view from which it was created.

Configuration Releases

A configuration release preserves the state of a working view, without creating a released view. As such, a configuration release is a summary of configuration information from which a released view subsequently can be constructed, if desired. You should make a configuration release when you want to keep a record of a particular configuration, but you do not need to execute the units immediately. Making a configuration release is faster and uses less storage than making a released view. However, reconstructing a released view from a configuration release requires complete recompilation of the Ada units within the view.

To make a configuration release:

1. Compile the view's units, if desired.
2. Designate the working view to be released. For example, select the view's entry in the subsystem image.
3. Enter the Cmvc.Release command, changing the default value of the Create_Configuration_Only parameter to true:

```
Cmvc.Release (Create_Configuration_Only => True);
```

Representation of Releases

Making any kind of release creates two objects in the subsystem's Configurations directory, namely:

- A *configuration object*, which essentially contains a list of particular generations of controlled objects in that view
- A *state description directory*, which contains files that store switch values, the names of exported and imported views, the name of the model world, and the like

When you make a released view, both of these objects are created in addition to the frozen view. In contrast, when you make a configuration release, only the objects in the Configurations directory are created.

For example, Figure 3-9 shows the Mail_Utilities subsystem and the Mail_Utilities-.Configurations directory after both a released view (Rev1_0_1) and a configuration release (Rev1_0_2) have been made. Note that a configuration object has the same simple name as the corresponding released view.

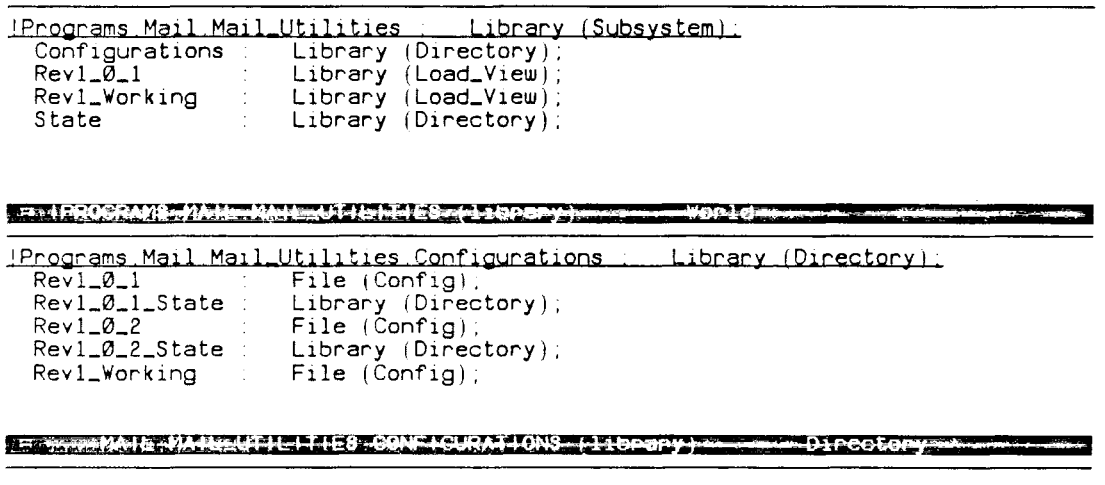


Figure 3-9. The Mail_Utilities Subsystem and Its Configurations Directory

Together, a configuration object and a state description directory contain all the information the Environment needs to construct a released view from the history stored in the CMVC database. (Use the Cmvc.Build command.) However, a view constructed from a configuration release may differ from a released view as follows:

- A released view contains a copy of every object from the working view, controlled or not.
- A view constructed from a configuration release contains only controlled objects for which source is saved in the CMVC database.

Because configuration objects are created for each released view, you can destroy released views to save space and later reconstruct them, if needed. (Use Cmvc.Destroy_View and Cmvc.Build, respectively.) Note once again that only controlled objects can be rebuilt. See "Managing Views" in the chapter entitled "Coordinating Development in a Subsystem."

Development Paths

A working view followed by a sequence of releases through time is called a *development path.* In a path, ongoing development continues in a working view and releases serve as "snapshots" of the working view made over time. As releases are made, a single subsystem comes to contain many views and configuration objects, each representing an alternative implementation of the program component encapsulated by the subsystem. Figure 3-10 shows the development path in Mail_Utilities with several releases, including a configuration release (shown with broken lines).

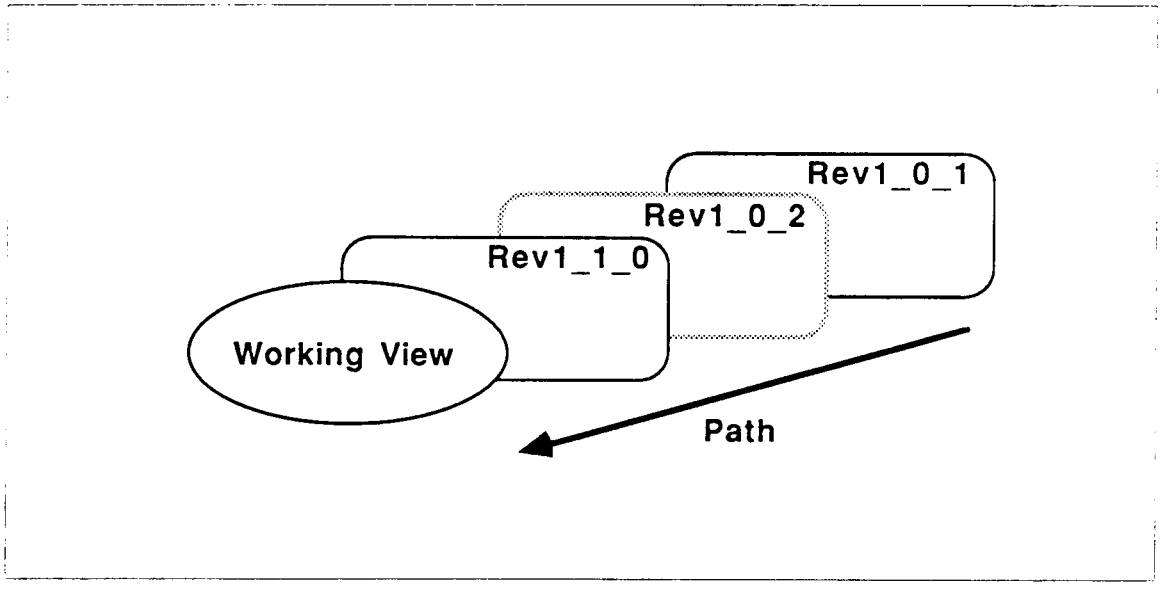


Figure 3-10. A Development Path in Mail_Utilities

Release Names

The Environment automatically constructs the names of released views and configuration objects (code-view names are entirely user-defined). A released view and its corresponding configuration object share the same simple name, which is constructed from two components:

- A *pathname prefix* (for example, "Rev1"). By convention, the pathname prefix of a view name is the portion of the name up to the first underscore.
- A set of *release level numbers* (for example, "_0_1").

By default, the pathname prefix in a release name is the same as the base name of the working view from which the release was made. In the Mail_Utilities example, "Rev1" appears in the release name because the working view base name is "Rev1".

Although the working view and the releases in a path share the same name prefix by default, it is possible to distinguish a special release by overriding autogenerated release names. To do this, you can specify a nondefault string for the Release_Name parameter of the Cmvc.Release command. The specified string is used as the entire release name, without adding release level numbers. For example, you may want to specify a release name like "Field_Release" for a release that is shipped to customers, using "Rev1" in the names of all interim development releases.

Release Level Numbers

The pathname prefix in a release name is followed by a set of release level numbers, which are separated by underscores. By default, two release level numbers are provided, which you can use to define a series of major and minor releases. The path in Figure 3-10 shows two releases at the minor level (Rev1_0_1 and Rev1_0_2) followed by one major level release (Rev1_1_0).

Release numbers are incremented automatically by the Environment at the minor level, unless you specify a different level using the Levels parameter in the Cmvc.Release command. In a released view, release numbers are incremented regardless of the kind of release.

The number of release levels that can be incremented is determined by a user-created file called Levels in the subsystem's model world. This file should contain a single integer. (If the model contains no Levels file, two release levels are used.)

Note that the release level numbers replace the "_Working" suffix that appears in the working view name:

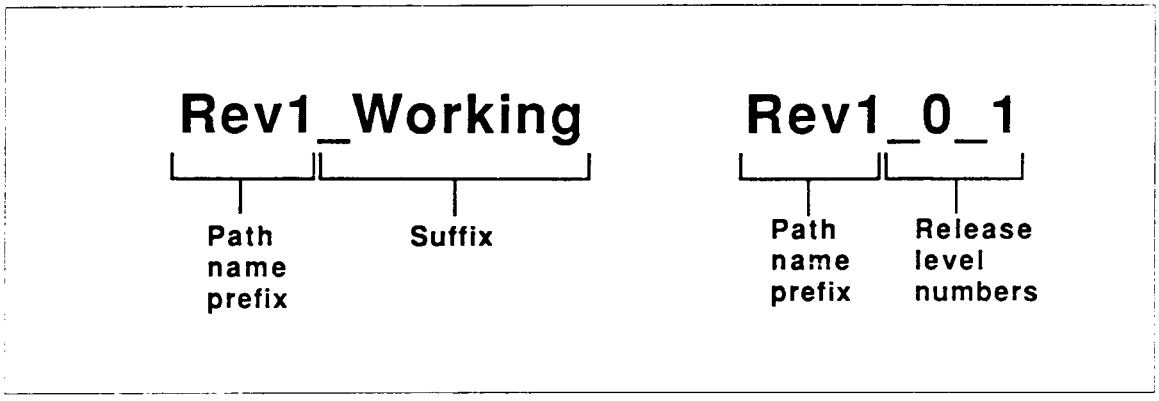


Figure 3-11. Structure of View Names

Library Management Operations for Controlled Objects

Library management operations, such as deleting, withdrawing, moving, and renaming objects, involve extra steps when those objects are controlled. In most cases, the extra steps are required to remove the objects in question from CMVC control so that the change can be made.

Deleting Objects

To delete a controlled object:

1. Remove the object from CMVC control with the Cmvc.Make_Uncontrolled command.
2. Delete the object using, for example, Common.Object.Delete.

Withdrawing Objects

To withdraw a controlled object:

1. Remove the object from CMVC control with the Cmvc.Make_Uncontrolled command.
2. Withdraw the object using, for example, Ada.Withdraw.

Moving Objects

Objects must be made uncontrolled for other operations that involve implicit deletion. For example, assume that you have several subdirectories within the Units directory of a view, and you want to move an object from one subdirectory to another. To do this:

1. Remove the object from CMVC control with the Cmvc.Make_Uncontrolled command.
2. Move the object to the desired directory using, for example, Common.Object_.Move.
3. Make the object controlled again using Cmvc.Make_Controlled.

Note that objects are known to the CMVC database by their pathname within the view. Moving an object from one subdirectory to another within a view changes that pathname and thus involves implicitly deleting one controlled object and creating a new one. History for the object in its original directory is still maintained by the CMVC database under the object's original name; the object in its new location is made controlled as generation 1.

Renaming Ada Units

Renaming an Ada unit entails withdrawing it from the library, which involves implicit deletion. To rename an Ada unit:

1. Remove the unit from CMVC control with the Cmvc.Make_Uncontrolled command.
2. Withdraw the unit from the library, using Ada.Withdraw. The unit's Ada name is replaced with a temporary name, such as "_Ada_6_".
3. Edit the unit to change its Ada name.
4. Install the unit with its new name in the library using, for example, Ada.Install_Stub.
5. Make the object controlled again using Cmvc.Make_Controlled.

Note that changing a unit's kind (for example, from function to procedure) without changing its name must be done with care. This kind of change requires that the CMVC database be expunged, which results in loss of history for that object. See the Cmvc_Maintenance.Expunge_Database command.

Coordinating Development in a Subsystem

When a subsystem encapsulates a program component that is large enough, it may be necessary to assign that subsystem to a team of developers rather than to a single developer. When a team of developers needs to work on units in the same subsystem, multiple development *subpaths* can be set up within the subsystem to facilitate parallel development. A separate subpath can be assigned to each developer on the subsystem team.

In addition to separate subpaths within a single path, multiple paths also can be set up to accommodate distinct development efforts within the subsystem. Multiple paths are necessary when different development efforts require variant implementations of the subsystem—for example, for multiple-target development. Whereas paths represent distinct variants of an application component, subpaths within a path are intended for the eventual release of a single variant.

Creating a Subpath

Subpaths are full copies of the working view in a path and, as such, are working views themselves. Within each subpath, units can be edited, compiled, and tested, as in any program library. Because each subpath contains a copy of every unit in the path, compilation and execution in one subpath do not interfere with compilation and execution in the others. Furthermore, the right to modify units is coordinated across subpaths so that only one copy of a unit can be edited at a time.

A subpath is created from an existing working view, such as Mail_Utilities.Rev1-_Working. By convention, the Environment constructs each subpath name from:

- The pathname prefix from the original working view (for example, "Rev1").
- The *subpath name extension* that you specify when you create the subpath. This extension typically is used to identify the developer to whom the subpath is assigned.
- A suffix such as "_Working". If releases are made from a subpath, then release numbers appear in place of the suffix.

For example, assume that two additional developers (Larry and Sue) are to help maintain the Mail_Utilities subsystem and that you are responsible for setting up subpaths for them. To create a subpath for Larry:

1. In the working view from which the subpath is to be created (Rev1_Working), make sure that the desired objects have been made controlled. (Objects that are controlled in Rev1_Working will be controlled automatically in the new subpath; see "Developing with Joined Objects," below.)
2. Display the subsystem and designate the working view from which the subpath is to be made (in this example, Rev1_Working).
3. Enter the Cmvc.Make_Subpath command, specifying the New_Subpath_Extension parameter. For example:

```
Cmvc.Make_Subpath (New_Subpath_Extension => "Larry");
```

As a result, a subpath view named Rev1_Larry_Working is created in Mail_Utilities. At this point, you can create a subpath for Sue and you can either set up a subpath for yourself or continue working in the main path, Rev1_Working. (Typically, the integrator for the subsystem continues to use the working view from the main path.) Figure 4-1 shows the Mail_Utilities subsystem with subpaths for Sue and Larry:

```
!Programs.Mail.Mail_Utilities      Library (Subsystem);
  Configurations                   Library (Directory);
  Rev1_0_1                         Library (Load_View);
  Rev1_Larry_Working               Library (Load_View);
  Rev1_Sue_Working                 Library (Load_View);
  Rev1_Working                     Library (Load_View);
  State                            Library (Directory);
```

Figure 4-1. The Mail_Utilities Subsystem with Subpath Views

Developing with Joined Objects

Development is coordinated across subpaths because subpath views are automatically *joined* with each other through the working view from which they were created. When subpaths are joined, each controlled object in a given subpath is joined with the corresponding objects in the other subpaths. Corresponding objects in different subpaths originate as copies of the same object and have the same name from the view name down. For example, the views Rev1_Working, Rev1_Larry_Working, and Rev1_Sue_Working each contain an object called Units.Destinations'Spec, and these three instances of Units.Destinations'Spec are joined. A set of joined objects is called a *join set*, as shown in Figure 4-2.

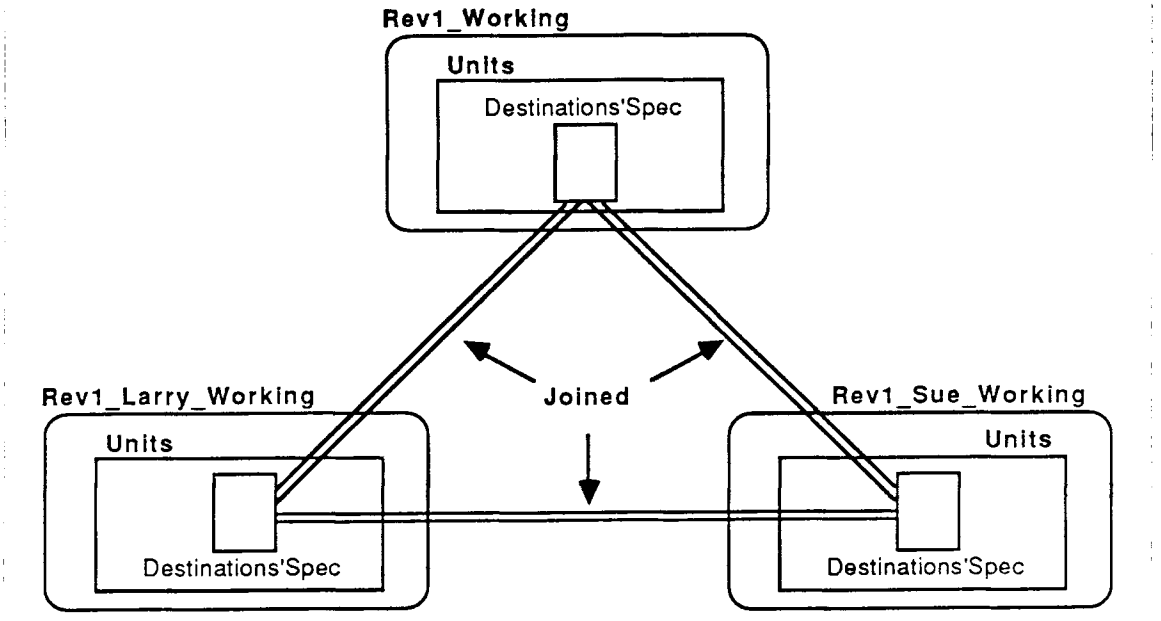


Figure 4-2. The Join Set for Destinations'Spec

Although the objects in a join set are separate library objects, they are treated as a single entity by the CMVC database:

- Objects in a given join set share a single reservation token. Consequently, a joined object can be checked out in only one view at a time.
- Objects in a given join set are represented as a single series of generations stored in the CMVC database. Consequently, changes made to one object in a join set can be propagated automatically to the other objects in the set.

Only controlled objects can be joined across views, so it is important to make the appropriate objects in a working view controlled before you create subpaths. Uncontrolled objects are copied into subpaths; however, because uncontrolled objects are not joined, they can be modified independently.

Checking Out a Joined Object

Because a joined object can be checked out in only one view at a time, the Cmvc.Check_Out command quits if you try to reserve an object that has been checked out in another view. For example, assume that Larry has checked out the unit Destinations'Body in the subpath Rev1_Larry_Working. If you now attempt to check out Destinations'Body in Rev1_Working, the Cmvc.Check_Out command quits with error messages informing you that the unit is already checked out.

To determine where a unit is checked out:

1. Designate the unit for which you want further information. In this example, you can select Destinations'Body in Rev1_Working.Units.
2. Enter the Cmvc.Show command using default parameter values.

As a result, a display as shown in Figure 4-3 appears in the output window.

```
88/02/23 14:38:27    [Cmvc.Show (Objects => "<CURSOR>")]
Views sharing tokens with !PROGRAMS.MAIL.MAIL_UTILITIES.REV1_WORKING.UNITS.DESTI
    !PROGRAMS.MAIL.MAIL_UTILITIES.REV1_SUE_WORKING
    !PROGRAMS.MAIL.MAIL_UTILITIES.REV1_LARRY_WORKING

      Object Name          Generation        Where          Chkd Out  By Whom
======================  ==========  ================  ========  ========
UNITS.DESTINATIONS'BODY     4 of 5    REV1_LARRY_WORKING     Yes      LARRY
88/02/23 14:38:32    [Show has finished]
```

Figure 4-3. Showing Where a Joined Unit Is Checked Out

The display in Figure 4-3 indicates, among other things, that Destinations'Body currently is checked out in the view Rev1_Larry_Working and that the user who checked it out was Larry. (Note that any user who has access to a view such as Rev1_Larry_Working can check out an object.) When Destinations'Body is checked in to Rev1_Larry_Working, you will be able to check out the corresponding unit in Rev1_Working. The Cmvc.Edit command also can be used to display checkout information; see the introduction to package Cmvc.

Keeping Joined Objects Updated

When a developer working in one subpath checks out a joined object, the other objects in the join set are rendered out of date by at least one generation. For example, the display in Figure 4-3 indicates that Destinations'Body in Rev1_Working is out of date, because it contains generation 4, whereas the latest generation for units in the join set is generation 5. (Generation 5 was created when Destinations'Body was checked out in Rev1_Larry_Working; this generation will be saved when Destinations'Body is checked in.)

The Environment provides several ways to propagate changes among joined objects to keep these objects up to date in each subpath:

- Changes are automatically propagated when objects are checked out. That is, the checkout operation ensures that you always have the latest generation for editing.
- Users can explicitly request that changes are propagated to objects without checking them out. This is called *accepting changes.*

Retrieving the Latest Generation at Checkout

The Cmvc.Check_Out command automatically retrieves the latest generation of an object from the CMVC database. For example, assume that Destinations'Body has been checked in, so that Rev1_Larry_Working has the latest generation (generation 5). If you now check out Destinations'Body in Rev1_Working, the checkout operation:

- Replaces the contents of Destinations'Body in Rev1_Working with the latest saved generation (generation 5)
- Creates the next generation (generation 6), which will store the changes you make while the unit is checked out

Accepting Changes

You can use the Cmvc.Accept_Changes command to update units in your subpath without having to check out those units. For example, assume that Rev1_Larry-_Working has created and saved generation 4 of package Lines'Body and that Sue wants to test the units in Rev1_Sue_Working against this generation. Sue does not want to check out Lines'Body because she does not want to edit it, nor does she want to create a new generation of it.

To update a unit to the latest generation without checking it out, she can:

1. Designate the unit to be updated. For example, she can select Lines'Body in Rev1_Sue_Working.
2. Enter the Cmvc.Accept_Changes command, using default values for all parameters.

As a result, Lines'Body in Rev1_Sue_Working is updated to the latest generation, which is generation 4.

When both a unit specification and body need to be updated, you must accept changes into the specification before accepting changes into the body.

You can use the Cmvc.Accept_Changes command to update multiple objects by specifying a naming string for the Destination parameter. If you specify a view name for the Destination parameter, all controlled objects in the view will be updated.

Permitting Demotion

By default, the Cmvc.Check_Out and Cmvc.Accept_Changes commands quit when they try to update the contents of a compiled unit, because such an update requires the demotion of that unit along with any units compiled against it.

To allow Cmvc.Check_Out and Cmvc.Accept_Changes to demote a compiled unit to update it:

1. Enter the desired command, specifying the value true for the Allow_Demotion parameter. For example:

```
Cmvc.Check_Out (Allow_Demotion => True);
```

As a result, one or more units are demoted to the source state, as needed.

Preventing Automatic Updating

Although the Environment provides for automatic propagation of changes, you may want to keep an older generation of a unit in your subpath—for example, for use when testing other units in the view.

If you want to keep an object at an older generation, you can avoid checking it out. A safeguard exists in the Cmvc.Check_Out command to permit an object to be checked out only if it is already at the latest generation. To use this safeguard:

1. Enter the Cmvc.Check_Out command, specifying the value false for the Allow-_Implicit_Accept_Changes parameter. For example:

```
Cmvc.Check_Out (Allow_Implicit_Accept_Changes => False);
```

As a result, the command can proceed only if the object does not need updating; otherwise, an error is reported and the command quits.

Creating New Joined Objects

When subpaths are created, the controlled objects within them are automatically joined. However, any objects that are created subsequently in one of several subpaths must be made controlled and propagated to the other subpaths explicitly. The Cmvc.Accept_Changes command can be used to copy new controlled objects from a source view into a destination view.

For example, assume that Sue has created a text file called To_Do in Rev1_Sue-_Working. To_Do contains a list of things to do on the project and is to be shared by all developers working in the Mail_Utilities subsystem. To propagate a new object across subpaths:

1. Make the object controlled where it was created (in this case, Rev1_Sue_Working).

2. Enter the Accept_Changes command to copy the object into one of the other subpaths. Specify the Source and Destination parameters with the object to be copied and the view into which it is to be copied. For example, from the context Rev1_Sue_Working.Units, you can enter:

```
Cmvc.Accept_Changes (Destination => "^^Rev1_Larry_Working",
                     Source => "To_Do",
```

3. Repeat step 2 for each other subpath (in this case, Rev1_Working).

Cmvc.Accept_Changes copies the controlled object into the Units directory of the destination view, makes the object controlled, and joins it to the source object. Thus, Cmvc.Accept_Changes automatically performs operations that can be performed individually using the Library.Copy, Cmvc.Make_Controlled, and Cmvc.Join commands.

If you use Library.Copy to propagate a controlled object that you intend to join, you must copy the object so that it has the same name within both views. In particular, if the two views have been restructured to contain further subdirectories, make sure that you copy the object into the corresponding subdirectory of the second view. Objects in different subdirectories cannot be joined, even if the objects themselves have the same simple name.

Accessing Controlled Objects Concurrently

If you and another user need concurrent access to a controlled object, you can *sever* the object in your view. When an object in a view is severed from its join set, that object acquires its own reservation token. Accordingly, the severed object can be checked out independent of the other objects in the join set, and a separate series of generations is stored in the CMVC database for that object. In effect, the severed object becomes a second join set, one that contains a single object.

For example, assume that you and Larry want to check out and edit package Messages'Body concurrently, presumably to modify different parts of the package. You decide to sever the object in your view (Rev1_Working), leaving the object in Rev1_Larry_Working in the original join set. To do this:

1. Select the object to be severed. Make sure that the object is checked in from Rev1_Working.
2. Enter the Cmvc.Sever command, using default parameters.

As indicated in Figure 4-4, a second join set for Messages'Body is created in the CMVC database. The current contents of the unit in Rev1_Working becomes generation 1 for the new join set.

Note that if an object is not meant to be shared, it can be severed in every subpath. Severed objects can be modified independently, while remaining controlled and tracked by the CMVC database.

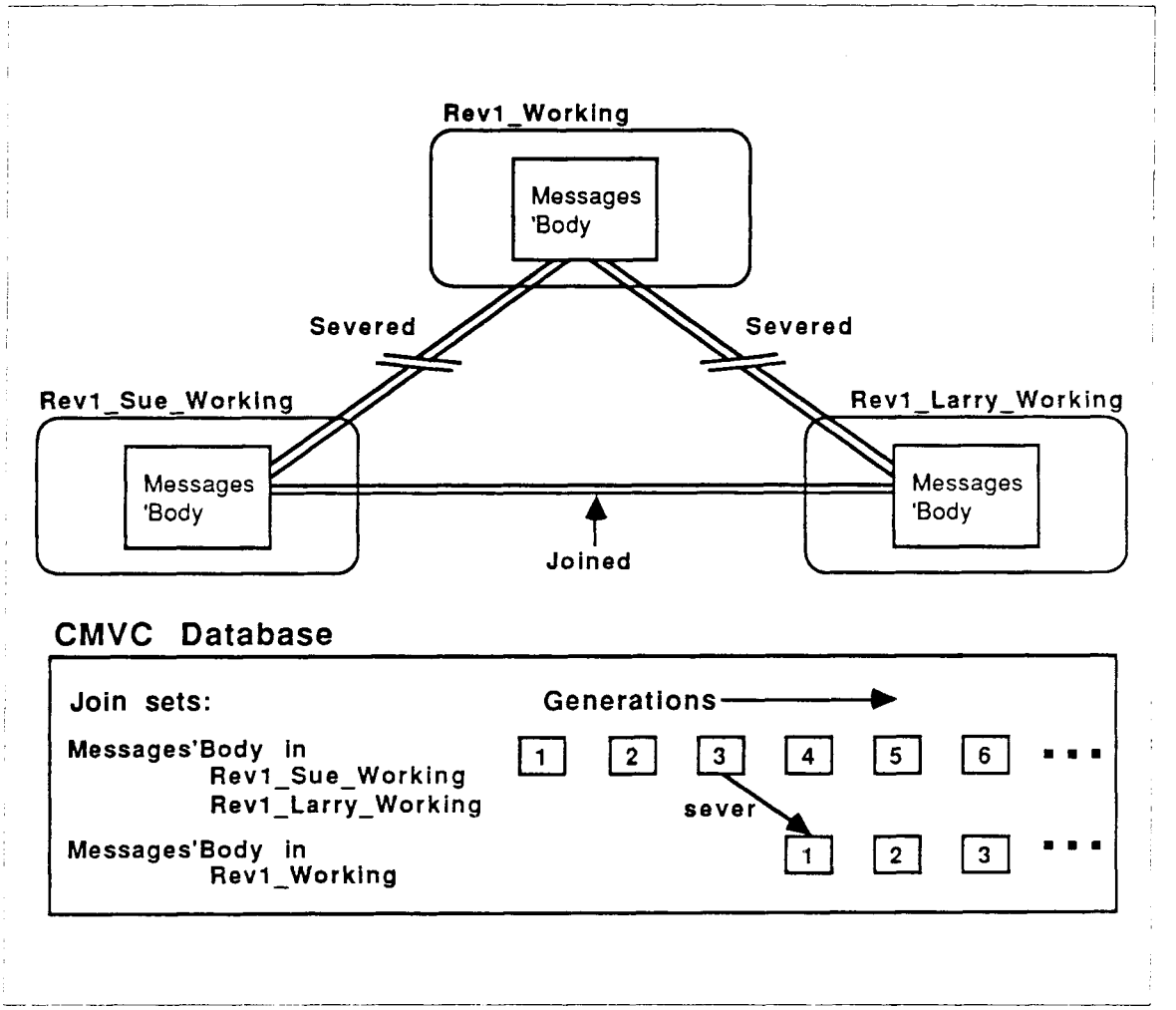


Figure 4-4. After Messages'Body Has Been Severed in Rev1_Working

Elements, Join Sets, and Reservation Tokens

In Figure 4-4, the three objects called Messages'Body are collectively represented as a single *element* in the CMVC database. This element is partitioned into two join sets, each having its own reservation token that connects corresponding objects from specific views.

By default, reservation tokens are named automatically by the Environment. However, through parameters to the Cmvc.Make_Controlled and Cmvc.Sever commands, you can define your own mnemonic reservation tokens to convey more information about the join sets into which an element has been partitioned. You can add a controlled object to a particular join set by specifying the appropriate reservation token as a parameter value of the Cmvc.Join command.

Merging Changes

Changes can be propagated between two severed objects with the Cmvc.Merge_Changes command. More specifically, the Cmvc.Merge_Changes command updates one of the two objects (the destination object) to include any changes that were made to the other (the source object). Changes that were previously made to the destination object are preserved, so this operation does not necessarily result in identical objects (that is, the source object is not updated with the changes that were made to the destination object). For example, assume that you want to merge changes from the unit Messages'Body in Rev1_Larry_Working into the corresponding severed unit in Rev1_Working. To merge changes:

1. Make sure both severed objects are checked in.
2. Select the destination object to be updated, Messages'Body in Rev1_Working.Units.
3. From the context Rev1_Working.Units, enter the Cmvc.Merge_Changes command, specifying the Source_View parameter with the name of the view containing the object to be merged:

```
Cmvc.Merge_Changes (Source_View => "^^Rev1_Larry_Working");
```

The Cmvc.Merge_Changes command finds the common ancestor of the two severed objects (the last generation they had in common before severing) and compares the source and destination objects to the common ancestor to determine which lines need to be merged. The destination object is updated accordingly. When conflicts exist, changes from both the source and destination objects are marked with the string *; (an asterisk followed by a semicolon). You must edit the destination object to resolve conflicts and remove the *; marks.

Note that the Cmvc.Merge_Changes command applies only to objects that belonged to the same join set at some time. You must use operations from package File_Utilities to synchronize objects that are not controlled.

Rejoining Severed Objects

The Cmvc.Merge_Changes command can be used to prepare two severed objects that you want to rejoin, since objects must be textually identical before they can be joined. To prepare two objects for joining:

1. Use Cmvc.Merge_Changes to merge the source object into the destination object.
2. Check out and edit the destination object to resolve any conflicts.
3. Check out the source object and copy the contents of the edited destination object into it.
4. When the two objects are textually identical, use the Cmvc.Join command to join them.

Integrating Subpaths into a Single Release

When a new release must be made, the person in charge of integration within the subsystem can consolidate the changes made in each subpath into a single working view. The next release can then be made from that single view.

For example, assume that you are the subsystem-level integrator and that you want to make a release from Rev1_Working that includes work done by Sue and Larry in their subpaths. To do this, you can:

1. Gather the changes from each subpath into Rev1_Working, using Cmvc.Accept_Changes to update joined units and Cmvc.Merge_Changes to update severed units.
2. Compile and test the updated units in Rev1_Working.
3. When appropriate, make a release from Rev1_Working using the Cmvc.Release command.

After the release, you can update the subpaths Rev1_Sue_Working and Rev1_Larry_Working so that they match the integration view Rev1_Working. For example, to update Rev1_Sue_Working from Rev1_Working:

1. Make sure all objects in both views are checked in.
2. Designate the destination view (the view to be updated). For example, if the Mail_Utilities subsystem is displayed, you can put the cursor on the entry for Rev1_Sue_Working.
3. Enter the Cmvc.Accept_Changes command, specifying the Source parameter with the name of the view to be matched. For example:

```
Cmvc.Accept_Changes (Source => "Rev1_Working");
```

As a result, Rev1_Sue_Working is made to look like Rev1_Working:

- Every controlled object in the source view updates the corresponding object in the destination view.
- New controlled objects in the source view are copied into the destination view, made controlled, and joined.

Note that uncontrolled objects in Rev1_Working are not copied into Rev1_Sue_Working and no objects are deleted when they exist in Rev1_Sue_Working but not in Rev1_Working.

Setting Up Multiple Development Paths

Development within a single subsystem can involve multiple major development efforts, each resulting in a variant implementation of the subsystem. For example:

- When an application is intended for multiple targets and the units need to have variants that contain target-specific code
- When a new major release of an application is developed while the existing release is maintained
- When a separate quality-assurance group conducts activities to turn a development release into a production release

To accommodate these variant implementations, you can set up multiple development paths. Multiple paths provide each development effort with its own working view from which releases can be made. Furthermore, because objects can be joined or severed across paths, paths enable some objects to be shared and others to be developed independently. Finally, multiple subpaths can be created from each path to provide a separate workspace for each developer involved in the same development effort.

Creating a Path

A path is created as a full copy of an existing working view and, as such, serves as a working view from which a series of releases can be made. Within the working view of a path, units can be edited, compiled, and tested, just as in subpaths. In fact, as copies of working views, paths and subpaths are fundamentally the same, although they are intended to support parallel development at different levels. Paths and subpaths differ with respect to what you can specify about them at creation:

- Paths and subpaths have different naming conventions. Therefore, when creating a path, you must specify a pathname prefix; creating a subpath involves specifying a subpath name extension.
- Paths can be created with their own library characteristics. Therefore, when creating a path, you can specify a different model world than the one used by other views in the subsystem. In contrast, subpaths use the same model as the path from which they were created.
- Paths can be joined with other views or not. That is, when creating a path, you must specify whether the controlled objects are to be joined to their counterparts in the source working view. In contrast, subpaths are always created with all controlled objects joined.

Assume that you want to set up a path in Mail_Utilities for quality-assurance activity. To do this:

1. Designate the source view to be copied as the beginning of the new path. For example, select the entry for Rev1_Working.
2. Enter the Cmvc.Make_Path command, specifying values for the following parameters, as needed:

   New_Path_Name — Specifies the pathname prefix for the new path. This prefix will be shared by all subpaths and releases made from the new path.

   Model — Specifies the name of a model to be used by views in the new path. You should specify a new model if the new path needs different links, compilation switches, and the like. By default, this parameter specifies the model world inherited from the source view.

   Join_Paths — Specifies whether to join the new path to the source view. You should specify true if you anticipate that most or all of the objects will be shared between the two views. You should specify false if you anticipate that most or all of the objects will need to be modified independently. By default, this parameter specifies that the views are joined.

   For example, to create a path called Qa_Working in which all objects are joined and which uses the inherited model, you can enter:

   ```
   Cmvc.Make_Path (New_Path_Name => "QA");
   ```

3. Use the Cmvc.Join or Cmvc.Sever commands to further fine-tune the objects that are shared between the new path and the source view.

Managing Views

Operations for managing views require the use of commands from package Cmvc. Such operations include deleting, rebuilding, copying, and renaming views.

Deleting Views

As you accumulate multiple views in a subsystem, you may want to delete some of them to save space. To do so, you must use the Cmvc.Destroy_View command; do not use other Environment deletion commands, such as Library.Delete, Library.Destroy, Compilation.Delete, or Compilation.Destroy. Deletion commands from packages Library or Compilation can delete only part of a view, leaving it in a damaged state. (See "Repairing Damaged Views," below.)

The Cmvc.Destroy_View command allows you to delete views permanently or to delete views so that they can be reconstructed again.

Deleting a View and Allowing Reconstruction

To delete a view and allow future reconstruction:

1. Select the view you want to delete.
2. Enter the Cmvc.Destroy_View command, using all default parameters (by default, the Destroy_Configuration_Also parameter is false). If the view contains compiled units, you will need to specify the nondefault value for the Demote_Clients parameter, as follows:

```
Cmvc.Destroy_View (Demote_Clients => True);
```

As a result, the view is deleted, although its configuration object and state description directory are preserved (see "Representation of Releases" in the chapter entitled "Getting Started"). If you delete a working view, a state description directory is created for it before the view is deleted.

Deleting a View Permanently

To delete a view permanently:

1. Select the view you want to delete.
2. Enter the Cmvc.Destroy_View command, specifying the nondefault value for the Destroy_Configuration_Also parameter, as follows:

```
Cmvc.Destroy_View (Destroy_Configuration_Also => True);
```

As a result, the view is deleted, along with its configuration object and state description directory (see "Representation of Releases" in the chapter entitled "Getting Started"). The deleted view cannot be reconstructed using the Cmvc.Build command.

Deleting a Configuration Object

Assume that you deleted a view to allow future reconstruction and then decided that the view should be deleted permanently. You can do this by deleting the configuration object that was left after you deleted the view.

To delete a configuration object:

1. Within the *subsystem*.Configurations directory, locate the appropriate configuration object and state description directory. These objects are named after the view you deleted.
2. Use Library.Delete to delete these objects.

Note that the CMVC database still contains information about the deleted view, even though the view's configuration object no longer exists. Because of this, you cannot create a new view with the same name as the deleted view without taking an extra step:

1. Enter the Cmvc_Maintenance.Expunge_Database command to remove all information about the deleted view from the CMVC database. You can now create the new view with the same name.

Building a View from a Configuration Object

After making a configuration release, or after deleting a view allowing future reconstruction, you can use the configuration object to rebuild a corresponding view. Configuration objects are located in the Configurations directory of the subsystem. To build the view corresponding to a configuration object named Configurations.Rev1_0_2:

1. From the context of the Configurations directory, enter the Cmvc.Build command, specifying the desired configuration object, as shown:

```
Cmvc.Build (Configuration => "Rev1_0_2");
```

Repairing Damaged Views

Attempting to delete a view using any command other than Cmvc.Destroy_View only partially destroys the view. Such damaged views cannot be completely deleted, even using Cmvc.Destroy_View. To repair a damaged view so that it can be destroyed completely:

1. Designate the view to be repaired. For example, select its entry in the subsystem.
2. Enter the Cmvc_Maintenance.Check_Consistency command with default parameters.

Renaming Views

To change a view's name:

1. Use the Cmvc.Make_Path or Cmvc.Make_Subpath command to copy the view, supplying the new name prefix or extension as appropriate.
2. Delete the original view using either of the methods listed above.

If you want to destroy and recreate the view again with the same name, you must destroy the view permanently and then expunge the database.

Developing Applications Using Multiple Subsystems

The concepts and operations presented so far (such as release, check in and check out, subpaths, paths) are relevant for the development of individual subsystems. This chapter covers what you need to know to compile and execute applications that consist of multiple subsystems. When dependencies exist across subsystems, you need to know how to:

- Define interfaces between subsystems to support compilation
- Specify combinations of views, one from each subsystem, for use during execution

The first section in this chapter covers the basic setup required for compiling and executing multiple subsystems. Subsequent sections expand on various aspects of this basic setup, describing ways of more precisely controlling interface definition and system execution.

This chapter concludes by taking a second look at setting up the subsystems in an application, incorporating the concepts covered in this and the preceding chapters. This second look provides a more sophisticated checklist of issues to be decided during subsystem creation.

Basic Compilation and Execution Setup

Before a multisubsystem application can be compiled, interfaces must be set up to support compilation dependencies among units in different subsystems. Subsystem-level interfaces are expressed as *exports*, which make a specific set of implemented units available for views in other subsystems to *import*. (The importing views are called *clients* of the exporting subsystem.)

Importing enables units in a client view to reference exported units in *with* clauses. In fact, compilation dependencies can hold between units of different subsystems only if:

- The referenced unit is among the exports of one subsystem
- Those exports are imported by the other subsystem (more specifically, by the view that contains the dependent unit)

Any other dependencies between subsystems are reported as errors. Exports and imports express and enforce design decisions by permitting or restricting visibility between subsystems.

Exports and imports must be set up to enable compilation, but an additional step is required before the application can be executed. Typically, each subsystem contains multiple releases of its implementations. (Furthermore, subsystems with multiple paths contain multiple variant implementations, each with multiple releases.) Therefore, an execution table called an *activity* must be set up to specify which implementation from each subsystem should be used for execution. Note that different precompiled implementations can be used during execution simply by changing entries in an activity.

Kinds of Views

Setting up exports, imports, and activities involves the creation and use of another kind of view along with the kind used for developing implementations:

- Subsystem implementations are developed in *load views*. Load views are program libraries that contain the specifications and bodies for all units in a given application component. The working views and released views described in previous chapters have all been load views.
- Subsystem exports are expressed in separate views called *spec views*. Spec views are special-purpose program libraries that contain a copy of the specification of each exported unit.

Figure 5-1 illustrates a load view from which one package is exported. Accordingly, the spec view contains a copy of the exported unit's specification:

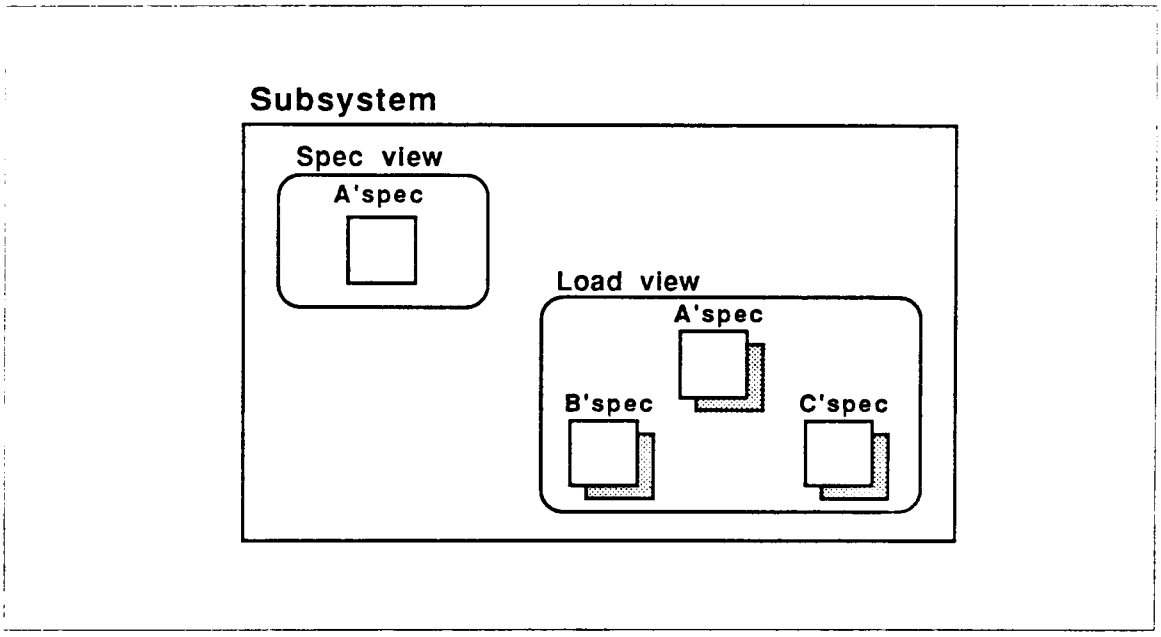


Figure 5-1. A Spec View That Expresses the Exports from a Load View

Together, the two kinds of views function similarly to Ada package bodies and specifications, in that spec views, like package specifications, make resources available to clients, whereas load views, like package bodies, implement those resources.

A spec view typically is created from, and therefore expresses the exports of, a particular working load view. As subsequent releases are made from that working view, the same spec view also can serve to express exports for each release that is *compatible* for use with that spec view. A load view is compatible if it implements all the resources promised by the spec view. (If implementation changes result in releases that are incompatible with an existing spec view, either the existing spec view must be changed or a new spec view must be made.) Thus, a single spec view typically is used to express the exports of a family of released load views.

Spec views are what views in other subsystems import to enable compilation. However, the units in spec views are not actually executed. Instead, the units in load views are used for execution, one load view for each imported spec view. An activity specifies which of the compatible load views are to be used for a given execution. One of the advantages of having separate spec views is that client views can remain compiled against spec views while changes are made and tested in load views.

A subsystem can contain multiple spec views. Multiple spec views result when new spec views are created to accommodate changed implementation. Furthermore, when a subsystem contains multiple development paths for a variant implementation, each development path typically requires its own exports and consequently has one or more spec views associated with it. A given spec view thus expresses the exports of a family of load views within a particular development path.

Note that a third kind of view, called *combined views*, combines exports and implementation. However, combined views are required only to accommodate:

- Designs that require *circular importing*
- Specific Ada structures in cross-development (that is, the development of programs that are to be executed on target processors other than the R1000)

In either case, combined views are used in place of spec and load views. Combined views are not recommended for more general use because they do not provide the advantages gained by expressing exports and implementation in separate views. The remainder of this chapter describes spec and load views; see the chapter entitled "Using CDFs with Subsystems" and the introduction to package Cmvc for information about combined views.

Defining Exports

To illustrate how to define exports, this and the following sections will use the sample mail program presented in previous chapters. Recall from Figure 3-2 in "Getting Started" that units in both the Command_Interpreter and the Mailboxes subsystems need to *with* various units implemented in the Mail_Utilities subsystem. These dependencies are indicated by heavy arrows in Figure 5-2.

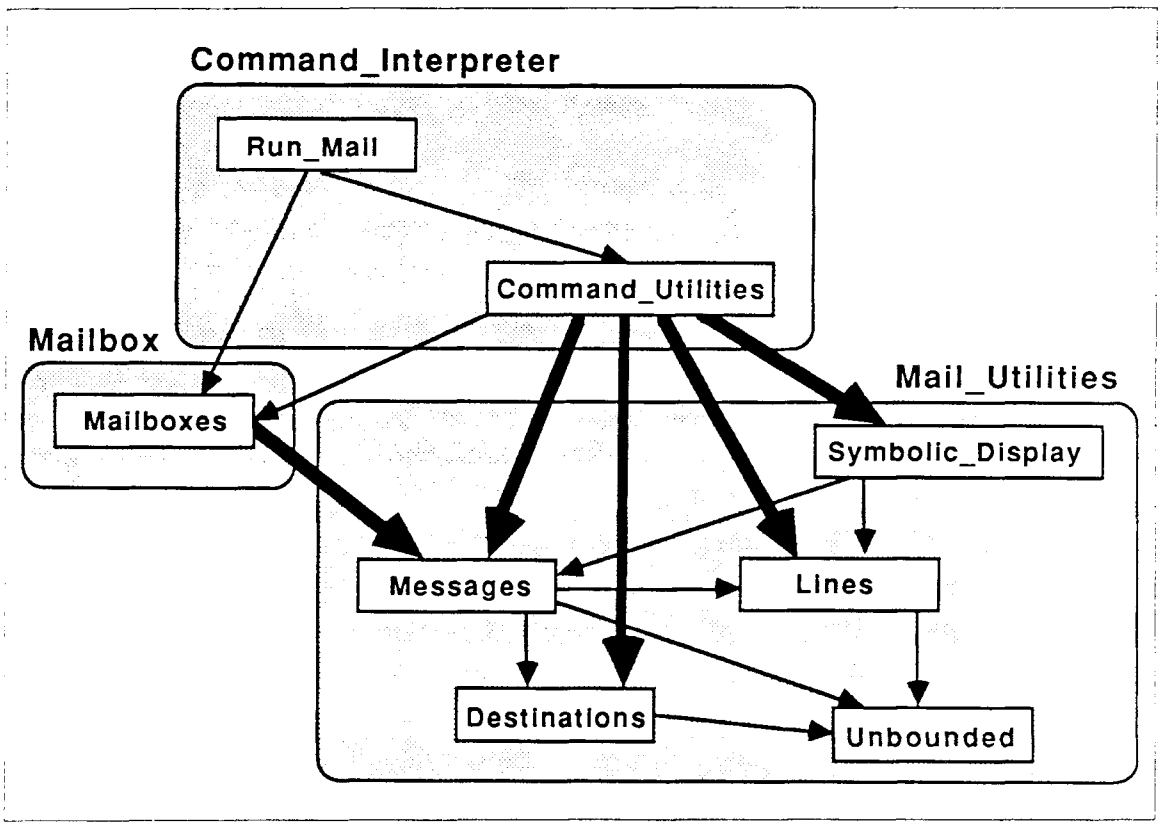


Figure 5-2. Dependencies between Units in Different Subsystems

From Figure 5-2, it is clear that the Mail_Utilities subsystem must export a total of four units to support the required compilation dependencies: Messages, Destinations, Lines, and Symbolic_Display.

Overview of Steps

To export units from a given load view, such as Rev1_Working, you must follow these general steps, which are clarified in subsequent sections:

1. Specify the desired exports as part of the state of the load view that implements them—in this case, Rev1_Working. This means editing the Exports file in the State directory of the load view to specify the units to be copied into the spec view.
2. Enter the Cmvc.Make_Spec_View command to create a spec view from the appropriate load view. The new spec view is created containing a copy of each unit listed in the load view's State.Exports file.
3. Make sure that the units in the newly created spec view are promoted to the coded state so that clients will be able to compile against them.

Locating the State.Exports File

Every view is created with a file called Exports in its State subdirectory. Do not confuse this file with the Exports directory in the view, which is covered in "Creating Export Restriction Files," below. The relevant portion of the directory structure within Rev1_Working is shown in Figure 5-3.

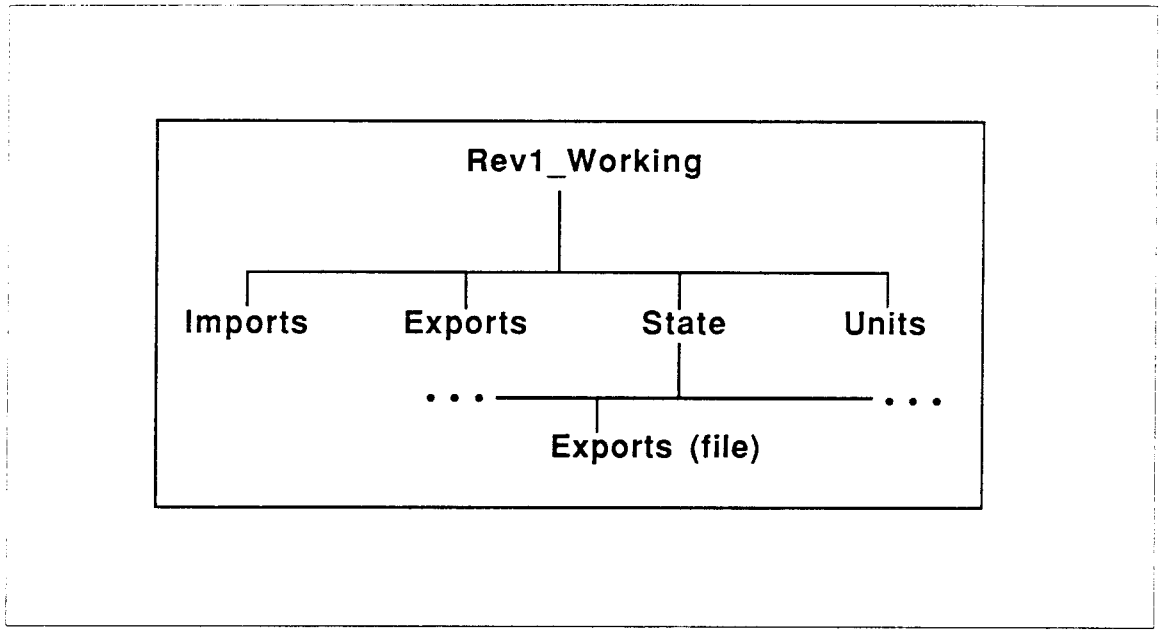


Figure 5-3. The Exports File and the Exports Directory

What to Put in the State.Exports File

The State.Exports file serves as an indirect file for the Cmvc.Make_Spec_View command by providing a list of the unit specifications that the command will automatically copy from the load view into the new spec view. Note that these unit specifications must exist in the load view, although the unit bodies need not. That is, exports can be created before the implementation is complete.

The first time you display the State.Exports file, it contains the naming string `?'spec`, as shown in Figure 5-4. If you leave the file as is, the new spec view will contain a copy of all unit specifications from the load view. That is, by default, all units are exported from a given view. However, if you determine that fewer units need to be put in the spec view, you can replace the default naming string with a list of names.

```
?'spec
```

```
=----REV1_WORKING STATE EXPORTS V1.2 (text)----
```

Figure 5-4. The Default State.Exports File

When determining the list of units to specify in the State.Exports file, bear in mind that you eventually will have to compile the units in the spec view. Therefore, the list in the State.Exports file must include the names of the following:

- The specifications of the units that other subsystems require for compilation
- Any additional unit specifications that are required to compile the spec view

For example, as indicated by Figure 5-2, the State.Exports file for Rev1_Working must in fact name all of the unit specifications in that view:

- The units Messages'Spec, Destinations'Spec, Lines'Spec, and Symbolic_Display-'Spec are included because these units are required by other subsystems.
- The unit Unbounded'Spec is included because three of the four unit specifications listed above depend on Unbounded'Spec.

Thus, for compilation purposes, a spec view may have to contain more units than you originally expected to export. In some cases, however, "extra" units can be omitted from the spec view—specifically, if the units that depend on them reference them only in private parts (see "Using Pragma Private_Eyes_Only," below). Furthermore, *export restrictions* can be used to prevent particular units in a spec view from being referenced in client views (see "Imposing Further Import and Export Controls," below).

Using Pragma Private_Eyes_Only

As shown above, the State.Exports file may need to include certain unit specifications solely to enable compilation within the spec view and not to support dependencies from other subsystems. When this is the case, you can check to see whether any of these "extra" unit specifications are required only within the private part of dependent units. If so, you can omit these "extra" unit specifications from the State.Exports file. You can omit them because, by default, private parts are closed, which means they are ignored when a spec view is compiled; the load view supplies the private parts at execution time. (See "More on Closed Private Parts," later in this chapter.)

For example, recall that Unbounded'Spec is required for compiling three of the unit specifications to be exported—namely, Messages'Spec, Destinations'Spec, and Lines'Spec. However, Unbounded'Spec is not referenced by units in other subsystems; therefore, you can:

1. Check each of the three dependent units to see where they use Unbounded'Spec. In this example, resources from Unbounded'Spec are used only in the private parts of these three units, such as that shown in the abbreviated representation of Destinations'Spec in Figure 5-5.

```
with Unbounded;

package Destinations is

    type User is private;

...

private
    type User is new Unbounded.Variable_String;
...

end Destinations;
-- DEV$:[WORKING.UNITS.DESTINATIONS]V[-1.-1](ada)    installed
```

Figure 5-5. Destinations'Spec with Reference to Unbounded in the Private Part

2. Edit each of the unit specifications that reference Unbounded'Spec and insert the following pragma before the relevant *with* clause:

   ```
   pragma Private_Eyes_Only;
   with Unbounded;
   ```

   Pragma Private_Eyes_Only applies to any *with* clause that follows it, so you may need to insert the pragma between *with* clauses.

3. Leave Unbounded'Spec out of the State.Exports file.

Editing the State.Exports File

When you edit the State.Exports file:

1. Enter either a list of unit specification names or one or more naming expressions that matches such a list. (The contents of this file have the same syntax as the contents of indirect files; see the "Naming" chapter.)
   - If you want the spec view to contain all unit specifications in the load view, use the naming string `?'spec`.
   - If you want to specify a list of names, place the names on consecutive lines with no delimiters.
2. Be sure to commit the file.

Figure 5-6 shows the State.Exports file for Rev1_Working.

```
destinations'spec
lines'spec
messages'spec
symbolic_display'spec

--- REV1_WORKING_STATE_EXPORTS;V(3) (text) ---
```

Figure 5-6. The State.Exports File for Rev1_Working

Creating the Spec View

After you have edited the State.Exports file of the exporting load view, you are ready to make a spec view from that view. To do this:

1. Designate the exporting load view. For example, if the Mail_Utilities subsystem is displayed, you can select the entry for Rev1_Working.
2. Enter the Cmvc.Make_Spec_View command, specifying the Spec_View_Prefix parameter and using default values for the other parameters.

   The Environment uses the value of the Spec_View_Prefix parameter when constructing the name for the new spec view. In this example, the specified value is the pathname prefix from the name of the exporting load view (for example, "Rev1" from "Rev1_Working"):

   ```
   Cmvc.Make_Spec_View (Spec_View_Prefix => "Rev1");
   ```

As shown in Figure 5-7, a spec view named Rev1_0_Spec is created within the Mail_Utilities subsystem. Spec views have the same internal directory structure as load views; as shown, the Units directory in Rev1_0_Spec contains a copy of each unit specification listed in the Rev1_Working.State.Exports file.

```
!Programs.Mail.Mail_Utilities :   Library (Subsystem);
  Configurations          :   Library (Directory);
  Rev1_0_1                :   Library (Load_View);
  Rev1_0_Spec             :   Library (Spec_View);
  Rev1_Larry_Working      :   Library (Load_View);
  Rev1_Sue_Working        :   Library (Load_View);
  Rev1_Working            :   Library (Load_View);
  State                   :   Library (Directory);
```

```
=-!PROGRAMS.MAIL_MAIL_UTILITIES (library)-*-World-
```

```
!Programs.Mail.Mail_Utilities.Rev1_0_Spec.Units :   Library (Directory);
  Destinations       :  C Ada (Pack_Spec);
  Lines              :  C Ada (Pack_Spec);
  Messages           :  C Ada (Pack_Spec);
  Symbolic_Display   :  C Ada (Gen_Proc);
```

```
=-MAIL_UTILITIES.REV1_0_SPEC.UNITS (library)-Directory-
```

Figure 5-7. The Spec View Rev1_0_Spec

Note that Rev1_0_Spec has no underlying connection to the load view from which it was created. Each view is a separate library structure; modifying or compiling units in Rev1_Working leaves the units in Rev1_0_Spec unaffected, and conversely.

Spec-View Names and Level Numbers

By default, spec-view names such as Rev1_0_Spec are constructed automatically from:

- The spec-view prefix specified by the Spec_View_Prefix parameter of the Cmvc.Make_Spec_View command (for example, "Rev1")
- One or more level numbers (for example, "_0")
- The suffix "_Spec"

The spec-view prefix can be any string, although a typical convention is to use the pathname prefix from the name of the exporting load view. By this convention, the association between a spec view and a particular development path is reflected in the view names. This convention is especially useful when a subsystem contains multiple paths.

The level numbers in spec-view names are related to the level numbers in release names. Recall that, by default, release names have two release level numbers, which you can use to define a series of major (level 1) and minor (level 0) releases.

A spec view normally expresses the exports for a series of minor releases. Therefore, in spec-view names, the rightmost (level 0) number is replaced by the "Spec" prefix, so that numbering within a spec-view name starts with the level 1 number. (If only two release levels are maintained, then spec-view names contain only the level 1 number.) Level numbers are shown in Figure 5-8.

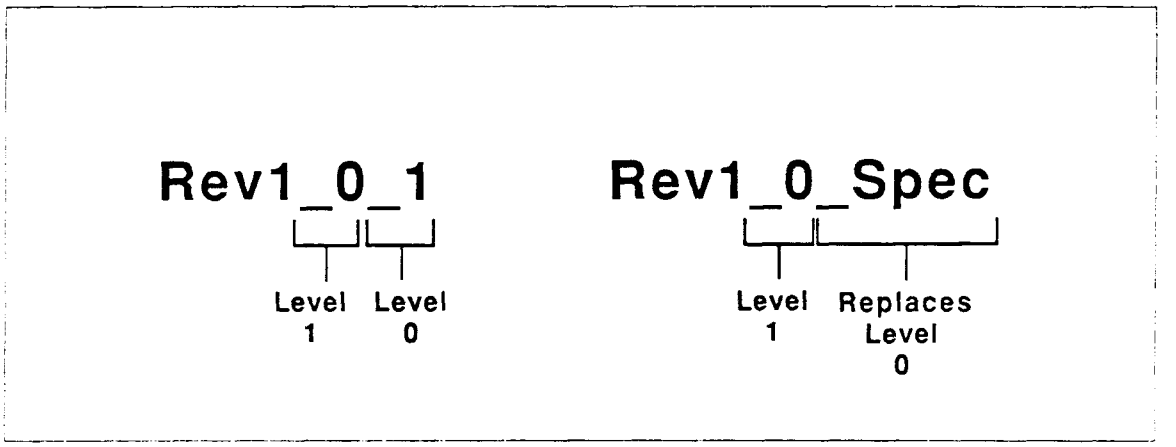


Figure 5-8. Structure of Spec-View and Release Names

By default, the Cmvc.Release command automatically increments the level 0 number in release names, creating a series like Rev1_0_1 and Rev1_0_2. Also by default, the Cmvc.Make_Spec_View command constructs spec-view names using the level 1 (and higher) numbers from the name of the last release. Thus, in the above example, the level 1 number in the name Rev1_0_Spec is _0 because that is the level 1 number in the name of the configuration release Rev1_0_2. In this way, level numbers can be used to correlate a spec view with a particular family of minor releases in a development path. In this case, Rev1_0_Spec correlates with Rev1_0_1 and Rev1_0_2.

Both the Cmvc.Release and the Cmvc.Make_Spec_View commands have a Level parameter that you can use to specify a level number other than 0 for incrementing. If you increment a nondefault level number when making a release, subsequent spec-view names will by default be constructed with that number; for example, when Rev1_1_1 is released (with an incremented level 1 number), then the next spec view will by default have the name Rev1_1_Spec. Conversely, if you increment a level number when making a spec view, subsequent release names will by default be constructed with the incremented number.

You can take advantage of the automatic level numbering to keep track of which spec and load views are compatible. Major releases (with incremented level numbers) should be made whenever the implementation has changed enough to require a new spec view to express its exports. (Note that level numbers are automatically coordinated only between releases and spec views that are made from the same working view. See also "Coordinating Level Numbers in Spec and Released View Names.")

As a final note, you can suppress level numbers in a spec-view name so that the spec-view prefix is followed directly by _Spec. To do this, specify the value Natural'Last for the Level parameter in the Cmvc.Make_Spec_View command.

Controlled Units within Spec Views

When a spec view is made, the units in it are deliberately left uncontrolled. By default, the Environment considers the working view as the place for ongoing changes; therefore the working view is where change history should be tracked.

Units in spec views can be changed, so some installations may choose to make units in spec views controlled in order to track change history. However, these units should never be joined to their counterparts in the working load view. As shown in later sections, clients ultimately will compile against units in spec views. Propagation of changes from load-view units to spec-view units will demote not only the spec-view units but also all clients compiled against them.

Compilation within Spec Views

By default, the Cmvc.Make_Spec_View command copies units into the spec view and then promotes them to the coded state. The promotion of the copied units is controlled by the Remake_Demoted_Units and Goal parameters of the Cmvc.Make_Spec_View command.

After they are copied, units in spec views can be demoted or promoted independently from their counterparts in working load views. However, units in a spec view must be in the installed or coded state for clients to compile against them.

Defining Imports

After exports have been defined in a subsystem, they are available for views in other subsystems to import. For example, assume that development of the Command_Interpreter subsystem is in progress and that a prototype of package Command_Utilities needs to be compiled and tested. At this point, the prototype of Command_Utilities depends only on units from the Mail_Utilities subsystem. (When complete, Command_Utilities also will depend on a unit implemented in the Mailbox subsystem; however, this dependency will be ignored for the present.)

To enable the prototype of Command_Utilities to compile, the working view containing Command_Utilities must import the spec view from Mail_Utilities. More specifically, Command_Interpreter.Rev1_Working must import (and become a client of) Mail_Utilities.Rev1_0_Spec, as shown in Figure 5-9.

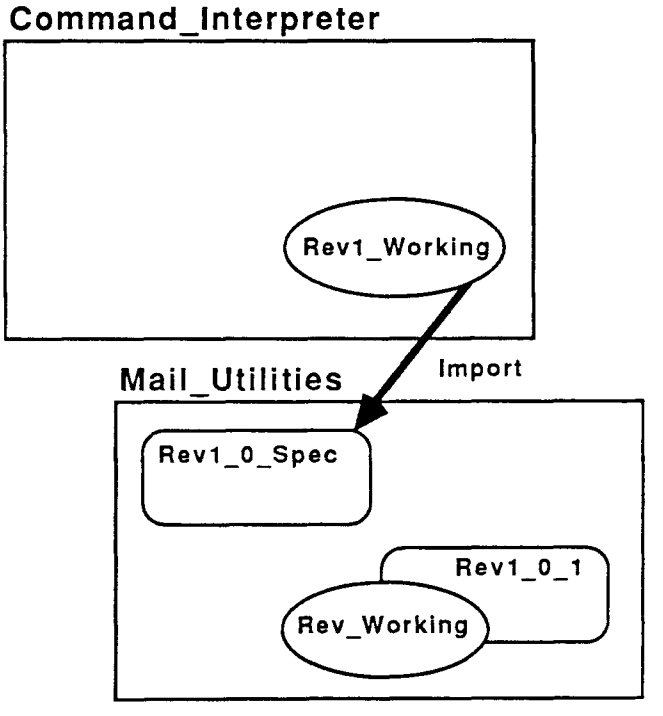


Figure 5-9. Importing Mail_Utilities.Rev1_0_Spec

Steps for Defining Imports

To import a spec view from one subsystem into a client view in another subsystem:

1. Display both subsystems (for example, Command_Interpreter and Mail_Utilities).
2. In the exporting subsystem, select the desired spec view (in this example, Mail_Utilities.Rev1_0_Spec).
3. Move the cursor to the importing subsystem and put the cursor on the desired client view (in this example, Command_Interpreter.Rev1_Working).
4. Open a Command window and enter the Cmvc.Import command, using all default parameter values.
5. In the client view, you can now compile units, such as the prototype package Command_Utilities.

Note that you can specify a naming expression for the View_To_Import parameter in order to import spec views from multiple subsystems.

Only spec views from other subsystems can be imported. Accordingly, a view's *imports* refers to a list of spec views. However, either spec views or load views can have imports, because either type of view may need to compile against units from other subsystems. A spec view's clients (also called *referencers*) are the spec and load views that import it.

By default, a view's imports are inherited by the releases, paths, and subpaths that are made from it. For example, when a release is made from Command_Interpreter.Rev1_Working, that release will automatically import Mail_Utilities.Rev1_0_Spec. Note that the Cmvc.Import command can be used to change the imports in the working view of any path or subpath; however, because releases are frozen, their imports cannot be changed.

Displaying a View's Imports

You can verify a view's imports by requesting an information display as follows:

1. Designate the view (for example, Command_Interpreter.Rev1_Working) whose imports are to be displayed.
2. Enter the Cmvc.Information command.

As a result, a list of imports is displayed along with other information about the view, as shown in Figure 5-10.

```
Information for view !PROGRAMS.MAIL.COMMAND_INTERPRETER.REV1_WORKING
  Model                : R1000
  Frozen               : FALSE
  View Kind            : LOAD
  Creation             : March 30, 1988 at 6:03:28 PM
  Imports              :
    !PROGRAMS.MAIL.MAIL_UTILITIES.REV1_0_SPEC
  Referencers          :
  Unit Summary         :
    coded = 0, installed = 0, source = 4, archived = 0, stubs = 0
88/03/31 12:26:04 ... [Information has finished]
--- _INTERPRETER & CMVC_INFORMATION (Text) ---
```

Figure 5-10. Information about Command_Interpreter.Rev1_Working

Note that you can also use this command to display a spec view's referencers.

Imports and Links

Imports enable compilation because they create and manage links. When a spec view is imported, links to its units are automatically created in the client view, thus enabling units in the client view to compile.

Within subsystems, you should manage links only through imports. This is because imports alone can manage links across releases, paths, and subpaths. You should never directly add or remove an individual link from a view using commands from package Links. Such changes do not get passed on to releases, paths, or subpaths.

The links for a given view should contain only:

- Links resulting from imports; these are links to units in other subsystems
- Links provided by the model world; these are links to units elsewhere in the Environment

Removing Imports

If you imported the wrong view or you want to make a design change, you can remove imports with the Cmvc.Remove_Import command. Removing an import automatically removes all links to the units in the imported spec view. An import cannot be removed if there are units compiled against any of the links it created. That is, if units are compiled against even one link in an import, none of the links from that import are removed. (Thus, removing an import removes either all or none of the relevant links.) The import can be removed if units in the client view are demoted to source.

Note that you can change the links provided by the model world by modifying the model world and using the Cmvc.Replace_Model command. Replacing a working view's model does not affect previous releases made from that view.

Using Activities for Execution

After imports have been defined, units in client views can compile against units in imported spec views. However, spec views do not contain unit bodies and are therefore insufficient for execution. To execute units that are compiled against imports, you must set up an execution table called an *activity* to specify the implementation from each subsystem that will actually be used for execution. That is, an activity must specify one load view for each subsystem from which a spec view is imported.

At this point in the example, recall that the client view Command_Interpreter.Rev1_Working has imported the spec view Mail_Utilities.Rev1_0_Spec (see Figure 5-9). Within the client view, the prototyped unit Command_Utilities has been promoted to the coded state, and subprograms in it are ready to be tested (assume that a test driver has been created in Command_Interpreter.Rev1_Working to execute subprograms from Command_Utilities).

Now you must decide which implementation of the Mail_Utilities subsystem to use for execution. You can choose between any of the working or released views in that subsystem (note that a configuration release cannot be executed because it is not a full program library). Assume that you want to test Command_Utilities against the released implementation Mail_Utilities.Rev1_0_1, because it is frozen and stable.

Overview of Steps

Having decided on the implementations to be executed, you must follow these general steps, which are clarified in subsequent sections:

1. Create an activity.
2. Add entries to the activity to specify the chosen subsystem implementations; in this case, you need one entry that specifies Rev1_0_1 for subsystem Mail_Utilities.
3. Set the activity as the *default activity* for your session.
4. Execute the desired unit (for example, the test driver for Command_Utilities).

Note that a simple activity will be created for single application use, which is sufficient for the present example. In practice, however, you will need to build a more general-purpose activity so that you can run Environment tools and commands along with your application; this is covered in "Using General-Purpose Activities," below.

Creating an Empty Activity

When creating a simple activity for a single application, you can begin with an empty activity. To create an empty activity:

1. Choose a convenient location for the activity. You can create a special subdirectory for activities within a working view, within a subsystem, or within the application library that contains the subsystem. In this example, the activity is created in the application library !Programs.Mail.
2. Enter the Activity.Create command, specifying the The_Activity parameter with the desired name and using default values for the other parameters:

```
Activity.Create (The_Activity => "Mail_Activity");
```

As a result, an empty activity is created, as shown in Figure 5-11.

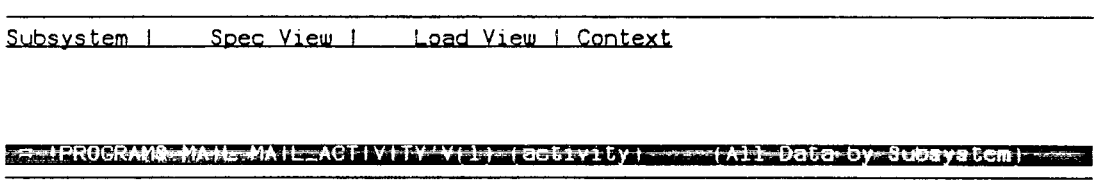


Figure 5-11. An Empty Activity

Adding Activity Entries

An activity must contain one entry for each subsystem that needs an implementation specified for it. To add an entry to an activity:

1. Display the activity.
2. Enter the Activity.Insert command, specifying two of the three parameters as follows:

Subsystem	The name of the subsystem for which an entry is to be added (in this case, Mail_Utilities)
Load_View	The name of the load view containing the implementation to be executed (in this case, Rev1_0_1)

   The third parameter, Spec_View, can be omitted for present purposes.
3. Use Common.Commit to save the contents of the activity.

For example, the following command creates the entry shown in Figure 5-12:

```
Activity.Insert (Subsystem => "!Program.Mail.Mail_Utilities",
                 Spec_View => "",
                 Load_View => "Rev1_0_1";
```

Subsystem	Spec View	Load View	Context
MAIL_UTILITIES		REV1_0_1	!PROGRAMS.MAIL

```
# !PROGRAMS.MAIL.MAIL_ACTIVITY'V(2) (activity) . . . (All Data by Subsystem) . .
```

Figure 5-12. An Activity with One Entry

Setting the Default Activity

After creating an activity that contains the appropriate entries, you must make it available for the Environment's use during execution. More specifically, you must make this activity the default for the session in which application and testing will take place.

To set the default activity for the current session:

1. Select the activity that is to be the default.
2. Enter the Activity.Set_Default command using default parameter values.
3. You can use the Activity.Current command to display the name of the activity that is currently the default.

Note that one default activity can be set for a given session. An error results if you try to execute units without a default activity.

CAUTION *The simple activity in the above example references only one subsystem. As long as an activity such as this is the default, you will not be able to execute other programs that are partitioned into subsystems, including many Environment tools and commands. To regain the use of Environment tools and commands, reset the default activity to the Environment default by entering:*

```
Activity.Set_Default ("!Machine.Release.Current.Activity");
```

The section "Using General-Purpose Activities" describes how to create an activity for your application that also references the Environment default activity.

The Execution Process

After creating an activity and setting it as the default for the current session, you can execute the test driver in Command_Interpreter.Rev1_Working to test subprograms from package Command_Utilities. It is during execution that the default activity is consulted.

Execution begins by *loading*, which is the process of determining the units to be executed, determining their elaboration order, and setting up the machine for execution. The loading process first looks for the main program (in this case, the test driver) and then looks for the units in the main program's *transitive closure*. (A unit's transitive closure is the set of units that are directly or indirectly *with*ed and that constitute the program to be executed.)

The loading process follows internal and external links to find the units in the transitive closure. When internal links lead to units in a load view (in this example, package Command_Utilities in Command_Interpreter.Rev1_Working), those units are earmarked for execution. In contrast, when external links lead to units in an imported spec view, the loading process consults the default activity to locate the load view that contains the actual units to be executed.

That is, when consulting the default activity, the loading process:

1. Determines which subsystem contains the imported spec view—in this case, Mail_Utilities.Rev1_0_Spec
2. Looks up the activity entry for that subsystem
3. Determines from that entry which load view to use—in this case, Mail_Utilities-.Rev1_0_1
4. Earmarks the appropriate units from that load view for execution

If the default activity contains no entry for the subsystem containing a spec view, an error such as the following is reported and the program is not executed:

```
1: ERROR Default activity does not define load view for subsystem of
                                                          spec unit
!PROGRAMS.MAIL.MAIL_UTILITIES.REV1_Ø_SPEC.UNITS.MESSAGES'V(1)
```

In sum, spec views are compiled against statically, but they are never actually executed. They serve as placeholders used for compilation, representing load views, which are actually executed.

Completing the Compilation and Execution Setup

As shown above, the basic compilation and execution setup includes defining exports, defining imports, and using activities. The following sections describe how to obtain more control over subsystem interfaces and more utility from activities.

At this point in the development of the mail program, an interface has been created between two subsystems (Command_Interpreter and Mail_Utilities). Remaining to be done are the following tasks, which serve as examples in subsequent sections:

- Create interfaces between each of these subsystems and the third subsystem under development (Mailbox)
- Build a general-purpose activity that specifies the subsystem implementations to be used during execution
- Decide how you will test the entire application and create an interface between test drivers and the application's main procedure

Imposing Further Import and Export Controls

A spec view such as Mail_Utilities.Rev1_0_Spec makes specific units available for reference within client views. However, a spec view may make too many units available for particular client views, for several reasons:

- A spec view may contain "extra" unit specifications that are required only for compilation within that view (see "What to Put in the State.Exports File," above). Such units include those that cannot be omitted by virtue of pragma Private_Eyes_Only. These units are present in the spec view even though no client needs them for compilation.
- A spec view must contain specifications for the entire set of units needed outside the subsystem. By design, however, a particular client view may require only a subset of these units for compilation.

Importing a spec view that contains too many units creates more links in the client view than are needed to support the design of the program. By virtue of these links, developers potentially can introduce dependencies that violate the program's design. To prevent this from happening, you can specify *export* and *import restrictions*, which make specific subsets of spec-view units available to specific client views. You can use these restrictions to minimize the links created in each of the spec view's clients.

For example, the spec view Mail_Utilities.Rev1_0_Spec contains four units, all of which are required for compilation by units in the client view Command_Interpreter-.Rev1_Working. However, only one of these four units (package Messages) is required for compilation within subsystem Mailbox, which implements a single package called Mailboxes (see Figure 5-2). Importing Mail_Utilities.Rev1_0_Spec as is creates links to three additional units, making them available for package Mailboxes to *with*. To prevent this, you can specify export and import restrictions so that when Mail_Utilities.Rev1_0_Spec is imported, only the desired link is created to package Messages in Mailbox.Rev1_Working.

The Environment offers several levels at which the availability of implemented units can be restricted:

- At the first level, the spec view defines which load view units are potentially available to any client view.
- At the second level, a spec view optionally can contain export restrictions that define subsets of exported units. Particular export restrictions can be requested by client views through their import restrictions.
- At the third level, import restrictions themselves optionally can be used to further restrict the subset of units from the requested export subset.

Overview of Steps

Export and import restrictions are created as files in particular subdirectories within the exporting and importing views. These files are consulted automatically by the import operation.

To import Mail_Utilities.Rev1_0_Spec into Mailbox.Rev1_Working using restrictions, you must follow these general steps, which are clarified in subsequent sections:

1. In the Exports subdirectory of the exporting (*supplier*) view, create an export restriction file that specifies the minimum set of spec-view units required by the client view.
2. In the Imports subdirectory of the client view, create an import restriction file that specifies, among other things, the export restriction file created in step 1.
3. Enter the Cmvc.Import command as before. The files created in steps 1 and 2 are used automatically to determine which links to create.

In the present example, only one export restriction file is created, because only one client requires a subset of exported units. However, in applications in which multiple clients require different subsets from a single supplier view, you must create multiple export restriction files, one for each subset.

Similarly, only one import restriction file is needed in this example because only one import needs to be restricted. However, in applications in which a given client view needs subsets from multiple imports, you must create multiple import restriction files, one for each import.

Creating Export Restriction Files

Every view is created with a subdirectory called Exports in which export restriction files can be created. The Exports subdirectory is at the same level of hierarchy as the view's Units directory. Do not confuse the Exports directory with the State.Exports file within the view (see "Locating the State.Exports File," above).

Export restriction files ultimately must reside in the Exports directory of the spec view to be imported (in this case, Mail_Utilities.Rev1_0_Spec). However, because ongoing development and maintenance occur in working views, it is recommended that you create and edit export restriction files in the working view containing the subsystem implementation and then copy the files into the appropriate subdirectory of the spec view. This makes it easier for you to create new spec views from the load view at any time. Note that export restriction files can be created in a working view before spec views have been made from it. In this case, the Cmvc.Make_Spec_View command automatically copies the files into the new spec view.

To create an export restriction file:

1. Use the Text.Create command to create a text file in the Exports directory of the relevant working view (in this case, Mail_Utilities.Rev1_Working).

   You can establish your own naming convention for export restriction files. In this example, a file called Subset_1 is created in the Mail_Utilities.Rev1_Working-.Exports directory.
2. In the empty file, specify the names of the exported units to be included in the restricted subset. For multiple names, enter one name per line or use a naming expression. In this example, only one name is entered—namely, Messages.
3. Commit the file.
4. If you want to be able to rebuild the file from a configuration release, you can make the file controlled using the Cmvc.Make_Controlled command.
5. Copy the export restriction file into the Exports directory of the spec view to be imported (Mail_Utilities.Rev1_0_Spec).

The resultant file and its parent directory are shown in Figure 5-13.

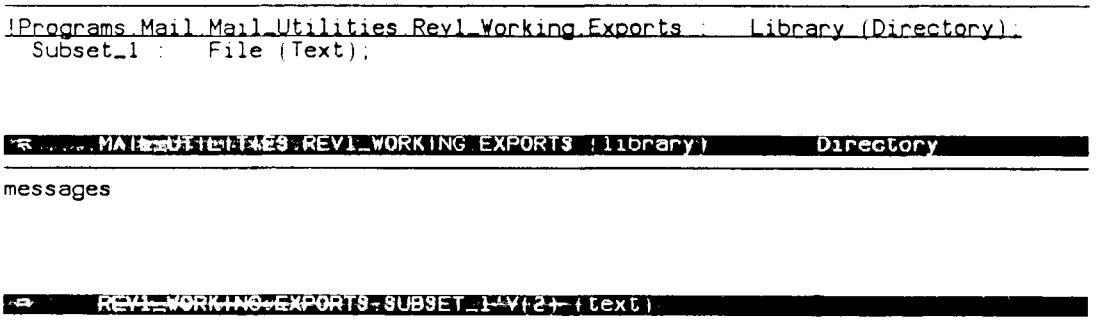


Figure 5-13. An Export Restriction File Called Subset_1

Name Resolution in the Export Restriction File

Names in the export restriction file are resolved against the Units directory within the view. Therefore, you can use a simple name like Messages for units located in the Units directory. If the Units directory contains subdirectories, you must use partly qualified names for units in those subdirectories. For example, for a unit called Message_Lists in the directory Units.Utilities, you would enter Utilities.Message_Lists in the export restriction file.

Note that you need to use the caret special character (^) in naming expressions to specify units in directories at the same level of hierarchy as the Units directory.

Export and Import Restriction Files

At this point, a subset of exported units has been defined by creating an export restriction file and copying it into the appropriate spec view. The next step is to create an import restriction file in the client view to enable that view to take advantage of the export subset. An import restriction file uses the following two conventions to request a particular export restriction file:

- The import restriction file must have the same name as the subsystem containing the supplier view. (See "Import Restriction Filenames," below.)
- The first line of the import restriction file specifies the simple name of the desired export restriction file. Omitting this line implicitly specifies an export restriction file named Default.

For example, because Mailbox.Rev1_Working requires the export restriction file Subset_1 from the supplier view Mail_Utilities.Rev1_0_Spec, you must create a corresponding import restriction file called Mail_Utilities that specifies Subset_1 in its first line. Note that when a given supplier view defines multiple export subsets, it is the correspondence between restriction files that enables a client view to specify which of many subsets to use during import.

Creating Import Restriction Files

Import restriction files are created in the Imports directory within the client view. Like the Exports directory, the Imports directory is at the same level of hierarchy as the view's Units directory. Note that because both spec and load views can be client views, you may need to create import restriction files in both kinds of views. Within a given client view, one import restriction file must be created for each supplier view from which an export subset is required.

To create an import restriction file:

1. Use the Text.Create command to create a text file in the Imports directory of the importing view (in this example, Mailbox.Rev1_Working).

   Because an import restriction file is named for the subsystem containing the supplier view, the new file in this example is called Mail_Utilities. (See "Import Restriction Filenames," below.)

2. On the first line of the empty file, enter the name of the desired export restriction file as shown (do not put blanks around or between the => characters):

   ```
   export_restriction=>subset_1
   ```

3. Starting on the second line of the file, enter one or more naming expressions that specify those units in the export restriction file for which links are to be created. Naming expressions should match simple names. In this example, a link is needed for the only name in the export subset, so the *at* sign (@) wildcard is entered (@ matches all names in the export restriction file). (See "What to Put in Import Restriction Files," below.)
4. Commit the file.
5. If you want to be able to rebuild the file from a configuration release, you can make the file controlled using the Cmvc.Make_Controlled command.

Figure 5-14 shows the resultant file and its parent directory.

```
!Programs.Mail.Mailbox.Rev1_Working.Imports : Library (Directory);
  Mail_Utilities : File (Text);

MAIL_MAILBOX.REV1_WORKING.IMPORTS (library)          Directory

export_restriction=>subset_1
@

WORKING.IMPORTS.MAIL_UTILITIES (V1) (text)
```

Figure 5-14. An Import Restriction File That References Subset_1

Import Restriction Filenames

Typically, the simple name of the supplier subsystem (for example, Mail_Utilities) is used for the name of the import restriction file.

In very large applications where subsystems in different worlds have the same simple name, you can derive the import restriction filename from the fully qualified subsystem name by:

- Omitting the preceding ! in the fully qualified name
- Changing the dot (.) between name components to underscores (_)

For example, Programs_Mail_Mail_Utilities is the import restriction filename derived from the fully qualified subsystem name !Programs.Mail.Mail_Utilities.

What to Put in Import Restriction Files

As shown above, the first line of an import restriction file can be used to identify which of the supplier view's export restriction files to use. The first line consists of the string `export_restriction=>` followed by the export restriction filename. Do not enter blank spaces in this line. Omitting this line implicitly specifies an export restriction file named Default, which is used if such a file exists; otherwise, the entire supplier view is used.

On subsequent lines in the file, you can enter naming expressions to specify a further subset of the units listed in the export restriction file. Enter one naming expression per line. Links are created in the importing view for the units that are matched by the naming expressions. Note that if no naming expressions are specified, no links are created.

Because the import restriction file essentially specifies a set of link names, only simple Ada names should be used in the naming expressions. This is true even for names that are qualified within the export restriction file. Whereas names in an export restriction file are resolved as library names, names in an import restriction file are resolved as link names.

You can use naming expressions to:

- Request links for all units in the export restriction file by entering `@` (as shown in Figure 5-14)
- Request links for subsets by using wildcard expressions such as `@_pkg`
- Exclude links to units by using expressions such as `~Unit_Name` (such an expression normally follows an expression such as `@`)
- Rename links to units by specifying the unit name followed by the new link name (see below)

If a unit in an imported view has the same simple name as a unit in the client view, the internal link that already exists in the client view prevents the creation of the external link from the import. In such a case, you can use the import restriction file to rename the link, thereby allowing the link to be created without renaming the imported unit.

For example, if a client view contains a unit named Interface and the import contains a unit named Interface, the following entry in the import restriction file creates an external link called Other_Interface instead of Interface:

```
interface  other_interface
```

In units within the client view, you use a statement such as `with Other_Interface` to reference the unit Interface from the supplier view.

Note that links should be renamed only if absolutely necessary. Instead, you should design your application so that the main program does not reference two units with the same simple name.

Summary of Import and Export Restriction Setup

At this point in the example, Mail_Utilities.Rev1_0_Spec contains an export restriction file called Subset_1, which is referenced by the import restriction file called Mail_Utilities in Mailbox.Rev1_Working. This setup is represented in Figure 5-15.

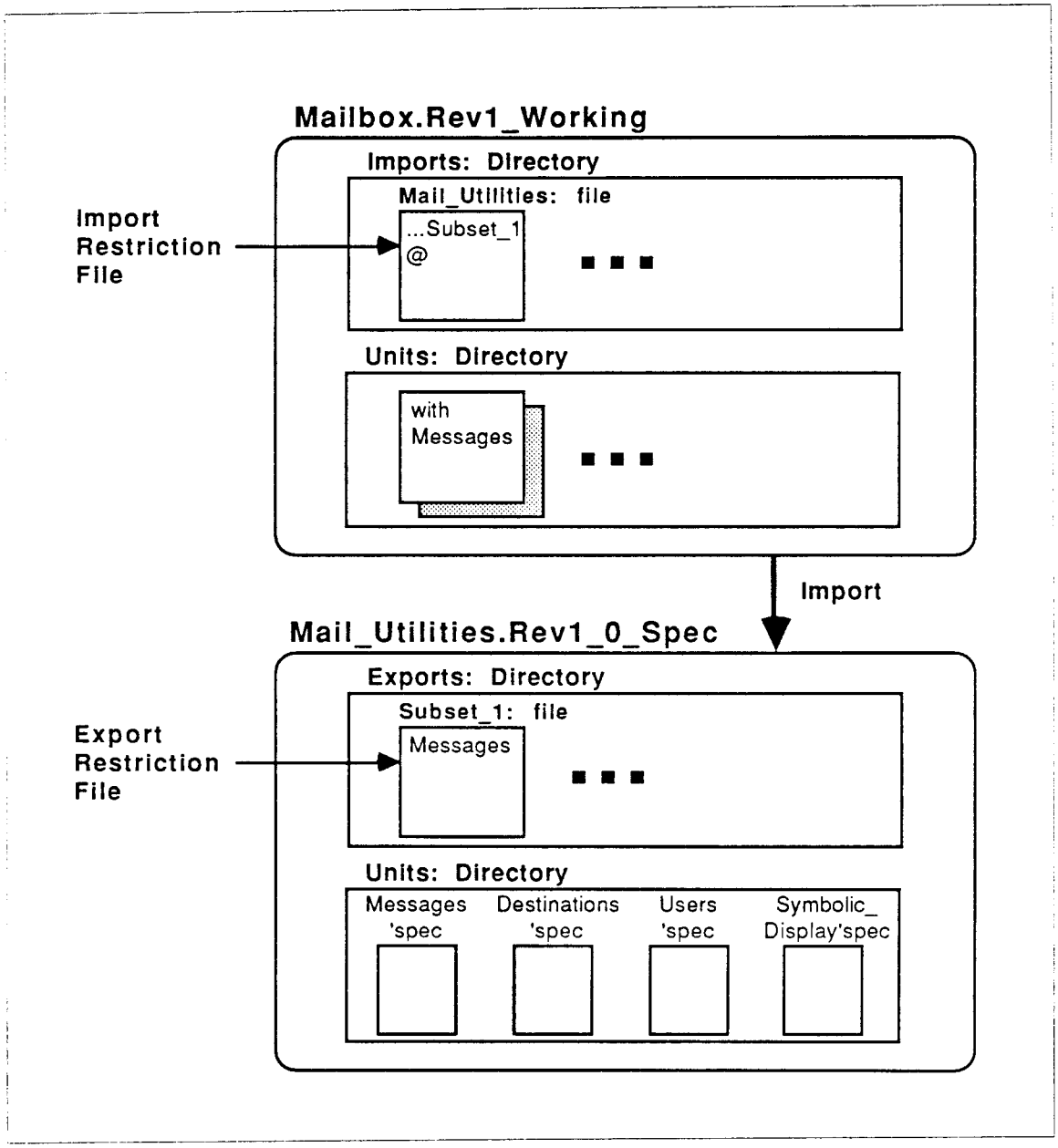


Figure 5-15. Import and Export Restriction Setup

When the Cmvc.Import Command Is Entered

To complete the import operation, enter the Cmvc.Import command as before, specifying the supplier view for the View_To_Import parameter and the client view as the Into_View parameter. Do not specify restriction filenames as parameters to the Cmvc.Import command.

When the Cmvc.Import command is entered, the information from the restriction files is used as follows:

1. From the name of the supplier view specified by View_To_Import, the import operation determines the name of the enclosing subsystem (in this case, Mail_Utilities).
2. In the Imports directory of the client view, the import operation looks for an import restriction file with the name that was obtained in step 1.
3. If no file is found with that name, links are made for units as specified by the supplier view. Specifically, links are made for the units named in an export restriction file named Default, if there is one; otherwise, links are made for all units in the supplier view.
4. If there is an import restriction file with that subsystem's name, the import operation looks inside the file to find the name of the export restriction file. (If no export restriction file is named, an export restriction file named Default is used, if one exists.)
5. The import operation gets a set of names from the export restriction file.
6. To this set of names, the import operation applies the naming expressions from the import restriction file, eliminating names as specified.
7. The import operation makes links for the resultant set of names.

More on Importing

Preparing an entire application for compilation typically involves many import operations. So far in the mail program, two clients have imported the spec view from Mail_Utilities. To set up imports for the entire mail program:

1. Create a spec view from the working view in subsystem Mailbox. By default, the Cmvc.Make_Spec_View command causes the new spec view to inherit imports; therefore, the spec view in Mailbox automatically imports the spec view from Mail_Utilities, as required.
2. Import the spec view from subsystem Mailbox into the load view in subsystem Command_Interpreter.
3. For completeness, create a spec view from the working view in subsystem Command_Interpreter. This spec view should contain only the specification for the main procedure, Run_Mail.

   Because Run_Mail'Spec contains no *with* clauses, the spec view requires no imports. To prevent the spec view from inheriting imports from the working view, enter the null string ("") for the View_To_Import parameter in the Cmvc.Make_Spec_View command.

The complete network of import relationships is represented in Figure 5-16.

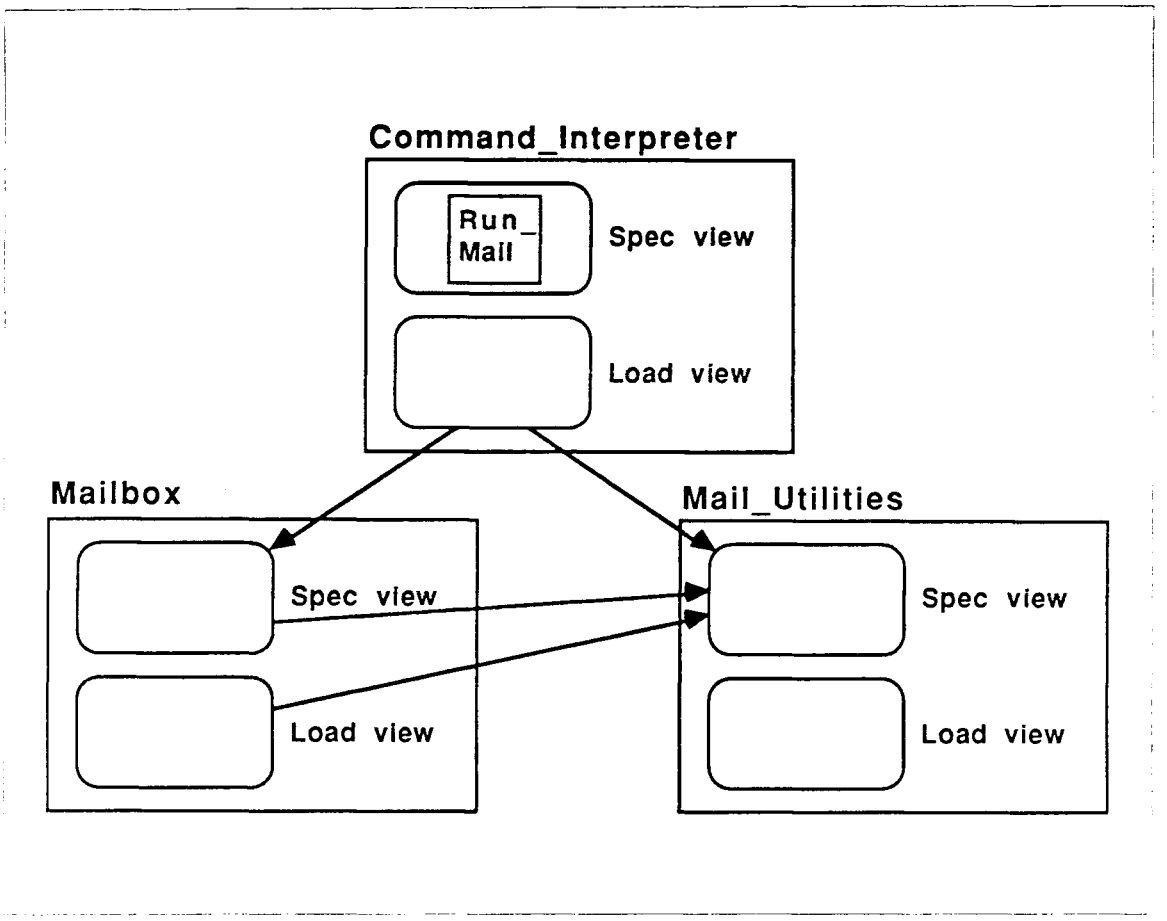


Figure 5-16. Import Relationships in the Mail Program

Within a single program, networks of import relationships among spec and load views must have the following properties, which are explained below:

- They must be *consistent.*
- They may not be *circular.*

Import operations automatically enforce these properties. In fact, import operations will fail if you attempt to create a network of imports that violate either of these properties.

Consistency

Within a consistent set of imports, no view can directly or indirectly import more than one spec view from the same subsystem. Thus, if a subsystem contains more than one spec view, only one of these spec views can be used throughout a single chain of imports. ("Making Non-Upward-Compatible Changes" describes conditions under which additional spec views are created.)

For example, assume that a second spec view is created in the Mail_Utilities subsystem to accommodate implementation changes. The diagram in Figure 5-17 illustrates an inconsistent network of import relationships.

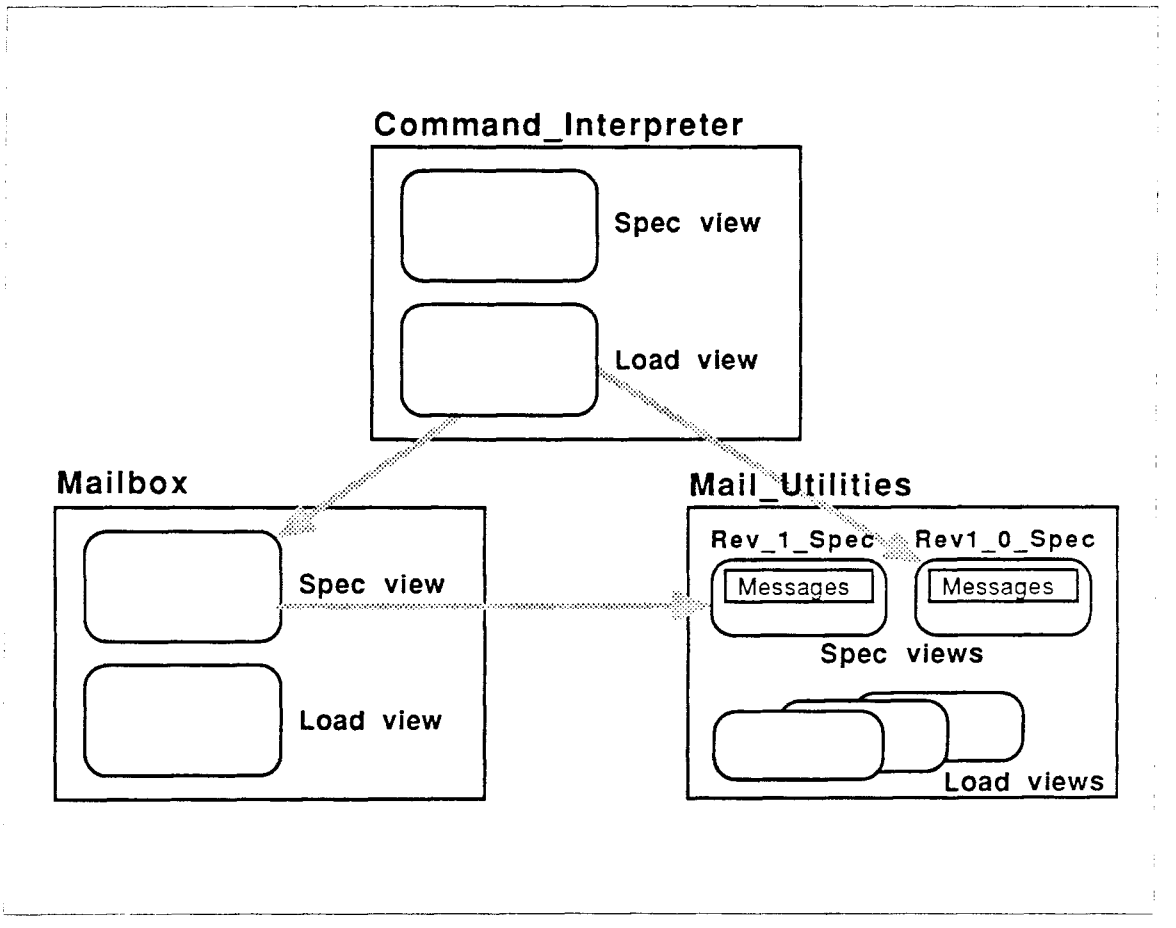


Figure 5-17. An Inconsistent Network of Import Relationships

This network is inconsistent because a view in Command_Interpreter directly imports Rev1_0_Spec and indirectly imports Rev1_1_Spec from Mail_Utilities. As a result, ambiguous references are introduced into the mail program; for example, a reference to package Messages is ambiguous because both spec views contain an

instance of package Messages. To prevent you from creating illegal Ada programs in this way, the Cmvc.Import command will fail if you try to set up inconsistent imports.

Circularity

Within a set of imports among spec and load views, no spec view can indirectly import itself. For example, the diagram in Figure 5-18 illustrates a circular network of import relationships.

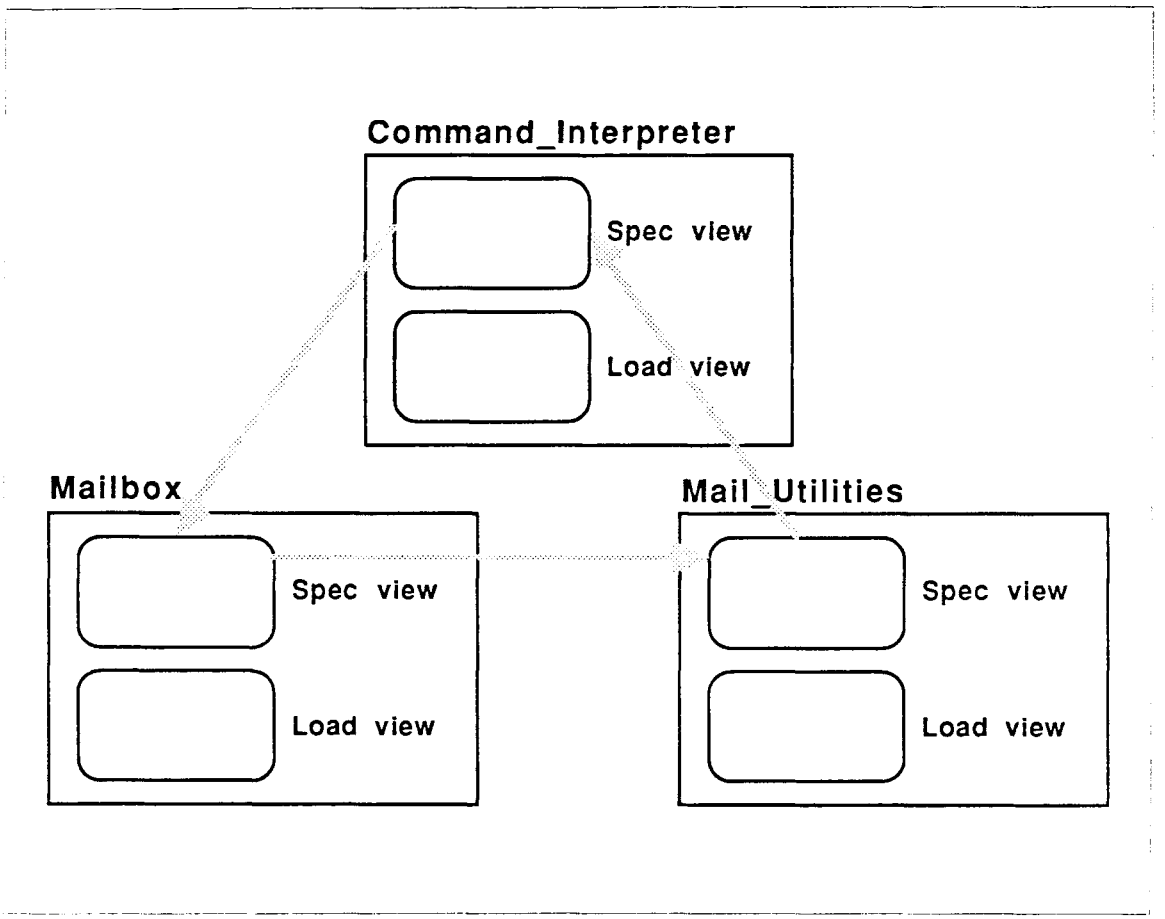
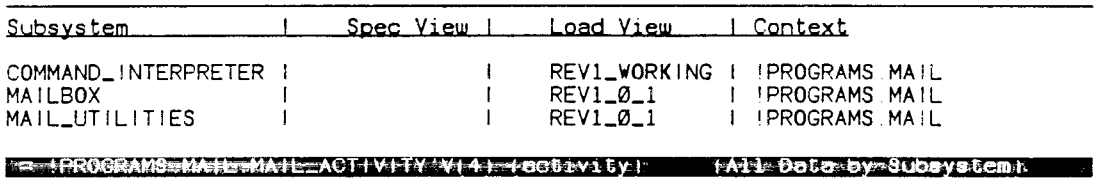


Figure 5-18. A Circular Network of Import Relationships

The Cmvc.Import command will fail if you try to set up circular imports. Note that some existing applications require circular imports among program components. Such applications should be developed in *combined views* within *combined subsystems*. These are discussed in the introduction to package Cmvc.

Using General-Purpose Activities

When you execute an application that is partitioned into subsystems, the Environment consults the default activity to look up which implementation to execute from each subsystem. Therefore, when you execute an entire application, the default activity must contain one entry for each subsystem in the application. For example, to execute the entire mail application with Mail_Activity as the default activity, you must add two more entries to it, such as those shown in Figure 5-19.



Subsystem	Spec View	Load View	Context
COMMAND_INTERPRETER		REV1_WORKING	!PROGRAMS.MAIL
MAILBOX		REV1_0_1	!PROGRAMS.MAIL
MAIL_UTILITIES		REV1_0_1	!PROGRAMS.MAIL

Figure 5-19. An Activity Containing Entries for the Entire Mail Application

While logged into a given session, however, you typically need to execute more than just one such application; in fact, many Environment and user-defined commands and tools are partitioned into subsystems. Therefore, the default activity for a session must accommodate more than just a single application; it must specify implementations for any subsystem-based application that you execute during that session.

One obvious way to construct such a default activity is to add all the required entries to a single-application activity such as Mail_Activity. However, this method typically is not practical because of the large number of entries required to support tools and commands. Another option is to reset the default activity each time you want to execute a different application, tool, or command. This option is impractical also, for example, when using user-defined tools to debug an application as it executes.

Instead, it is recommended that you construct a separate general-purpose activity that references other, more special-purpose activities. Such an activity references other activities by containing pointers to their entries. In this example, you can create a general-purpose default activity that references:

- The Environment's standard activity (!Machine.Release.Current.Activity), which contains the required entries for tools and commands provided by the Environment
- Mail_Activity, which contains entries for the subsystems in the mail program

Figure 5-20 illustrates an activity called Test_Activity that accommodates both the mail program and standard Environment tools and commands. As shown, Test_Activity contains pointers to entries in Mail_Activity and to entries in the Environment's standard default activity, !Machine.Release.Current.Activity. (Thus, Test_Activity constitutes a superset of the standard default activity.)

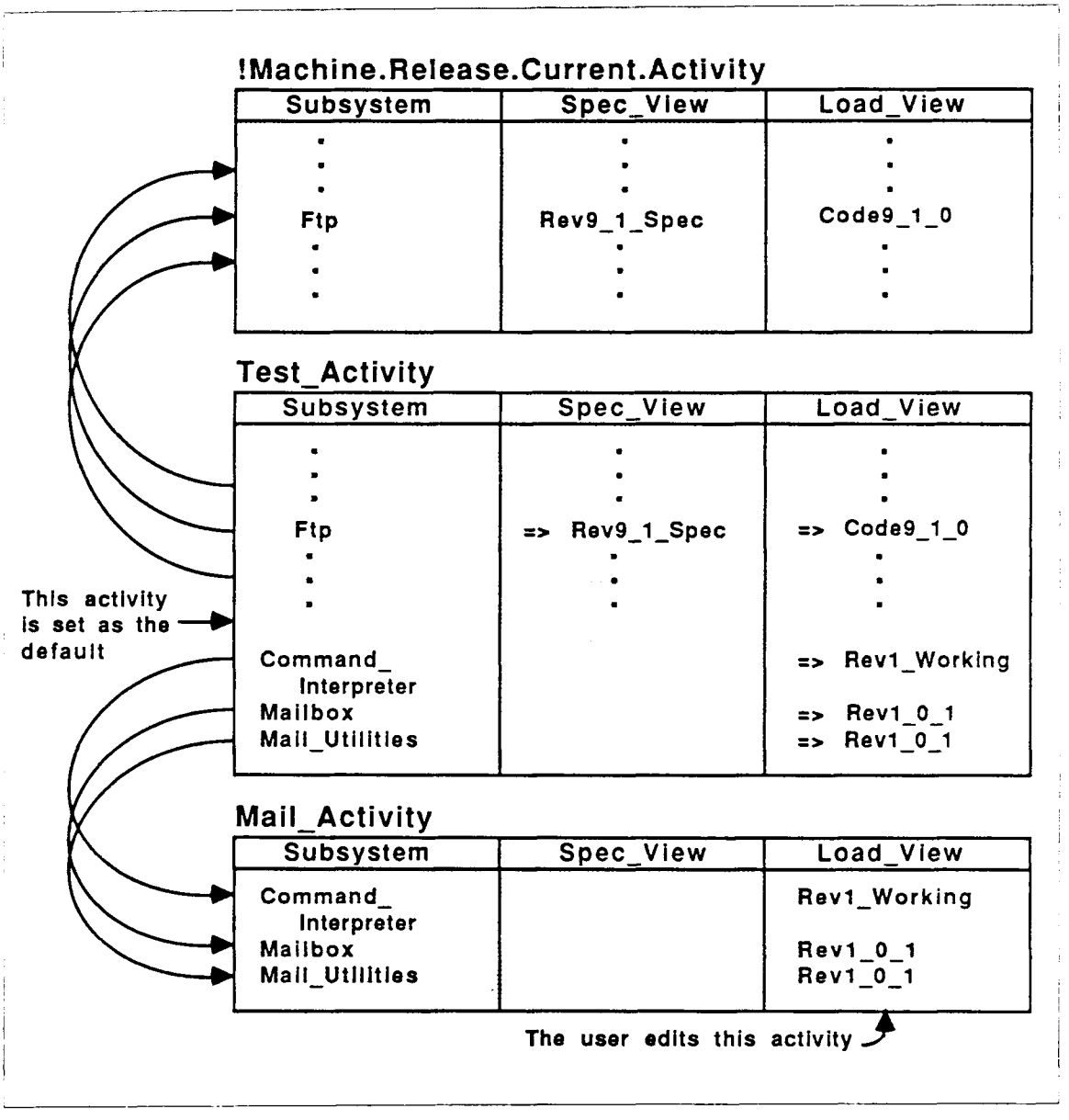


Figure 5-20. A General-Purpose Activity That Points to Two Other Activities

Modes for Creating Activity Entries

Recall that new entries, such as those in Mail_Activity, are created directly, using the Activity.Insert command. Alternatively, entries in an activity such as Test_Activity (shown in Figure 5-20) are derived from entries in other existing activities. There are three modes for deriving entries from a source activity:

`Differential`	Entries created in differential mode are pointers to a source activity's entries. Test_Activity contains differential entries.
`Exact_Copy`	Entries created in exact-copy mode are copies of the entries in the source activity.
`Value_Copy`	Entries created in value-copy mode are the dereferenced values of the entries in the source activity. (This is useful when the source activity itself contains pointers to entries in other activities.

Several commands allow you to specify a mode for creating activity entries, including Activity.Add, Activity.Create, and Activity.Merge. Note that you can insert new entries among derived entries; furthermore, all types of entries can be deleted or changed (see package Activity).

Differential entries are especially useful because they allow you to manage special-purpose activities (for example, Mail_Activity) separately, automatically reflecting any changes made to the referenced activities.

Creating an Activity with Differential Entries

As shown in Figure 5-20, Test_Activity is to contain differential entries from two other activities, !Machine.Release.Current.Activity and Mail_Activity. Differential entries from one source activity are created when Test_Activity is created; differential entries from the other source activity are subsequently merged in a separate step, as follows:

1. In the appropriate context (in this case, !Programs.Mail), enter the Activity-.Create command, specifying the name of the new activity, one of the source activities from which entries are derived, and the mode for deriving these entries:

```
Activity.Create (The_Activity => "Test_Activity",
                 Source => "!Machine.Release.Current.Activity",
                 Mode => Activity.Differential);
```

2. Select the new activity and create a Command window.
3. Enter the Activity.Merge command, specifying the name of the remaining source activity and the mode for deriving entries from the source activity (use default values for the remaining parameters):

```
Activity.Merge (Source => "Mail_Activity",
                Mode => Activity.Differential);
```

4. Use Activity.Set_Default to set Test_Activity as the default activity.

Preserving the Default Activity between Logins

When you set a default activity for a session, that activity is recorded automatically in your session switches. In this way, your default activity is preserved from login to login.

If, however, you find that your default activity has been reset the next time you log in, check the login procedure(s) that are executed for your login. These may include an Environment default login procedure (!Machine.Release.Current.Commands.Login) or a customized login procedure of your own. Login procedures typically call Activity.Set_Default to ensure that a default activity is set when you log in. In particular, the default Environment login procedure resets the default activity to be !Machine.Release.Current.Activity.

To prevent your default activity from being reset inadvertently, delete the call to Activity.Set_Default from your own login procedure, if you have one; otherwise, define a login procedure of your own (see the *Rational Environment Basic Operations*) that does not call Activity.Set_Default. (Do not edit the Environment default login procedure.)

Executing the Entire Application

To execute an entire application, you must execute its main program, which is the procedure that serves as the root of the application's dependency closure. (For applications that execute on R1000 targets, main programs can be subprograms in the library or in packages; for targets other than the R1000, main programs typically are parameterless procedures.) For example, the procedure Run_Mail is the main program for the mail application.

Within an application that is partitioned into subsystems, the specification for the main program typically exists in multiple contexts—specifically, in one or more spec and load views. Figure 5-21 shows the main program Run_Mail in each of several views within the subsystem Command_Interpreter.

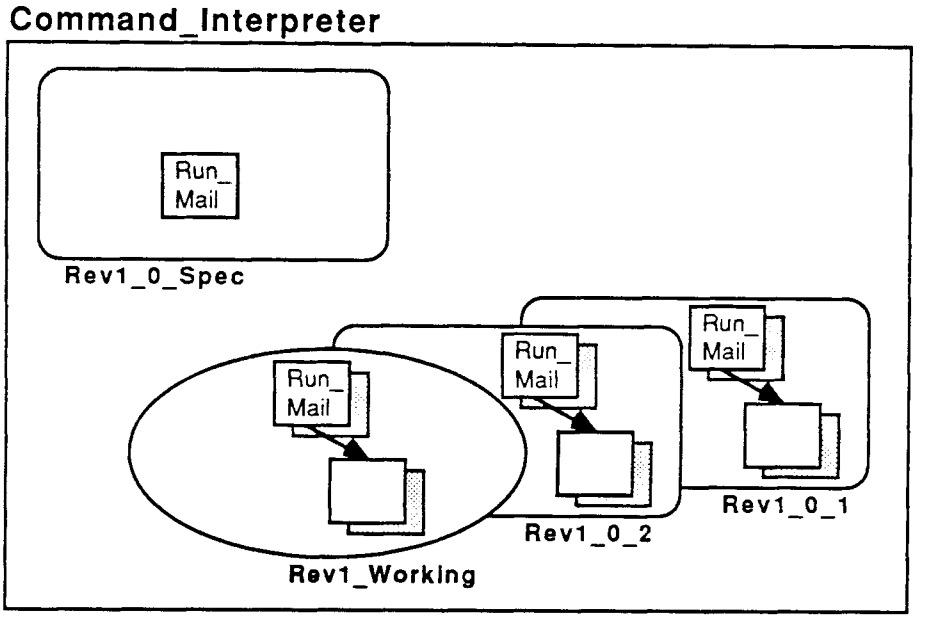


Figure 5-21. Multiple Instances of an Application's Main Program

Although the main program can be executed from any of these contexts, it is recommended that you execute the instance of the main program in the spec view.

When you do this, the execution process consults the default activity to look up and use the implementation specified for the top-level subsystem. In contrast, if you execute the main program from a load view, the implementation in that load view is executed, regardless of the implementation specified by the default activity.

For example, if you execute Run_Mail from the spec view Rev1_0_Spec, the execution process consults the default activity to find out which implementation to use from the Command_Interpreter subsystem. Recall that the default activity references Mail_Activity, which specifies the implementation Rev1_Working, as shown in Figure 5-20 above. If, on the other hand, you execute Run_Mail from the load view Rev1_0_1, then the implementation in Rev1_0_1 is executed instead of the one specified in the default activity. In both cases, the default activity is consulted for every imported spec view. (The role of activities during execution is covered in "The Execution Process," above.)

Testing an Application

Testing typically involves test drivers that call the application's main program. There are several strategies for setting up test drivers. One strategy is to create test drivers in a world external to the application subsystems. For example, you can create test drivers in the application world !Programs.Mail. In this case, you must use Links.Add to create a link to the appropriate instance of the main program Run_Mail—namely, to the instance in Command_Interpreter.Rev1_0_Spec. (If you link to an instance of Run_Mail in a load view, the execution process will bypass the default activity when loading the Command_Interpreter subsystem.)

Alternatively, you can create a subsystem in which to maintain test drivers. In this case, you can put all test drivers in a working load view that imports the spec view containing the main program. (Note you must use imports instead of creating links when the test drivers are in a subsystem.)

Recombinant Testing

You can test different combinations of implementations by changing entries in the default activity (or by changing entries that are referenced by the default activity). For example, assume that you want to test the entire application using a new, unreleased implementation of Mail_Utilities—namely, Rev1_Working. To do this:

1. Display the activity that contains entries for the application (in this case, Mail_Activity).
2. Select the entry to be changed (in this case, the Mail_Utilities subsystem).
3. Enter the Activity.Change command, specifying the Load_View parameter with the name of the desired implementation (in this case, Rev1_Working).
4. Use Common.Commit to save the contents of the activity.

   The default activity (Test_Activity) is updated automatically because it contains differential entries that point to Mail_Activity.

Assuming that the units in Mail_Utilities.Rev1_Working are compiled, you can now execute Run_Mail (or a test driver that calls Run_Mail) to run the entire application.

By changing activity entries, you can test alternative implementations without recompiling the entire application. That is, for applications partitioned into subsystems, a "system build" amounts to specifying a particular combination of precompiled load views in an activity. Without having to recompile the application, you can easily isolate the effects of a new release by testing it against proven releases in other subsystems.

Note that you can specify alternative test combinations in separate activities. Then, instead of changing entries in an activity, you can reset the default activity to test the desired combination.

Making Implementation Changes

Implementation changes are those changes that affect only a working view within a single subsystem. Such changes do not affect the units in spec views or the client views that are compiled against spec views. Several types of implementation changes are described in the following sections. Changes that affect spec views and their clients are covered in "Making Design Changes," below.

Changing Nonexported Units

The simplest type of implementation change involves nonexported units in a working view. You can make arbitrary changes to any unit body in the working view as well as to any unit specs that occur only in the working view. For example, in the Mail_Utilities subsystem, the nonexported units are: Destinations'Body, Line'Body, Messages'Body, Symbolic_Display'Body, Unbounded'Body, and Unbounded'Spec.

Assume that you need to fix a problem that affects both Unbounded'Body and Unbounded'Spec. To make the necessary implementation changes:

1. Within Rev1_Working, change the relevant portions of Unbounded'Body and Unbounded'Spec. You can demote the units to the source state or edit them incrementally.
2. Recompile the affected units in Rev1_Working. This includes all units in the present example, because all units in this view depend on Unbounded'Spec.
3. If desired, make a new release (Rev1_0_3) from Rev1_Working. Because only nonexported units were changed, Rev1_0_3 is compatible with Rev1_0_Spec and therefore can be executed with views that import Rev1_0_Spec.
4. To test the new release, change the appropriate activity entry to reference Rev1_0_3 and execute the application. (See "Recombinant Testing," above.)

Note that because clients are compiled against spec views, not load views, the units in a load view can be recompiled without affecting the clients. In effect, the presence of a spec view serves to minimize the recompilation required after making changes.

Changing Private Parts in Exported Units

The Environment provides support for the conceptual separation of private parts from visible package declarations, in that private parts in spec view units are *closed* by default. Closed private parts are ignored when spec views are compiled; the load view supplies the private parts at execution time. Accordingly, private parts in exported units can be changed in the load view without affecting the corresponding spec view and without requiring the recompilation of client views. Thus, changing the private part in an exported unit is effectively an implementation change.

For example, the Mail_Utilities subsystem exports a package called Destinations, which contains a private type called User. Assume that you need to optimize the completion of Destinations.User by implementing it as a linked list instead of a variable string. To do this:

1. Make the change in the private part of Destinations'Spec in the load view Rev1_Working. (This private part is shaded in Figure 5-22.) You can edit the unit specification or use incremental operations.
2. Make the necessary adjustments to Destinations'Body.
3. Recompile the units in Rev1_Working and test the application.

As the preceding steps indicate, changes to an exported unit's private part need to be made only in the load view, which means that the unit in the working view now differs from the corresponding unit in the spec view. This difference does not affect the compatibility between the working view and spec view, and the working view still can be executed as a valid implementation of the spec view.

As a matter of user preference, you can keep units textually identical across spec and load views by changing the corresponding private part in the spec view. This extra step is not necessary, however, because closed private parts are ignored when spec views and their clients are compiled. Note that if you do modify a unit in the spec view, demoting the unit to source will entail the demotion of dependent units in client views as well.

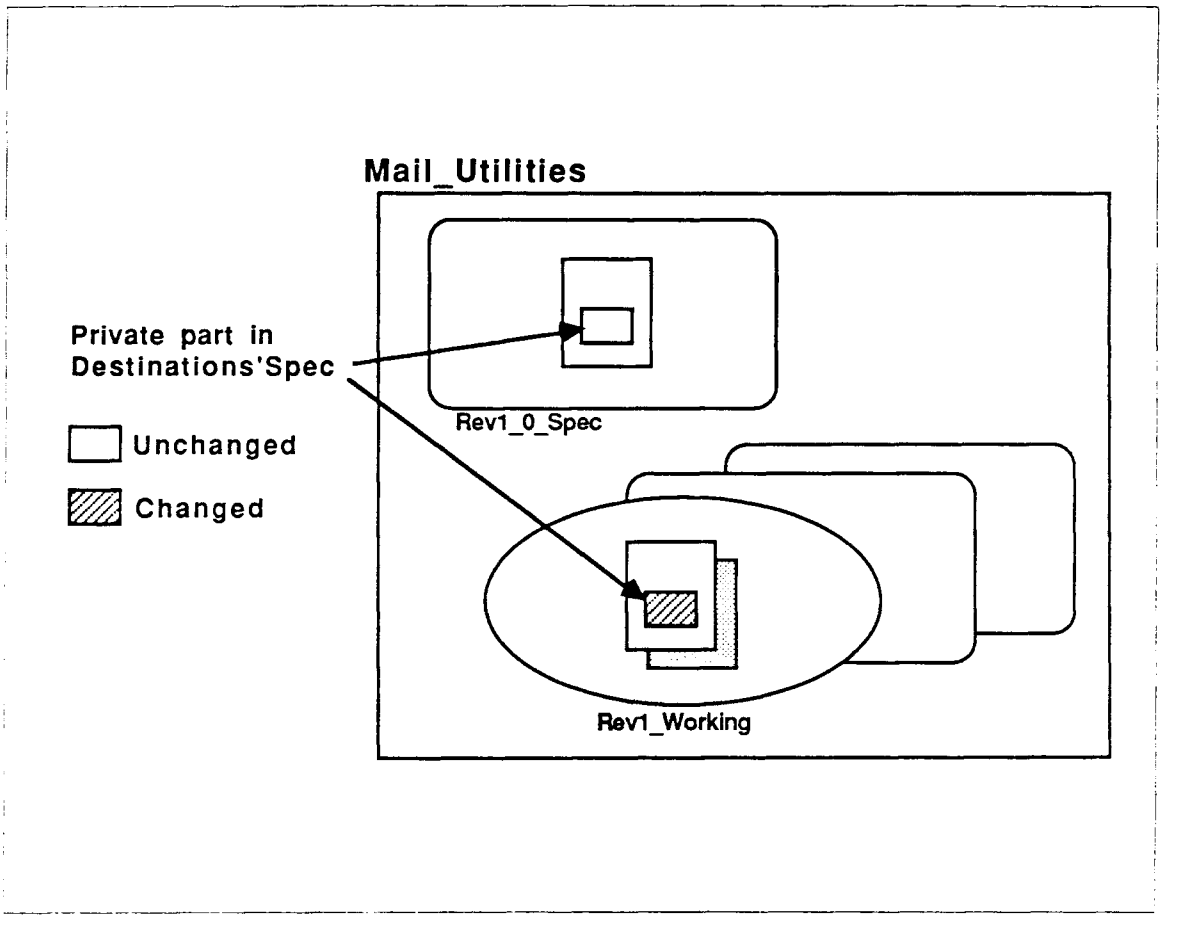


Figure 5-22. Compatible Spec and Load Views with Different Private Part

More on Compatibility

A load view is compatible with a spec view if it implements the unit specifications in that spec view. When a spec view is created from a load view, the exported unit specifications are identical across the two views. The preceding discussion shows that corresponding units in spec and load views can be different without rendering the two views incompatible. Following are the specific differences that can exist between corresponding units in compatible spec and load views:

- Declarations in the private part can be completely different. (This is true only when private parts are closed. See "More on Closed Private Parts," below.)
- Context clauses (*with* and *use*) can be different.
- Declarations can appear in different order, although they must have the same form.
- Unit specifications in a load view can contain additional declarations that are not present in their spec-view counterparts.

More on Closed Private Parts

Private parts are closed only in spec views that have target key R1000. Furthermore, private parts are closed only in units that are not generics. Accordingly, private parts are *open* in generics and in units in spec views that have other target keys. Open private parts are compiled along with the rest of the spec view. Therefore, when private parts are open:

- Pragma Private_Eyes_Only has no effect. (See "Using Pragma Private_Eyes_Only," above.)
- Changes to private parts must be made not only in the working view but also in the corresponding spec view; otherwise, the spec and working view are rendered incompatible. See "Making Non-Upward-Compatible Changes," below, for more information.

Making Design Changes

Design changes are those changes that affect subsystem interfaces, including changes to exported unit specifications in spec views. Design changes potentially entail the recompilation of units in client views. Design changes include:

- Upward-compatible changes to unit specifications—for example, adding new declarations
- Non-upward-compatible changes to unit specifications—for example, changing the parameter profile in a subprogram declaration
- Adding or removing unit specifications from spec views
- Changing a view's imports
- Changing a view's model

Making Upward-Compatible Changes

Upward-compatible design changes include adding new declarations to exported units as well as deleting unused declarations (declarations that have no dependents). Such changes are upward-compatible in that they introduce no conflicts with existing declarations and dependencies in the program.

For example, assume that you are required to add a new field to mail messages in the mail program. You can do this by adding new selectors and constructors; that is, by adding declarations for new procedures and functions to the unit specification of package Messages. To do this:

1. Add the new declarations to Messages'Spec in Mail_Utilities.Rev1_Working, using incremental operations.
2. Make the necessary additions to Messages'Body.
3. Recompile units in Rev1_Working as necessary and test the changes within the working view. Note that testing is limited to the working view because the new declarations are not yet available for units in client views to reference.

4. If desired, make a new release (Rev1_0_4) from Rev1_Working. Note that Rev1_Working and Rev1_0_4 are compatible with Rev1_0_Spec, even though they implement additional declarations.
5. Export the new declarations by using incremental operations to add them to Messages'Spec in Rev1_0_Spec. If, for some reason, the addition cannot be done incrementally (for example, if the new declaration introduces a naming conflict), continue with step 3 in either of the methods described in "Making Non-Upward-Compatible Changes," below. See also "Effects of Demotion in a Spec View with Clients," immediately below.
6. Modify those client view units that need to reference the new declarations. Note that:
   - Clients can take advantage of the new feature without having to reimport the Rev1_0_Spec.
   - Clients that do not need to reference the new declarations do not need to be changed or recompiled.

Note that these steps call for implementing the new declarations before exporting them. Depending on the methodology at your installation, you can also export the new declarations first and then implement them. This amounts to doing step 5 before any of the other steps.

Effects of Demotion in a Spec View with Clients

Step 5 above recommends the use of incremental operations to change the unit in the spec view and advises against demoting the unit to source. This is necessary because demoting a unit to source entails demoting its transitive closure, which consists of all units that directly or indirectly depend on the unit to be changed. Demoting a unit in a spec view poses two problems when that spec view has clients:

- If the spec view's clients include released views, then the required demotion cannot take place, because units in releases are frozen. In this case, the unit in question cannot be demoted and changed.
- If all of the spec view's clients are working views, then units in the transitive closure can be demoted as required and the unit in question can be changed. However, clients cannot execute while a spec view is demoted, and, depending on the size of the transitive closure, the recompilation cost can be high.

In contrast, incremental operations require neither the demotion of an entire unit nor the demotion of a unit's transitive closure. Therefore, changes that can be made incrementally do not affect units in client views. When changes cannot be made incrementally (because they introduce a conflict), you can use the method described in "Making Non-Upward-Compatible Changes," below.

To prevent inadvertent demotion of units in spec views, make these units either uncontrolled or controlled but not joined. This is necessary because controlled units must be checked out to perform incremental operations, as in step 5. However, if the unit is joined to its counterpart in the load view, the checkout operation will attempt automatically to demote the unit to source and then update it to the latest generation.

Implications for Prior Releases

When you make an upward-compatible change to a unit in a spec view, client releases that were made before the change still can be executed against any implementation of the changed spec view. For example, assume that the Command_Interpreter subsystem contains several releases that import Mail_Utilities-.Rev1_0_Spec. These releases were compiled against the spec view before the new declarations were added to Messages'Spec. Upward-compatibility guarantees that you can specify any of these prior releases in an activity for successful execution after the spec-view change.

Furthermore, the Mail_Utilities subsystem itself contains several releases that were compatible with Rev1_0_Spec before the new declarations were added. These prior releases are now technically incompatible with the changed spec view because they do not implement the new declarations. However, you can specify any of these prior releases for execution, provided that you specify prior releases from client subsystems as well.

Making Non-Upward-Compatible Changes

Non-upward-compatible changes are changes to existing declarations in spec-view units. Such changes include changing the parameter profile of a subprogram or changing a nonprivate type. Such changes typically require the demotion and recompilation of other units.

This section presents several methods for making non-upward-compatible changes. Each method involves making a new spec view to avoid the demotion problems imposed by frozen, released clients. The methods differ with respect to the amount of editing and recompilation required. Therefore, the method you should choose depends on the size of your program and the nature of the changes to be made. Note that any upward-compatible change also can be made using any of these methods.

Method I

Method I involves generating a new spec view from a working view. This method is easiest for keeping units textually identical across spec and load views, with a minimum amount of editing effort. However, because this method has the maximum recompilation cost, as shown below, it is preferable in the following situations:

- When making many changes to one or more exported units
- When relatively few units depend directly or indirectly on the units in the changed spec view (that is, when the transitive closure for the entire view is small)

For example, assume that you need to completely rewrite package Messages in the Mail_Utilities subsystem. To do so:

1. Rewrite package Messages in Mail_Utilities.Rev1_Working and test the changed package in that view.
2. If desired, make a release from Rev1_Working.

   Note that the changes to package Messages make this release incompatible with the existing spec view (Rev1_0_Spec). Therefore, you can increment the release's level 1 number to indicate the start of a new family of compatible releases. To do this, specify the value 1 for the Level parameter in the Cmvc.Release command. The resulting release is called Rev1_1_1.
3. Make a new spec view from Rev1_Working. The new spec view automatically contains a copy of the changed Messages'Spec.

   Note that if you followed step 3 to create a new release with an incremented level 1 number, you do not have to specify the Level parameter in the Cmvc.Make_Spec_View command. Instead, the previously incremented number is used automatically in the new spec view name, Rev1_1_Spec.
4. Switch the imports of all unfrozen client views from the old spec view (Rev1_0_Spec) to the new spec view (Rev1_1_Spec). (This means switching imports for client working and spec views, not released views.)

   In this example, the imports of three client views need to be switched. To do so, select the entry for Rev1_1_Spec within the Mail_Utilities subsystem and enter the Cmvc.Import command, specifying the Into_View parameter and using default values for the remaining parameters:

   ```
   Cmvc.Import (Into_View =>
        "^[Mailbox.[Rev1_Working,Rev1_0_Spec],Command_Int@.Rev1_Working]");
   ```

   Imports for all clients must be switched in a single operation; multiple client views can be specified by using a naming expression for the Into_View parameter. (The expression in the example is resolved relative to the context !Programs.Mail.Mail_Utilities.) Import operations fail if you try to switch imports one client at a time, because doing so results in an inconsistent import closure.
5. To verify that the imports were successfully changed, you can select the new spec view, Rev1_1_Spec, and enter the Cmvc.Information command. The display should list the three client views specified above as referencers for the new spec view.

Switching a client's imports creates links to the units in the newly imported spec view. Links to units in the previously imported spec view are deleted. To permit the change of links, all units that depend directly or indirectly on the imported view's units are demoted to the source state. (That is, the transitive closure of the entire spec view is demoted.) The Cmvc.Import command automatically repromotes the demoted units to the coded state when default values are used for the Remake_Demoted_Units and Goal parameters.

Method II

Method II involves generating a new spec view from the existing spec view, not from the working view. This method potentially reduces recompilation cost when compared to Method I. However, this method involves editing units in both the spec and the load view. This method is preferable:

- When changing relatively few exported units
- When the cost of recompiling the changed units is smaller than the cost of recompiling the entire spec view

For example, assume that you need to change declarations in package Symbolic_Display in the Mail_Utilities subsystem. To do so:

1. Make and test the required changes to package Symbolic_Display in Mail_Utilities.Rev1_Working.
2. If desired, make a release from Rev1_Working.

   As before, you can increment the release's level 1 number to indicate the start of a new family of compatible releases. In this case, the resulting release is called Rev1_2_1.
3. Use Cmvc.Make_Spec_View to make a copy of the latest spec view, which is Rev1_1_Spec in this example.

   Because the newly created spec view ultimately will be compatible with the release made in step 2, you should use the Level parameter to increment the level 1 number, thus creating Rev1_2_Spec. Note that, in Method II, you must increment the level number explicitly in both this and the preceding step (see "Coordinating Level Numbers in Spec and Released View Names," below).
4. In the new spec view (Rev1_2_Spec), make the same changes to Symbolic_Display'Spec as you made in the working view in step 1. If many changes must be made, you can use Library.Copy to copy the changed unit specification from the working view to the new spec view.
5. Compile the units in Rev1_2_Spec.
6. Switch the imports of all unfrozen client views from the previous spec view (Rev1_1_Spec) to the new spec view (Rev1_2_Spec). Use the Cmvc.Import command to switch imports for three client views, as shown in step 4 in Method I, above. By default, the Cmvc.Import command will perform any necessary recompilation to the coded state.

Relocation

Making a spec view from another spec view (as in step 3 of Method II) takes advantage of an Environment optimization called ***relocation***. Relocation enables the Environment to copy compiled units, preserving the internal representation of at least their installed state. When imports are switched, units in client views that were compiled against the source spec view can depend on the relocated units of the new spec view without requiring recompilation. However, when a change of any kind (even incremental) is made to a relocated unit, the changed unit loses its preserved internal representation. Consequently, any client-view units that depend on the changed unit must be recompiled when imports are switched.

In the above example, the transitive closure of Symbolic_Display'Spec must be recompiled when the imports are switched because that unit was changed in step 4. The transitive closure of Symbolic_Display'Spec consists of two units in Command_Interpreter.Rev1_Working. However, under Method I, which does not use relocation, all units in the transitive closure of the new spec view would require recompilation. That is, using Method I for this change causes the additional recompilation of units in Mailbox.Rev1_Working and Mailbox.Rev1_0_Spec.

Coordinating Level Numbers in Spec and Released View Names

You can indicate which families of releases are compatible with a given spec view by coordinating the level numbers in the view names. (See "Spec-View Names and Level Numbers," earlier in this chapter.) In the above examples, the level 1 number is incremented for each set of compatible views. However, the two methods differ with respect to how these level numbers are coordinated.

In Method I, the level number was explicitly incremented once, by specifying the Level parameter in the Cmvc.Release command. The subsequently created spec view was named automatically using the same incremented level number. Automatic coordination is possible in this case because both views were created from the same working view. In contrast, Method II required that you specify the Level parameter both when creating the release and when creating the spec view. This is necessary because each view is created from a different source view.

Level numbers for newly created releases or spec views are recorded in a predefined file within the source view. This predefined file is called State.Last_Release_Name. The next time a release or spec view is created from that source view, the file is consulted and the level numbers in it are incremented as specified. When spec views and releases are created from the same working view, both operations consult the working view's State.Last_Release_Name file, so level numbers are coordinated automatically. When spec views and releases are created from different source views, different State.Last_Release_Name files are consulted.

Note that State.Last_Release_Name files can be edited to resynchronize level numbers. The first digit in the file represents the number of levels that can be incremented; subsequent digits represent the current number at each level.

Specifying Compatible Load Views in an Activity

In each of the above examples, a new spec view is created to accommodate non-upward-compatible changes in the working view. The changed working view and subsequent releases made from it therefore are compatible with the new spec view, whereas releases made before the changes are not compatible with the new spec view. Thus, the subsystem now contains a "compatibility family" of load views for each spec view.

When imports are switched so that clients import a new spec view, the application can be executed only if the activity is also changed so that it specifies a load view that is compatible with the new spec view. For example, after the change made in Method II above, clients import Rev1_2_Spec from Mail_Utilities. Therefore, the activity entry for Mail_Utilities must be changed to specify either Rev1_Working

or Rev1_2_1, which are compatible with Rev1_2_Spec. An error message is displayed if execution is attempted while the activity entry for Mail_Utilities specifies Rev1_0_1.

In general, the activity entry for each subsystem must specify a load view that is compatible with the spec view imported from that subsystem. If you attempt to execute an application and the default activity specifies an incompatible load view for some subsystem, an error message is displayed.

Adding or Removing Units from Spec Views

Adding or removing whole units from spec views basically involves the same methods used for adding or removing individual declarations from exported units.

If the change is upward-compatible (that is, adding or deleting a unit that has no dependents), you can make the change directly to an existing spec view, as in the following steps:

1. Add or delete the unit from the load view.
2. Edit the State.Exports file in the load view. If necessary, edit any export restriction files. (This step has no direct effect on following steps; however, it is recommended to avoid confusion in the long run.)
3. Make a new release from the working view, if desired.
4. Add or delete the unit from the current spec view. (If units in the spec view need to be recompiled at this point, you are not making an upward-compatible change; see the next set of steps in this section.)
5. Refresh the imports in each client view by entering the Cmvc.Import command with the View_To_Import parameter set to the null string (""). This parameter value causes the same spec view to be reimported, thereby adding a new link or disallowing the use of an old link, as appropriate.
6. Change units in client views as necessary to take advantage of any newly added units.

If the change is not upward-compatible (that is, deleting a unit on which client units depend or adding a unit on which other spec-view units depend), you should create a new spec view to which to make the change, as in the following steps:

1. Add or delete the unit in question from the working view.
2. Edit the State.Exports file (and any export restriction files) in the load view. Note that this step may have a direct effect on step 4.
3. Make a new release from the working view, if desired.
4. Use the Cmvc.Make_Spec_View command to create a new spec view from the working view.

   Alternatively, if you want to take advantage of relocation, you can make a new spec view from the current spec view. In this case, you must edit export restriction files as necessary in the new spec view, because these files will not have been copied from the load view.

5. Add or delete the unit from the new spec view, and compile the new spec view.
6. Switch the imports of client views to the new spec view.
7. Change units in client views as necessary and recompile.

Replacing the Model in a Path

As you work within a development path, you may find that you need to change certain of the predefined library characteristics that are determined by the model. You can change these characteristics by replacing the current model with a different model that has the desired characteristics. To replace a path's model, use the Cmvc.Replace_Model command in the working view of the path; the units in the working view must be in the source state.

You can replace the model for a path to:

- Add, change, or delete links to units that are not in subsystems (for example, to Environment tools required for compilation).
- Change the number of release levels that are represented in the names of release and spec views.
- Change the library switches.

Note that you can replace a model in order to change a target key; however, the target key can only be changed to a compatible target key, as defined by the Cross-Development Facility (CDF) for that target.

Setting Up Subsystems: A Second Look

Having familiarized yourself with the concepts in the preceding chapters, you can use the following checklists to help you to partition an application into subsystems and then set up those subsystems.

Planning

Before creating subsystems:

- Examine the application design and consider the development team to determine the best partitioning. Ideally:
  - — Each subsystem is a complete, logical component of the application.
  - — Each subsystem has well-defined, restricted interfaces.
  - — The application is partitioned into a manageable number of subsystems.
  - — Each subsystem eventually contains a manageable amount of code (5–25K lines).
  - — Each subsystem has one to five developers working in it.
  - — Each subsystem interface exports private types and avoids reexporting declarations from other subsystem interfaces.

- Decide how many development paths each subsystem will contain. Plan on one path per target.
  - — If any non-R1000 target exports generics, its path must contain combined views instead of spec and load views. This is determined when new paths are created. See the chapter entitled "Using CDFs with Subsystems."
  - — Decide whether units will be joined across these paths.
- Determine the internal directory structure for views in each subsystem. This can be specified as part of the model; see "Setting Up Model Worlds," below.
- Establish application-wide naming conventions:
  - — Determine where the application will reside in the Environment. Locate or create a project library.
  - — Choose appropriate names for each subsystem.
  - — Choose the pathname prefix for the views in each path (depending on your development conventions, you may want to specify more descriptive base names than "Rev1").
  - — Set up the release structure for each subsystem. Determine whether to use release level numbers. If release level numbers are to be used, decide how many levels to use and when to increment each level. (This is specified as part of the model; see "Setting Up Model Worlds," below.)

    If desired, establish naming conventions for sets of spec and load views.
- Determine the external resources needed by units in the subsystems. Different paths may require different sets of links to units not in subsystems.
- Determine the interfaces between subsystems, mapping out the network of imports among subsystems. Which subsystems will need to import spec views from which other subsystems?
  - — If imports are hierarchic, use spec/load subsystems (the default subsystem type).
  - — If imports are circular, consider changing the application design so that imports are hierarchic; otherwise, you must use *combined subsystems* (different from combined views; see the introduction to package Cmvc).

Setting Up Model Worlds

After planning the basic subsystem elements, you can create or choose a model for the initial development path in each subsystem. (You may need additional models when you set up additional paths.) Models are Environment worlds that provide specific library characteristics for each view in the path. You can create project-specific model worlds or choose among the predefined Environment models located in !Models. Project-specific models can be created anywhere.

The model for a given development path must have:

- Links to the external resources needed by units in the path
- The library switches needed for compiling units in the path
- The desired target key

Additionally, models determine:

- The number of release levels to be used in automatically generated names of spec and released views. By default, two release levels are used. Alternatively, you can create in the model world a file called Levels that contains an integer representing the desired number of levels.
- The user-defined directory structure to be created in each view in the path. Such structure is created in addition to the Environment-defined subdirectories. If you want directories at the same level as the Units directory in each view, create appropriately named directories in the model world. If you want the Units directory in each view to contain two subdirectories, create a directory called Units in the model world, and then create the two subdirectories within that Units directory.

Creating Subsystems from the Bottom Up

After model worlds are in place, you can create the subsystems for an application. The Cmvc.Initial command creates each subsystem containing the working view for one path; other paths must be created separately. The following parameters specify some of the information that was determined during the planning phase:

- Working_View_Base_Name: Specifies the base name for the working view in the initial path.
- System_Object_Type: Determines whether circular imports are permitted within an application. If your application requires only hierarchic imports, as is recommended, all subsystems in the application should be of type Cmvc.Spec_Load_Subsystem. Circular imports are permitted only among subsystems of type Cmvc.Combined_Subsystem.
- View_To_Import: Specifies the views to be imported from other subsystems (if any) for the initial path. Using this parameter is equivalent to using the Cmvc.Import command after the subsystem is created. (However, do not use this parameter if export and import restrictions files will need to be created.)
- Create_Load_View: Specifies the type of working view to be created within a spec/load subsystem. Use the default value (true) to create load views, as in the previous chapters. Specify false if you want to create combined views. Paths for cross-development in spec/load subsystems must contain combined views when generics are exported.
- Model: Specifies the model for the initial path.

The following overview of the steps for creating the subsystems in an application starts with the lowest-level subsystems. These steps assume you are using spec and load views within spec/load subsystems:

1. Create any subsystem(s) that do not require imports.
2. Copy or create units in the working view of each subsystem, and compile the units.
3. Export units from each subsystem:
   - — Edit and commit the State.Exports file to include names of the units to be exported.
   - — If desired, create export restriction files in the Exports subdirectory.
   - — Enter the Cmvc.Make_Spec_View command to make a spec view from each working view.
   - — Compile the units in each spec view.
4. Create subsystems for the next layer in the application design. These are the subsystems that import spec views from the first set of subsystems.
5. Import spec views from the lower layer of subsystems into the working views of the next layer:
   - — If no import restrictions are needed, imports can be created by the same operation that creates the higher-level subsystems (specify the View_To-_Import parameter in the Cmvc.Initial command).
   - — If import restrictions are needed, create the subsystems first, then create the import restriction files, and finally perform the import operation using the Cmvc.Import command.
6. Copy or create units in the working view of each subsystem. Note that these units will not compile unless the imported spec views are compiled.
7. Repeat steps 3–6 to create subsequent layers of subsystems. Use the Cmvc.Import command to create imports for spec views. At the top layer, put the main program in a spec view, where test drivers can link to it.

After Subsystems Are Created

After a given subsystem is created containing an initial development path, you can:

1. Control the desired objects in each working view.
2. Make additional development paths.
3. Join or sever controlled objects across paths as desired.
4. Make subpaths within paths to accommodate multiple developers. Decide how to integrate work from each subpath.

RATIONAL

Developing Applications Using Multiple Hosts

A single application that is partitioned into subsystems can be developed on multiple host R1000s. This is useful when:

- The application is too large to be developed on a single host.
- Parts of the application are to be developed by subcontractors, typically on hosts at different sites.

This chapter covers the basic aspects of multiple-host development.

Overview of Multiple-Host Development

Assume that three host R1000s are to be used for developing the mail program described in previous chapters. This application consists of three subsystems, each of which is allocated to a specific machine for development. In particular, the Mail_Utilities subsystem is to be developed on Machine_1, the Mailbox subsystem on Machine_2, and the Command_Interpreter subsystem on Machine_3.

A copy of each subsystem resides on each host machine. However, only one copy of a given subsystem can support ongoing development. This copy is called the *primary subsystem*. The other copies, called *secondary subsystems*, are essentially local copies for execution and test. As shown in Figure 6-1, the primary subsystem for Mail_Utilities resides on Machine_1, whereas Machine_2 and Machine_3 host secondary subsystems for Mail_Utilities. Similarly, the primary subsystem for Mailbox resides on Machine_2, with secondary subsystems on each of the other hosts.

Note that a given machine typically hosts multiple primaries; if the mail program were to be developed using two machines, two primaries could reside on one of those machines with the third primary on the other machine.

Development proceeds in each primary subsystem as described in the preceding chapters. When a new release or spec view is made in a given subsystem, that release or spec view can be copied, via network or tape, to the corresponding secondary subsystems on each of the other hosts. Views in the primary subsystem on each host can then import and compile against the copied spec views in the secondary subsystems; the default activity on each host can specify the copied releases for execution.

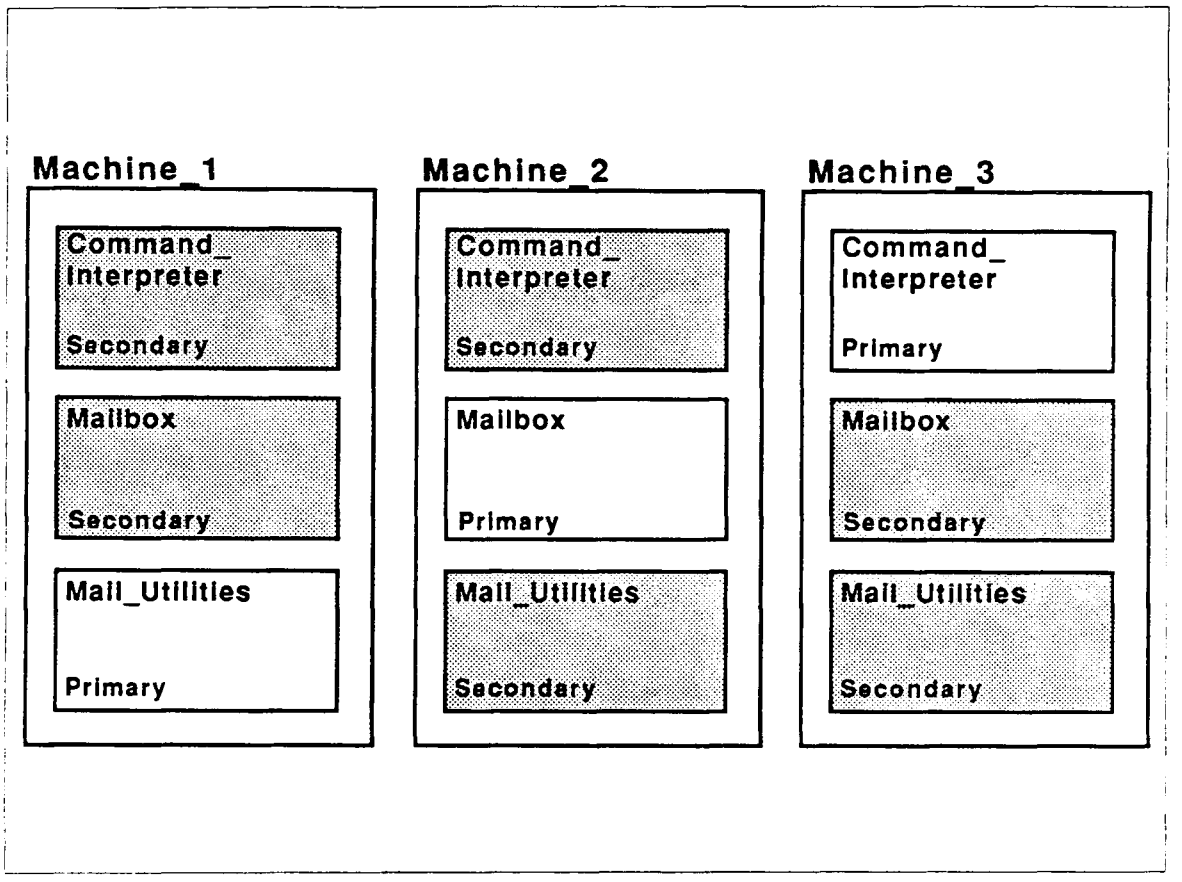


Figure 6-1. Primary and Secondary Subsystems

In R1000 development scenarios, *code views*, rather than released views, can be copied from primary to secondary subsystems. Code views are copies of views that store executable code in place of Ada units. Code views thus require the minimum amount of storage necessary to permit execution of the view. Furthermore, no recompilation is required when copying code views between machines.

Because code views do not contain Ada units, full source-level debugging is not available. Furthermore, the program source cannot be browsed directly as in released views, although source representation from the CMVC database can be viewed through configuration and generation images (see the introduction to package Cmvc). Note that a code view can be browsed in a secondary subsystem only if the following objects have been copied from the primary subsystem: the configuration object for the code view and the CMVC database. (See also "More about Copying between Subsystems," below.)

Setting Up Primary and Secondary Subsystems

To set up the mail application as described above:

1. Locate or create the desired application libraries on each host. For example, you can create a library called !Programs.Mail on each R1000. Be sure that the access-control list for these application libraries will permit you to create and copy subsystems, views, links, switches, and the like.
2. Use the Cmvc.Initial command to create the each primary subsystem on the appropriate machine. That is, create Mail_Utilities on Machine_1, Mailbox on Machine_2, and Command_Interpreter on Machine_3. Note that all subsystems created by Cmvc.Initial are primary.
3. Using the Archive.Copy command or the Archive.Save and Archive.Restore commands, copy the model world for each primary subsystem onto the other machines. This model is used when the secondary subsystems are created.
4. Using the Archive.Copy command or the Archive.Save and Archive.Restore commands, create secondary subsystems on the appropriate machines. By default, the Archive.Copy and Archive.Restore commands create secondary copies from primary subsystems.

   For example, you can use the Archive.Copy command to copy !!Machine_1-!Programs.Mail.Mail_Utilities onto Machine_2 and Machine_3, thereby creating secondary subsystems on those machines.

In this example, the secondary subsystems are created from a primary subsystem that contains only an empty working view. Note that secondary subsystems also can be created from a primary subsystem that already contains spec views and releases, and these are copied in the process of creating the secondaries.

Copying Views among Hosts

Development now proceeds within the working view of each of the primary subsystems created above. When appropriate, a spec view is created to express the exports from the primary subsystem Mail_Utilities on Machine_1. Compiled releases also can be made in Mail_Utilities because it is the bottommost layer in the application (it has no imports). However, compiled releases cannot be made in either of the other primary subsystems until the working views in these subsystems import the spec view from Mail_Utilities. To be imported into subsystems on other hosts, this spec view must be copied into the secondary Mail_Utilities subsystems on those hosts.

To copy a view from a primary subsystem into a secondary subsystem:

1. Within the primary subsystem, make sure that the units in the view have been promoted to the coded state. This step is necessary because it ensures that certain compilation information is recorded in the primary subsystem.
2. Use the Archive.Copy command (or Archive.Save and Archive.Restore) to copy the desired view into each secondary subsystem.

For example, in the context !!Machine_1!Programs.Mail.Mail_Utilities, the following command copies Rev1_0_Spec into the corresponding secondary subsystem on Machine_2. Note that specifying Promote for the Options parameter causes the view to be recompiled in the secondary subsystem:

```
Archive.Copy (Objects => "Rev1_Ø_Spec",
              Use_Prefix => "!!Machine_2",
              Options => "Promote");
```

The working views in the primary subsystems on Machine_2 and Machine_3 can import and compile against the spec view after it has been copied into the secondary subsystems. After all the necessary spec views have been created, copied into secondary subsystems, and imported, the entire application can be compiled.

Copying Views and Subsystem Identification Numbers

The first time units are compiled in a primary subsystem, that subsystem is assigned a unique *subsystem identification number.* Furthermore, the first time views are copied from a primary subsystem into an empty secondary subsystem, that secondary inherits the primary subsystem's identification number. A shared subsystem identification number thus defines a family of associated subsystems within which views and associated compilation information can be copied.

Because the association between primary and secondary subsystems is not formed until the first time a compiled view is copied, you must be careful to copy the first views into the correct secondary subsystems. Failure to do so will associate the wrong primary and secondary subsystems; to recover, you must destroy each incorrectly associated secondary subsystem and recreate it.

Note that when a secondary subsystem is created from a primary that already contains compiled units, the secondary subsystem is created with the primary subsystem's identification number. In this case, there is no danger of associating the wrong subsystems.

Copying Releases and Code Views

As development proceeds in each primary subsystem, releases (or code views) can be made from the compiled working views in all of the primary subsystems. However, the application cannot be executed on any machine until releases and code views are copied from the primary subsystems into the corresponding secondary systems.

You can copy releases and code views using commands from package Archive, as in the steps above. Code views are copied in the coded state. However, when released views are copied, all compiled units in the resulting copies require recompilation. Therefore, releases should be copied using the Promote option, which unfreezes, recompiles, and refreezes the copied release in the secondary subsystem.

When a given host has a complete set of releases or code views, activities can be set up on that host to specify these releases for execution.

The Compatibility Database

Spec views and releases can be copied into secondary subsystems in any order relative to each other, as long as they are both compiled in the primary subsystem. This is necessary because each primary subsystem maintains a *compatibility database* (CDB) which collects compilation information about the views compiled in the subsystem.

The CDB maintains compilation consistency between the load views and spec views, ensuring that, for each usage of a given declaration in an exported spec view, the correct declarations are executed in the compiled load view. More specifically, the CDB assigns a unique label, or *declaration number*, to every declaration in every unit in a spec view. These same declaration numbers represent the corresponding declarations in load-view units. When a unit in a client view compiles against a spec view, references to exported declarations are represented as calls to unique declaration numbers, which are matched with load-view declarations for execution.

CDBs are used in single-host development (where every subsystem is a primary subsystem) to enable compatible load views and code views to execute in place of imported spec views. In multiple-host development, the CDB additionally serves to ensure consistent compilation across machines. That is, when views are copied using commands from package Archive, the CDB from the primary subsystem is copied automatically into the secondary subsystem. When compilation on the secondary host involves the copied views, the copied CDB is consulted for the appropriate declaration numbers. The copied CDB is especially important when code views are copied into a secondary subsystem. The CDB ensures that spec views on the secondary host are compiled consistently with the precompiled code views.

A CDB is identified by the subsystem identification number of the primary subsystem that contains it. A CDB can be copied only into secondary subsystems sharing that identification number. Whereas Archive.Copy allows you to copy views from a primary subsystem into a secondary subsystem with a different identification number, the CDB is not copied along with it. Failure to copy the CDB is reported as a warning in the Archive.Copy log. The subsystem identification number for a subsystem can be displayed using the Cmvc_Maintenance.Display_Cdb command.

Propagating Changes across Hosts

Ongoing development is permitted in a primary subsystem because its CDB can be updated to reflect new or changed declarations. In contrast, secondary subsystems contain read-only copies of the primary subsystem's CDB. Therefore, in a secondary subsystem, declarations cannot be changed or added to unit specifications in any kind of view. Such changes must be made in the primary subsystem and propagated to the secondary subsystems.

The following methods are most appropriate for propagating changes among spec views or combined views (for information on combined views, see the chapter entitled "Using CDFs with Subsystems"). When you change a working load view in a primary subsystem, you should propagate changes in any of the following ways:

- Make a new release of the load view on the primary subsystem and copy it into the secondary subsystem. Note that when releases are copied, all units in the resulting copies require recompilation; using the Promote option in the Archive commands automatically unfreezes, recompiles, and refreezes copied releases in the secondary subsystem.
- Make a code view from the load view on the primary subsystem and copy it into the secondary subsystem. Copying a code view is faster than copying a released view because the code view does not need to be recompiled on the secondary subsystem. Note, however, that source-level debugging of the code contained in the code view is not possible.
- Move the new and changed units in the load view on the primary subsystem to a working view on the secondary subsystem. This provides source-level debugging for the code in the view and minimizes the recompilation that results from the copy operation because only the changed units and the units that depend on them need to be recompiled. Note that this alternative is the recommended approach for moving changes into target paths in cross-development (see "Method III: For Development on Multiple Hosts," in the chapter entitled "Using CDFs with Subsystems").

It is important to be aware that units in a secondary subsystem are not necessarily frozen, even though the CDB is frozen. Leaving units unfrozen in a secondary subsystem allows you to use incremental operations according to Method I in the following section. However, it is possible to freeze all units in secondary subsystems to prevent any kind of change from being made there. In this case, you must use Method II exclusively to propagate changes.

Method I: Propagating Incremental Changes

Incremental changes can be made to units in the primary subsystem and then propagated to the secondary subsystem as follows:

1. Make the incremental change to the appropriate units in the primary subsystem—for example, incrementally adding a declaration in a unit in a spec view. When you promote the change to the coded state, compilation information is recorded in the CDB.
2. From the secondary subsystem, enter the Cmvc_Maintenance.Update_Cdb command to copy the CDB from the primary subsystem into the secondary subsystem.

   Alternatively, if the two machines are not on the same network, you can use the Archive.Save and Archive.Restore commands with the Cdb option (see "More about Copying between Subsystems," below).
3. Make the same incremental change to the appropriate units in the spec view in the secondary subsystem. The copied CDB is consulted when the change is promoted. Note that the insertion window cannot be promoted unless the CDB has been copied.

For completeness in this example, note that you also should change the working load view on the primary subsystem so that it is compatible with the changed spec view. Then you can make a new release from the working load view and use Archive.Copy to copy the new release into the secondary subsystem.

Method II: Propagating Changed Units or Views

Changes that are not made incrementally can be propagated as follows:

1. Make the desired changes to the appropriate units in the primary subsystem.
2. Promote the changed units to the coded state. This operation records compilation information in the CDB.
3. Use the Archive.Copy command to copy either the changed units or the view containing the changed units into the secondary subsystem. The CDB is copied automatically by this operation.

For example, assume that you changed units in Rev1_0_Spec in the primary subsystem Mail_Utilities. The following command overwrites the corresponding units in Rev1_0_Spec in the secondary subsystem on Machine_2:

```
Archive.Copy (Objects => "Rev1_0_Spec",
              Use_Prefix => "!!Machine_2",
              Options => "Changed_Objects,Replace,Remake");
```

The values for the Options parameter have the following effects:

- Changed_Objects: Causes new and modified objects to be copied.
- Replace: Permits units with dependents to be demoted and overwritten. Dependent units also are demoted.
- Remake: Repromotes all units that were demoted by the Replace option.

If necessary, delete any units from the spec view in the secondary subsystem that had been deleted from the spec view in the primary subsystem. Following are special considerations when units are controlled in the secondary subsystem:

- Controlled units in the secondary subsystem must be checked out before the Archive.Copy command is entered. Otherwise, the changed units cannot be overwritten with the updated units.
- The Archive.Copy command does not automatically make controlled any new units that it copies. New units will have to be made controlled on the secondary subsystem as a separate step.

Note that history information maintained by CMVC is valid only in the primary subsystem.

Moving a Primary to Another Host

Occasionally it is necessary to move the primary development of a subsystem to a different host R1000. For example, assume that, because of machine loads, the primary subsystem Mail_Utilities is to be moved to Machine_2 and the primary subsystem Mailbox is to be moved to Machine_1.

To rehost a primary subsystem:

1. Find or create an associated secondary subsystem on the desired host. In this example, a secondary subsystem for Mail_Utilities already exists on Machine_2.
2. Enter the Cmvc_Maintenance.Update_Cdb command to copy the CDB from the primary subsystem into the secondary subsystem.
3. Convert the secondary subsystem into a primary subsystem using the Cmvc_Maintenance.Make_Primary command with the Moving_Primary parameter set to true. This causes the converted subsystem to retain its original subsystem identification number and thus its previous association with other subsystems.
4. Convert the original primary subsystem to a secondary subsystem with the Cmvc_Maintenance.Make_Secondary command. This step must be done to prevent corruption of the CDB.

   CAUTION *If the CDB is corrupted, it must be destroyed in all associated primaries and secondaries, which demotes all views in those subsystems to the source state and destroys all code views in those subsystems.*

It is crucial that the subsystem identification number be preserved when converting the secondary subsystem to a primary subsystem. (This is achieved by setting the Moving_Primary parameter to true when entering the Cmvc_Maintenance.Make_Primary command.) If the subsystem identification number is allowed to change (by leaving the Moving_Primary parameter false), then the new primary subsystem is effectively severed from its former associates. The changed subsystem identification number means that the CDB cannot be copied between the new primary and what were intended to be its secondary subsystems.

More on the CDB

A CDB exists in a subsystem only after units have been promoted to the installed or coded state in that subsystem. Accordingly, the CDB contains declaration numbers only for declarations that have been compiled. The CDB contains a reference to all declarations that were ever compiled in the subsystem, even those that have been deleted.

The CDB for a subsystem consists of objects in the State.Compatibility directory of that subsystem. The CDB is corrupted if any object in this directory is deleted. The CDB is also corrupted if compilation takes place in multiple primary subsystems that share the same subsystem identification number (see "Moving a Primary to Another Host," above).

A corrupted CDB may need to be destroyed using the Cmvc_Maintenance.Destroy_Cdb command (consult your Rational technical representative). If you destroy a CDB in a primary subsystem, you must also destroy the CDB in all the associated secondary subsystems. Destroying a subsystem's CDB:

- Demotes all views to the source state in that subsystem
- Destroys all code views in that subsystem

In a primary subsystem, the CDB is recreated automatically the next time you compile units in the subsystem. However, the CDB is recreated with a new subsystem identification number, which severs the subsystem from any associated subsystems.

More about Copying between Subsystems

To copy a view or an object from a primary subsystem into a secondary subsystem, you can:

- Use the Archive.Save and Archive.Restore commands to copy via tape.
- Use the Archive.Copy command to copy via the network.

For complete information about commands in package Archive, see Library Management (LM).

With respect to copying among subsystems, the Archive.Save, Archive.Restore, and Archive.Copy commands have the same default behavior:

- The CDB is copied automatically whenever a subsystem, a view, or individual units in a view are specified.
- The CMVC database is not copied automatically when a view is copied, so objects that are controlled in the source are not controlled in the copy. To cause copies to be controlled, you must copy the CMVC database explicitly by naming it in the command.
- When subsystems are copied, secondary subsystems are created unless otherwise specified.
- The corresponding configuration object is created automatically for each specified view. However, the configuration object is of no use unless the CMVC database is copied.

In the Archive.Restore and Archive.Copy commands, an Options parameter accepts the following values pertaining to subsystems:

- Cdb: Causes only the CDB for the specified subsystem to be moved. Thus, the following command is equivalent to using the Cmvc_Maintenance.Update_Cdb command with the appropriate subsystems specified:

```
Archive.Copy (Objects => "Mail_Utilities",
              Options => "Cdb");
```

- Ignore_Cdb: Causes the specified objects, views, or subsystems to be copied without copying the CDB. When a subsystem is specified, this option creates a copy that is not associated with the source subsystem. Note that the CDB is always ignored when you copy views or objects into unrelated subsystems, in which case this option merely saves time.
- Primary: Causes the specified subsystem to be copied as a primary (with read/write access to CDB). Otherwise, a secondary is created with a read-only CDB.

  Specifying Primary creates a primary copy that has the same subsystem identification number and is therefore equivalent to using the Cmvc_Maintenance.Make_Primary command with the Moving_Primary parameter set to true. Note that when you have two primary subsystems with the same subsystem identification number, you must make one of them a secondary; otherwise, the CDB will be corrupted. If the CDB is corrupted, it must be destroyed in all associated primaries and secondaries, which demotes all views in those subsystems to the source state and destroys all code views in those subsystems.
- Revert_Cdb: Allows a less recently updated version of a CDB to be restored over a more recently updated version. This may be useful when restoring a subsystem from tape over a subsystem that contains a corrupted CDB (for example, if any of the objects in the *subsystem*.State.Compatibility directory are deleted). However, apart from correcting overt corruption of the CDB, there is no reason to use this option, because all versions of a CDB are consistent.

Using CDFs with Subsystems

With Rational's family of Cross-Development Facility (CDF) products, you can develop applications on an R1000 for execution on specific target processors. As described in the CDF user's manual for each target, the basic cross-development scenario is to:

1. Develop the Ada units in the application on the R1000, where you can take advantage of the Environment for language-specific editing, compilation management, and functional execution testing.
2. Use the CDF on the R1000 to cross-compile, cross-assemble, and link the application into an executable module for the desired target.
3. Download and execute the executable module on the target.
4. From the R1000, debug the application as it runs on the target.

Applications intended for non-R1000 targets can be developed either in worlds or in subsystems. This chapter will cover the basic aspects of using subsystems as the environment in which cross-development takes place.

Overview of Cross-Development in Subsystems

Using subsystems for cross-development allows you take advantage of *development paths*, which are working views for developing variant implementations. Paths support the development of code that is common to each variant as well as code that is specific to individual variants. More specifically:

- Controlled units in a given path can be joined to their counterparts in other paths, so that changes made in one path can be propagated to the other paths through the Cmvc.Check_Out and Cmvc.Accept_Changes commands.
- Controlled units in a given path can be severed (or left unjoined), so that they can be checked out and modified independently, without propagating changes.

Each path contains a separate series of releases that are made from its working view. (For more information on paths, see "Setting Up Development Paths" in the chapter entitled "Coordinating Development in a Subsystem.")

Applications developed for non-R1000 targets typically are partitioned into subsystems that contain one path for R1000 development and another path for target development. Ada units containing target-independent code are joined across these paths

so that changes can be propagated automatically. Figure 7-1 shows two subsystems, each containing paths for two targets—namely, the R1000 and a Motorola® MC68020 microprocessor. These paths contain load views whose exports are expressed as spec views; note that views in each path import views from the corresponding paths in the other subsystem.

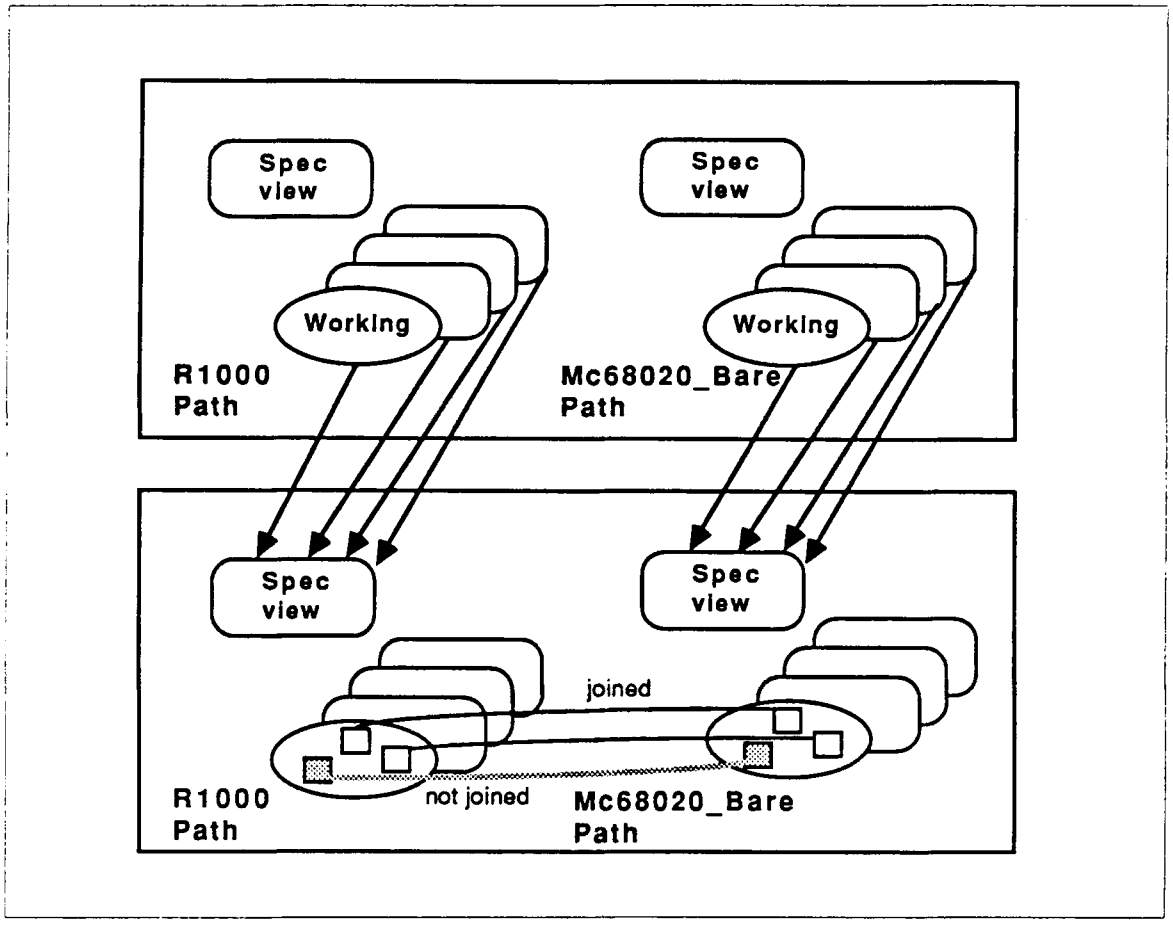


Figure 7-1. Paths for Two Targets

Typically, the target-independent units in the application are developed and tested as much as possible in the R1000 path. When appropriate, changes are propagated to the working view of the target path, where target-specific code is also being developed. Units in the target-specific path are cross-compiled and cross-linked using the CDF for that target. The resulting executable module is downloaded to the target processor for execution. As it executes on the target, the application can be debugged using the cross-debugger invoked from the target-specific path.

Target Keys

Each path is identified for a particular target by a *target key.* The target key defines the compilation mode within that path, enabling Ada units to be compiled for the target named by the key. That is, the CDF for a target such as the MC68020 microprocessor is invoked only when you compile units in a working view that has the Mc68020_Bare target key. For targets other than the R1000, the target key for a path is displayed in the window banner for any view in that path. (The R1000 target key designation is not displayed.)

The target key for a path is supplied by the model world used to create the path. Thus, a path for R1000 development is created with a model such as !Model.R1000, which has an R1000 target key. Predefined model worlds exist for each target; these are located in !Model. See the appropriate CDF user's manual for more information about target-specific model worlds.

Differences and Restrictions

For the most part, subsystem and CMVC usage is the same for cross-development as it is for R1000 target development, which is presented in the preceding chapters. However, there are some differences and restrictions in the areas listed below.

Kinds of Views in Target Paths

Typically, the path for each target contains load views whose exports are expressed as spec views. However, in the following two cases, the target path must be created using *combined views* instead of spec and load views:

- If generics are to be exported
- If inlined subprograms are to be exported

Combined views are discussed in "Using Combined Views," below. Note that spec and load views should be used when possible because combined views entail special release considerations. Combined views do not have the advantages of spec/load views with respect to minimized recompilation requirements and flexible testing combinations.

Closed Private Parts

In spec views that have target key R1000, private parts are closed, which means that they are not compiled along with the rest of the spec view. Closed private parts are useful because they can be changed without requiring recompilation of clients. (See "Changing Private Parts in Exported Units" in the chapter entitled "Developing Applications Using Multiple Subsystems.")

In contrast, private parts are open in spec views that have non-R1000 target keys. This is true because cross-compilers are limited by the target architecture for which they must generate code. When private parts are open:

- Changes to private parts must be made not only in the working view but also in the corresponding spec view; otherwise, the spec and load views are rendered incompatible. Spec and load views must be made compatible before you can link an executable module.
- Pragma Private_Eyes_Only has no effect.

See "More on Closed Private Parts" in the chapter entitled "Developing Applications Using Multiple Subsystems."

Code Views

Currently the capability for generating code views is not available in paths for targets other than R1000.

Note, however, that the executable module for a main program can be copied without copying the program source. Furthermore, host/target machine-level debugging is available for executable modules through the CDF for each target.

Execution and Activities

When a path contains spec and load views, an activity is required for execution, regardless of the path's target key. However, exactly when the activity is consulted depends on whether pragma Main is used in the application's main program. Applications that are to be cross-compiled for execution on a non-R1000 target must have main programs containing pragma Main; applications that are to be compiled for execution on the R1000 may, but need not, have main programs containing pragma Main.

The sequence of events for execution on the R1000 (without pragma Main) follows:

1. Units are promoted to the coded state.
2. The application is executed; the activity is consulted during this step.

The sequence of events for a non-R1000 target (with pragma Main) follows:

1. Units are promoted to the coded state and linked using the CDF. Because an executable module is made as a result of cross-compiling and cross-linking, the activity is consulted during this step.
2. The application is downloaded to the target processor and executed.

When pragma Main is not used (the first sequence), changing the activity between steps 1 and 2 affects execution in step 2. When pragma Main is used (the second sequence), changing the activity between steps 1 and 2 has no effect on execution in step 2.

Note that pragma Main causes an executable module to be created containing code for all units in the application. This executable module sets up an implicit dependency between the main program's code and the code of the other units executed in the application—if any unit in the application is subsequently demoted below the coded state, then the main program is demoted automatically to the installed

state. Hence the main program should never be released after being executed with units in working views, because demoting units in the working views would require demoting the executable module in the frozen release.

Setting Up Subsystems for Cross-Development

If the application to be developed has not yet been partitioned into subsystems, start with step 1. If the application is already partitioned into subsystems whose working views have the R1000 target key, you can start with step 2.

1. Use the Cmvc.Initial command to create the desired subsystems, one for each logical program component. Specify models as appropriate so that the initial working view in each subsystem has an R1000 target key. This initial working view defines an R1000 path.
2. Develop and test Ada units as much as possible in the R1000 path. Use the Cmvc.Make_Controlled command to put these units under CMVC.
3. When desired, create a target path from the R1000 path in each subsystem. To do this, select the working view of the R1000 path and enter the Cmvc.Make_Path command, specifying at least the following parameters:
   - New_Path_Name: Specify the name prefix for the new path, according to your project's naming conventions. For example, the path name can indicate the path's target.
   - Model: Specify a model that has the appropriate target key. Predefined model worlds exist in !Model and are described in the CDF user's manual for each target. Predefined models can be used to create your own project-specific models.
   - Create_Load_View: Specify true if a working load view is to be created.
   - Create_Combined_View: Specify true if a working combined view is to be created. You should use load views if possible; however, you must use combined views if generics or inlined subprograms are exported.
   - Join_Paths: Specify true if all or most of the controlled units are to be shared (joined) between paths; specify false if none or few of the units are to be joined.
4. Use the Cmvc.Sever or Cmvc.Join commands as necessary so that all target-independent units are joined between the two paths and target-specific units are severed.

The resulting target-specific path is a working view that contains a copy of the units from the R1000 path. Development can now continue in either path, as appropriate. Changes can be accepted between paths for the joined units only.

Using Combined Views

Combined views are a third kind of view that must be created in spec/load subsystems. When a path is created using combined views, that path contains a working combined view from which releases can be made.

Combined views combine characteristics of spec and load views:

- Like load views, combined views are used for execution because each contains a complete subsystem implementation, including specifications and bodies for all units.
- Like spec views, combined views can be imported because they contain the specifications of exported units; in fact, unless export restrictions are imposed, every unit specification in a combined view is exported.

Combined views are similar in content to load views, yet they can be imported directly by client views.

When to Use Combined Views

A path for a non-R1000 target must contain combined views when:

- Generics are to be exported from the path.
- Inlined subprograms (units containing pragma Inline) are to be exported from the path.

This is necessary because target compilation requires the bodies of generics and inlined subprograms to be in the same view as their specifications.

A target path should contain spec and load views whenever possible because combined views do not have the recompilation and testing advantages that are gained by separating exports and implementation.

Note that a single subsystem can contain paths containing spec/load views and paths containing combined views. Furthermore, within a single application, the path for a given target may contain load views in some subsystems and combined views in others.

Defining Exports Using Export Restrictions

Because combined views are imported directly, every unit specification in an imported combined view is by default available for client-view units to reference. To export a smaller subset of unit specifications from a combined view, you can use export and import restrictions. More specifically, you must create an export restriction file in the Exports directory of the combined view, and then create the corresponding import restriction file in the Imports directory of the client view. (See "Imposing Further Import and Export Controls" in the chapter entitled "Developing Applications Using Multiple Subsystems.")

Importing among Target Views

You can set up imports among target-specific views using the Cmvc.Import command. Any kind of view (spec, load, or combined) can import combined views; furthermore, combined views can import either spec views or other combined views. Note that a client view can import only a view that has the same target key.

Figure 7-2 shows a sample network of imports among working views in Mc68020-_Bare target paths in three subsystems. Note that the target path in the topmost subsystem contains load views, whereas the target path in the lower subsystems contains combined views.

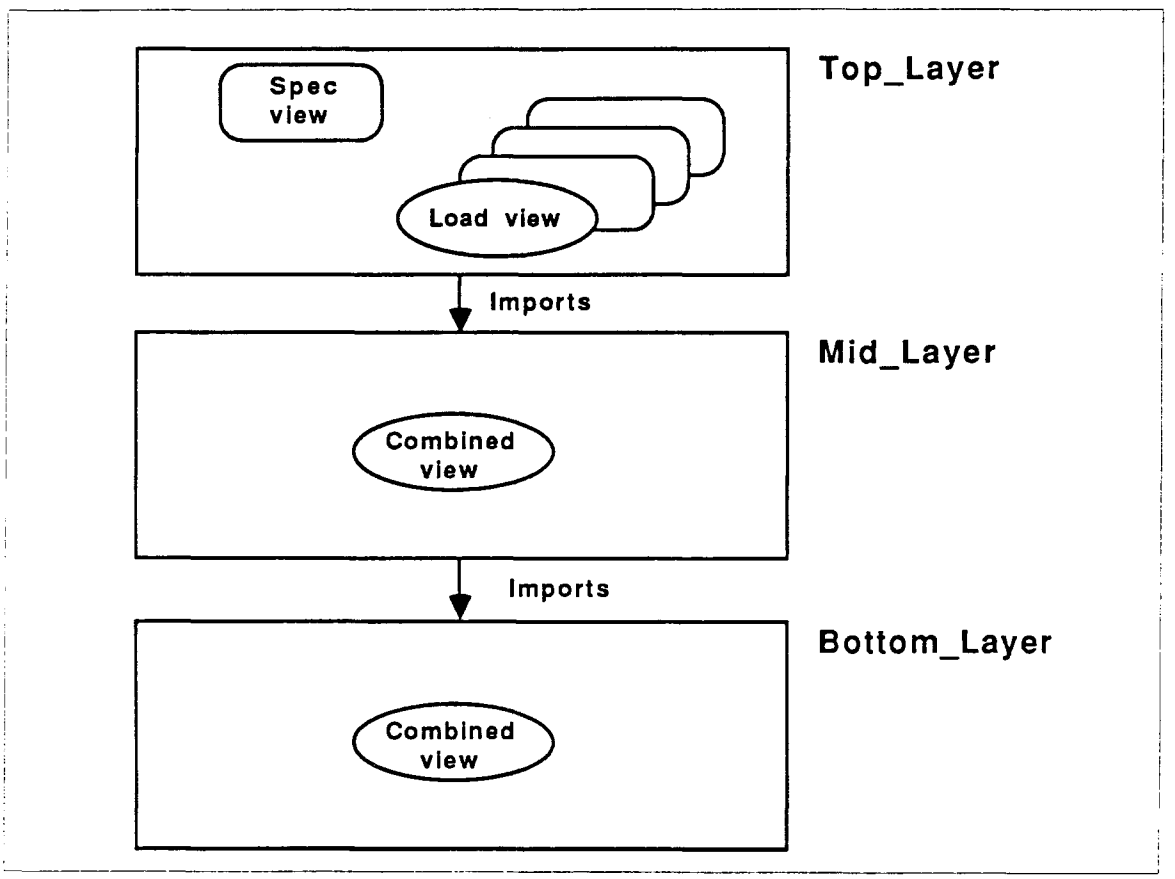


Figure 7-2. An Import Network That Contains Load and Combined Views

Because these imports are among views in spec/load subsystems, this import network must be hierarchic (that is, no view can directly or indirectly import itself). However, import relationships can be circular among combined views that are in *combined subsystems* (see the introduction to package Cmvc).

Consequences of Using Combined Views

The fact that combined views do not separate exports from implementation has consequences for compilation and for execution. As a result, using combined views has implications for making, testing, and releasing changes.

Consequences for Compilation

Because combined views are imported, units in client views depend directly on units in imported combined views. Therefore, compilation obsolescence in an imported combined view can propagate to its client views. Consequently, demoting and recompiling unit specifications in an imported combined view must be coordinated with the development of the client views. In contrast, units in load views can be recompiled with no direct impact on clients (unless client units contain pragma Main) because the clients are compiled against spec views; new spec views can be imported at the client's convenience.

As a further consequence of direct dependencies, no released view should ever import a working combined view, because frozen client releases cannot be recompiled to accommodate changes made in working combined views. An imported combined view can be thought of as an extension of its clients; if a client is a frozen release, any combined views it imports should also be frozen releases.

Consequences for Execution

Because combined views contain both exports and implementation, imported combined views are used not only for compilation but also for building executable modules. Consequently, when a combined view is imported from a subsystem, no activity entry is needed for that subsystem in order to specify an implementation for execution.

Testing in paths that contain combined views is less flexible than in paths that contain spec/load views. When spec views are imported, alternative implementations can be tested by changing activity entries; no recompilation is required. In contrast, when combined views are imported, executing an alternative implementation means importing a different combined view, and changing imports entails recompilation of client views.

Note, however, that client views can change imports only if they are working views, because frozen released views cannot be recompiled. Therefore, to test a new release of a combined view against existing released client views, you must release not only the combined view you want to test but also all clients of that view. In effect, separate sets of releases must be made for each desired combination of combined views.

Methods for Using Combined Views

Because of the direct dependencies on units in imported combined views, development methods for combined view paths differ from development methods for spec/load view paths. Two general methods are presented below. The first method is workable for a small number of subsystems (two or three) in which changes to

exported units are made fairly infrequently (for example, once a week). The second method is preferred for larger applications or when changes to exported units occur more frequently.

Method I: For Smaller Applications

Method I is similar to development using spec and load views, in that releases are used to facilitate parallel development in the target path. However, because Method I involves the highest recompilation cost when changes are made, this method is suitable only for smaller applications (two or three subsystems) in which changes are made fairly infrequently (approximately once a week).

Under Method I, development proceeds in parallel in the working views of the target paths. To facilitate this, working views from the higher layers import releases made from the lower layers. In this method, released combined views play a role similar to that of spec views, providing stable unit specifications against which views in higher layers can compile.

For example, consider the application shown in Figure 7-2 above. This application contains three subsystems: Top_Layer, Mid_Layer, and Bottom_Layer. Each subsystem contains an R1000 path and a path for an MC68020 target; the target path in Top_Layer contains load views and the target paths in Mid_Layer and Bottom_Layer contain combined views. Finally:

- Top_Layer imports Mid_Layer.
- Mid_Layer imports Bottom_Layer.

Under Method I, development proceeds as in the following steps (these steps are represented by number in Figure 7-3):

1. Develop the Ada units in the working combined view in Bottom_Layer until these units are ready for use by the higher-level subsystems. Then make a release of the working combined view in Bottom_Layer.
2. make the working view in Mid_Layer import the released combined view from Bottom_Layer. Compilation and testing can now proceed in Mid_Layer.
3. After appropriate development, make a release from the working combined view in Mid_Layer. Note that, by default, the new release inherits the working view's imports, so that the release in Mid_Layer imports the release in Bottom_Layer.
4. Make the working view in Top_Layer import the released combined view in Mid_Layer. Compilation and testing can now proceed in Top_Layer.
5. If a complete set of releases is desired, you can now make a release from the working view in Top_Layer. A release in Top_Layer is not required for purposes of parallel development but is useful for complete system testing or for identifying a major system release. Note that, by default, this release imports the released view from Mid_Layer.

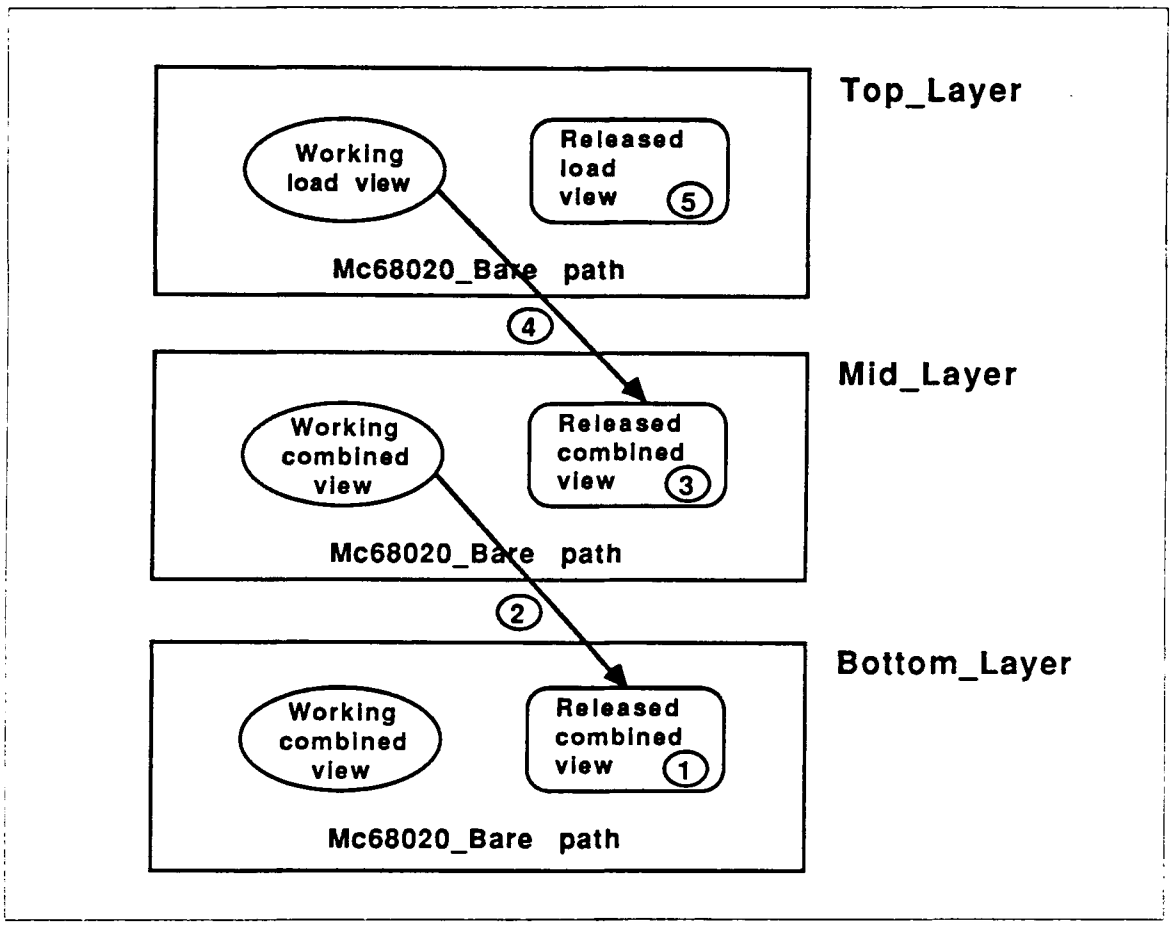


Figure 7-3. Method I Development Steps

The steps shown above serve as a "bottom-up" method for making releases serially from combined views. Making releases from the bottom up prevents releases from depending on working combined views.

When a new release is made from a lower-level combined view, these steps must be repeated to take advantage of the new release. That is, all clients need to be re-released to execute a new lower-level release.

For example, as shown in Figure 7-4, a new release is made in Bottom_Layer (1). This new release must be imported (2) by the working view in Mid_Layer. At this point, however, Mid_Layer contains the only working view that can test against the new release, because the working view in Top_Layer still indirectly imports the original release in Bottom_Layer.

To solve this, a new release (3) must be made in Mid_Layer, which imports the new release in Bottom_Layer. Finally, the new Mid_Layer release must be imported by the working view in Top_Layer (4). Only then can the working view in Top_Layer test against the most current release in Bottom_Layer.

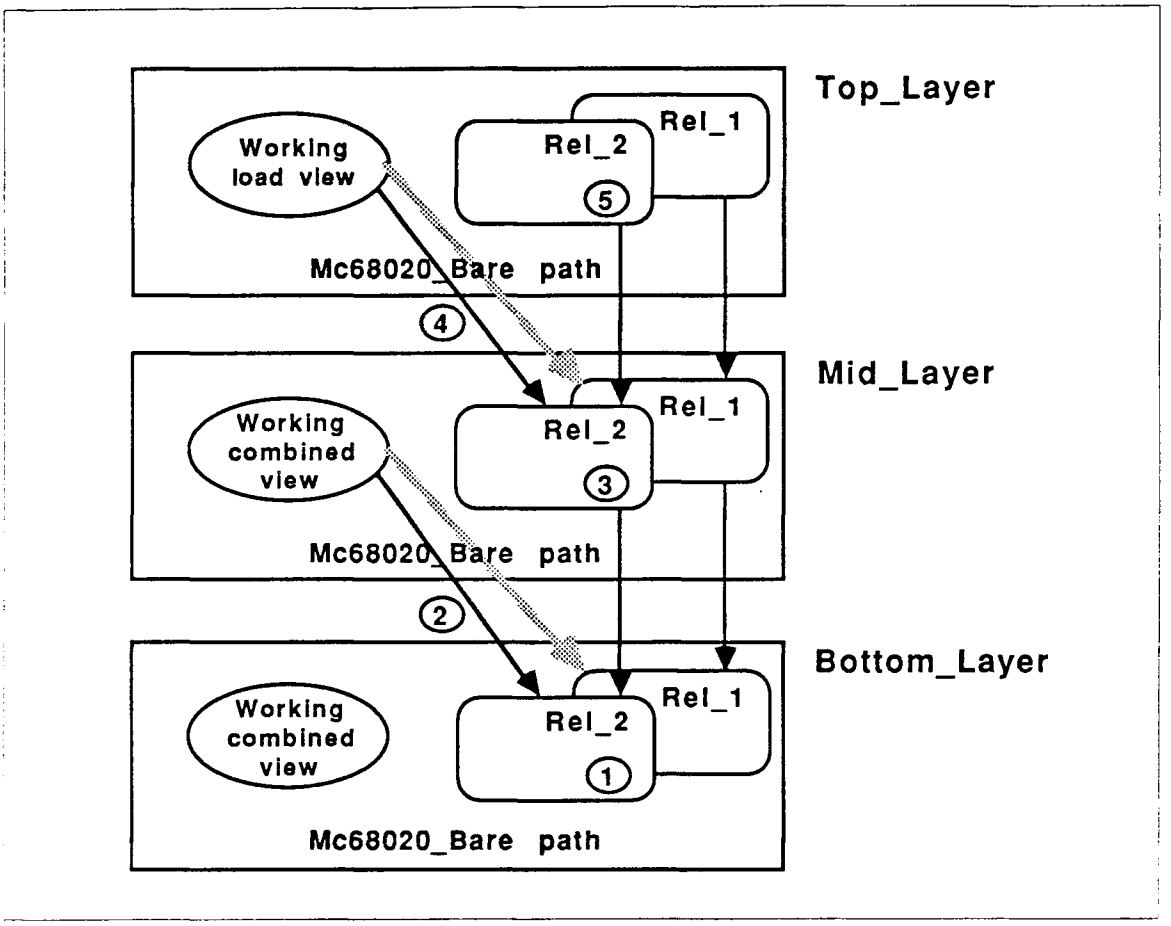


Figure 7-4. Taking Advantage of a New Release in Bottom_Layer

Method II: For Larger Applications

Method II is preferred for larger applications or when changes to exported units occur more frequently. In this method, the bulk of development and functional testing takes place in an R1000 path, where spec and load views facilitate parallel development. The target path is reserved for final target integration and minor changes. In Method II, released combined views do not play a role in facilitating parallel development; instead, releases are made from the target path only at major release points.

Consider once again the application containing the subsystems Top_Layer, Mid_Layer, and Bottom_Layer. Assume that an initial R1000 version has been implemented and tested in an R1000 path in each subsystem. Each R1000 path contains spec and load views; views from Top_Layer import the spec view from Mid_Layer and views from Mid_Layer import the spec view from Bottom_Layer. Assume also that the target paths in Mid_Layer and Bottom_Layer must contain combined views because of exported generics or inlining, whereas the target path in Top_Layer can contain load views.

Under Method II, development proceeds according to the following general guidelines (steps 1, 3, and 5 are numbered in Figure 7-5):

1. After ensuring that units in the R1000 path are controlled, set up the target paths from the bottom up, joining each with the R1000 path:

   a. From the working view in the R1000 path in Bottom_Layer, create a target path containing a working combined view.

   b. From the working view in the R1000 path in Mid_Layer, create a target path that contains a working combined view. The newly created view should import the working combined view in Bottom_Layer.

   c. From the working view in the R1000 path in Top_Layer, create a target path that contains a working load view. The newly created view should import the working combined view in Mid_Layer.

2. In the R1000 paths, continue development or maintenance as necessary. Perform functional testing by setting up an activity and executing the application on the R1000.

3. When ready for final target integration, propagate changes between R1000 and target paths in each subsystem, from the bottom up. That is, operations such as the following should be done first for Bottom_Layer, then for Mid_Layer, then for Top_Layer:

   - Use the Cmvc.Accept_Changes command to update the working view in the target path from the working view in the R1000 path. (Under this method, most if not all units should be controlled and joined between the R1000 and target paths.)

     The Allow_Demotion parameter must be set to true so that units with compilation dependencies can be updated. As a result, accepting changes into combined views demotes the updated units and their dependents in client views. However, by default, all affected units are recompiled automatically by the Cmvc.Accept_Changes command; in the target path, they are recompiled using the appropriate cross-compiler.

   - If necessary, delete any units from the target path that had been deleted from the R1000 path. (Note that any new controlled units in the R1000 path are copied automatically into the target path by the Cmvc.Accept_Changes command; however, no units are deleted by this command.)

   - If necessary, refresh the imports of client views to take advantage of added or deleted units.

4. At major release points, use the Cmvc.Release command to make a set of releases from the target paths of all subsystems. (Note that the From_Working_View parameter in the Cmvc.Release command accepts a list of views to release at the same time.)

   When you release a set of combined views among which import relations hold, the imports are adjusted automatically so that the new releases reference each other as appropriate, instead of referencing working views.

   Note that if source-level debugging is not required for the released application, you can save time and space by copying the executable module into an appropri-

ate location and then making a set of configuration releases instead of released views. Retaining the executable module allows you to execute the released application; the configuration objects can be used to rebuild the set of released views from the bottom up, if necessary.

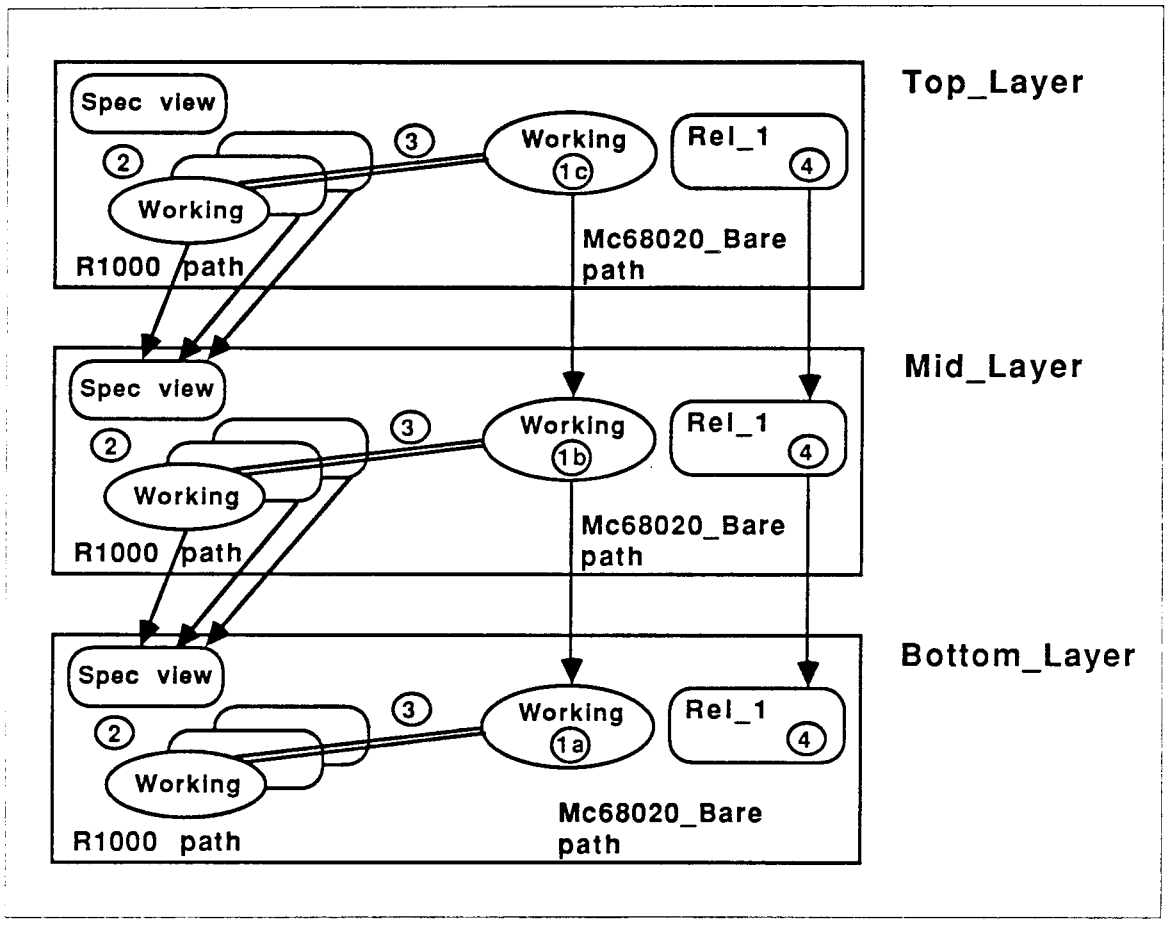


Figure 7-5. Setup for Development under Method II

Because accepting changes may cause the demotion and recompilation of units in client views, Method II requires more synchronized development efforts. That is, changes should be accumulated in the R1000 working view and propagated to the target working view only at synchronization points that are agreed upon by developers of all affected subsystems.

However, the advantage of Method II is that recompilation requirements are minimized. That is, when changes are accepted, the only units that require recompilation are the changed units themselves and any units in their transitive closures. In contrast, under Method I, changes are propagated by making and then importing new releases; changing imports potentially causes entire client views to be recompiled.

For example, assume that a statement in a unit body is changed in Bottom_Layer. Under Method I, this change is made available to clients by making a new release from the target path in Bottom_Layer, changing the imports of the target working view in Mid_Layer, and then making a release in Mid_Layer and changing the imports of Top_Layer. Under this method, two new releases need to be made and two working views need to be recompiled as a result of changing imports.

Under Method II, the changed unit body is accepted from the R1000 path into the working view of the target path in Bottom_Layer, where that unit is recompiled. No further recompilation is necessary because no units depend on unit bodies.

Method III: For Development on Multiple Hosts

Method III builds on Method II to accommodate cross-development on multiple hosts. Assume that primary subsystems for Bottom_Layer and Mid_Layer exist on separate machines and that secondary subsystems have been created for each primary subsystem on the appropriate machines. Like the primary subsystems, each secondary subsystem is set up with two paths (as shown in Figure 7-6):

- An R1000 path containing spec views and releases copied from the R1000 path in the associated primary subsystem
- A target path containing working combined (or load) views copied from the target path in the associated primary subsystem

Imports are set up on the secondary subsystem as they are on the primary.

Method II is used for development within each primary subsystem. For example, assume that changes have been made in the R1000 path in the primary subsystem for Bottom_Layer on Machine_1. These changes are propagated from the R1000 path to the target path within the primary subsystem, using the Cmvc.Accept_Changes command. Now you must propagate these changes from the target path in the primary subsystem (on Machine_1) to the target path in a secondary subsystem (on Machine_2). (Note that this is also covered as "Method II: Propagating Changed Units or Views" in the chapter entitled "Developing Applications Using Multiple Hosts.")

To propagate changes to the secondary subsystem:

1. Use the Archive.Copy command to copy only the changed units from the target working view on the primary subsystem to the target working view on the secondary:

```
Archive.Copy (Objects => "Rev1_Working",
              Use_Prefix => "!!Machine_2",
              Options => "Changed_Objects,Replace,Remake");
```

   In this command, the Changed_Objects option causes new and modified objects to be copied. The Replace option permits units with dependents to be demoted and overwritten. Dependent units are also demoted. The Remake option repromotes all units that were demoted by the Replace option.

   Because only changed units are copied, recompilation in the subsystems on Machine_2 is limited to the transitive closure of the changed units.

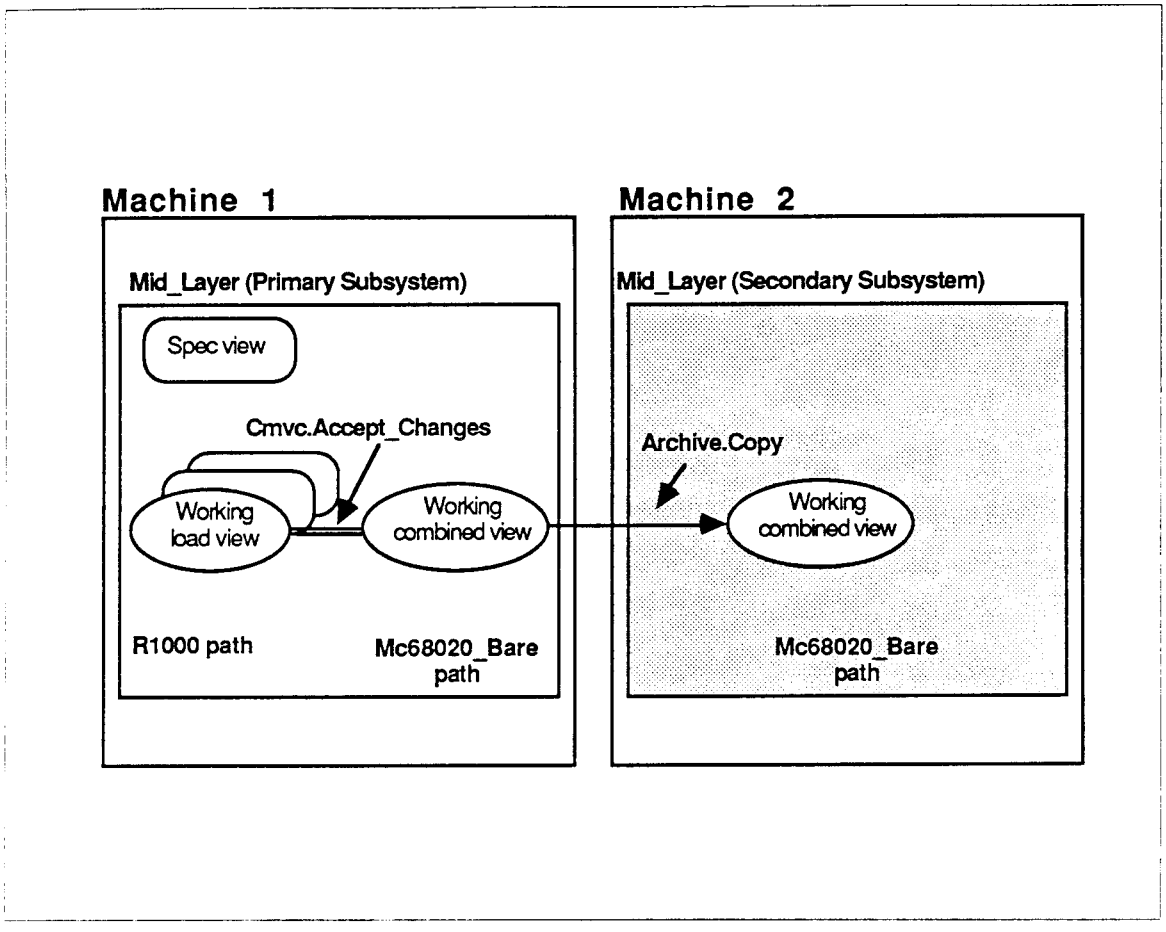


Figure 7-6. Setup for Multiple-Host Cross-Development

2. If necessary, delete any units from the target working view in the secondary subsystem that had been deleted from the target working view in the primary subsystem.

Following are special considerations when units are controlled in the secondary subsystem:

- Controlled units in the secondary subsystem must be checked out before the Archive.Copy command is entered. Otherwise, the changed units cannot be overwritten with the updated units.
- Unlike the Cmvc.Accept_Changes command, the Archive.Copy command does not automatically make controlled any new units that it copies. New units will have to be made controlled on the secondary subsystem as a separate step.

Note that history information maintained by CMVC is valid only in the primary subsystem.

RATIONAL

Naming

Many commands in the Rational Environment require a way of *naming* objects in the Environment to move those objects or to perform operations on those objects. The Environment uses two forms of naming: *Ada names* and *string names*. Ada names are used in program units or when executing a command. String names typically are used in the parameters to Environment commands.

Ada names are used to call an Environment command in a Command window or to reference an Ada unit in a program. Ada names are the extended Ada names as defined in the *Reference Manual for the Ada Programming Language*. Ada names are used to reference Ada units only. Files, worlds, directories, and other non-Ada objects in the Environment cannot be referenced with an Ada name.

String names are used as arguments to commands. These strings are very similar to Ada names but can be used to reference any object in the Environment. Also, string names have five important additions: *special names*, *parameter placeholders*, *wildcards*, *special characters*, and *attributes*. The ability also exists to create a set of names using simple set notations and to substitute characters.

Special Names

Special names are used as parameter values for many Environment operations to specify text, objects, and regions. Special names allow you to specify selections and designations without providing a pathname. Anywhere that a string name can be used, special names can be used. They take the form "*<special name>*", where *special name* specifies the text, object, region, or activity, as described below:

"<SELECTION>"	References the highlighted object if the cursor is located in a highlighted area.
"<REGION>"	References the highlighted object.
"<CURSOR>"	References the object on which the cursor is located, whether or not there is a highlighted area in the window.
"<IMAGE>"	References the highlighted object if the cursor is in a highlighted area. If the cursor is not located in the highlighted area, this special name references the image on which the cursor is located.

"<TEXT>"	References the highlighted text in the image in the window.
"<ACTIVITY>"	References the default activity. If an activity is highlighted and the cursor is in the highlight, this special name references that activity rather than the default activity.

Special names are used as default parameter values to many operations. The user can replace them with another special name or other form of string name, as accepted by that operation.

Special Values

Many operations in the Environment have a Response parameter that specifies how the command should respond to errors.

Error Reactions

When errors are discovered in a command, the system can respond by:

- Ignoring the error and trying to continue.
- Issuing a warning message and trying to continue.
- Raising an exception and abandoning the operation.

For each job, the Rational Environment maintains a default action for commands in package !Tools.Profile (documented in SJM) to take if an error occurs. There are commands to specify and display the default error reaction for a job. Regardless of the default error reaction, any error reaction can be specified for any command.

The Environment has *special values* used as parameters to commands for which *profile* it should use when responding to errors in a command. These are "<PROFILE>", "<SESSION>", and "<DEFAULT>", which refer, respectively, to the job response profile, the session response profile, and the default profile returned by the Profile.Default_Profile function. See SJM, package Profile, for further information on profiles.

Parameter Placeholders

Many Environment commands use parameter placeholders as default parameter values. They take the form *">>parameter placeholder<<"*. This naming convention is used, as its name suggests, as a placeholder indicating the type of string name that must be entered to replace it. Executing a command without replacing a parameter placeholder results in an error. Parameter placeholders include:

```
">>FILE NAME<<"
">>PATH NAMES<<"
">>ACTIVITY NAME<<"
```

For example, an operation that has the ">>FILE NAME<<" parameter placeholder requires a filename, such as "!Users.John.File_1".

Wildcards

Wildcards allow for both the abbreviation of names and the specifying of several objects with one name. The wildcards are: pound sign (#), *at* sign (@), question mark (?), and double question mark (??).

Wildcard

The pound sign (#) represents any single identifier character in a name, including the underscore (_). It can be used several times within a single name. For example, F### will match the name Food.

Any wildcard can be used to represent a set of named objects. For example, if there are objects in the directory !Users.Stooges called Larry, Curly, and Moe, a single string, such as !Users.Stooges.####y, can be created to refer to the first two of them.

Wildcard @

The *at* sign (@) represents zero or more identifier characters in a name, including the underscore (_). It does not match any subunits of Ada units. It can be used several times within a single name. For example, the name !Users.Fred.Food can be written !U@.@.Food if that abbreviation is unambiguous.

This wildcard can be used to represent a set of named objects. For example, if there are objects in the directory !Users.Stooges called Larry, Curly, and Moe, a single string, such as !Users.Stooges.@, can be created to refer to all three of them.

This wildcard can be combined with the special characters, discussed in the next section, to create very short names that represent sets of objects in the current context. As before, if there are three Ada units in the current context called Larry, Curly, and Moe, the string @ can be used to represent all three Ada units, but it would not include their subunits.

Wildcard ?

The question mark (?) represents zero or more components in a name, which are not worlds or objects contained by those worlds. For example, the name !Users.Stooges? represents the Ada units called Larry, Curly, and Moe and any of their subunits.

Also note that periods before and after the wildcard are optional. For example, the name A.?.B is equivalent to the name A?B.

Wildcard ??

The double question mark (??) represents zero or more components in a name, including worlds or objects contained by those worlds. For example, the name !Users?? represents the home worlds of all users and the contents of those worlds; !Users.Bill represents everything in his home world, including worlds and the objects within those worlds. As another example, consider that !?? matches all objects in the directory system on a given machine.

Note that periods before and after the wildcard are optional. For example, the name A.??.B is equivalent to the name A??B.

Substitution Characters

Similar to the way in which wildcard characters can be used to specify a source group of objects, substitution characters can be used to create target names from source names.

The substitution characters and their definitions are described below. Note that if a substitution character is encountered after all segments/wildcards have been exhausted, the characters are replaced by the null string. If the character # or ? is replaced by the null string, an immediately following period (.) is also elided from the resulting string.

Substitution Character

The pound sign (#) is replaced by the next complete segment in a name. For example, if there are Ada units in the world !Users.Stooges called Larry, Curly, and Moe, and the user wants to copy them into !Users.Stooges.New_World, the user could build the target name parameter (from the !Users.Stooges source name parameter) using substitution characters as follows: !#.#.New_World.#.

Substitution Character @

The *at* sign (@) is replaced by the portion of the current segment that is matched by a wildcard in the source name. If there is more than one wildcard in the segment, a separate @ is needed in the target to match each one. If the current segment has no wildcards, the next character that is followed by any of the special (not wildcard) characters covered in this section is not eligible as the source of the substitution. (For the purpose of this matching, @, #, ?, and ?? are considered to be wildcards.)

For example, there is a world called !Users.Gzc containing files File_1 through File_50. The user wants to rename these objects My_File_1 through My_File_50. The source name parameter would be !Users.Gzc.File_@. The target name parameter, using substitution parameters, would be !#.#.My_File_@.

Substitution Character ?

The question mark (?) is replaced by successive full segments until the segment for a world is encountered. For example, to copy everything in a world up through the next-level world !Users.Mary to !Users.John, the source string would be !Users.Mary?? and the target string would be !Users.John?.

Special Characters in Names

Special characters can be used in names to specify either relative or absolute contexts or to specify indirect files of names. These special characters apply to names used throughout the Environment.

A special character in a name determines the context in which the remaining portion of the name will be interpreted. A special character of exclamation (!), caret (^), dollar sign ($), double dollar sign ($$), percent (%), underscore (_), period (.), backslash (\), or grave (`) causes an explicit interpretation of the remainder of the name as described below.

Character pairs are also used to enclose a name and to give that name an additional meaning. Character pairs are brackets ([ ]) and braces ({ }), which are also described below.

Special Character !

The exclamation mark (!) specifies that the context for resolving the remainder of the name should be set to the root of the directory system. This creates a *fully qualified name.* This character represents the root of the library system in any context.

Special Character ^

The caret (^) specifies that the context should be set to the immediately enclosing object. This climbs the hierarchy of objects and eventually reaches the root of the directory system. This prefix can be used repeatedly to define the context to be several units above the current context. The parent object of the root of the directory system is itself.

A special use of this character occurs in combination with a bracketed name. A name component of the form ^[some_unit] resolves to the closest containing object whose simple name is Some_Unit. Brackets normally are used for creating sets of objects.

The caret also can be used as a shorthand method for referring to objects in a parent unit. For example, if the current context is !Users.Pete, another user named Joe can be referred to as !Users.Joe or simply ^Joe.

Special Character $

The dollar sign ($) specifies that the context should be set to the immediately enclosing library. A library is either a directory or a world. If the current context is a library, this character has no effect.

A special use of this character occurs in combination with a bracketed name. A name component of the form $[some_library] resolves to the closest containing library whose simple name is Some_Library.

Special Character $$

The double dollar sign ($$) specifies that the context should be set to the immediately enclosing world. This is more restrictive than the single dollar sign ($), which is either a world or a directory. If the current context is a world, this character has no effect.

A special use of this character occurs in combination with a bracketed name. A name component of the form $$[some_world] resolves to the closest containing world whose simple name is Some_World.

Special Character _

The underscore (_) is interpreted as an indirect file prefix when used in some Environment commands. If the first character after the underscore is an alphabetic character, then it is assumed to be the first character of the name of a file that contains other names. This provides a way of building lists of objects and referring to that list in a name. (See "Indirect Files," below.) The underscore also must be used when specifying an activity file as an indirect file.

Special Character .

The period (.) is used both as a name component separator and as a name prefix. As a separator, it is used just as in Ada names to separate components of a name. For example, in the name Commands.Ada, the period separates the two components of the name.

Special Character \

The backslash (\) specifies that the next name component be evaluated in the current searchlist. For example, a name such as Larry would be evaluated in the current context. However, a name such as \Larry would be evaluated in each of the contexts of the searchlist in turn until all occurrences of the name Larry are found in those contexts. If more than one occurrence is found, a menu is displayed.

More information about searchlists can be found in Session and Job Management (SJM).

Special Character `

The grave (`) is used to evaluate names using the current context and the set of links associated with the current context. The grave evaluates the name as if it were the name of an Ada unit in a *with* clause of a unit in the library that contains the current context. For example, the name `Moe resolves to an Ada unit called Moe in the containing library. Moe could be a link to some other library.

This kind of naming does not allow for renamed packages or instances of generic packages or subprograms to be used. It does not "look through" renaming declarations.

More information about links can be found in Library Management (LM).

Special Characters [ ]

Brackets ([ ]) define a set notation. Sets are created by enclosing a series of name components, separated by commas, in brackets. For example, the name [Larry, Curly, Moe] represents only those three objects in the current context.

The semicolon character also can be used to separate name components. Commas and semicolons cannot be mixed. If semicolons are used, each name component in the set must resolve to at least one object. For example, Foo?['C(Lib), 'Spec] matches any component of Foo that is either a library or an Ada spec. Foo[A;B] must match A and B in Foo.

Names also can be excluded from a set with the tilde (~). For example, the name [@, ~Curly] represents all names in the current context except the name Curly.

The special string [ ] represents the current context, whether that context is a directory, world, Ada unit, or other object.

Special Characters { }

Braces ({}) denote objects that have been deleted but not expunged as well as objects that have not been deleted. For example, if the object Curly is deleted but not expunged, the name @ refers only to Larry and Moe, but the name {@} refers to Larry, Curly, and Moe.

Indirect Files

Indirect files are text files that contain one or more object names or naming expressions. When an indirect file is given as a parameter value, the Environment converts the file's contents into set notation (see "Special Characters [ ]," above). An indirect file is thus a way of maintaining a list of objects that can then be referenced using a single name.

In an indirect file, you can put each name or naming expression on a separate line:

```
Larry
Curly
Moe
@_pkg
```

Alternatively, you can separate name components with commas or semicolons (with semicolons, each name component in the set must resolve to at least one object):

```
Larry,Curly
Moe,@_pkg
```

When resolving the contents of an indirect file, the Environment inserts commas in place of new lines and preserves any existing commas or semicolons.

To specify an indirect file as a parameter value, it must be prefixed with an underscore (_) (see "Special Character _", above). For example, to specify an indirect file called Archive_List, enter:

```
Objects => "_Archive_List"
```

RATIONAL

package Activity

An activity maintains a mapping between subsystems and pairs of views. The pair consists of a spec view and a load view from that subsystem. An activity typically is used to specify an implementation from each subsystem to be used for execution.

This package provides operations for creating, viewing, and manipulating activities and for identifying which activity is the current activity for a running job or session.

Editing Activities

In addition to the commands relating to activities, an editor provides editing operations specific to activities. Many of the operations in package !Commands.Common apply to activities. An activity can be viewed with the Edit command (or simply by getting the definition of the activity) and then can be edited with common editing operations. This section describes the commands from package !Commands.Common that apply to activities. Operations from package Common that do not apply to activities produce a message to that effect in the Message window.

Changes to activities are not made permanent until committed. When an activity is changed, but not yet committed, the # symbol appears in the window banner. Committing the activity makes all changes to the activity permanent, and the = symbol appears in the window banner.

Commands from Package !Commands.Common

procedure Common.Abandon

Ends editing of the activity and removes the window from the screen. Because all changes to activities are not made permanent until committed, any uncommitted changes will be lost.

procedure Common.Commit

Makes permanent any changes made to the activity.

procedure Common.Create_Command

Creates a Command window below the current window. The *use* clause in the Command window includes package Activity, so operations in package Activity are directly visible without qualification in the Command window.

procedure Common.Definition

Finds the definition of the subsystem corresponding to the selected entry or the entry on which the cursor resides, in the compressed form of an activity. For expanded entries (that is, those expanded to three lines: one each for the subsystem, the spec view, and the load view), this command finds the definition of the corresponding subsystem, spec view, or load view. This procedure creates a window containing that subsystem or view.

procedure Common.Edit

Prompts the user for changes to the selected entry, or to the entry on which the cursor resides when Edit is pressed, by creating a Command window and placing in it the command:

```
Change (Spec_View => "", Load_View => "");
```

The user fills in values for one or both parameters, as desired.

Spec- or load-view entries also can be specified indirectly through another activity. By specifying the name of an activity rather than the name of an actual view, the user indicates that the name of the desired view should be derived from the subsystem's corresponding entry in the specified activity.

procedure Common.Release

Makes any changes to the activity permanent, releases control of (unlocks) the activity, and then destroys the window.

procedure Common.Sort_Image

Sorts the activity image according to the specified sort format. These formats are specified by number:

1 Sorts by subsystem
2 Sorts by kind and subsystem
3 Sorts by kind and value
4 Sorts by kind and view
5 Sorts by value and subsystem
6 Sorts by value and kind
7 Sorts by view and subsystem
8 Sorts by view and kind

procedure Common.Object.Child

Selects the entry in the activity on which the cursor currently resides. If an entry is already selected, this command has no effect.

procedure Common.Object.Delete

Deletes the selected entry or the entry on which the cursor resides.

procedure Common.Object.Elide

Controls the level of detail displayed in the image of the current activity. Successive uses display successively less information about the activity entries, proceeding from top to bottom in the following list:

- All data by subsystem
- Load data by subsystem (indirections are identified)
- Spec data by subsystem (indirections are identified)
- Both views by subsystem
- Load views by subsystem (indirections are not identified)
- Spec views by subsystem (indirections are not identified)
- Subsystems by subsystem

procedure Common.Object.Expand

Controls the level of detail displayed in the image of the current activity. Successive uses display successively more information about the activity entries, proceeding from bottom to top in the list given under Common.Object.Elide.

procedure Common.Object.Explain

Uncompresses a subsystem entry, separating each component (subsystem name, spec view, and load view) of the entry onto separate lines.

procedure Common.Object.First_Child

Selects the first entry of the activity.

procedure Common.Object.Insert

Inserts a new subsystem entry or modifies an existing entry in the activity by prompting the user. Creates a Command window and places in it the command:

```
Insert (Subsystem => "", Spec_View => "", Load_View => "");
```

The user fills in values for parameters, as desired. If the subsystem name is omitted, it will be derived from the view names, provided that these are full pathnames.

Spec- or load-view entries also can be specified indirectly through another activity. By specifying the name of an activity rather than the name of an actual view, the user indicates that the name of the desired view should be derived from the subsystem's corresponding entry in the specified activity.

procedure Common.Object.Last_Child

Selects the last entry of the activity.

procedure Common.Object.Next

Selects the next entry in the activity if an entry is selected. If no entry is selected, this command selects the entry on which the cursor currently resides. If all entries are selected, this procedure produces an error.

procedure Common.Object.Parent

Selects the entry in the activity on which the cursor currently resides. If an entry is already selected, the procedure selects all entries in the activity. Otherwise, the procedure has no effect.

procedure Common.Object.Previous

Selects the previous entry in the activity if an entry is selected. If no entry is selected, the procedure selects the entry on which the cursor currently resides. If all entries are selected, this procedure produces an error.

subtype Activity_Name

```
subtype Activity_Name is String;
```

Description

Defines a string pathname that resolves to an activity in the directory system.

procedure Add

```
procedure Add
   (Subsystem     : Subsystem_Name          := "<CURSOR>";
    Load_Value    : View_Or_Activity_Name   := Activity.Nil;
    Spec_Value    : View_Or_Activity_Name   := Activity.Nil;
    The_Activity  : Activity_Name           := Activity.The_Current_Activity;
    Mode          : Creation_Mode           := Activity.Exact_Copy;
    Response      : String                  := "<PROFILE>");
```

Description

Modifies the activity specified by The_Activity parameter by updating an existing entry for a subsystem or by adding a new entry if an entry for the specified subsystem does not already exist.

Parameters

`Subsystem : Subsystem_Name := "<CURSOR>";`

Specifies the subsystem name for the new entry. This name is resolved in the current context. The default is the subsystem name on which the cursor is located.

`Load_Value : View_Or_Activity_Name := Activity.Nil;`

Specifies the name of a load view within the specified subsystem or an activity from which the load view can be derived (based on the Mode parameter) for the new entry. The view's name is resolved within the context of the specified subsystem. The default is the empty activity, indicating no load-view component.

`Spec_Value : View_Or_Activity_Name := Activity.Nil;`

Specifies the name of a spec view within the specified subsystem or an activity from which the spec view can be derived (based on the Mode parameter) for the new entry. The view's name is resolved within the context of the specified subsystem. The default is the empty activity, indicating no spec-view component.

`The_Activity : Activity_Name := Activity.The_Current_Activity;`

Specifies the activity to which the new entry will be added. The default indicates the current selection or image.

`Mode : Creation_Mode := Activity.Exact_Copy;`

Specifies the mode by which the new entry shall be derived, if either the Spec_Value or the Load_Value parameter specifies the name of an activity and not a view.

```
Response :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example

```
Add (Subsystem_Name => "User_Interface",
     Load_Value => "Rev1_0_4",
     Spec_Value => "Rev1_0_Spec",
     The_Activity => "!Application_Name.Current_Release");
```

References

type Creation_Mode

---

procedure Change

```
procedure Change (Spec_View : View_Or_Activity_Name := "";
                  Load_View : View_Or_Activity_Name := "");
```

Description

Modifies the spec-view and/or load-view components of the currently selected subsystem entry or the entry on which the cursor currently resides.

The !Commands.Common.Edit command prompts the user with this command. This command is meaningful only in a Command window associated with an activity.

Parameters

`Spec_View : View_Or_Activity_Name := "";`

Specifies the name of the new spec view. The null string indicates that the spec view should not be changed.

`Load_View : View_Or_Activity_Name := "";`

Specifies the name of the new load view. The null string indicates that the load view should not be changed.

Example

The following command changes only the load-view component of the designated entry to Rev1_0_7:

```
Change (Load_View => "Rev1_0_7");
```

The following command changes only the spec-view component of the designated entry to Rev1_1_Spec:

```
Change (Spec_View => "Rev1_1_Spec");
```

The following command changes both spec- and load-view components of the designated entry:

```
Change (Spec_View => "Rev1_0_Spec", Load_View => "Rev1_0_7");
```

The following command changes the spec-view component in the designated entry to be the spec view used in the corresponding entry in the named activity:

```
Change (Spec_View => "Some_Activity_Name");
```

References

EST, procedure Common.Edit

---

procedure Create

```
procedure Create (The_Activity : Activity_Name := ">>ACTIVITY NAME<<";
                 Source       : Activity_Name := Activity.Nil;
                 Mode         : Creation_Mode := Activity.Exact_Copy;
                 Response     : String        := "<PROFILE>");
```

Description

Creates a new activity.

The created activity may be derived from the source activity based on the Mode parameter.

Parameters

`The_Activity : Activity_Name := >>ACTIVITY NAME<<;`

Specifies the name of the new activity. The default parameter placeholder ">>ACTIVITY NAME<<" must be replaced or an error will result.

`Source : Activity_Name := Activity.Nil;`

Specifies the name of the activity from which the new activity is to be created. The default is an empty activity.

`Mode : Creation_Mode := Activity.Exact_Copy;`

Specifies the mode by which the entries shall be derived from the source activity.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example

The following command creates a new activity with indirect entries for all subsystems in the source activity !Release.Current.Activity:

```
Create (Activity => "My_Private_Activity",
        Source => "!Release.Current.Activity",
        Mode => Activity.Differential);
```

References

type Creation_Mode

---

type Creation_Mode

```
type Creation_Mode is (Differential, Exact_Copy, Value_Copy);
```

Description

Defines three modes for the creation of spec-view and load-view references for subsystem entries.

Enumerations

`Differential`

Indicates that the new entry should be formed as an indirect reference to the source activity. The created entry will not be the name of a view but the name of another activity that specifies an actual view or another activity from which to derive the view. With this mode, changes made to the source activity will be reflected in the target activity.

`Exact_Copy`

Indicates that the new entry should be formed as an exact copy of the entry in the source activity. Thus, if the source entry contains the name of an actual view, the new entry also will contain the actual view. If the source entry contains an indirect reference, the new entry will contain an identical indirect reference.

`Value_Copy`

Indicates that the new entry should be formed as the dereferenced value of the corresponding source entry. Indirect (differential) references will be resolved until an actual view is found.

procedure Current

```
procedure Current (Response : String := "<PROFILE>");
```

Description

Displays the name of the activity that is associated with the current job.

If no activity has been associated with the job, the procedure returns the activity currently associated with the running session.

The current activity is set by the Set and Set_Default procedures.

Parameters

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Set

procedure Set_Default

procedure Display

```
procedure Display
          (Subsystem     : Subsystem_Name  := "?";
           Spec_View     : View_Name       := "?";
           Load_View     : View_Name       := "?";
           Mode          : Creation_Mode   := Activity.Value_Copy;
           The_Activity  : Activity_Name   := Activity.The_Current_Activity;
           Response      : String          := "<PROFILE>");
```

Description

Displays an image of the specified activity.

Only the mappings that match the patterns (Environment naming conventions, including wildcards) given in the Subsystem, Spec_View, and Load_View parameters are listed. In Value_Copy mode, all indirect references are resolved; only the resolution is displayed. In Exact_Copy mode, indirect mappings are not resolved; the name of the source activity is displayed. In Differential mode, the indirect mappings are resolved; both the resolution and the original indirect activity are displayed.

Parameters

`Subsystem : Subsystem_Name := "?";`

Specifies the name of the subsystem entry to be displayed. The default indicates that all subsystem entries should be displayed.

`Spec_View : View_Name := "?";`

Specifies a pattern for spec-view entries. Only patterns that match are displayed. The default indicates that all spec views are acceptable.

`Load_View : View_Name := "?";`

Specifies a pattern for load-view entries. Only patterns that match are displayed. The default indicates that all load views are acceptable.

`Mode : Creation_Mode := Activity.Value_Copy;`

Specifies the mode by which the image of each entry shall be derived from the activity.

```
The_Activity :  Activity_Name := Activity.The_Current_Activity;
```

Specifies the name of the activity to be displayed. The default indicates the activity associated with the running job. If no activity has been associated with the job, the procedure returns the activity associated with the running session.

```
Response :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

function The_Current_Activity

---

procedure Edit

```
procedure Edit (The_Activity : Activity_Name := "<ACTIVITY>");
```

Description

Invokes the activity object editor on the specified activity.

The default is to edit the current activity.

Parameters

```
The_Activity :  Activity_Name := "<ACTIVITY>";
```

Specifies the name of the activity to be edited. The default indicates the activity for the current job or session.

procedure Enclosing_Subsystem

```
procedure Enclosing_Subsystem (View     : View_Name := "<IMAGE>";
                               Response : String    := "<PROFILE>");
```

Description

Displays the name of the subsystem that contains the specified view.

The default is the currently selected view, the view containing the current selection, or the view containing the current context.

The view may be either a spec or a load view.

Parameters

`View : View_Name := "<IMAGE>";`

Specifies the name of the view whose enclosing subsystem is desired. The default is the currently selected view, the view containing the current selection, or the view containing the current context.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Enclosing_View

```
procedure Enclosing_View (Unit     : Unit_Name := "<IMAGE>";
                          Response : String    := "<PROFILE>");
```

Description

Displays the name of the view that contains the specified unit.

The default is the currently selected unit or unit image.

Parameters

`Unit : Unit_Name := "<IMAGE>";`

Specifies the name of the unit for which the enclosing view is desired. The default is the currently selected unit or unit image.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Insert

```
procedure Insert
                (Subsystem : Subsystem_Name          := ">>SUBSYSTEM NAME<<";
                 Spec_View : View_Or_Activity_Name   := "";
                 Load_View : View_Or_Activity_Name   := "");
```

Description

Modifies an activity to update an existing entry for a subsystem or adds a new entry if one does not already exist for the specified subsystem.

The !Commands.Common.Object.Insert command prompts the user with this command. This command is meaningful only in a Command window associated with an activity.

Parameters

`Subsystem : Subsystem_Name := ">>SUBSYSTEM NAME<<";`

Specifies the name of the subsystem. The default parameter placeholder ">>SUBSYSTEM NAME<<" must be replaced or an error will result.

`Spec_View : View_Or_Activity_Name := "";`

Specifies the name of the spec-view for the subsystem entry. A null string specifies no entry if an entry does not already exist for the subsystem or no change if the subsystem does exist.

`Load_View : View_Or_Activity_Name := "";`

Specifies the name of the load view for the subsystem entry. A null string specifies no entry if an entry does not already exist for the subsystem or no change if the subsystem does exist.

Name resolution: The subsystem name is resolved in the current context. Names within spec- and load-view indications are resolved within the context of the specified subsystem.

Example 1

```
procedure Insert (Subsystem => "User_Interface",
                  Spec_View => "Rev1_0_Spec",
                  Load_View => "Rev1_0_5");
```

Example 2

```
procedure Insert (Subsystem => "User_Interface",
                  Spec_View => "",
                  Load_View => "Current_Release");
```

where Current_Release is the name of an activity in !My_Application.User_Interface.

References

EST, procedure Common.Object.Insert

---

procedure Merge

```
procedure Merge (Source     : Activity_Name   := ">>ACTIVITY NAME<<";
                 Subsystem  : Subsystem_Name  := "?";
                 Spec_View  : View_Name       := "?";
                 Load_View  : View_Name       := "?";
                 Mode       : Creation_Mode   := Activity.Exact_Copy;
                 Target     : Activity_Name   := "<ACTIVITY>";
                 Response   : String          := "<PROFILE>");
```

Description

Copies into the specified target those subsystem entries defined in the source activity that match the patterns specified in the Subsystem, Spec_View, and Load_View parameters.

New subsystem entries are added as necessary; existing subsystem entries are replaced.

Patterns for the Subsystem, Spec_View, and Load_View parameters are the standard Environment naming conventions and wildcards.

Parameters

`Source : Activity_Name := ">>ACTIVITY NAME<<";`

Specifies the name of the activity from which entries are to be copied. The default parameter placeholder ">>ACTIVITY NAME<<" must be replaced or an error will result.

`Subsystem : Subsystem_Name := "?";`

Specifies the subsystem entries to be copied. The default indicates that all subsystem entries should be copied.

`Spec_View : View_Name := "?";`

Specifies a pattern for spec-view entries. Only patterns that match are copied. The default indicates that all spec views are acceptable.

`Load_View : View_Name := "?";`

Specifies a pattern for load-view entries. Only patterns that match are copied. The default indicates that all load views are acceptable.

`Mode : Creation_Mode := Activity.Exact_Copy;`

Specifies the mode by which entries are derived from the source activity.

```
Target : Activity_Name := "<ACTIVITY>";
```

Specifies the name of the activity into which the new entries are to be copied. The default target activity is the current activity for the job or session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

function Nil

```
function Nil return Activity_Name;
```

Description

Returns the name of an empty activity.

procedure Remove

```
procedure Remove
          (Subsystem     : Subsystem_Name  := "<SELECTION>";
           The_Activity  : Activity_Name   := Activity.The_Current_Activity;
           Response      : String          := "<PROFILE>");
```

Description

Deletes a subsystem entry from an activity.

The default activity is the current activity for the job or session.

Parameters

`Subsystem : Subsystem_Name := "<SELECTION>";`

Specifies the name of the subsystem entry to be deleted. The default is the current selection.

`The_Activity : Activity_Name := Activity.The_Current_Activity;`

Specifies the name of the activity from which the entry is to be deleted. The default indicates the current activity for the job or session.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Set

```
procedure Set (The_Activity : Activity_Name := "<ACTIVITY>";
               Response     : String       := "<PROFILE>");
```

Description

Changes the current activity for the running job to the specified activity.

A session may have a current activity associated with it. The Set_Default procedure is used to form this association. When a job begins execution, its current activity is that of the current session. The Set procedure changes a job's current activity without changing the session's activity. Thereafter, until the job terminates, the new activity is consulted when necessary, instead of the current session's activity.

Note that this procedure cannot be used to affect the loading of a subsequent command in the same job. Loading is done for the entire job before execution begins and thus would be unaffected by the execution of the Set command. In the following example:

```
Activity.Set ("New_Activity_Name");
Command_Requiring_Loading;
```

the Command_Requiring_Loading command will be loaded with the current session's activity and not with New_Activity_Name.

By contrast, in the example:

```
Activity.Set ("New_Activity_Name");
Program.Run ("Command_Requiring_Loading");
```

the loading for the command via Program.Run is performed after the execution of the Set procedure and thus will use New_Activity_Name.

Parameters

```
The_Activity :  Activity_Name := "<ACTIVITY>";
```

Specifies the name of the activity to make current for this job. The default indicates the activity for the current session.

```
Response :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Set_Default

---

procedure Set_Default

```
procedure Set_Default (The_Activity : Activity_Name := "<ACTIVITY>";
                       Response     : String       := "<PROFILE>");
```

Description

Makes the specified activity the current activity for the current session.

This procedure sets the value of the Profile.Activity_File session switch. If the current activity of the job that executes Set_Default is nil, the procedure sets this activity as well.

The default activity for a session is also preserved across logouts.

Parameters

`The_Activity : Activity_Name := "<ACTIVITY>";`

Specifies the name of the activity to make current. The default indicates the current activity.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Set

procedure Set_Load_View

```
procedure Set_Load_View
   (Load_View     : View_Or_Activity_Name  := "<CURSOR>";
    Subsystem     : Subsystem_Name         := "";
    Mode          : Creation_Mode          := Activity.Differential;
    The_Activity  : Activity_Name          := Activity.The_Current_Activity;
    Response      : String                 := "<PROFILE>");
```

Description

Modifies the load view for the specified subsystem entry in The_Activity parameter.

If an entry for the specified subsystem does not exist, one is added to the activity.

Parameters

```
Load_View :  View_Or_Activity_Name := "<CURSOR>";
```

Specifies the name of the new load-view entry. Name resolution is performed within the context of the specified subsystem. The default is the load-view name or activity name on which the cursor is located.

If the Load_View parameter designates a view, that view is associated with the subsystem that contains it. The value of Load_View must be the simple name of a load view.

If the Load_View parameter designates an activity, that activity must contain an entry for the subsystem specified by the Subsystem parameter. The load view associated with Subsystem in this activity becomes the load view associated with Subsystem in the activity named by The_Activity.

```
Subsystem :  Subsystem_Name := "";
```

Specifies the name of the subsystem entry. Name resolution is performed relative to the current context. The default value ("") resolves to the current context and therefore can be used only when the current context is the subsystem that contains the view specified by the Load_View parameter. Otherwise, the Subsystem parameter must name the subsystem that contains the view designated by the Load_View parameter.

```
Mode :  Creation_Mode := Activity.Differential;
```

Specifies the mode by which the entry shall be derived if an activity is designated by the Load_View parameter.

```
The_Activity :  Activity_Name := Activity.The_Current_Activity;
```

Specifies the name of the activity to be modified. The default indicates the current selection or image.

```
Response :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example 1

```
Set_Load_View (Load_View => "Rev1_2_4",
               Subsystem => "User_Interface",
               The_Activity => "Current_Release");
```

is equivalent to:

```
Set_Load_View (Load_View => "User_interface.Rev1_2_4",
               Subsystem => "",
               The_Activity => "Current_Release");
```

Example 2

```
Set_Load_View (Load_View => "Working_Activity",
               Subsystem => "User_Interface",
               Mode => Activity.Value_Copy,
               The_Activity => "Current_Release");
```

This command changes the load view for the User_Interface subsystem in the Current_Release activity. The command will set the load view to the same value as that specified for the User_Interface subsystem in Working_Activity.

---

procedure Set_Spec_View

```
procedure Set_Spec_View
   (Spec_View     : View_Or_Activity_Name  := "<CURSOR>";
    Subsystem     : Subsystem_Name         := "";
    Mode          : Creation_Mode          := Activity.Differential;
    The_Activity  : Activity_Name          := Activity.The_Current_Activity;
    Response      : String                 := "<PROFILE>");
```

Description

Modifies the spec view for the specified subsystem entry in The_Activity parameter.

If an entry for the specified subsystem does not exist, one is added to the activity.

Parameters

`Spec_View : View_Or_Activity_Name := "<CURSOR>";`

Specifies the name of the new spec-view entry. Name resolution is performed within the context of the specified subsystem. The default is the spec-view name or activity name on which the cursor is located.

If the Spec_View parameter designates a view, that view is associated with the subsystem that contains it. The value of Spec_View must be the simple name of a spec view.

If the Spec_View parameter designates an activity, that activity must contain an entry for the subsystem specified by the Subsystem parameter. The spec view associated with Subsystem in this activity becomes the spec view associated with Subsystem in the activity named by The_Activity.

`Subsystem : Subsystem_Name := "";`

Specifies the name of the subsystem entry. Name resolution is performed relative to the current context. The default value ("") resolves to the current context and therefore can be used only when the current context is the subsystem that contains the view specified by the Spec_View parameter. Otherwise, the Subsystem parameter must name the subsystem that contains the view designated by the Spec_View parameter.

`Mode : Creation_Mode := Activity.Differential;`

Specifies the mode by which the entry shall be derived if an activity is designated by the Spec_View parameter.

```
The_Activity :  Activity_Name := Activity.The_Current_Activity;
```

Specifies the name of the activity to be modified. The default indicates the current selection or image.

```
Response :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Example 1

```
Set_Spec_View (Spec_View => "Rev1_2_Spec",
               Subsystem => "User_Interface",
               The_Activity => "Current_Release");
```

is equivalent to:

```
Set_Spec_View (Spec_View => "User_interface.Rev1_2_Spec",
               Subsystem => "",
               The_Activity => "Current_Release");
```

Example 2

```
Set_Spec_View (Spec_View => "Working_Activity",
               Subsystem => "User_Interface",
               Mode => Activity.Value_Copy,
               The_Activity => "Current_Release");
```

This command changes the spec view for the User_Interface subsystem in the Current_Release activity. The command will set the spec view to the same value as that specified for the User_Interface subsystem in Working_Activity.

---

subtype Subsystem_Name

```
subtype Subsystem_Name is String;
```

Description

Defines a string pathname that resolves to a subsystem in the directory system.

function The_Current_Activity

```
function The_Current_Activity return Activity_Name;
```

Description

Returns the name of the current activity associated with the running job.

If no activity has been associated with the running job, this function returns the activity associated with the running session.

function The_Enclosing_Subsystem

```
function The_Enclosing_Subsystem (View : View_Name := "<IMAGE>")
                                                    return Subsystem_Name;
```

Description

Returns the name of the subsystem that contains the specified view.

The default is the currently selected view, the view containing the current selection, or the view containing the current context.

Parameters

`View : View_Name := "<IMAGE>";`

Specifies the name of the view whose enclosing subsystem is desired. The default is the currently selected view, the view containing the current selection, or the view containing the current context.

`return Subsystem_Name;`

Returns the name of the subsystem that contains the specified view.

function The_Enclosing_View

```
function The_Enclosing_View (Unit : Unit_Name := "<IMAGE>")
                                                    return View_Name;
```

Description

Returns the name of the view that contains the specified unit.

Parameters

`Unit : Unit_Name := "<IMAGE>";`

Specifies the name of the unit whose enclosing view is desired. The default is the currently selected unit or unit image.

`return View_Name;`

Returns the name of the view that contains the specified unit.

subtype Unit_Name

```
subtype Unit_Name is String;
```

Description

Defines a string pathname that resolves to an Ada compilation unit in the directory system.

subtype View_Name

```
subtype View_Name is String;
```

Description

Defines a string pathname that resolves to a view of a subsystem.

subtype View_Or_Activity_Name

```
subtype View_Or_Activity_Name is String;
```

Description

Defines a string pathname that resolves either to a view of a subsystem or to an activity in the directory system.

subtype View_Simple_Name

```
subtype View_Simple_Name is String;
```

Description

Defines a string that is the simple name of a view of a subsystem.

A simple name is an unqualified name not prefixed with the name of the object's parent.

Example 1

```
Rev1_0_5

Rev2_0_Spec
```

not:

```
User_Interface.Rev3_4_7
```

procedure Visit

```
procedure Visit (The_Activity : Activity_Name := "<ACTIVITY>");
```

Description

Invokes the activity editor on the specified activity and replaces the old activity if one is currently being edited.

This procedure is identical to the !Commands.Common.Edit command, except that if the command is given on an activity window, the new activity is displayed in that window rather than in a new one.

Parameters

```
The_Activity : Activity_Name := "<ACTIVITY>";
```

Specifies the name of the activity to be visited. The default is the current activity for the job or session.

References

EST, procedure Common.Edit

procedure Write

```
procedure Write (File : Activity_Name := "<ACTIVITY>");
```

Description

Copies the contents of an activity window into a new activity in the directory system.

This command is valid only in an activity window.

Parameters

```
File :  Activity_Name := "<ACTIVITY>";
```

Specifies the name of the new activity. The name is resolved relative to the current context. The default is the current activity for the job or session.

end Activity;

RATIONAL

package Check

Package Check provides interfaces for checking the compatibility between spec and load views in a subsystem. Compatibility is defined in the Key Concepts to this book. Command-oriented interfaces and programmatic interfaces with status values are provided. Interfaces are available for comparing units in load views with their corresponding units in spec views or for comparing a set of spec/load-view pairs.

The compatibility checking done by this package checks that every declaration exported by a spec-view unit is also exported by the corresponding load-view unit and that the spec and load views have the same target key. These declarations do not need to be in the same order or textually identical. Two declarations are considered equivalent if they match according to the subprogram specification conformance rules of the *Reference Manual for the Ada Programming Language*, section 6.3.1. All units involved in the check must be in the installed or coded state.

It is possible to construct an activity such that the spec/load-view pairs named by the activity are compatible but the set of load views specified would not execute correctly together. For example, this could happen if two load views in the activity import different spec views of the same subsystem. The checks done by this package will not catch those situations, but the loader will check for this and report these types of problems at load time.

procedure Activity

```
procedure Activity (The_Activity : String  := "<ACTIVITY>";
                    Menu         : Boolean := False;
                    Response     : String  := "<PROFILE>");

function  Activity (The_Activity : String := "<ACTIVITY>";
                    Response     : String := "<PROFILE>") return Status;
```

Description

Checks the compatibility of all spec-view and load-view pairs specified in an activity.

For each subsystem entry in the activity, each unit in the spec view is checked for compatibility with the corresponding unit in the load view. If an entry for a subsystem does not specify both a spec and a load view, that subsystem will not be checked.

Two interfaces are provided: a procedure interface for Command window usage and a functional interface returning a status value for use in building tools.

Parameters

`The_Activity : String := "<ACTIVITY>";`

Specifies the name of the activity whose spec-view and load-view pairs should be checked for compatibility.

`Menu : Boolean := False;`

Specifies whether to display a menu output. This parameter applies only to the procedure interface.

If true, any incompatible units will be reported in a menu image instead of in messages in a log file. Traversing to a spec-view unit from this menu displays the unit with the incompatible declarations underlined. You can also change the elision level of the menu to display these declarations. When this parameter is true, no log file is produced unless there are errors in the command's execution.

If false, a log file is produced as specified by the Response parameter.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

type Status

```
type Status is (Compatible, Incompatible, Error);
```

Description

Defines the range of possible outcomes of a compatibility check.

Enumerations

`Compatible`

Indicates that the unit or units included in the check are all compatible with each other.

`Error`

Indicates that the operation failed to complete successfully. For example, the name of a unit or view may not be resolvable. The log output can be consulted to determine the reason for the error.

`Incompatible`

Indicates that at least one unit included in the check is not compatible with another unit. The log output can be consulted to determine which unit or units are not compatible.

procedure Units

```
procedure Units (Load_View_Units  : String  := "<CURSOR>";
                 Spec_Views       : String  := "<ACTIVITY>";
                 Menu             : Boolean := False;
                 Response         : String  := "<PROFILE>");

function Units (Load_View_Units  : String := "<CURSOR>";
                Spec_Views       : String := "<ACTIVITY>";
                Response         : String := "<PROFILE>") return Status;
```

Description

Checks the compatibility of a set of units in load views with their corresponding units in the specified spec views.

The Load_View_Units parameter specifies the set of units to be checked, and the Spec_Views parameter specifies the set of spec views used to perform the check.

Two interfaces are provided: a procedure interface for Command window usage and a functional interface returning a status value for use in building tools.

Parameters

```
Load_View_Units :  String := "<CURSOR>";
```

Specifies one or more units in load views to be checked. Multiple units can be specified by using wildcards, context characters, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts section of this book.)

```
Spec_Views :  String := "<ACTIVITY>";
```

Specifies one or more spec views to be used to perform the check. Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts section of this book.) Furthermore, Spec_Views can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Menu : Boolean := False;
```

Specifies whether to display a menu output. This parameter applies only to the procedure interface.

If true, any incompatible units will be reported in a menu image instead of in messages in a log file. Traversing to a spec-view unit from this menu displays the unit with the incompatible declarations underlined. You can also change the elision level of the menu to display these declarations. When this parameter is true, no log file is produced unless there are errors in the command's execution.

If false, a log file is produced as specified by the Response parameter.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Restrictions

The naming expressions for the Spec_Views parameter must resolve to subsystem views or an activity and must not specify multiple spec views from the same subsystem. The naming expressions for the Load_View_Units parameter must resolve to compilation units. Spec_Views must specify a spec view for each subsystem that contains one or more of the units specified by Load_View_Units.

Example

In a Command window off a window displaying a unit in a working view of a subsystem, the command:

```
Check.Units (Load_View_Units => "<CURSOR>",
             Spec_Views => "<ACTIVITY>");
```

will check compatibility of the unit in the working view with the unit in the spec view as specified in the user's default activity. Using the default values for the Menu and Response parameters will result in log file output.

---

procedure Views

```
procedure Views (Load_Views  : String  := "<CURSOR>";
                 Spec_Views  : String  := "<ACTIVITY>";
                 Menu        : Boolean := False;
                 Response    : String  := "<PROFILE>");

function Views (Load_Views  : String := "<CURSOR>";
                Spec_Views  : String := "<ACTIVITY>";
                Response    : String := "<PROFILE>") return Status;
```

Description

Checks the compatibility of all units in one or more spec/load-view pairs.

The Load_View parameter specifies the set of views to be checked, and the Spec_Views parameter specifies the set of spec views used to perform the check.

Two interfaces are provided: a procedure interface for Command window usage and a functional interface returning a status value for use in building tools.

Parameters

```
Load_Views :  String := "<CURSOR>";
```

Specifies one or more load views to be checked. Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts section of this book.) Furthermore, Load_Views can name an activity as an indirect file, which is equivalent to naming the load view associated with each subsystem listed in the activity.

```
Spec_Views :  String := "<ACTIVITY>";
```

Specifies the spec views used to perform the check. Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts section of this book.) Furthermore, Spec_Views can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Menu : Boolean := False;
```

Specifies whether to display a menu output. This parameter applies only to the procedure interface.

If true, any incompatible units will be reported in a menu image instead of in messages in a log file. Traversing to a spec-view unit from this menu displays the unit with the incompatible declarations underlined. You can also change the elision level of the menu to display these declarations. When this parameter is true, no log file is produced unless there are errors in the command's execution.

If false, a log file is produced as specified by the Response parameter.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Restrictions

The naming expressions for the Spec_Views and Load_Views parameters must resolve to subsystem views or an activity. The Spec_Views parameter should not specify multiple spec views of the same subsystem, and it should specify a spec view for each subsystem that contains one or more of the views specified by the Load_Views parameter. Combined views named by these parameters are not considered for checking.

Example

In a Command window off a window displaying the Rev1_Working view of My_Subsystem, the command:

```
Check.Views (Load_Views => "<CURSOR>",
             Spec_Views => "<ACTIVITY>");
```

will check compatibility of units in the spec view for My_Subsystem, as specified by the user's default activity, with the units in Rev1_Working. Using the default values for the Menu and Response parameters will result in log file output.

---

end Check;

RATIONAL

package Cmvc

Package Cmvc defines a set of operations that support the following activities of project management:

- Partitioning projects into components using subsystems and managing the interfaces among these components
- Creating and releasing alternative implementations (views) of individual project components
- Placing the objects within project components under source control to record generations of change history and to coordinate the work of multiple developers
- Coordinating parallel development efforts both within and between subsystems

The Key Concepts section of this book provides a guide to project development using subsystems, views, and source control.

The following sections in this package introduction provide:

- A list of CMVC commands grouped by topic
- A summary of the types of objects you can create and manage through CMVC commands
- A guide to the images and operations for managing configuration information interactively
- A list of switches and commands from package Common that pertain to CMVC

Commands Grouped by Topic

The commands in package Cmvc fall into several functional groups. They are listed here by group for your convenience. (Note that the reference entries for these commands are arranged in alphabetical order by command name.)

- Commands for creating and destroying subsystems and systems:

  Initial
  Destroy_Subsystem
  Destroy_System

- Commands for creating, releasing, destroying, and recreating views:

  Build
  Copy
  Destroy_View
  Initial
  Make_Code_View
  Make_Path
  Make_Spec_View
  Make_Subpath
  Release

- Commands for managing source control:

  Abandon_Reservation
  Accept_Changes
  Check_In
  Check_Out
  Join
  Make_Controlled
  Make_Uncontrolled
  Merge_Changes
  Revert
  Sever

- Commands for managing subsystem interfaces:

  Import
  Imported_Views
  Remove_Imports
  Remove_Unused_Imports
  Replace_Model

- Commands for interactively viewing notes and configuration information:

  Def
  Edit
  Notes

- Commands for displaying reports in I/O windows:

  Information
  Show
  Show_All_Checked_Out
  Show_All_Controlled
  Show_All_Uncontrolled
  Show_Checked_Out_By_User
  Show_Checked_Out_In_View
  Show_History
  Show_History_By_Generation
  Show_Image_Of_Generation
  Show_Out_Of_Date

- File-oriented commands for managing notes:

  Append_Notes
  Create_Empty_Note_Window
  Get_Notes
  Put_Notes

System Object and View Types

The Cmvc.Initial command can create several types of *system objects* (see also System_Object_Enum type). A system object refers to both *systems* and *subsystems*:

- Subsystems provide a means of partitioning applications into components to facilitate parallel development, minimize recompilation dependencies, and enforce design decisions. Each subsystem contains the units that implement a component of an application. As development progresses within a given subsystem, *releases* can be made of its implementation.
- Systems provide an optional means of grouping the subsystems that compose an application; within a system, operations are available for tracking the latest release from each subsystem and for referencing those releases for execution. These releases are referenced by *release activities* that are built and maintained within the system. Systems can form a hierarchy by including other systems.

There is only one type of system, and systems contain only one type of view—namely, *system views*. In contrast, there are two types of subsystem—*spec/load subsystems* and *combined subsystems*. Spec/load subsystems can contain *spec* and *load views*, which function together, as well as *combined views*. Combined subsystems can contain only combined views.

Within a spec/load subsystem, load views and combined views can be created using the Initial, Copy, or Make_Path command; spec views are created using the Make_Spec_View or Copy command:

- A *load view* contains an implementation of a subsystem. Load views are specified in activities and are actually used for execution.
- A *spec view* expresses a subsystem's exports. Exports are the specifications of implemented units that are made available for units in other subsystems to reference in *with* clauses. When a spec view is imported by a view in another subsystem, units in the importing view can compile against the units in the imported view.
- A *combined view* combines characteristics of spec and load views. A combined view both contains a subsystem implementation and expresses the exports from that implementation. When a combined view is imported, units in the importing subsystem can compile against the combined view's units; at execution time, the units in that combined view are executed.

Spec and load views provide greater flexibility than combined views during development and test. Using spec and load views minimizes the recompilation required after changes are made and eliminates the need for recompilation during recombinant testing. Using combined views involves no such reduction of recompilation requirements; from a recompilation point of view, development in combined views is equivalent to development in worlds. (Note, however, that development in combined views makes CMVC operations available, which are not available in worlds.)

In spite of the advantages of spec and load views, combined views must be used in spec/load subsystems under certain circumstances—namely, when generics or inlined subprograms are exported from implementations for non-R1000 targets.

Combined views also must be used in combined subsystems. Combined subsystems must be used when import relationships in an application need to be circular—that is, when a given view must be within its own import closure (for example, when two views must import each other). In contrast, import relationships among views in spec/load subsystems must be hierarchic.

Within any system object (subsystem or system), there are *working views* within which ongoing development and maintenance proceeds. A view is recognized as a working view through a naming convention—namely, the _Working suffix. A load view, a combined view, or a system view can serve as a working view.

At any time during development and maintenance, a *release* can be made from a working view. A release is a frozen copy of the working view; releases typically are made after the implementation in the working view is compiled and tested.

Managing CMVC Information Interactively

When objects are controlled in one or more views in a subsystem, you typically need to know the following information, which is managed by the CMVC database:

- Which objects are controlled?
- Which objects are checked out and to which views?
- Which objects are joined to objects in other views?
- Which views contain the other objects in the join set?
- Which objects in a join set are out of date and which view contains the latest generation of these objects?

Furthermore, because multiple generations typically exist for a given object in a view, it is useful to be able to:

- View images of past generations for the object.
- View the line-by-line differences between two successive generations.
- Find out when a given generation was created.
- Keep a scratchpad of notes recording the changes that were made to each generation.
- Review the comments that were supplied each time the object was checked out and checked in.

The Def, Edit, and Notes procedures bring up three kinds of images in which such information is displayed. These images are *configuration images*, *generation images*, and *history images*. These images not only provide several levels of information but also make available commands from package Common, which you can use to traverse to other images and perform certain CMVC operations.

Configuration Images

The Edit procedure displays a configuration image for a specified view or configuration object or for the view enclosing a specified controlled object. A configuration image for a view is a library-like display of CMVC information pertaining to the configuration embodied by that view. (A configuration is a set of generations, one for each controlled object in the view.) Note that Edit can be used to display a configuration image for a configuration object that has no view associated with it (for example, a configuration release).

The information displayed in a configuration image represents the contents of the CMVC database at the time the Edit procedure is entered. Subsequent CMVC operations can change the CMVC database without automatically updating the configuration image. You can refresh a configuration image using the Common.Format or Common.Revert command.

Levels of Information in Configuration Images

Several levels of information are available in a configuration image. For example, Figure 11-1 shows the configuration image displayed for the view !Programs.Mail-.Mail_Utilities.

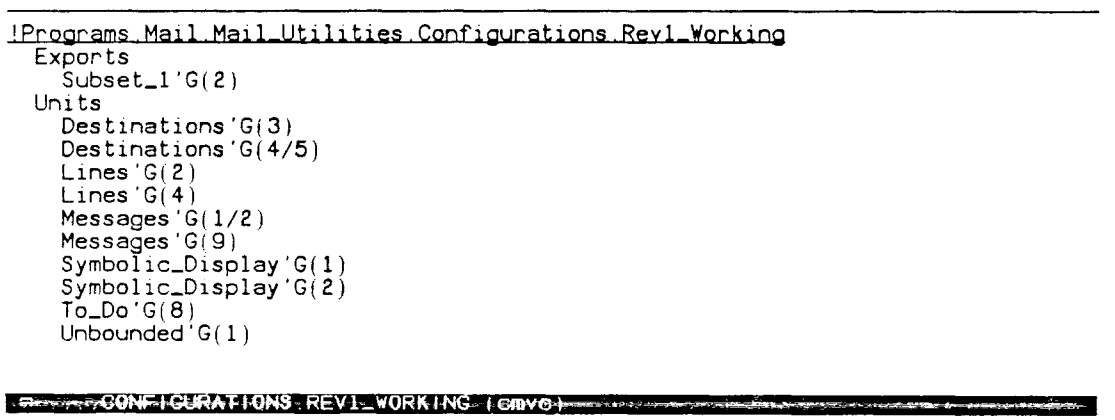


Figure 11-1. The First Level of Information in a Configuration Image

This image contains the first level of configuration information. This level contains an entry for each controlled object in the view's configuration. Each entry indicates the generation of the object that is present in the view. An entry also indicates the latest generation that exists for the object in any view, if the object is out of date. Thus, Rev1_Working contains generation 4 out of a possible five generations for Destinations'Body.

With the cursor on the first line in the configuration image, the Common.Expand command displays the second level of configuration image, as shown in Figure 11-2.

```
!Programs.Mail.Mail_Utilities.Configurations.Rev1_Working
  Exports                          :    Lib
    Subset_1'G(2)                  :    In  88/05/11 16:50:07 ANDERSO
  Units                            :    Lib
    Destinations'G(3)              :    In  88/05/11 16:32:48 ANDERSO
    Destinations'G(4/5)            : *  In  88/05/11 16:33:26 ANDERSO
    Lines'G(2)                     :    In  88/02/23 11:42:01 ANDERSO
    Lines'G(4)                     :    In  88/02/23 11:41:51 ANDERSO
    Messages'G(1/2)                : *  In  88/02/25 10:42:15 ANDERSO
    Messages'G(9)                  :    In  88/02/23 11:44:16 ANDERSO
    Symbolic_Display'G(1)          :    In  88/02/23 10:57:31 ANDERSO
    Symbolic_Display'G(3)          :    Out 88/05/20 19:41:59 ANDERSO *current_view*
    To_Do'G(8)                     :    In  88/03/30 11:57:08 ANDERSO
    Unbounded'G(1)                 :    In  88/02/23 10:57:32 ANDERSO

CONFIGURATIONS REV1_WORKING (Cmvc)
```

Figure 11-2. The Second Level of Information in a Configuration Image

Each entry in this expanded configuration image contains the following additional information (from left to right):

- An asterisk indicating whether the object is out of date in the view
- An indication of whether the unit is currently checked out (`Out`) or checked in (`In`); libraries are indicated as `Lib`
- The date and time at which the object was checked out (if the object is currently checked out) or checked in (if the object is currently checked in)
- The user who performed the last checkout or checkin
- The view in which a given object is currently checked out

When objects are checked out or out of date, portions of their entries are underlined, so you can use the Editor.Cursor.Next and Editor.Cursor.Previous commands to move the cursor among these objects.

Using Common.Expand again displays a third level of configuration information, as shown in Figure 11-3. Each entry now displays the reservation token associated with each controlled object in the view.

```
!Programs.Mail.Mail_Utilities.Configurations.Rev1_Working
  Exports                    :
    Subset_1'G(2)            : REV1
  Units                      :
    Destinations'G(3)        : REV1
    Destinations'G(4/5)      : REV1
    Lines'G(2)               : REV1
    Lines'G(4)               : REV1
    Messages'G(1/2)          : REV1
    Messages'G(9)            : REV1
    Symbolic_Display'G(1)    : REV1
    Symbolic_Display'G(3)    : REV1
    To_Do'G(8)               : REV1
    Unbounded'G(1)           : REV1

CONFIGURATIONS.REV1_WORKING.(CMVC)
```

Figure 11-3. The Third Level of Information in a Configuration Image

Finally, using Common.Expand again displays a fourth level of configuration information, as shown in Figure 11-4. At this level, the entry for a given joined object displays the views containing other objects in the join set.

```
!Programs.Mail.Mail_Utilities.Configurations.Rev1_Working
  Exports                    :
    Subset_1'G(2)            : REV1 => Rev1_Working
  Units                      :
    Destinations'G(3)        : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Destinations'G(4/5)      : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Lines'G(2)               : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Lines'G(4)               : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Messages'G(1/2)          : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Messages'G(9)            : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Symbolic_Display'G(1)    : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Symbolic_Display'G(3)    : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    To_Do'G(8)               : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor
    Unbounded'G(1)           : REV1 => Rev1_Sue_Working Rev1_Larry_Working Rev1_Wor

CONFIGURATIONS.REV1_WORKING.(CMVC)
```

Figure 11-4. The Fourth Level of Information in a Configuration Image

Operations in Configuration Images

At any level of expansion, a configuration image provides a convenient way to:

- Check objects in, using the Common.Promote command
- Check objects out, using the Common.Demote command
- Accept changes on objects, using the Common.Complete command
- Access generation and history images, using the Common.Definition and Common.Explain commands, respectively
- Traverse to the designated object in the associated view, using the Cmvc.Def command

A complete list of operations is given in "Commands from Package !Commands-.Common," below.

Restricting Operations in Configuration Images

Operations in configuration images can be restricted using the Edit command. When creating a configuration image, you can set the Allow_Check_Out, Allow-_Check_In, and Allow_Accept_Changes parameters to false to prevent the corresponding operations from accessing objects through the configuration image.

You can also use the Edit command to reset the restrictions on these operations in an existing configuration image. For example, if a checkout operation currently is not permitted in a given configuration image, you can enter the Edit command with Allow_Check_Out set to true.

Alternative Ways of Displaying a Configuration Image

The basic way to create a configuration image is to enter the Edit command from a view or object. Following are two alternative ways of creating a configuration image:

- Within the Configurations directory in a subsystem, put the cursor on the name of a configuration object and enter the Common.Definition command.
- From a generation image (see "Generation Images," below), enter the Common.Enclosing command.

In both of these cases, checkin, checkout, and accept-changes operations are automatically restricted in the configuration image. However, the Edit command can be entered from the existing configuration image to change these restrictions as specified by the Allow_Check_Out, Allow_Check_In, and Allow_Accept_Changes parameters.

Generation Images

Generation images are textual representations of particular generations of controlled objects. Generation images can be displayed even for generations of objects that do not currently exist outside the CMVC database. For example, using generation images, you can browse the text of past generations from configuration-only releases or from code views, which no longer contain source objects. A given generation image can be expanded to show differences between that generation and the previous one. Generation images are available only for controlled objects for which source is saved.

Accessing Generation Images

Generation images can be accessed in several ways. They can be accessed from configuration images:

1. Display the configuration image for a view, code view, or configuration object.
2. With the cursor on the configuration image entry for the desired object, enter the Common.Definition command.

Alternatively, you can access a generation image for a given object directly from view, as follows:

1. Put the cursor on the object's entry in the view.
2. Enter the Cmvc.Def command.

Generation images contain text reconstructed from the CMVC database and does not have the underlying structure of an Ada unit. Therefore, commands such as Common.Object.Parent select text structures such as lines rather than Ada structures. A generation image is identified in the window banner by the generation attribute following the object's name and by the string `(cmvc)`.

Accessing Next and Previous Generation Images

An object's generations form a sequence from the starting generation to the latest generation. When the image of a particular generation is displayed, you can access images for the previous and next generations in the sequence as follows:

- With the cursor in the generation image, enter the Common.Undo command to access the image for the previous generation in the sequence. Repeated uses of Common.Undo iterate toward the starting generation.
- With the cursor in the generation image, enter the Common.Redo command to access the image for the next generation in the sequence. Repeated uses of Common.Redo iterate toward the latest generation.

Displaying the Differences between Consecutive Generations

A given generation image can be expanded to show the differences between it and the previous generation. Enter the Common.Expand command to expand a generation image. (The Common.Elide command removes the differences from the display.) For example, Figure 11-5 shows the result of using Common.Expand in the generation image for generation 4 of Destinations'Body.

```
 |with System_Utilities;
 |with String_Utilities;
 |
 |package body Destinations is
 |
 |    procedure Define (New_User : String) is
 |    begin
 |        [statement]
 |    end Define;
 |
 |    function Image (The_User : User) return String is
 |    begin
-|_       return Unbounded.Image (Unbounded.Variable_String (The_User));
+|        return (Unbounded.Image (Unbounded.Variable_String
+|                                    (The_User.User_Names)));
 |    end Image;
 |
= UNITS.DESTINATIONS'BODY'Diff(3-4) (Cmvc)-  @
```

Figure 11-5. Differences between Generations 3 and 4 of Destinations'Body

Differences are shown on a line-by-line basis:

- A line beginning with the minus sign (−) indicates that the line was deleted from the previous generation.
- A line beginning with the plus sign (+) indicates that the line was added to the previous generation.
- One or more lines beginning with the minus sign immediately followed by one or more lines beginning with the plus sign indicate changed lines.

Regions of difference begin with an underline so that you can use the Editor.Cursor-.Next and Editor.Cursor.Previous commands to move the cursor among such regions.

Other operations available in generation images are listed in "Commands from Package !Commands.Common," below.

History Images

The CMVC database stores history information pertaining to each generation of a controlled object. The history image for a given generation displays this stored information. Figure 11-6 shows the history image for generation 4 of Destinations'Body.

The history image for a generation of an object contains:

- The history for the generation, which lists the time of checkout and checkin and the user who performed these operations
- The notes for the generation, which contains comments provided to various CMVC commands as well as arbitrary commentary associated with that generation

The CMVC database also stores release history for each configuration. Release history contains comments provided through the Cmvc.Release command and also lists the date and time at which spec views and releases were created.

```
History for Units.Destinations'Body

Checked-out on 88/05/20 19:41:59 by ANDERSON
Checked-in  on 88/05/24 14:58:46 by ANDERSON

Notes for Generation 4
CHECK_OUT: Changing return statement in function image.
CHECK_IN:  Change has been tested
-- Notes from 88/05/24 15:06:19 by ANDERSON --
Still need to implement procedure Define.
--

-- UNITS.DESTINATIONS'BODY'G:4 | History | CMVC |
```

Figure 11-6. The History Image for Generation 4 of Destinations'Body

Accessing History Images

History images can be accessed in several ways. They can be accessed from configuration images:

1. Display the configuration image for a view, code view, or configuration object.
2. With the cursor on the configuration image entry for the desired object, enter the Common.Explain command.

   If the cursor is on the header line of a configuration image, then Common.Explain displays the release history for the configuration.

   If the cursor is on an underline other than the header line, an explanation of the underline is displayed in the Message window. Move the cursor off the underline to display a history window.

History images also can be accessed from generation images:

1. With the cursor in the appropriate generation image, enter the Common.Explain command.

Finally, a history image for a given object can be accessed directly, as follows:

1. Put the cursor on the object or on its directory entry.
2. Enter the Cmvc.Notes command.

Displaying History from Other Generations

From a history image, the Common.Undo and Common.Redo commands iterate through history images of the previous and next generations, respectively.

Furthermore, using the Common.Expand command in a history image displays the cumulative history and notes for a range of previous generations within the same image. The number of previous generations for which history is displayed

is determined by the Repeat parameter of the Common.Expand command. The Common.Elide command reduces the amount of cumulative history by the number of generations specified by its Repeat parameter.

Managing Notes through History Images

History images provide an interactive way to manage notes. From a history image, new notes can be added and saved. The Common.Edit command displays a prompt in which additional notes can be entered. The Common.Save or Common.Commit commands save the new notes in the CMVC database.

The window banner for a history image contains the object name followed by a generation attribute (for example `'G(3)`), followed by the attribute `'History`. Furthermore, the window banner contains the string `(cmvc)`.

Traversing between Library and CMVC Images

Subsystems, views, configuration objects, and objects such as files and Ada units are all part of the Environment library system. Associated with these library objects are configuration images, generation images, and history images, which display information managed by the CMVC database.

As shown in Figure 11-7, the Cmvc.Edit, Cmvc.Def, and Cmvc.Notes commands traverse between objects in the library system and images managed by CMVC. Commands from package Common traverse among images within each group.

Figure 11-8 shows the use of Common.Undo and Common.Redo to access generation images for different generations of the same object.

Session Switches

A number of session switches have names that begin with the prefix "Cmvc_". All but one of these pertain to objects that are managed by commands in package Work_Order and are documented in that package. The remaining switch, Cmvc_Enable_Relocation, is for use by Rational personnel only.

Commands from Package !Commands.Common

Commands from Package Common in Configuration Images

procedure Common.Complete

Equivalent to entering the Accept_Changes command to update the designated object (or the objects in the designated configuration) to the latest generation. The Accept_Changes operation is performed with default parameter values, except that Allow_Demotion has the value true. The configuration image is updated to reflect the operation. The operation performed by the command is subject to restriction according to the Allow_Accept_Changes parameter of the Cmvc.Edit command.

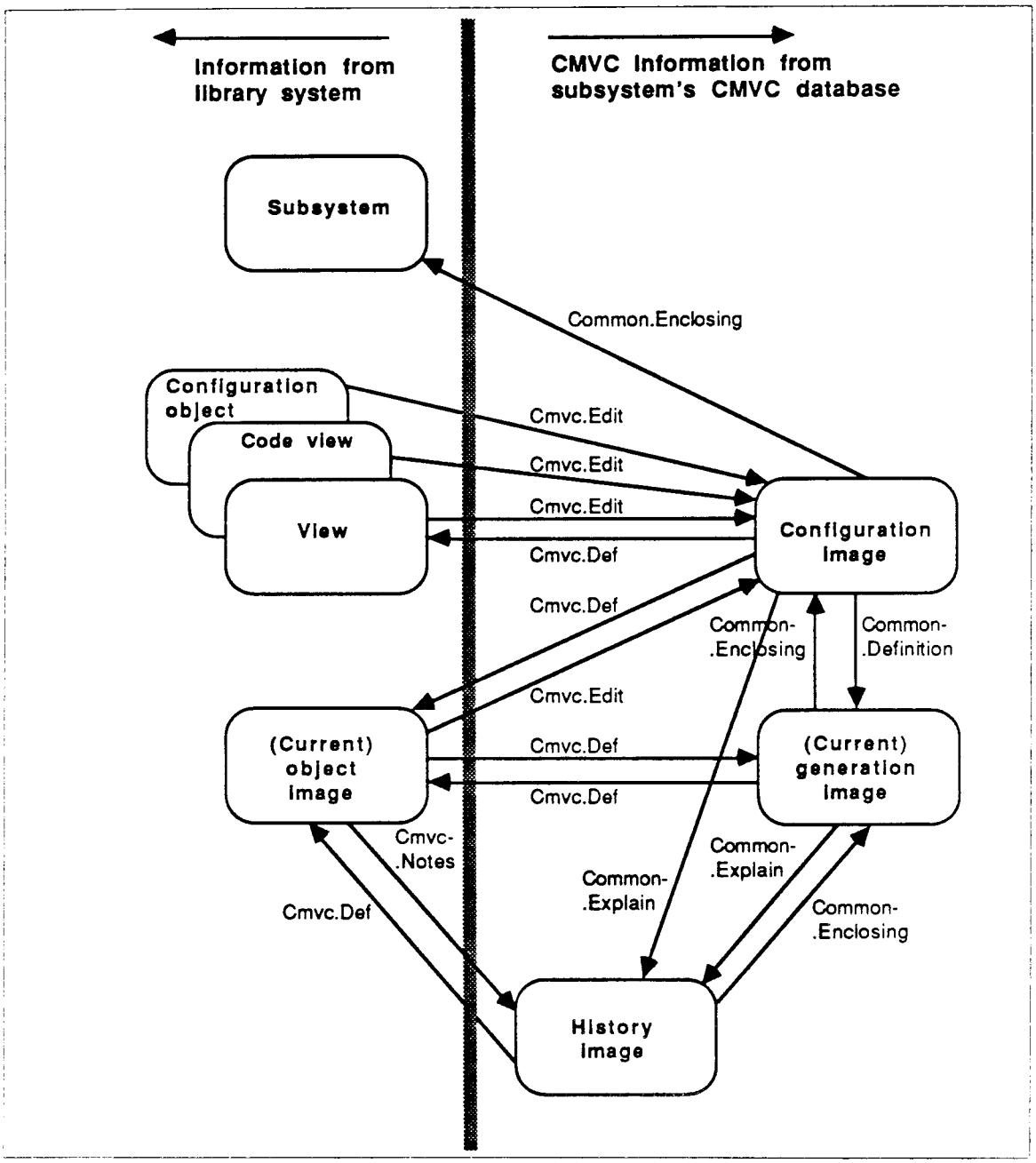


Figure 11-7. Summary of Traversal Commands

procedure Common.Definition

Displays the generation image for the current generation of the object whose entry is designated in a configuration image. An In_Place parameter specifies whether the current frame should be used.

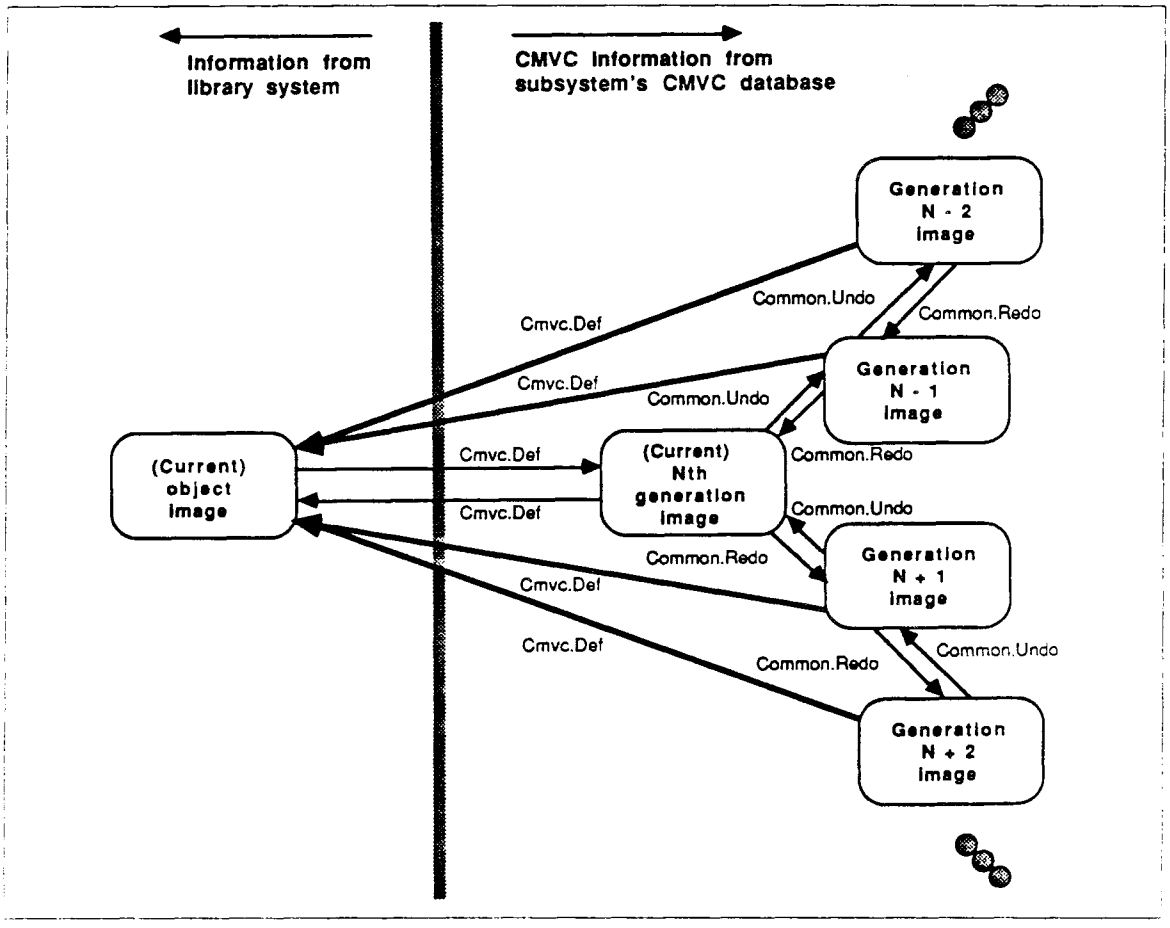


Figure 11-8. Traversing between Generation Images

procedure Common.Demote

Equivalent to entering the Check_Out command to check out the designated object (or the objects in the designated configuration). The Check_Out operation is performed with default parameter values, except that Allow_Demotion has the value true. The configuration image is updated to reflect the operation. The operation performed by the command is subject to restriction according to the Allow_Check_Out parameter of the Cmvc.Edit command.

procedure Common.Edit

Checks out the object whose entry is designated in the configuration image and then displays the object. The object is not opened for editing, in case it is an Ada unit to which you want to make incremental changes. The Check_Out operation is performed with default parameter values, except that Allow_Demotion has the value true. The configuration image is updated to reflect the operation. The operation performed by the command is subject to restriction according to the Allow_Check_Out parameter of the Cmvc.Edit command.

procedure Common.Elide

Reduces the level of information displayed in the configuration image. As designated by the cursor, the level can be reduced for an individual entry or for the entire image (the cursor must be on the top header line of the image). See "Levels of Information in Configuration Images," above.

procedure Common.Enclosing

Displays the subsystem that contains the configuration represented in the configuration image. An In_Place parameter specifies whether the current frame should be used.

procedure Common.Explain

Displays the history image for the generation of the designated object. If the cursor is on the top header line in the configuration image, release history for the configuration is displayed. If the cursor is on an underline (other than the header), an explanation of the underline is given.

procedure Common.Expand

Increases the level of information displayed in the configuration image. As designated by the cursor, the level can be increased for an individual entry or for the entire image (the cursor must be on the top header line of the image). See "Levels of Information in Configuration Images," above.

procedure Common.Format

Updates the configuration image with current information from the CMVC database. Note that the configuration image is updated automatically after Common.Promote, Common.Demote, or Common.Complete is executed, but it is not updated when the CMVC database is changed by any other operation.

procedure Common.Promote

Equivalent to entering the Check_In command to check in the designated object (or the objects in the designated configuration). The configuration image must have a view associated with it. The Check_In operation is performed with default parameter values. The configuration image is updated to reflect the operation.

procedure Common.Revert

Updates the configuration image with current information from the CMVC database. Note that the configuration image is automatically updated after Common.Promote, Common.Demote, or Common.Complete is executed, but it is not updated when the CMVC database is changed by any other operation.

Commands from Package Common in Generation Images

procedure Common.Definition

Displays the controlled object associated with the generation in the generation image containing the cursor. An In_Place parameter specifies whether the current frame should be used.

procedure Common.Elide

Removes from the generation image the differences that were displayed by the Common.Expand command.

procedure Common.Enclosing

Displays the configuration image for the last configuration that was visited. An In_Place parameter specifies whether the current frame should be used.

procedure Common.Explain

Displays the history image for the generation represented in the current generation image. If the cursor is on an underline, an explanation of the underline is given.

procedure Common.Expand

Displays the differences between the generation in the generation image and the previous generation. Differences are shown on a line-by-line basis. Lines beginning with a minus sign (−) indicate lines deleted from the previous generation. Lines beginning with a plus sign (+) indicate lines added to the previous generation. The start of each difference region is underlined.

procedure Common.Redo

Displays the generation following the generation represented in the current generation image. A Repeat parameter specifies which succeeding generation is displayed, relative to the currently displayed generation.

procedure Common.Undo

Displays the generation previous to the generation represented in the current generation image. A Repeat parameter specifies which preceding generation is displayed, relative to the currently displayed generation.

Commands from Package Common in History Images

procedure Common.Commit

Saves the new notes entered through the prompt given by the Common.Edit command.

procedure Common.Definition

Displays the controlled object associated with the generation for which history is displayed. An In_Place parameter specifies whether the current frame should be used.

procedure Common.Edit

Provides a prompt in the current history image in which new notes can be entered.

procedure Common.Elide

Reduces the cumulative history that is displayed in the current history image. The Repeat parameter specifies the number of generations by which the cumulative history should be reduced.

procedure Common.Enclosing

Displays the generation image for the generation associated with the current history image. An In_Place parameter specifies whether the current frame should be used.

procedure Common.Expand

Expands the cumulative history that is displayed in the current history image. The Repeat parameter specifies the number of generations by which the cumulative history should be increased.

procedure Common.Format

Updates the history image with current information from the CMVC database.

procedure Common.Promote

Saves the new notes entered through the prompt given by the Common.Edit command.

procedure Common.Redo

Displays the history image for the generation following the current generation. A Repeat parameter specifies which succeeding history image is displayed, relative to the generation of the current history image.

procedure Common.Revert

Updates the history image with current information from the CMVC database.

procedure Common.Undo

Displays the history image for the generation previous to the current generation. A Repeat parameter specifies which preceding history image is displayed, relative to the generation of the current history image.

procedure Abandon_Reservation

```
procedure Abandon_Reservation
      (What_Object           : String                    := "<SELECTION>";
       Allow_Demotion        : Boolean                   := False;
       Remake_Demoted_Units  : Boolean                   := True;
       Goal                  : Compilation.Unit_State    := Compilation.Coded;
       Comments              : String                    := "";
       Work_Order            : String                    := "<DEFAULT>";
       Response              : String                    := "<PROFILE>");
```

Description

Abandons the reservation on one or more checked-out objects, effectively canceling the checkout of those objects.

Any changes made during the canceled checkout are discarded.

When an object is checked out, a new generation is created. The Abandon_Reservation procedure cancels this newly created generation and causes the object to revert to the last checked-in generation.

Note that checking out an object automatically updates that object to the latest checked-in generation, accepting changes as necessary. This procedure does not undo the implicit accept changes, so the object remains at the latest generation.

Parameters

```
What_Object : String := "<SELECTION>";
```

Specifies the object or objects whose reservations are to be abandoned. Objects that are not checked out are ignored. The default is the currently selected object. View names cannot be specified.

Multiple objects must be in the same view. Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Allow_Demotion : Boolean := False;
```

Specifies whether the Abandon_Reservation procedure is allowed to demote Ada units in the process of reverting units to the last checked-in generation.

If the Allow_Demotion parameter is true, the Abandon_Reservation procedure is permitted to demote Ada units if necessary. If this parameter is false, the command can proceed only if no demotion is required; otherwise, an error is reported and the command quits.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted in the process of reverting units to the last checked-in generation.

If true (the default value), demoted units are recompiled to the state specified by the Goal parameter. If false, units remain demoted.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, the demoted units are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified, and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date on which the reservation was abandoned, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment is also entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Check_Out

---

procedure Accept_Changes

```
procedure Accept_Changes
      (Destination          : String                    := "<CURSOR>";
       Source               : String                    := "<LATEST>";
       Allow_Demotion       : Boolean                   := False;
       Remake_Demoted_Units : Boolean                   := True;
       Goal                 : Compilation.Unit_State    := Compilation.Coded;
       Comments             : String                    := "";
       Work_Order           : String                    := "<DEFAULT>";
       Response             : String                    := "<PROFILE>");
```

Description

Updates the object(s) specified in the Destination parameter to the generation(s) indicated by the Source parameter; that is, the destination objects are changed to reflect any modifications that have been made to the corresponding source objects.

When changes to individual Ada units are being accepted, unit specifications should be updated before their corresponding bodies to ensure that the units compile correctly.

Typically, the Accept_Changes procedure is used to update each destination object to the latest generation. The procedure thus is a means of synchronizing the development of controlled objects that are joined to objects in other views. When an object in a join set is checked out and then checked in, a new generation is created, rendering the other objects in the set at least one generation out of date. This procedure can be used on the out-of-date objects to update them to the latest generation. (Checking out an out-of-date object implicitly accepts changes.)

The Accept_Changes procedure also can be used to "go backward in time." If the name of a previous configuration is given as the Source parameter, the objects specified by the Destination parameter are changed to the generations given in that configuration. Unless such objects are subsequently severed, however, checking them out automatically updates them to the latest generation.

The Accept_Changes procedure also can be used to copy new controlled objects from the source view into the destination view. This is more effective than using Library.Copy to propagate new objects across views.

If Ada units are compiled against a specified unit, accepting changes to that unit may require the demotion of the other dependent units. The value of the Allow_Demotion parameter controls whether the command actually performs the demotion and updates the unit.

The configuration image displayed by the Edit command uses an asterisk to indicate objects that require updating. Alternatively, the Show_Out_Of_Date_Objects command can be used to determine the objects that may require updating.

Parameters

```
Destination : String := "<CURSOR>";
```

Specifies one or more objects to be updated. If multiple objects are named, they must be in the same view. A view name can be used to specify all the objects in that view. The default is the object on which the cursor is located.

Destination objects must be controlled. If uncontrolled objects are named, they are noted in the output log generated by the command. If a destination object is checked out, it is not updated and a warning message is issued.

If a destination object was made controlled without saving source, the object can be updated only if the Source parameter names an object that exists in some view. (For example, when updating such a destination object, the Source parameter may not name a configuration object that has no view associated with it.)

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Source : String := "<LATEST>";
```

Specifies the object(s) to which the corresponding destination object(s) are updated. The Source parameter can be the special name "<LATEST>", the name of one or more objects, a view name, or a configuration name. Note that subdirectories (for example, Units) are not accepted as object names; instead, you must use naming expressions that resolve to the contents of such subdirectories (for example, Units.@).

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

The Source and Destination parameters interact as follows:

- Source =>"<LATEST>"

  When the Source parameter is the default special name "<LATEST>", the Destination parameter can name a set of objects or a view. Each destination object is updated to the most recently checked-in generation. If the most recent generation is currently checked out, the previous generation is used and a warning message appears in the output log.

- Source => *object(s) or view*

  When the Source and Destination parameters name a set of objects or a view, each source object must have a more recent generation than the corresponding destination object. (The source object can, but need not, have the latest generation.) If the destination is more recent than the source, the destination is not changed and a warning note appears in the output log.

As before, if a source object is currently checked out, the most recent *checked-in* generation is used, and a warning message appears in the output log.

The following combinations of Source and Destination parameters are permitted. Note that when the Source parameter names a set of objects, the Destination parameter must name a view.

— Source => *object(s)*; Destination => *view*: If the Source parameter names a set of objects and the Destination parameter names a view, the command updates the objects in the destination view that correspond to the source objects. If Source names controlled objects that are new, these objects are copied into the Destination view, where they are made controlled and joined to the original source objects.

— Source => *view*; Destination => *object(s)*: If the Source parameter names a view and the Destination parameter names a set of objects, the command updates the destination objects to match the corresponding objects in the source view.

— Source => *view*; Destination => *view*: If the Source and Destination parameters each name a view, the destination view is made to look like the source view. Every controlled object in the source view updates the corresponding object in the destination view. New controlled objects in the source view are copied into the destination view. The copied objects are automatically controlled and joined to the corresponding source-view objects.

- Source => *configuration*

  When the Source parameter names a configuration, the Destination parameter can name a set of objects or a view. The command causes each destination object to have the generation of the corresponding object in the specified configuration. Consequently, naming an older configuration causes the destination objects to "go back in time" to earlier generations.

  Naming a source configuration is the same as naming a view, except that naming a view always updates destination objects to more recent generations, whereas naming a configuration can change the destination objects to older generations. (The name of a previously released view cannot be used in place of a configuration in order to go back in time.)

  Note that changing a destination object to an older generation does not cause that generation to become the latest one (see the Revert command). Checking out such an object updates it to the latest generation.

```
Allow_Demotion : Boolean := False;
```

Specifies whether the Accept_Changes procedure should be allowed to demote Ada units in order to update the specified destination objects.

If the Allow_Demotion parameter is true, the Accept_Changes procedure is permitted to demote Ada units if necessary. If this parameter is false, the command proceeds only if no demotion is required; otherwise, an error is reported and the command quits.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted in the process of updating the destination objects.

If true (the default value), demoted units are recompiled to the state specified by the Goal parameter. If false, units remain demoted.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, the demoted units are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified, and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date on which changes were accepted, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment is also entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Errors

An error is noted in the output if an attempt is made to accept changes into a previous generation by naming objects other than a configuration.

References

procedure Revert

procedure Show_Out_Of_Date_Objects

---

procedure Append_Notes

```
procedure Append_Notes (Note        : String := "<WINDOW>";
                        What_Object : String := "<CURSOR>";
                        Response    : String := "<PROFILE>");
```

Description

Appends the specified string to the end of the notes for the specified controlled object.

The notes for a controlled object are stored in the CMVC database. An object's notes can be used as a scratchpad for arbitrary commentary to be associated with particular generations.

The contents of a file can be appended by specifying the filename as an indirect file.

Append_Notes is one of a set of file-oriented commands for managing notes. That is, these commands, including Get_Notes, Create_Empty_Note_Window, and Put_Notes, are most useful for managing notes through files. However, these commands also manage special-purpose notes windows (identified by the `Notes for` string in the banner) in which the Append_Notes command can be used as follows:

- If the Create_Empty_Note_Window procedure has been used to display an empty notes window for an object, text entered in this window can be appended to the object's existing notes using the Append_Notes procedure. In this case, Append_Notes must be entered (with default parameter values) from a Command window attached to the window that was created by the Create_Empty_Note_Window procedure.

Note that modified notes windows retain the * symbol in their window banners, even after their contents have been entered in the CMVC database using Append_Notes or Put_Notes. Accordingly, the Quit command reports these windows as changed images when logout is attempted. Because these windows cannot be committed, use the Common.Abandon procedure to remove these windows.

The Notes command provides an interactive alternative to Create_Empty_Note_Window, Append_Notes, and the like. The Notes command displays a history image (identified by `'History` attribute following the object name and generation in the window banner), which allows interactive operations for managing an object's notes.

Parameters

```
Note : String := "<WINDOW>";
```

Specifies a string to be appended to an object's existing notes. If the Note parameter names an indirect file, the contents of that file are appended to the existing notes for the specified controlled object.

If the default special name "<WINDOW>" is used, it refers to the contents of a notes window created by either the Get_Notes or the Create_Empty_Note_Window command. When the default value is used, Append_Notes must be entered from a Command window attached to the notes window. The first line of the notes window contains the name of the object associated with the notes; therefore, the What_Object parameter is ignored.

```
What_Object : String := "<CURSOR>";
```

Specifies the object whose notes are to be augmented. The specified object must be both controlled and checked out; otherwise, the command quits.

The What_Object parameter is ignored if the Note parameter's default value is used.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Create_Empty_Note_Window

procedure Get_Notes

procedure Notes

procedure Put_Notes

procedure Build

```
procedure Build
     (Configuration   : String                    := ">>CONFIGURATION NAME<<";
      View_To_Import  : String                    := "<INHERIT_IMPORTS>";
      Model           : String                    := "<INHERIT_MODEL>";
      Goal            : Compilation.Unit_State    := Compilation.Installed;
      Limit           : String                    := "<WORLDS>";
      Comments        : String                    := "";
      Work_Order      : String                    := "<DEFAULT>";
      Volume          : Natural                   := 0;
      Response        : String                    := "<PROFILE>");
```

Description

Builds views from the specified configuration objects.

Views corresponding to the specified configuration objects must not already exist.

Whenever a view is created or released, a configuration object is created for it automatically. The configuration object for a view lists the specific generations of the controlled objects in that view and provides an index into the CMVC database where the source for these generations is stored. Thus, views are realizations of configuration objects, in that views contain library structure and compilable units, whereas configuration objects merely summarize the contents of the corresponding views.

Because configuration objects provide enough information to reconstruct views, space can be saved by creating or keeping only the configuration objects for views whose units do not need to be compiled and executed frequently:

- The Release command creates only a configuration object without creating the corresponding released view if the Create_Configuration_Only parameter is true.
- The Destroy_View command destroys only a view without destroying the corresponding configuration object if the Destroy_Configuration_Also parameter is false.

The Build command is used when it is necessary to build a view for a released configuration object or rebuild a destroyed view.

Note that when a view is built (or rebuilt) from a configuration object, the only objects that can be recreated are controlled objects for which source is saved in the CMVC database. (Controlled objects for which source is not saved cannot be rebuilt.)

Configuration objects reside in the directory *subsystem_name*.Configurations. Each configuration object has the same simple name as the view to which it corresponds.

Parameters

```
Configuration :  String := ">>CONFIGURATION NAME<<";
```

Specifies one or more configuration objects from which views are to be built. If the command is executed in the subsystem library, the configuration names can be specified using relative naming—for example, Configurations.Rev1_0_1 names the configuration for which the corresponding view Rev1_0_1 is built.

Multiple configuration objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
View_To_Import :  String := "<INHERIT_IMPORTS>";
```

Specifies one or more views to be imported by each of the newly built views. The views specified by this parameter must be spec or combined views.

If the View_To_Import parameter is the default special name "<INHERIT_IMPORTS>", imports are determined by information in the state description directory associated with each configuration object. (State description directories are created automatically for released views and are named *subsystem*.Configurations.*release_name*_State.)

If the View_To_Import parameter is the null string (""), no views are imported.

If the View_To_Import parameter specifies one or more views, only the specified views are imported, and any imports listed in a corresponding state description directory are ignored.

The Imported_Views function can be used to return another view's imports as the value of the View_To_Import parameter. This is a convenient way of setting the newly built view's imports to be the same as another view's imports.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, the View_To_Import parameter can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Model : String := "<INHERIT_MODEL>";
```

Specifies a model world for each newly built view. If the specified name cannot be resolved in the context !Model, the name is resolved relative to the current context.

If the Model parameter is the default special name "<INHERIT_MODEL>", each newly built view uses the model that was recorded in the state description directory associated with the relevant configuration object. (State description directories are created automatically for released views and are named *subsystem*.Configurations-.*release_name*_State.)

```
Goal : Compilation.Unit_State := Compilation.Installed;
```

Specifies the state to which units in the view are compiled. The goal can be any of the enumerations of the Compilation.Unit_State type. By default, the compilation goal is the installed state. To set the compilation goal to the coded state, specify Compilation.Coded.

```
Limit : String := "<WORLDS>";
```

Specifies the units that can be compiled to the state specified by the Goal parameter. Because views are worlds, the default special value "<WORLDS>" means that only units within the newly built views can be compiled. Other values for this parameter are given as enumerations of the Compilation.Change_Limit subtype. For example, the string "<ALL_WORLDS>" permits the compilation of units in other subsystems in order to compile the units in the newly built views.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified, and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of the build operation as well as the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to build the views. The default value specifies that the views should be built on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Destroy_View

procedure Release

---

procedure Check_In

```
procedure Check_In (What_Object  : String := "<CURSOR>";
                    Comments     : String := "";
                    Work_Order   : String := "<DEFAULT>";
                    Response     : String := "<PROFILE>");
```

Description

Releases the reserved right to update the specified object or set of objects and stores the text of the new generation(s) in the CMVC database.

An object that is checked in cannot be modified until it is checked out again. Only controlled objects can be checked in or out.

Because checked-in objects cannot be modified in any way, it is recommended that all incremental additions or changes to Ada units be promoted before those units are checked in. Errors will result from attempting to compile the checked-in units that contain insertion points, because promoting insertion points would require the modification of checked-in units.

Note that checking in an object that was made controlled without saving source simply releases the right to update that object; no text is recorded in the CMVC database.

Parameters

```
What_Object :  String := "<CURSOR>";
```

Specifies one or more objects to be checked in. These objects must be controlled. If uncontrolled objects are named, they are noted in the output log generated by the command and ignored.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Comments :  String := "";
```

Specifies a comment to be stored in the CMVC database with the notes for the specified object(s). Notes can be displayed using the Get_Notes command. This comment also appears in the display generated by the Show_History_By_Generation command.

In addition, the specified comment is logged to the work order specified by the Work_Order parameter.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment is also entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Restrictions

The Check_In procedure takes time proportional to the size of the object(s) being checked in. Large objects may exhaust the job page limit and fail. In this case, the job page limit must be increased.

References

procedure Check_Out

procedure Make_Controlled

---

procedure Check_Out

```
procedure Check_Out
  (What_Object                    : String                    := "<CURSOR>";
   Comments                       : String                    := "";
   Allow_Implicit_Accept_Changes  : Boolean                   := True;
   Allow_Demotion                 : Boolean                   := False;
   Remake_Demoted_Units           : Boolean                   := True;
   Goal                           : Compilation.Unit_State    :=
                                                    Compilation.Coded;
   Expected_Check_In_Time         : String                    := "<TOMORROW>";
   Work_Order                     : String                    := "<DEFAULT>";
   Response                       : String                    := "<PROFILE>");
```

Description

Reserves the right to modify the specified controlled object or objects.

Controlled objects can be modified only while they are checked out. However, objects need not be checked out in order to be compiled.

When objects are joined across multiple views, they share the same reservation token, so that only one of the joined objects can be checked out at a time. Checking out a joined object in one view renders the corresponding objects in the other views unavailable for update. (In contrast, objects that do not share the same reservation token can be checked out and modified independently.)

A new generation of an object is created when it is checked out. The new generation can be preserved by the Check_In command or abandoned by the Abandon_Reservation command. When one object in a join set is checked out and then checked in, the other objects in the set are rendered at least one generation out of date. Checking out one of the out-of-date objects automatically updates it to the latest generation, unless the Allow_Implicit_Accept_Changes parameter has been set to false, in which case the checkout operation fails. Setting this parameter to false allows an object to be checked out only if it is at the latest generation already. (Note that if an object was made controlled without saving source, Check_Out can implicitly update it only if an object in some view actually contains the latest generation; see the Make_Controlled command.)

If Ada units are compiled against a unit that requires updating, checking out that unit may require the demotion of the other dependent units. In this case, the value of the Allow_Demotion parameter controls whether the command actually performs the demotion and checks out the unit.

Various commands can be used to determine whether objects are currently checked out, including Show, Show_All_Checked_Out, Show_Checked_Out_In_View, and Show_Checked_Out_By_User. Other related information, such as the checkout date, time, and user, can be displayed using the Show_History_By_Generation command.

The reservation obtained by the Check_Out procedure can be abandoned using the Abandon_Reservation command.

Parameters

```
What_Object : String := "<CURSOR>";
```

Specifies one or more objects to be checked out. If multiple objects are specified, all must belong to the same view. If a view name is specified, the Check_Out procedure attempts to check out all the objects in the view.

If the Check_Out procedure encounters an object that is checked out in another view, an error is reported at that point and the command quits without looking at any more specified objects. Checkouts made before the command quits are abandoned.

Objects must be controlled to be checked out. If uncontrolled objects are named, they are noted in the output log generated by the command.

If multiple objects are specified, they must be in the same view. Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Comments : String := "";
```

Specifies a comment to be stored in the CMVC database with the notes for the specified object(s). The notes can be displayed using the Get_Notes command. This comment also appears in the display generated by the Show_History_By_Generation command.

In addition, the specified comment is logged to the work order specified by the Work_Order parameter.

```
Allow_Implicit_Accept_Changes : Boolean := True;
```

Specifies whether the Check_Out procedure is allowed to update the specified objects to the latest generation.

If this parameter is true, the Check_Out procedure is permitted to update the objects. If it is false, the command proceeds only if the specified objects are already at the latest generation; otherwise, an error is reported and the command quits.

```
Allow_Demotion : Boolean := False;
```

Specifies whether the Check_Out procedure is allowed to demote Ada units in order to update the specified objects to the latest generation.

If this parameter is true, the Check_Out procedure is permitted to demote Ada units if necessary. If it is false, the command proceeds only if no demotion is required; otherwise, an error is reported and the command quits.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted in the process of updating the specified objects to the latest generation.

If true (the default value), demoted units are recompiled to the state specified by the Goal parameter. If false, units remain demoted.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, the demoted units are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Expected_Check_In_Time : String := "<TOMORROW>";
```

Specifies the anticipated date and time at which the objects will be checked in. The value of this parameter can be any string accepted by the !Tools.Time_Utilities.Value function (documented in PT). The default value, "<TOMORROW>", supplies the date and time for the next day. The expected checkin time can be viewed using commands such as Show.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkout, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment is also entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Abandon_Reservation

procedure Check_In

procedure Show

procedure Show_All_Checked_Out

procedure Show_Checked_Out_By_User

procedure Show_Checked_Out_In_View

procedure Show_History_By_Generation

---

procedure Copy

```
procedure Copy
   (From_View                 : String                    := "<CURSOR>";
    New_Working_View          : String                    := ">>SUB/PATH NAME<<";
    View_To_Modify            : String                    := "";
    View_To_Import            : String                    := "<INHERIT_IMPORTS>";
    Only_Change_Imports       : Boolean                   := True;
    Join_Views                : Boolean                   := True;
    Reservation_Token_Name    : String                    := "<AUTO_GENERATE>";
    Construct_Subpath_Name    : Boolean                   := False;
    Create_Spec_View          : Boolean                   := False;
    Create_Load_View          : Boolean                   := False;
    Create_Combined_View      : Boolean                   := False;
    Level_For_Spec_View       : Natural                   := 0;
    Model                     : String                    := "<INHERIT_MODEL>";
    Remake_Demoted_Units      : Boolean                   := True;
    Goal                      : Compilation.Unit_State    := Compilation.Coded;
    Comments                  : String                    := "";
    Work_Order                : String                    := "<DEFAULT>";
    Volume                    : Natural                   := 0;
    Response                  : String                    := "<PROFILE>");
```

Description

Creates one or more new views by copying the specified view or views.

By default, the Copy command makes new spec views, new working load views, or new working combined views, depending on the kinds of source views named by the From_View parameter. This procedure also can be used to make views of a specific type, depending on the values of the Create_Spec_View, Create_Load_View, and Create_Combined_View parameters. (At most, only one of these three parameters can be true.)

The Copy command can be used to make new paths, subpaths, and spec views, although specialized commands (Make_Path, Make_Subpath, and Make_Spec_View) exist for this purpose. (Note that all of the special-purpose commands call the Copy command.)

Objects in new working load or combined views are made controlled if the corresponding objects were controlled in the source views. Objects in new spec views are left uncontrolled.

Controlled objects in a new view can, but need not, be joined to the corresponding objects in the view from which it is copied. Two views should be joined (using the Join_Views parameter) if the majority of the controlled objects in them are to be joined. (Joined objects cannot be checked out and modified independently.) The controlled objects that need to be modified independently can be severed subsequently with the Sever command.

A new view should not be joined to the view from which it is created if most of the controlled objects in these two paths are to be modified independently. (Note that changes can be propagated across unjoined objects with the Merge_Changes command.) Although the new path is not created joined, individual objects in it subsequently can be joined to the corresponding objects in other views (see the Join command).

By default, each new view has the same imports as the view from which it was copied. It is also possible to specify different imports in the process of creating the new paths by using the View_To_Import and Only_Change_Imports parameters. Import adjustments are subject to the same consistency checking that is performed by the Import command.

Parameters

```
From_View :  String := "<CURSOR>";
```

Specifies the source view or views from which copies are to be made. The default is the view on which the cursor is located. This parameter can name:

- Combined, load, or spec views
- Working or released views

All controlled objects in a view named by the From_View parameter must be checked in. If the parameter names multiple views, a new view is copied from each of the named views. Each new view is created in the same subsystem as the view from which it is copied.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. The named views can be in the same or in different subsystems. (For further information, see "Naming" in the Key Concepts in this book.)

```
New_Working_View :  String := ">>SUB/PATH NAME<<";
```

Specifies the string to be used in constructing the names of the new views.

The string specified by the New_Working_View parameter is used in several ways, depending on the values of the Construct_Subpath_Name and Create_Spec_View parameters:

- If both Create_Subpath_Name and Create_Spec_View are false, the string specified by New_Working_View is used as a pathname prefix for a working view—for example, Rev2 in Rev2_Working.
- If Create_Subpath_Name is true, the string specified by New_Working_View is used as a subpathname extension for a working view—for example, Anderson in Rev1_Anderson_Working. The new name is constructed using the pathname (Rev1) from the source view.
- If Create_Subpath_Name is false and the New_Working_View string contains an underscore, the string specified by New_Working_View is used as both the pathname prefix and subpath extension—for example, Rev2_Miyata in Rev2_Miyata_Working.
- If Create_Spec_View is true, the string specified by New_Working_View is used as a spec view prefix—for example, Rev1 in Rev1_0_Spec.

Other portions of the constructed names, such as _Working and _0_Spec, are supplied automatically.

If the From_View parameter names multiple views, all of the new views will use the same name prefix or extension.

New_Working_View can be any string that constitutes a legal Ada identifier. Note that a string containing an underscore is interpreted as a path prefix followed by a subpath extension and not merely as a path prefix containing an underscore. This has consequences for subsequent CMVC operations. For example, if the New_Working_View parameter specifies the string "Target_2" and the Construct_Subpath_Extension parameter is false, the Copy command creates a view named Target_2_Working. If another subpath view is subsequently created from this view, the string "2" will be replaced by the new subpath extension.

```
View_To_Modify : String := "";
```

Specifies one or more spec, load, or combined views whose imports should be changed to refer to the new views, provided that those new views are combined or spec views. The imports of the views specified by this parameter also are updated using the views named by the View_To_Import parameter. The View_To_Modify views are updated by View_To_Import views as if Only_Change_Imports were true, regardless of this parameter's actual value.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
View_To_Import : String := "<INHERIT_IMPORTS>";
```

Specifies one or more spec or combined views to be imported by the new views. The views named by this parameter also are used to update the imports of the views named by the View_To_Modify parameter.

If View_To_Import specifies the default special name "<INHERIT_IMPORTS>", each new view uses the same imports as the view from which it was copied. (However, if the From_View parameter names multiple combined views among which import relations hold, the imports are automatically adjusted so that the working views in the new paths reference each other as appropriate, instead of referencing the working views in the original paths.)

If View_To_Import specifies the null string (""), no views are imported.

If View_To_Import specifies one or more views, the specified views are imported by the new views in the manner specified by the Only_Change_Imports parameter.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, View_To_Import can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Only_Change_Imports : Boolean := True;
```

Specifies the manner in which the views specified by the View_To_Import parameter are actually used as imports by the new views. Only_Change_Imports has no effect if View_To_Import specifies "<INHERIT_IMPORTS>" or the null string.

If this parameter is false, the entire list of views given by View_To_Import is imported by each new view created by the Copy command. No imports are inherited.

If the parameter is true (the default value):

- Each new view inherits its imports from the view from which it was copied.
- The list of views in View_To_Import is compared to the inherited views. If a View_To_Import view is from the same subsystem as an inherited view, the View_To_Import view replaces that inherited view.

Thus, if Only_Change_Imports is true, the list of views in View_To_Import is used to update the inherited imports of each new view. In this way, the replacement imports for every new view can be specified in a single list without forcing each new view to import everything in the list.

```
Join_Views : Boolean := True;
```

Specifies whether to join each new view to the view from which it was copied. Only new working views (either load or combined) are joined. (That is, the value of Join_Views is ignored if the Create_Spec_View parameter is true.)

If Join_Views is true (the default value), the controlled objects in each copied working view are joined to the corresponding objects in each source view named by the From_View parameter. The reservation tokens from the source views are used. If a source view contains no controlled objects, then no objects can be joined. Note that Join_Views affects only controlled objects that exist at the time the Copy command is executed. Objects created after the new views are made must be controlled explicitly and joined using the Make_Controlled and Join commands.

If Join_Views is false, new reservation tokens are created for all of the controlled objects. The value for New_Working_View is used as the reservation token, unless Reservation_Token_Name specifies a nonnull value.

```
Reservation_Token_Name : String := "<AUTO_GENERATE>";
```

Specifies the name of the reservation token to be associated with each specified object. The value of this parameter is used only if the Join_Views and Create_Spec_View parameters are false.

The default value "<AUTO_GENERATE>" means that the reservation token is generated automatically by the Environment. Names of reservation tokens that are generated automatically are derived from the first portion of the enclosing view name (up to the first underscore character). For example, the controlled objects in a view called Rev1_Working would have Rev1 as their automatically generated token name. (Where necessary, a number is appended to produce a unique name for the reservation token—for example, Rev1_1.)

A user-defined token name can be supplied instead to provide subsequent join sets with more meaningful or mnemonic token names.

Note that supplying an existing token name cannot be used to join the newly controlled objects to any other objects.

```
Construct_Subpath_Name : Boolean := False;
```

Specifies whether each new view should be named as a subpath of the corresponding source view.

If true, the string specified by the New_Working_View parameter is used as a subpathname extension in each new view's name. Each new view name is constructed from the pathname prefix of the source view followed by the string specified by New_Working_View. The string "_Working" is automatically added to the name's end. For example, if From_View names a source view called "Rev1_4_5" and New_Working_View specifies the string "Anderson", then setting Construct_Subpath_Name to true causes the new view to be called Rev1_Anderson_Working.

If false (the default value), the string specified by the New_Working_View parameter is used either as a pathname prefix or (if Create_Spec_View is true) as a spec-view prefix in the new view names.

The value of Construct_Subpath_Name is ignored if the Create_Spec_View parameter is true.

```
Create_Spec_View : Boolean := False;
```

Specifies whether to create spec views instead of working load or combined views.

If Create_Spec_View is false (the default value), the type of view created depends on the values of the Create_Load_View and Create_Combined_View parameters. If all three parameters are false, each new view is the same type as the source view from which it was copied.

If Create_Spec_View is true, a new spec view is created from each of the source views specified by From_View. In this case, the values of Create_Spec_View and Create_Combined_View must be false. Objects in the new spec views are uncontrolled.

Each new spec view is created with only those units named in the Exports file of the corresponding source view. (This file is located in the *view_name*.State directory.) The new spec view contains a copy of the specifications of those units. If no units are specified in the Exports file, the new spec view copies the specifications of all of the units in the source view.

When Create_Spec_View is true, the string specified by New_Working_View is used as a spec-view prefix. The name of each new view thus is constructed from the specified string, followed by one or more release level numbers (as determined by the Level_For_Spec_View parameter), followed by the string "_Spec". For example, if From_View names a source view called "Rev1_Anderson_Working" and New_Working_View specifies the string "Rev2", then setting Create_Spec_View to true causes the new view to be called Rev2_*n*_Spec (where *n* represents the current release level number).

When Create_Spec_View is true, the values of the Join_Views, Reservation_Token_Name, and Construct_Subpath_Name parameters are ignored.

The value of Create_Spec_View is ignored when the Copy command is entered in a system or in a combined subsystem. Systems can contain only system views and combined subsystems can contain only combined views.

```
Create_Load_View : Boolean := False;
```

Specifies whether to create working load views instead of spec views or working combined views.

If Create_Load_View is false (the default value), the type of view created depends on the values of the Create_Spec_View and Create_Combined_View parameters. If all three parameters are false, each new view is the same type as the source view from which it was copied.

If Create_Load_View is true, a new load view is created from each of the source views specified by From_View. In this case, the values of Create_Spec_View and Create_Combined_View must be false.

The value of Create_Load_View is ignored when the Copy command is entered in a system or in a combined subsystem. Systems can contain only system views and combined subsystems can contain only combined views.

```
Create_Combined_View : Boolean := False;
```

Specifies whether to create working combined views instead of spec views or working load views.

If Create_Combined_View is false (the default value), the type of view created depends on the values of the Create_Spec_View and Create_Load_View parameters. If all three parameters are false, each new view is the same type as the source view from which it was copied.

If Create_Combined_View is true, a new combined view is created from each of the source views specified by From_View. In this case, the values of Create_Spec_View and Create_Load_View must be false.

The value of Create_Combined_View is ignored when the Copy command is entered in a system. Systems can contain only system views.

```
Level_For_Spec_View : Natural := 0;
```

Specifies which level number to increment when creating spec-view names. If the value of this parameter is Natural'Last, spec-view names are generated without level numbers.

Level numbers in a spec-view name are generated from the level numbers in the name of the most recently released view in that subsystem. Note that a released-view name contains as many numbers as there are release levels; the rightmost number is the 0th level. In a spec-view name, the string "_Spec" replaces the rightmost (0th level) number, so a spec-view name has one number less than a released-view name.

If Level_For_Spec_View is 0, no release level numbers are incremented, because the 0th-level number has been replaced. In this case, the spec-view name contains the same numbers (starting with level 1) as the most recent release. If Level_For_Spec-_View is 1, the first-level number in the most recent release name is incremented before the appropriate level numbers are inserted into the spec-view name. The number of levels that can be incremented is determined by the Levels file within the model world for the view. The Copy command quits if the value of the Level-_For_Spec_View parameter is a number other than Natural'Last that exceeds the total number of levels specified by the Levels file.

For example, assume that there are two release levels and the most recently released view is called Rev1_4_2. If Create_Spec_View is true and Level_For_Spec_View is 1, the name generated for the new spec view is Rev1_5_Spec (assuming that the New_Working_View parameter specifies the string "Rev1").

The value of Level_For_Spec_View is ignored if the Create_Spec_View parameter is false.

```
Model : String := "<INHERIT_MODEL>";
```

Specifies a model world for each new working view. If the specified name cannot be resolved in the context !Model, the name is resolved relative to the current context. By default, the new working view uses the same model as the view from which it was copied.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted by adjusting imports.

If true (the default value), units are recompiled to the state specified by the Goal parameter.

If false, any units demoted by adjusting imports are left in the demoted state.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, all units in the view are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date when the new working view was copied and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to make the new working views. The default value specifies that the new working views should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Make_Path

procedure Make_Spec_View

procedure Make_Subpath

procedure Merge_Changes

---

procedure Create_Empty_Note_Window

```
procedure Create_Empty_Note_Window (What_Object : String := "<CURSOR>";
                                    Response    : String := "<PROFILE>");
```

Description

Creates an empty window for the purpose of composing notes for the specified controlled object. The banner of the created window identifies it as Notes For followed by the object's name.

The notes for a controlled object are stored the CMVC database. An object's notes can be used as a scratchpad for arbitrary commentary to be associated with particular generations.

After the notes window has been edited:

- The Append_Notes command can be used to append the window's contents to the object's existing notes.
- The Put_Notes command can be used to replace the object's existing notes with the window's contents.

Modified notes windows retain the * symbol in their window banners, even after their contents have been entered in the CMVC database using Append_Notes or Put_Notes. Accordingly, the Quit command reports these windows as changed images when logout is attempted. Because these windows cannot be committed, use Common.Abandon to remove these windows.

The Notes command provides an interactive alternative to Append_Notes, Create_Empty_Note_Window, and the like. The Notes command displays a history image (identified by the 'History attribute following the object name and generation in the window banner), which allows interactive operations for managing an object's notes.

Parameters

```
What_Object : String := "<CURSOR>";
```

Specifies the object for which an empty notes window is to be created. The specified object must be controlled.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Append_Notes

procedure Get_Notes

procedure Notes

procedure Put_Notes

---

procedure Def

```
procedure Def (What_Object : String  := "<CURSOR>";
               In_Place    : Boolean := False);
```

Description

Traverses between various objects managed by the Environment library system and images managed by the CMVC editor.

In some contexts, the Def command serves as the inverse of the Edit command:

- Entering Edit from a view or from a controlled object in the view displays the configuration image for that view.
- Entering Def from the configuration image for a view displays the view itself or a controlled object in the view (depending on the location of the cursor within the configuration image).

In other contexts, the Def command serves as the inverse of the Notes command:

- Entering Notes from a controlled object in a view displays the history image for the object.
- Entering Def from the history image for a controlled object displays the object itself.

Finally, Def traverses back and forth between a controlled object in a view and its current generation image. If images of other generations are displayed subsequently, Def also displays the controlled object from any of these other generation images.

A particularly useful application of Def is to use it to display an object's current generation image and then use Common.Expand to see the differences between the current generation and the previous generation.

Parameters

```
What_Object :  String := "<CURSOR>";
```

Specifies the object or image from which to traverse. Objects must be controlled. Images include configuration, generation, and history images.

The default is the object or image on which the cursor is currently located.

```
In_Place : Boolean := False;
```

Specifies whether the current frame should be used to display the image. The default specifies that the least recently used frame should be used.

---

procedure Destroy_Subsystem

```
procedure Destroy_Subsystem (What_Subsystem : String := "<SELECTION>";
                             Comments       : String := "";
                             Work_Order     : String := "<DEFAULT>";
                             Response       : String := "<PROFILE>");
```

Description

Destroys the specified subsystem or subsystems.

All views in each subsystem must be destroyed (with the Destroy_View command) before that subsystem can be destroyed.

Parameters

`What_Subsystem : String := "<SELECTION>";`

Specifies one or more subsystems to be destroyed. There can be no views in the specified subsystems. By default, the selected subsystem is destroyed.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

`Work_Order : String := "<DEFAULT>";`

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date when the subsystem is destroyed and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Destroy_System

```
procedure Destroy_System (What_System : String := "<SELECTION>";
                          Comments    : String := "";
                          Work_Order  : String := "<DEFAULT>";
                          Response    : String := "<PROFILE>");
```

Description

Destroys the specified system or systems.

All views in each system must be destroyed (with the Destroy_View command) before the subsystem can be destroyed.

Parameters

`What_System : String := "<SELECTION>";`

Specifies one or more systems to be destroyed. There can be no views in the specified systems. By default, the selected system is destroyed.

Multiple systems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

`Work_Order : String := "<DEFAULT>";`

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date the system is destroyed and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Destroy_View

```
procedure Destroy_View
                    (What_View                      : String  := "<SELECTION>";
                     Demote_Clients                 : Boolean := False;
                     Destroy_Configuration_Also     : Boolean := False;
                     Comments                       : String  := "";
                     Work_Order                     : String  := "<DEFAULT>";
                     Response                       : String  := "<PROFILE>");
```

Description

Destroys the named view or views and all of their subdirectory structure, including the Ada units in the Units directories.

This procedure destroys views in subsystems and in systems.

All objects are first unfrozen if they are currently frozen, and then they are deleted and expunged from the directory system. A view cannot be destroyed in any of the following cases:

- The view contains controlled objects that currently are checked out.
- The view is currently imported by client views.
- The view is included in a system as a result of operations in the Cmvc_Hierarchy package.

Destroy_View is the only command that should be used to destroy a view. In particular, neither the Library.Destroy nor the Compilation.Destroy command should be used, because these commands cannot destroy the entire view structure. If an attempt was made to destroy a view using any command other than Destroy_View, you can recover as follows:

1. Enter the Cmvc_Maintenance.Check_Consistency command to repair the partially destroyed view.
2. Enter the Destroy_View command to destroy the view completely.

By default, views are destroyed so that they can be rebuilt using the Build command. Views can be destroyed permanently by setting the Destroy_Configuration_Also parameter to true.

Parameters

```
What_View : String := "<SELECTION>";
```

Specifies one or more views to be destroyed. The default, "<SELECTION>", means that the selected view is destroyed. The specified views cannot contain controlled objects that currently are checked out.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Demote_Clients : Boolean := False;
```

Specifies whether a view can be destroyed if other views import it. If false (the default value), the Destroy_View command quits if the specified view is imported by other views. If true, the specified view is removed from the imports of any referencing views and then destroyed. Note that units in the referencing views may be demoted as a result of removing the import.

```
Destroy_Configuration_Also : Boolean := False;
```

Specifies whether to destroy the configuration object associated with each specified view. If false (the default value), the configuration object is preserved for each destroyed view. In addition, the state description directory is preserved for each released view and a state description directory is created for each spec and working view. (State description directories exist in the *subsystem*.Configurations directory along with configuration objects.)

As a result, any view destroyed while this parameter is false can be reconstructed using the Build command. (Note that only the controlled objects in a view can be reconstructed by Build.) Destroying a view while this parameter is false is useful for saving space without losing information. Note that as long as the configuration object exists, a new view with the same name cannot be created in the same subsystem.

If true, the configuration object is expunged from the CMVC database. The destroyed view cannot be reconstructed, although a new view with the same name can be created.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date the view is destroyed and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Restrictions

A view cannot be destroyed if it contains controlled objects that currently are checked out, if it is imported by other views, or if it is included in a system.

---

procedure Edit

```
procedure Edit (View_Or_Config        : String  := "<CURSOR>";
                In_Place              : Boolean := False;
                Allow_Check_Out       : Boolean := True;
                Allow_Check_In        : Boolean := True;
                Allow_Accept_Changes  : Boolean := True);
```

Description

Displays a configuration image for the specified view or configuration object or for the view enclosing the specified object.

A configuration image for a view is a library-like display of CMVC information pertaining to that view.

Every view embodies a specific configuration, where a configuration is a combination of generations, one for each controlled object in the view. A configuration image for a view thus contains an entry for each controlled object in the view, indicating the generation of the object that is present in the configuration embodied by the view. Each entry also indicates the latest generation that exists for that object in any view.

A configuration image for a view provides a convenient way to:

- Check in, check out, and accept changes on controlled objects.
- Determine whether an object is checked out, to whom it is checked out, whether it is out of date in a given view, and which other views contain objects in the same join set. (The Common.Expand command displays increasing levels of information from this image.)
- Access generation images (textual representations of previous generations) and history images (the notes stored in the CMVC database for each controlled object). Note that a given generation image can be expanded to show differences between that generation and the previous one. Generation images are available only for controlled objects for which source is saved.

Note that Edit can be used to display a configuration image for a configuration object that has no view associated with it (for example, a configuration release). In this case, the configuration image provides access to generation images and history images even for objects that may not still exist outside the CMVC database. This is a useful means for browsing past generations of objects.

Similarly, the Edit command can be used to display configuration images for code views, which do not contain source objects.

The Def command traverses from a configuration image for a view to the view itself (or to an object in that view).

By default, commands from package Common can be used to perform checkin, checkout, and accept-changes operations in a configuration image that was created by the Edit command. To restrict such operations, set the Allow_Check_Out, Allow_Check_In, and Allow_Accept_Changes parameters to false when you enter the Edit command.

As an alternative to using the Edit command, configuration images can be created using commands from package Common. In this case, checkin, checkout, and accept-changes operations are restricted automatically. However, the Edit command can be entered from the configuration image to change these restrictions as specified by the Allow_Check_Out, Allow_Check_In, and Allow_Accept_Changes parameters.

Parameters

```
View_Or_Config : String := "<CURSOR>";
```

Specifies the view or object for which to display a configuration image. A configuration object also can be specified, even if the corresponding view no longer exists. If an object is specified, it must be controlled.

The default is the object or view on which the cursor is currently located.

```
In_Place : Boolean := False;
```

Specifies whether the current frame should be used to display the image. The default specifies that the least recently used frame should be used.

```
Allow_Check_Out : Boolean := True;
```

Specifies whether to permit checkout operations in configuration images. If true (the default), commands from package Common can be used to check out objects from a configuration image. If false, checkout operations are not permitted.

```
Allow_Check_In : Boolean := True;
```

Specifies whether to permit checkin operations in configuration images. If true (the default), commands from package Common can be used to check in objects from a configuration image. If false, checkin operations are not permitted.

```
Allow_Accept_Changes : Boolean := True;
```

Specifies whether to permit changes to be accepted in configuration images. If true (the default), commands from package Common can be used to accept changes into objects from a configuration image. If false, accept-changes operations are not permitted.

References

procedure Def

procedure Get_Notes

```
procedure Get_Notes (To_File     : String := "<WINDOW>";
                     What_Object : String := "<CURSOR>";
                     Response    : String := "<PROFILE>");
```

Description

Retrieves the notes for the current generation of the specified controlled object.

The Get_Notes command retrieves an object's notes from the CMVC database and displays them in a special-purpose window or writes them into a file. An object's notes can be used as a scratchpad for arbitrary commentary to be associated with particular generations.

Get_Notes is one of a set of file-oriented commands for managing notes. That is, these commands, including Put_Notes, Create_Empty_Note_Window, and Append_Notes, are most useful for managing notes through files. However, these commands also manage special-purpose notes windows, which are identified in the banner by the string Notes For followed by the object's name. The contents of the window can be edited; however, the edited text in the window can be saved into the CMVC database only as follows:

- The Append_Notes command can be used to append the window's contents to the object's existing notes.
- The Put_Notes command can be used to replace the object's existing notes with the window's contents.

Note that modified notes windows retain the * symbol in their window banners even after their contents have been entered in the CMVC database using Append_Notes or Put_Notes. Accordingly, the Quit command reports these windows as changed images when logout is attempted. Because these windows cannot be committed, use Common.Abandon to remove these windows.

The Notes command provides an interactive alternative to Get_Notes, Put_Notes, and the like. The Notes command displays a history image (identified by the 'History attribute following the object name and generation in the window banner), which allows interactive operations for managing an object's notes.

Parameters

```
To_File : String := "<WINDOW>";
```

Specifies where to put the retrieved notes. If a new filename is specified, a file is created and the notes are written into it. If an existing filename is specified, the contents of that file are replaced with the notes.

If the default special name "<WINDOW>" is used, a window is opened in which the notes are displayed. Note that this window is not a normal text file; changes to this window can be saved only by using the Put_Notes command.

```
What_Object : String := "<CURSOR>";
```

Specifies the object whose notes are to be retrieved. Only controlled objects have notes. The default is the object on which the cursor is located.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Append_Notes

procedure Create_Empty_Note_Window

procedure Notes

procedure Put_Notes

procedure Import

```
procedure Import
      (View_To_Import        : String                    := "<REGION>";
       Into_View             : String                    := "<CURSOR>";
       Only_Change_Imports   : Boolean                   := False;
       Import_Closure        : Boolean                   := False;
       Remake_Demoted_Units  : Boolean                   := True;
       Goal                  : Compilation.Unit_State    := Compilation.Coded;
       Comments              : String                    := "";
       Work_Order            : String                    := "<DEFAULT>";
       Response              : String                    := "<PROFILE>");
```

Description

Imports the specified spec or combined views into the designated view(s).

Some or all of the views specified by the View_To_Import parameter are imported by a given view, depending on the value of the Only_Change_Imports parameter.

The Import command can be used to:

- Add new imports
- Change an existing import by importing a different view from the same subsystem
- Refresh a view's existing imports after new specifications have been added to the imported views

Consistency checking is done to ensure that no view directly or indirectly imports more than one view from the same subsystem. The import operation checks the closure of the importing view and the closures of all views that import it. An error results if any new or changed import would cause an inconsistency.

Furthermore, within spec/load subsystems, circularity checking is done to ensure that no view directly or indirectly imports itself. (Circular importing is permitted among views in combined subsystems, however.)

An import operation succeeds only if the target key of the importing (client) view is compatible with the target key of the imported (supplier) view. For example, a view with target key R1000 cannot import a view with target key Mc68020_Bare.

Importing operations create and manage links among subsystems. When one view imports another, links are created in the client view to each of the units in the supplier view. Imports alone enable links to be managed across paths, subpaths, and releases; links should never be added individually through commands from package Links.

An import operation will create links to a subset of the units in a supplier view if export and import restrictions exist. Users create export and import restrictions as text files in the supplier and client views, respectively.

An *export restriction file* is a text file in the Exports subdirectory within the supplier view. An export restriction file defines a subset of the units in the supplier view either through a list of unit names (one per line) or through naming expressions. Names in an export restriction file are resolved against the Units directory within the view. The Exports subdirectory can contain multiple export restriction files that define alternative subsets of the view.

An *import restriction file* is a text file in the Imports subdirectory within the client view. A given import restriction file determines which subset to use from a particular supplier view. A client view may have multiple import restriction files, one for each of its supplier views. The following rules pertain to the creation of an import restriction file that corresponds to a particular supplier view:

- The import restriction file must have the same name as the subsystem containing the supplier view. Typically the subsystem's simple name is used; however, a fully qualified subsystem name can be converted to a filename by omitting the preceding ! and changing the dots (.) between name components to underscores (_). For example, an import restriction file for a supplier view in the subsystem !Programs.Mail.Mail_Utilities can be named either Mail_Utilities or Programs_Mail_Mail_Utilities.
- The first line of the file must consist of the string `export_restriction=>` followed by the simple name of the desired export restriction file from the supplier view. No blanks should appear in this line. Omitting this line implicitly specifies an export restriction file named Default, if such a file exists; otherwise, the entire supplier view is used.
- Subsequent lines in the import restriction file can contain names or naming expressions to specify a further subset of the units listed in the export restriction file. Links are created in the client view for the units that are matched by the naming expressions. If no naming expressions are specified, no links are created.

  Because an import restriction file essentially specifies a set of link names, only simple Ada names should be used in the naming expressions. This is true even for names that are qualified within the export restriction file. Whereas names in an export restriction file are resolved as library names, names in an import restriction file are resolved as link names.

  Naming expressions can be used to:

  — Request links for all units in the export restriction file by entering @

  — Request links for subsets by using wildcard expressions such as @_pkg

  — Exclude links to units by using expressions such as ~Unit_Name (which should follow an expression such as @)

  — Rename links to units by specifying the unit name followed by the new link name

---

Parameters

```
View_To_Import : String := "<REGION>";
```

Specifies one or more views to be imported by the views named in the Into_View parameter. The views specified by View_To_Import must be spec or combined views. If View_To_Import specifies a set of views, these views are imported in the manner specified by the Only_Change_Imports parameter.

If View_To_Import is the null string (""), the existing imports of Into_View are refreshed to include any new unit specifications that have been added.

If both View_To_Import and Into_View name the same set of combined views, the named views import each other.

The Imported_Views function can be used to return another view's imports as the value of View_To_Import. This is a convenient way of setting the imports of Into_View to be the same as another view's imports.

Note that this parameter accepts only view names; export and import restriction files never need to be specified explicitly.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, View_To_Import can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Into_View : String := "<CURSOR>";
```

Specifies the view or views to which imports are to be added. The default is the view on which the cursor is located. Into_View can specify spec, load, or combined views.

If both the View_To_Import and Into_View parameters name the same set of combined views, the named views import each other.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Only_Change_Imports : Boolean := False;
```

Specifies the manner in which the views specified by the View_To_Import parameter are actually used as imports.

If false (the default value), the entire list of views given by View_To_Import is imported into each view specified by Into_View. Existing imports are not affected unless a View_To_Import view is from the same subsystem as an existing import. In this case, the View_To_Import view replaces the corresponding existing import.

If true, a View_To_Import view is imported only if it is from the same subsystem as an existing import. The View_To_Import view then replaces the existing import. Thus, if Only_Change_Imports is true, the list of views in View_To_Import is used to update existing imports rather than add new imports. In this way, all replacement imports can be specified in a single list without forcing every view to import everything in the list.

```
Import_Closure : Boolean := False;
```

Specifies whether to import not only the views named by View_To_Import but also the views in their closures.

If false (the default value), imports are limited to the views named by View_To-_Import.

If true, imports include the views in the closures of the View_To_Import views, subject to Only_Change_Imports.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted by the import operation.

If true (the default value), units are recompiled to the state specified by the Goal parameter.

If false, any units demoted by the import operation are left in the demoted state.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The compilation goal can be any of the enumerations of the Compilation.Unit-_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To compile units to the installed state, specify Compilation.Installed. If Compilation.Source is specified, the demoted units are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. The command's action is recorded only if the Comments parameter is specified. In addition to the comment, the work order records the time and date as well as the username and session in which the command was entered.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

function Imported_Views

---

function Imported_Views

```
function Imported_Views (Of_View                 : String  := "<CURSOR>";
                         Include_Import_Closure  : Boolean := False;
                         Include_Importer        : Boolean := False;
                         Response                : String  := "<WARN>")
                                                    return String;
```

Description

Returns a string that names all the views that are imported by the specified view.

Parameters

`Of_View : String := "<CURSOR>";`

Specifies one or more views whose imports are to be returned in a naming string. If multiple views are specified, the Imported_Views function returns the union of all the imports of the specified views.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Include_Import_Closure : Boolean := False;`

Specifies whether to return the names of the indirectly imported views in addition to directly imported views.

If false (the default value), only the views directly imported by the Of_View parameter are listed. If true, the returned naming string lists all of the views in Of_View's import closure.

`Include_Importer : Boolean := False;`

Specifies whether to return the names of the views specified by the Of_View parameter. If false (the default value), only the names of imported views are listed. If true, the returned naming string includes the names specified by Of_View as well.

`Response : String := "<WARN>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is a response profile that puts negative messages, warning messages, error messages, and exception messages in the log.

```
return String;
```

Returns a string that names all the views that are imported by the specified view.

---

procedure Information

```
procedure Information (For_View                    : String  := "<CURSOR>";
                       Show_Model                  : Boolean := True;
                       Show_Whether_Frozen         : Boolean := True;
                       Show_View_Kind              : Boolean := True;
                       Show_Creation_Time          : Boolean := True;
                       Show_Imports                : Boolean := True;
                       Show_Referencers            : Boolean := True;
                       Show_Unit_Summary           : Boolean := True;
                       Show_Controlled_Objects     : Boolean := False;
                       Show_Last_Release_Numbers   : Boolean := False;
                       Show_Path_Name              : Boolean := False;
                       Show_Subpath_Name           : Boolean := False;
                       Show_Switches               : Boolean := False;
                       Show_Exported_Units         : Boolean := False;
                       Response                    : String  := "<PROFILE>");
```

Description

Displays various kinds of information about the specified view in the output window.

Each parameter specifies whether to display a particular kind of information.

Parameters

`For_View : String := "<CURSOR>";`

Specifies the view for which information is to be displayed. The default is the view on which the cursor is located. The specified view can be in a subsystem or in a system.

`Show_Model : Boolean := True;`

Specifies whether to display the name of the view's model. If true (the default value), the name is displayed.

`Show_Whether_Frozen : Boolean := True;`

Specifies whether to display the view's status with respect to freezing. If true (the default value), this information is displayed.

`Show_View_Kind : Boolean := True;`

Specifies whether to display the view's kind—for example, spec, load, or combined. If true (the default value), the kind is displayed.

```
Show_Creation_Time : Boolean := True;
```

Specifies whether to display when the view was created. If true (the default value), the creation time is displayed.

```
Show_Imports : Boolean := True;
```

Specifies whether to display a list of all imported views. If true (the default value), the imports are displayed.

```
Show_Referencers : Boolean := True;
```

Specifies whether to display a list of all subsystems that import this view. If true (the default value), this information is displayed.

```
Show_Unit_Summary : Boolean := True;
```

Specifies whether to display a summary of the compilation states of all units in the view. If true (the default value), the number of coded units, installed units, source units, and empty stubs is displayed. If For_View specifies a system view, this parameter causes the view's release activity to be displayed.

```
Show_Controlled_Objects : Boolean := False;
```

Specifies whether to display a list of all controlled objects in the view. If false (the default value), this information is not displayed. If true, the display includes the same information as the Show_All_Controlled command—namely, the number of generations that exist for each controlled object, whether the object is checked out, and by whom.

```
Show_Last_Release_Numbers : Boolean := False;
```

Specifies whether to display the level numbers that appear in the name of the most recently released view. These numbers will be incremented if another released view or spec view is created. If false (the default value), this information is not displayed.

```
Show_Path_Name : Boolean := False;
```

Specifies whether to display the string within the view's name that serves as the pathname. If false (the default value), this information is not displayed.

```
Show_Subpath_Name : Boolean := False;
```

Specifies whether to display the string within the view's name that serves as the subpathname. If false (the default value), this information is not displayed.

```
Show_Switches : Boolean := False;
```

Specifies whether to display the settings for all switches associated with the view. If true, switch settings are displayed, along with the view's target key. If false (the default value), this information is not displayed.

```
Show_Exported_Units : Boolean := False;
```

Specifies whether to display a list of all exported units. If false (the default value), this information is not displayed.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Initial

```
procedure Initial
 (System_Object           : String              := ">>SYSTEM OBJECT NAME<<";
  Working_View_Base_Name  : String              := "Rev1";
  System_Object_Type      : System_Object_Enum  := Cmvc.Spec_Load_Subsystem;
  View_To_Import          : String              := "";
  Create_Load_View        : Boolean             := True;
  Model                   : String              := "R1000";
  Comments                : String              := "";
  Work_Order              : String              := "<DEFAULT>";
  Volume                  : Natural             := 0;
  Response                : String              := "<PROFILE>");
```

Description

Builds a new system or a new subsystem of the specified type—namely, spec/load or combined.

Subsystems partition a project or application into high-level components by grouping Ada units or other objects. A system pulls an application's components together by logically grouping particular releases from the component subsystems. Operations for systems are in package Cmvc_Hierarchy.

The new subsystem or system contains an empty working view that has the specified imports. The Initial command also can be used to create an empty view in an existing subsystem or system.

The initial view is set up according to the specified model. This includes the setting of the switches and initial links for the view. The model also may contain a file named Levels whose integer contents specify the number of levels for automatic name generation for released and spec views. Furthermore, the model may contain user-defined directory structure to be created in the view in addition to the predefined directories.

The name of the initial view of the subsystem or system is:

```
[System_Object].[Working_View_Base_Name]_Working
```

Parameters

`System_Object : String := ">>SYSTEM OBJECT NAME<<";`

Specifies the name of the subsystem or system to be created. The default parameter placeholder ">>SYSTEM OBJECT NAME<<" must be replaced or an error will result.

```
Working_View_Base_Name : String := "Rev1";
```

Specifies the base name of the initial view in the subsystem or system. If the default value is used, the initial view is named Rev1_Working.

The string given for Working_View_Base_Name can be any legal Ada identifier. By convention, if this string contains no underscores, it serves as a pathname prefix; if the string contains an underscore, it serves as a pathname prefix followed by a subpathname extension.

```
System_Object_Type : System_Object_Enum := Cmvc.Spec_Load_Subsystem;
```

Specifies whether to create a system or one of two types of subsystem—namely, spec/load or combined.

Systems are an optional device for creating logical groupings of releases from component subsystems in an application. Operations for systems are in package Cmvc_Hierarchy.

Subsystems partition applications into high-level components. The two types of subsystem determine the kinds of views that can be created as well as whether hierarchic importing is enforced. The default value, Cmvc.Spec_Load_Subsystem, causes the Initial procedure to build a subsystem that can contain either spec/load or combined views. Within such a subsystem, all imports must be hierarchic, in that no view is permitted to be in its own import closure. If Cmvc.Combined_Subsystem is specified, the Initial procedure builds a subsystem that can contain only combined views, among which circular import relations may hold.

```
View_To_Import : String := "";
```

Specifies one or more views to be imported by the new working view. The views specified by View_To_Import must be spec or combined views.

If View_To_Import is the null string (""), the default value, no views are imported.

The Imported_Views function can be used to return another view's imports as the value of View_To_Import. This is a convenient way of setting the new working view's imports to be the same as another view's imports.

Note that if export and import restrictions will be needed, it is recommended that you do not use this parameter to create imports. Instead, after the subsystem is created, you can create the export/import restriction files and use the Cmvc.Import command to perform the import operation.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, View_To_Import can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Create_Load_View : Boolean := True;
```

Specifies whether to create a load view for the initial working view. If true, the initial working view is a load view. If false, the initial working view is a combined view.

The value of this parameter is used only when the System_Object_Type parameter has the value Cmvc.Spec_Load_Subsystem; otherwise, this parameter is ignored. (When a combined subsystem is created, the initial view is a combined view; when a system is created, the initial view is a system view.)

```
Model : String := "R1000";
```

Specifies a model world for the initial view in the subsystem. If the specified name cannot be resolved in the context !Model, the name is resolved relative to the current context. By default, the view uses the model !Model.R1000.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, if the Comments parameter is specified, the work order records the time and date when the subsystem was created, the username and session in which the command was entered, the specified comment, and the creation of the release history file.

If the Comments parameter is not specified, only the creation of the release history file is logged.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to make the new subsystem. The default value specifies that the new subsystem should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

type System_Object_Enum

---

procedure Join

```
procedure Join (What_Object              : String := "<SELECTION>";
                To_Which_View            : String := ">>VIEW NAME<<";
                Reservation_Token_Name   : String := "";
                Comments                 : String := "";
                Work_Order               : String := "<DEFAULT>";
                Response                 : String := "<PROFILE>");
```

Description

Joins the specified controlled objects to the corresponding objects in the designated view.

When objects are joined across views, they form a join set. Objects in a join set have the same pathname within their respective views and share a single reservation token, so that only one object in the set can be checked out at a time. Thus, joining allows synchronized changes to an object when there are instances of the same object in multiple working views.

The objects to be joined must be textually identical. The Merge_Changes command can be used to prepare objects for joining.

There are two alternative ways to specify the join set to which objects are to be joined. One is to specify a view that contains an object in the desired join set. The other is to specify the reservation token associated with the desired join set. (See the To_Which_View and Reservation_Token_Name parameters, below.)

Parameters

```
What_Object :  String := "<SELECTION>";
```

Specifies one or more objects to be joined to the corresponding objects in the designated view. This parameter must specify controlled objects. The text of the specified objects must be identical to the text of the objects to which they are to be joined. The Merge_Changes command can be used to prepare objects for joining.

If What_Object names an object that already belongs to a join set, the Join command implicitly severs that object from its original join set before joining the object to the new join set.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
To_Which_View : String := ">>VIEW NAME<<";
```

Specifies the view containing the objects to which the specified objects are to be joined. Objects in the specified view must be checked in.

The default parameter placeholder ">>VIEW NAME<<" must be replaced unless a value is given for the Reservation_Token_Name parameter.

```
Reservation_Token_Name : String := "";
```

Specifies the reservation token of the join set to which the specified objects are to be joined. This parameter is used only if no value is specified for the To_Which_View parameter.

Reservation tokens are displayed in expanded configuration images (see the Cmvc.Edit command).

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Make_Code_View

```
procedure Make_Code_View (From_View       : String  := "<CURSOR>";
                          Code_View_Name  : String  := "";
                          Comments        : String  := "";
                          Work_Order      : String  := "<DEFAULT>";
                          Volume          : Natural := 0;
                          Response        : String  := "<PROFILE>");
```

Description

Makes a code view from the specified load view.

Code views are copies of views that store executable code in place of Ada units. Code views thus require the minimum amount of space necessary to permit execution of the view. The executable code is stored in an object called Code_Database within the view.

The Units directory of a code view contains a copy of any non-Ada objects from the original view.

Because Ada units in code views are stored as executable code, these units cannot be modified or browsed except through configuration and generation images (see the Cmvc.Edit command).

Parameters

```
From_View : String := "<CURSOR>";
```

Specifies one or more views from which code views are to be made. The named views must be load views. The default is the view on which the cursor is located.

All units in the named views must be coded and must contain bodies for all specifications that require them. All controlled units in the named views must be checked in.

If multiple views are named, they must be in different subsystems. Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Code_View_Name : String := "";
```

Specifies the simple name of the new code view. No part of a code-view name is automatically generated, so the string specified by Code_View_Name constitutes the entire name.

If multiple views in different subsystems are named, each will have the name specified by Code_View_Name. The name can be any legal Ada identifier.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date when the code view is created and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to make the new code views. The default value specifies that the new views should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Make_Controlled

```
procedure Make_Controlled
                    (What_Object              : String  := "<CURSOR>";
                     Reservation_Token_Name   : String  := "<AUTO_GENERATE>";
                     Join_With_View           : String  := "<NONE>";
                     Save_Source              : Boolean := True;
                     Comments                 : String  := "";
                     Work_Order               : String  := "<DEFAULT>";
                     Response                 : String  := "<PROFILE>");
```

Description

Makes the specified object or objects controlled by the CMVC system and therefore subject to reservation.

Once controlled, an object must be checked out before it can be modified and it must be checked in before various commands can access it.

When an object is controlled with the Save_Source parameter set to true, the textual changes from one generation to the next are stored in the CMVC database. This permits the reconstruction of previous generations through, for example, the Revert command or by rebuilding a view from a configuration object. (Note that because changed lines are determined textually, changing an Ada unit's pretty-printing causes all lines to be stored as changed lines.)

When an object is controlled with the Save_Source parameter set to false, no textual representation is stored in the CMVC database. This is useful for binary objects that have no ASCII representation or for very large files (when storage space is an issue). Even though previous generations cannot be reconstructed when objects are controlled without saving source, such objects still need to be checked out before they can be modified. (Generation numbers thus record the number of times objects were checked out and checked in.)

Controlling an object associates a reservation token with it. The Check_In and Check_Out procedures operate by manipulating reservation tokens, and joined objects share not only the same name but also a single reservation token.

Parameters

```
What_Object : String := "<CURSOR>";
```

Specifies one or more objects to be controlled. A view name can be specified, although specifying a view causes the unnecessary controlling of objects outside the Units directory. If a named object is already controlled, a note appears in the output log. Before a subunit can be controlled, its parent must be controlled.

If multiple objects are specified, all must be in the same subsystem. Naming multiple views not only controls those views but also effectively joins them under a single reservation token.

Objects in the State subdirectory of a view cannot be controlled; attempting to do so produces an error message. Similarly, derived objects resulting from cross-target development cannot be controlled (the names of such objects are enclosed in angle brackets in directory displays).

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Reservation_Token_Name : String := "<AUTO_GENERATE>";
```

Specifies the name of the reservation token to be associated with each specified object. This is useful for associating mnemonic names of reservation tokens with particular join sets. Note that an existing name of a reservation token can be used to implicitly join the newly controlled objects to other objects.

The value of Reservation_Token_Name is used only if the Join_With_View parameter has its default value "<NONE>".

The default special name "<AUTO_GENERATE>" means that the reservation token is generated by the Environment.

```
Join_With_View : String := "<NONE>";
```

Specifies a view to which the specified objects are joined. That is, if Join_With-_View names a view, the objects named by What_Object are joined to the corresponding objects in the named view, and the reservation token name for the objects is taken from that view. (In this case, the Reservation_Token_Name parameter is ignored.)

To be joined, the objects named by What_Object must be identical in content to the corresponding objects in the view named by Join_With_View. Furthermore, the corresponding objects in the view named by Join_With_View must already be controlled.

The default special name "<NONE>" means that the newly controlled objects are not joined to any objects in any other views. (In this case, the Reservation_Token_Name parameter determines the reservation token name.)

```
Save_Source : Boolean := True;
```

Specifies whether source is saved in the CMVC database for a controlled object.

If true (the default value), the textual changes from one generation to the next are stored in the CMVC database. Consequently, previous generations can be reconstructed by the Revert command or by the Build command. Furthermore, when source is saved for joined objects, out-of-date objects can be updated explicitly with the Accept_Changes command or implicitly with the Check_Out command.

If false, no textual representation is stored in the CMVC database, although objects must still be checked out before they can be modified. This parameter is typically set to false when controlling binary objects that have no ASCII representation or when controlling very large objects (when storage space is an issue).

When source is not saved for a controlled object, previous generations cannot be reconstructed—for example, when rebuilding a view from a configuration object. Furthermore, the Accept_Changes and Check_Out commands will update such a controlled object to the latest generation only if an object in some view actually contains that generation.

If instances of an object exist in multiple views, all of the controlled instances of the object must save source or else none of them can save source.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date when the objects were controlled, the object affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Restrictions

The Make_Controlled procedure cannot join the specified objects with the view named by the Join_With_View parameter unless the specified objects are identical in content to the corresponding objects in the view.

---

procedure Make_Path

```
procedure Make_Path
    (From_Path              : String                    := "<CURSOR>";
     New_Path_Name          : String                    := ">>PATH NAME<<";
     View_To_Modify         : String                    := "";
     View_To_Import         : String                    := "<INHERIT_IMPORTS>";
     Only_Change_Imports    : Boolean                   := True;
     Create_Load_View       : Boolean                   := False;
     Create_Combined_View   : Boolean                   := False;
     Model                  : String                    := "<INHERIT_MODEL>";
     Join_Paths             : Boolean                   := True;
     Remake_Demoted_Units   : Boolean                   := True;
     Goal                   : Compilation.Unit_State    := Compilation.Coded;
     Comments               : String                    := "";
     Work_Order             : String                    := "<DEFAULT>";
     Volume                 : Natural                   := 0;
     Response               : String                    := "<PROFILE>");
```

Description

Creates a copy of each of the specified views, starting new development paths.

A path is a logically connected series of views within a subsystem or a system. For each view specified, the Make_Path command creates a new working view that serves as the start of such a series of views.

A subsystem or a system can contain multiple paths. For example, if an application has multiple targets, a path can be made for each target. Similarly, if a new major release of an application must be developed while the existing release is maintained, a separate path can be made for the new major release.

A new path can, but need not, be joined to the view (and hence to the path) from which it is created. Two paths should be joined (using the Join_Paths parameter) if the majority of the controlled objects in them are to be joined. (Joined objects cannot be checked out and modified independently.) The controlled objects that need to be modified independently can be severed subsequently with the Sever command. For example, if an application has two targets, the target-independent code is shared and the target-dependent code is not. Assuming that a path already exists for one of the targets, a joined path can be created for the second target and then the target-dependent units can be severed.

A new path should not be joined to the path from which it is created if most of the controlled objects in these two paths are to be modified independently. For example, if a new major release of an application is developed while the previous major release is maintained, the objects in the two paths typically need to be modified independently, so the paths are not joined. (Note that changes can be propagated across unjoined objects with the Merge_Changes command.) Although the new path is not joined when created, individual objects in it subsequently can be joined to the corresponding objects in other views (see the Join command).

By default, the working view for each new path has the same imports as the view from which it was copied. It is also possible to specify different imports in the process of creating the new paths by using the View_To_Import and Only_Change-_Imports parameters. Import adjustments are subject to the same consistency checking that is performed by the Import command.

Parameters

```
From_Path :  String := "<CURSOR>";
```

Specifies the view or views that are to be copied as the beginning(s) of new path(s). The default is the view on which the cursor is located. The From_Path parameter can name:

- Combined, load, or spec views
- Either working or released views

All controlled objects in a From_Path view must be checked in. If From_Path names multiple views, a new path is made from each of the named views. Multiple views can be in the same or in different (sub)systems, creating a family of new paths across multiple (sub)systems.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
New_Path_Name :  String := ">>PATH NAME<<";
```

Specifies the pathname prefix to be used in constructing the names of the views in the new paths. Because the Make_Path procedure creates new working views, the names of these views are constructed from New_Path_Name and the "_Working" suffix. For example, if New_Path_Name has the value "Rev2", the working view created for the new path is "Rev2_Working" (the underscore is supplied automatically). If the From_Path parameter names multiple views, all of the new paths will have the same pathname prefix.

The New_Path_Name parameter can be any string that constitutes a legal Ada identifier and therefore can contain one or more underscore characters. However, other CMVC operations (such as generating reservation tokens or creating subpaths) conventionally consider a view's pathname prefix to be the portion of a view name up to (but not including) the *first* underscore in the name. Therefore, if the New-_Path_Name string contains an underscore (for example "Target_2"), only the first portion of that string ("Target") is actually considered to be the pathname. If a subpath is created from this path, the "2" will be replaced with the subpathname.

```
View_To_Modify : String := "";
```

Specifies one or more spec, load, or combined views whose imports should be changed to refer to the new working views, provided that the new working views are combined views.

The imports of the views specified by View_To_Modify are also updated using the views named by the View_To_Import parameter. The View_To_Modify views are updated by View_To_Import views as if the Only_Change_Imports parameter were true, regardless of this parameter's actual value.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
View_To_Import : String := "<INHERIT_IMPORTS>";
```

Specifies one or more spec or combined views to be imported by the new working views. The views named by View_To_Import are also used to update the imports of the views named by the View_To_Modify parameter.

If View_To_Import specifies the default special name "<INHERIT_IMPORTS>", each new working view uses the same imports as the view from which it was copied. (However, if the From_Path parameter names multiple combined views among which import relations hold, the imports are automatically adjusted so that the views in the new paths reference each other as appropriate, instead of referencing the views in the original paths.)

If View_To_Import specifies the null string (""), no views are imported.

If the View_To_Import parameter specifies one or more views, the specified views are imported by the new working views in the manner specified by Only_Change-_Imports.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, View_To_Import can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Create_Load_View : Boolean := False;
```

Specifies whether to create working load views instead of working combined views.

If Create_Load_View is false (the default value), the type of view created depends on the value of the Create_Combined_View parameter. However, if both parameters are false, a new combined view is created from each combined view specified by From_Path, and a new load view is created from each load or spec view specified by From_Path.

If Create_Load_View is true, a new load view is created from each of the source views specified by From_View. In this case, the value of Create_Combined_View must be false.

The value of Create_Load_View is ignored when the Make_Path command is entered in a system or in a combined subsystem. Systems can contain only system views and combined subsystems can contain only combined views.

```
Create_Combined_View : Boolean := False;
```

Specifies whether to create working combined views instead of working load views.

If Create_Combined_View is false (the default value), the type of view created depends on the value of the Create_Load_View parameter. However, if both parameters are false, a new combined view is created from each combined view specified by From_Path, and a new load view is created from each load or spec view specified by From_Path.

If Create_Combined_View is true, a new combined view is created from each of the source views specified by From_View. In this case, the value of Create_Load_View must be false.

The value of Create_Combined_View is ignored when the Make_Path command is entered in a system. Systems can contain only system views.

```
Only_Change_Imports : Boolean := True;
```

Specifies the manner in which the views specified by the View_To_Import parameter are actually used as imports by the new working views. Only_Change_Imports has no effect if View_To_Import specifies "<INHERIT_IMPORTS>" or the null string.

If this parameter is false, the entire list of views given by View_To_Import is imported by each new working view created by the Make_Path procedure. No imports are inherited.

If the parameter is true (the default value):

- Each new working view inherits its imports from the view from which it was copied.
- The list of views in View_To_Import is compared to the inherited views. If a View_To_Import view is from the same subsystem as an inherited view, the View_To_Import view replaces that inherited view.

Thus, if Only_Change_Imports is true, the list of views in View_To_Import is used to update the inherited imports of each new working view. In this way, the replacement imports for every new working view can be specified in a single list without forcing each new view to import everything in the list.

```
Model : String := "<INHERIT_MODEL>";
```

Specifies a model world for the views in the new path. If the specified name cannot be resolved in the context !Model, the name is resolved relative to the current context. By default, the new working view uses the same model as the view from which it was copied.

```
Join_Paths : Boolean := True;
```

Specifies whether to join each new working view to the view from which it was copied.

If true (the default value), the controlled objects in each From_Path view are joined to the corresponding objects in the copied working view. The reservation token from the From_Path view is used. If a From_Path view contains no controlled objects, then no objects can be joined. Note that Join_Paths affects only controlled objects that exist at the time the Make_Path command is executed. Objects created after the path is made must be controlled explicitly and joined using the Make_Controlled and Join commands.

If false, new reservation tokens are created for all of the controlled objects. The value for New_Path_Name is used as the reservation token.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted by adjusting imports.

If true (the default value), units are recompiled to the state specified by the Goal parameter.

If false, any units demoted by adjusting imports are left in the demoted state.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The compilation goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, all units in the view are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date the path was made and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to make the new paths. The default value specifies that the new paths should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Join

procedure Merge_Changes

procedure Sever

---

procedure Make_Spec_View

```
procedure Make_Spec_View
    (From_Path              : String                    := "<CURSOR>";
     Spec_View_Prefix       : String                    := ">>PREFIX<<";
     Level                  : Natural                   := 0;
     View_To_Modify         : String                    := "";
     View_To_Import         : String                    := "<INHERIT_IMPORTS>";
     Only_Change_Imports    : Boolean                   := True;
     Remake_Demoted_Units   : Boolean                   := True;
     Goal                   : Compilation.Unit_State    := Compilation.Coded;
     Comments               : String                    := "";
     Work_Order             : String                    := "<DEFAULT>";
     Volume                 : Natural                   := 0;
     Response               : String                    := "<PROFILE>");
```

Description

Creates a new spec view from each of the specified views in a spec/load subsystem.

Each new spec view is created with only those units named in the Exports file of the corresponding source view. (This file is located in the *view_name*.State directory.) The new spec view contains a copy of the specifications of those units. If no units are specified in the Exports file, the new spec view copies the specifications of all of the units in the source view. Units in each new spec view are compiled according to the Remake_Demoted_Units and Goal parameters.

By default, units in spec views are not made controlled. If these units are subsequently made controlled for purposes of history tracking, they should not be joined to their counterparts in working views.

Portions of each new spec view's name are automatically generated unless the Spec_View_Prefix and Level parameters specify otherwise. An automatically generated spec-view name consists of a spec-view prefix, one or more level numbers that correlate with a particular numbered release, and the _Spec suffix—for example, Rev1_1_Spec.

By default, each spec view has the same imports as the view from which it was copied. It is also possible to specify different imports in the process of creating the spec views by using the View_To_Import and Only_Change_Imports parameters. Import adjustments are subject to the same consistency checking that is performed by the Import command.

Parameters

```
From_Path : String := "<CURSOR>";
```

Specifies the view or views from which spec views are to be made. The default is the view on which the cursor is located. The From_Path parameter can name any view in a spec/load subsystem:

- Either load, spec, or combined views
- Either working or released views
- Views belonging to paths or views belonging to subpaths

All controlled objects in a From_Path view must be checked in. If From_Path names multiple views, a new spec view is made from each of the named views. Multiple views can be in the same or in different subsystems.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Spec_View_Prefix : String := ">>PREFIX<<";
```

Specifies the string that replaces both the path and subpath portion of the names listed in the From_Path parameter. For example, if From_Path specifies the string "Rev1_Anderson_Working", the value of Spec_View_Prefix replaces "Rev1_Anderson" in the name of the new spec view.

The default parameter placeholder `">>PREFIX<<"` must be replaced or an error will result.

```
Level : Natural := 0;
```

Specifies which level number to increment within each spec view's name. The automatic insertion of level numbers can be suppressed by setting Level to Natural'Last.

Level numbers in a spec-view name are generated from the level numbers in the name of the most recently released view. Note that a released-view name contains as many numbers as there are release levels; the rightmost number is the 0th level. In a spec-view name, the string "_Spec" replaces the rightmost (0th level) number, so a spec-view name has one number less than a released-view name.

If Level is 0, no release level numbers are incremented, because the 0th-level number has been replaced. In this case, the spec-view name contains the same numbers (starting with level 1) as the most recent release. If Level is 1, the first-level number in the most recent release name is incremented before the appropriate level numbers are inserted into the spec-view name. The number of levels that can be incremented is determined by the Levels file within the model world for the view. The Make_Spec_View command quits if the value of the Level parameter exceeds the total number of levels specified by the Levels file.

For example, assume that there are two release levels and the most recently released view is called Rev1_4_2. If a new spec view is created and Level is 1, the name generated for the spec view is Rev1_5_Spec (assuming that the Spec_View_Prefix parameter specifies the string "Rev1").

```
View_To_Modify : String := "";
```

Specifies one or more spec, load, or combined views whose imports should be changed to refer to the new spec views. The imports of the views specified by View_To_Modify are also updated using the views named by the View_To_Import parameter. The View_To_Modify views are updated by View_To_Import views as if the Only_Change_Imports parameter were true, regardless of this parameter's actual value.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
View_To_Import : String := "<INHERIT_IMPORTS>";
```

Specifies one or more spec or combined views to be imported by the new spec views. The views named by View_To_Import are also used to update the imports of the views named by the View_To_Modify parameter.

If View_To_Import specifies the default special name "<INHERIT_IMPORTS>", each new spec view uses the same imports as the view from which it was copied.

If View_To_Import specifies the null string (""), no views are imported.

If View_To_Import specifies one or more views, the specified views are imported by the new spec views in the manner specified by the Only_Change_Imports parameter.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, View_To_Import can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Only_Change_Imports : Boolean := True;
```

Specifies the manner in which the views specified by the View_To_Import parameter are actually used as imports by the new spec views. Only_Change_Imports has no effect if View_To_Import specifies "<INHERIT_IMPORTS>" or the null string.

If this parameter is false, the entire list of views given by View_To_Import is imported by each new view created by the Make_Spec_View command. No imports are inherited.

If the parameter is true (the default value):

- Each new spec view inherits its imports from the view from which it was copied.
- The list of views in View_To_Import is compared to the inherited views. If a View_To_Import view is from the same subsystem as an inherited view, the View_To_Import view replaces that inherited view.

Thus, if Only_Change_Imports is true, the list of views in View_To_Import is used to update the inherited imports of each new spec view. In this way, the replacement imports for every new spec view can be specified in a single list without forcing each new view to import everything in the list.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted by adjusting imports.

If true (the default value), units are recompiled to the state specified by the Goal parameter.

If false, any units demoted by adjusting imports are left in the demoted state.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The compilation goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, all units in the view are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date when the spec view was made and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to make the new spec views. The default value specifies that the new views should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Make_Subpath

```
procedure Make_Subpath
   (From_Path              : String                    := "<CURSOR>";
    New_Subpath_Extension  : String                    := ">>SUBPATH<<";
    View_To_Modify         : String                    := "";
    View_To_Import         : String                    := "<INHERIT_IMPORTS>";
    Only_Change_Imports    : Boolean                   := True;
    Remake_Demoted_Units   : Boolean                   := True;
    Goal                   : Compilation.Unit_State    := Compilation.Coded;
    Comments               : String                    := "";
    Work_Order             : String                    := "<DEFAULT>";
    Volume                 : Natural                   := 0;
    Response               : String                    := "<PROFILE>");
```

Description

Creates a copy of each of the specified views in order to start new development subpaths.

A subpath is a series of working views that constitutes an extension of a path. Multiple subpaths in a single path support parallel development within that path, allowing multiple developers to make and test changes without conflict. Parallel development can proceed because the controlled objects in each subpath are automatically joined to the corresponding objects in the other subpaths and in the parent path. A controlled object therefore can be checked out and modified in only one subpath view at a time.

Subpaths share the same model as their parent path, which means that they share the same target key and initial links.

By default, the working view for each new subpath has the same imports as the view from which it was copied. It is also possible to specify different imports in the process of creating the new subpaths by using the View_To_Import and Only_Change_Imports parameters. Import adjustments are subject to the same consistency checking that is performed by the Import command.

Subpaths can be created in systems as well as subsystems.

Parameters

```
From_Path : String := "<CURSOR>";
```

Specifies the view or views that are to be copied as the beginning(s) of new subpath(s). The default is the view on which the cursor is located. The From_Path parameter can name:

- Combined or load views only (not spec views).
- Either working or released views.
- Views belonging to paths or views belonging to subpaths. However, if From_Path names a subpath, the new subpaths are created at the same level, not as "subsubpaths."

All controlled objects in a From_Path view must be checked in. If From_Path names multiple views, a new subpath (with the same subpathname extension) is made from each of the named views. Multiple views can be in the same or in different (sub)systems.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
New_Subpath_Extension : String := ">>SUBPATH<<";
```

Specifies the subpathname extension to be used in constructing the names of the views in the new subpaths. Because the Make_Subpath procedure creates new working views, the names of these views are constructed by inserting New_Subpath_Extension between the pathname prefix and the "_Working" suffix. For example, if the From_Path parameter specifies a view called "Rev2_Working" and New_Subpath_Extension has the value "Anderson", the working view created for the new subpath is "Rev2_Anderson_Working". If From_Path names multiple views, all of the new paths will have the same subpathname extension.

The New_Subpath_Extension parameter can be any string that constitutes a legal Ada identifier and therefore can contain one or more underscore characters. However, the underscores preceding and following the subpathname extension are inserted automatically.

The New_Subpath_Extension is inserted after a pathname prefix, which, by convention, is the portion of a view name up to the *first* underscore in the name. The New_Subpath_Extension thus replaces any characters between the first underscore and the "_Working" suffix. For example, if From_Path is "Target_2_Working" and New_Subpath_Extension is "Anderson", the subpathname is "Target_Anderson_Working".

```
View_To_Modify : String := "";
```

Specifies one or more spec, load, or combined views whose imports should be changed to refer to the new working views, if those new views are combined views. The imports of the views specified by View_To_Modify are also updated using the views named by the View_To_Import parameter. The View_To_Modify views are updated by View_To_Import views as if the Only_Change_Imports parameter were true, regardless of this parameter's actual value.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
View_To_Import : String := "<INHERIT_IMPORTS>";
```

Specifies one or more spec or combined views to be imported by the new working views. The views named by View_To_Import are also used to update the imports of the views named by the View_To_Modify parameter.

If View_To_Import specifies the default special name "<INHERIT_IMPORTS>", each new working view uses the same imports as the view from which it was copied. (However, if the From_Path parameter names multiple combined views among which import relations hold, the imports are automatically adjusted so that the working views in the new subpaths reference each other as appropriate, instead of referencing the working views in the original paths.)

If View_To_Import specifies the null string (""), no views are imported.

If View_To_Import specifies one or more views, the specified views are imported by the new working views in the manner specified by the Only_Change_Imports parameter.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.) Furthermore, View_To_Import can name an activity as an indirect file, which is equivalent to naming the spec view associated with each subsystem listed in the activity.

```
Only_Change_Imports : Boolean := True;
```

Specifies the manner in which the views specified by the View_To_Import parameter are actually used as imports by the new working views. Only_Change_Imports has no effect if View_To_Import specifies "<INHERIT_IMPORTS>".

If this parameter is false, the entire list of views given by View_To_Import is imported by each new working view created by the Make_Subpath command. No imports are inherited.

If the parameter is true (the default value):

- Each new working view inherits its imports from the working view from which it was copied.
- The list of views in View_To_Import is compared to the inherited views. If a View_To_Import view is from the same subsystem as an inherited view, the View_To_Import view replaces that inherited view.

Thus, if Only_Change_Imports is true, the list of views in View_To_Import is used to update the inherited imports of each new working view. In this way, the replacement imports for every new working view can be specified in a single list without forcing each new view to import everything in the list.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted by adjusting imports.

If true (the default value), units are recompiled to the state specified by the Goal parameter.

If false, any units demoted by adjusting imports are left in the demoted state.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The compilation goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, all units in the view are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date when the subpath was made and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to make the new subpaths. The default value specifies that the new subpaths should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Make_Uncontrolled

```
procedure Make_Uncontrolled (What_Object  : String := "<CURSOR>";
                             Comments     : String := "";
                             Work_Order   : String := "<DEFAULT>";
                             Response     : String := "<PROFILE>");
```

Description

Makes the specified objects uncontrolled, so that change information about them is no longer collected in the CMVC database.

Existing history for these objects remains in the CMVC database until the database is expunged using the Cmvc_Maintenance.Expunge_Database command. Objects can be made controlled again using the Make_Controlled command; if the CMVC database has not been expunged, the history for the recontrolled objects continues where it stopped.

Because controlled objects cannot be deleted or withdrawn, the Make_Uncontrolled procedure is used to prepare a controlled object for deletion. Similarly, an Ada unit's kind cannot be changed (for example, from procedure to function) while the unit is controlled. Therefore, the unit must be made uncontrolled and then the database must be expunged (using Cmvc_Maintenance.Expunge_Database) before the unit's kind can be changed.

Parameters

```
What_Object : String := "<CURSOR>";
```

Specifies the object(s) to be made uncontrolled. Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Make_Controlled

procedure Cmvc_Maintenance.Expunge_Database

---

procedure Merge_Changes

```
procedure Merge_Changes
                    (Destination_Object       : String  := "<SELECTION>";
                     Source_View              : String  := ">>VIEW_NAME<<";
                     Report_File              : String  := "";
                     Fail_If_Conflicts_Found  : Boolean := False;
                     Comments                 : String  := "";
                     Work_Order               : String  := "<DEFAULT>";
                     Response                 : String  := "<PROFILE>");
```

Description

Merges two objects that previously were joined and then severed from each other.

The object named by the Destination_Object parameter is updated to include any changes that have been made to the corresponding object located in the view named by the Source_View parameter. The updated destination object is left in the source state; the source object is left unchanged.

The Merge_Changes procedure can succeed only if the views named by the Source_View and Destination_Object parameters were created from a common view (for example, by commands such as Make_Path). The configuration object for the common view must still exist. Merge_Changes uses the common ancestor of the two objects to determine the changes from the source object that need to be merged into the destination object.

Merge_Changes compares both the destination object and the source object with the common ancestor to determine the lines that need to be merged. Lines that have been added, deleted, or changed in the source object are correspondingly added, deleted, or changed in the destination object. Lines that have been added, deleted, or changed in the destination object are left as is.

Conflicts exist when the same lines have been changed in both the source and destination objects. When conflicts exist, the destination object is updated to contain the changed lines from both the destination and the source objects. These changed lines are marked with the string "*;". When a unit contains lines marked with "*;", the unit must be edited to remove these marks before it can be compiled.

Besides updating the destination object, the Merge_Changes procedure writes a report containing the text of the Destination_Object in which the following conventions indicate the lines that were affected by the merge:

- Added lines are marked by the + character.
- Deleted lines are redisplayed, marked with the - character.
- Each changed line is indicated as a deleted line followed by an added line.

- Conflicting lines are bracketed by *** START CONFLICT and *** END CONFLICT.

Following the + or - symbol is a number or letter indicating the origin of the modified line:

- The number 1 indicates changes that were merged from the source object.
- The number 2 indicates changes that existed in the destination object.
- The letter B indicates changes that were made in both the source and the destination objects.

The Fail_If_Conflicts_Found parameter can be set to true to cause the command to produce the merge report without actually updating the destination object.

The Merge_Changes procedure is used for updating objects that are not joined—for example, objects in unjoined paths or severed objects in joined paths. In contrast, the Accept_Changes command is used for updating objects that are joined.

Merge_Changes can be used to prepare two objects for joining since objects must be textually identical before they can be joined. To prepare two objects for joining:

1. Merge the source object into the destination object.
2. Check out and edit the destination object to resolve any conflicts.
3. Check out the source object and copy the contents of the destination object into it.

Parameters

```
Destination_Object : String := "<SELECTION>";
```

Specifies the object into which changes are to be merged. If a member of a join set, the specified object must be at the most recent generation (that is, all changes must already be accepted from the other objects in the join set). If the object named by the Destination_Object parameter currently is checked out, the Merge_Changes command automatically checks it in.

The default is the currently selected object.

```
Source_View : String := ">>VIEW_NAME<<";
```

Specifies the view containing the object whose changes are to be merged into the destination object. The object in the designated view must be checked in.

```
Report_File : String := "";
```

Specifies the name for the report file generated by the merge operation. The default value ("") allows the command to generate the filename by appending the string _Merging_Report to the simple name of the destination object. The file is created in the same library as the destination object.

```
Fail_If_Conflicts_Found : Boolean := False;
```

Specifies whether the command should fail to update the destination object if conflicting changes are found.

If true, the command produces the report file without actually updating the destination object. If false (the default value), the command both updates the destination object and produces the report file even if conflicts are found.

```
Comments : String := "";
```

Specifies a comment to be stored in the CMVC database with the notes for the specified object(s). This comment appears in the display generated by the Show_History_By_Generation command.

In addition, the specified comment is logged in the work order indicated by the Work_Order parameter.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Accept_Changes

---

procedure Notes

```
procedure Notes (What_Object : String  := "<CURSOR>";
                 In_Place    : Boolean := False);
```

Description

Displays the history image for the current generation of the specified controlled object.

A history image for a generation contains:

- The history for the generation, which lists the time of checkout and checkin, the user who performed these operations, and comments provided to various CMVC commands
- The notes for the generation, which can be used as a scratchpad for arbitrary commentary to be associated with that generation

History images provide an interactive way to manage notes. From a history image, new notes can be added and saved using the Common.Edit and Common.Commit or Common.Promote commands. Furthermore, operations are available in a history image for displaying notes from other generations.

The Notes procedure thus provides an interactive alternative to the set of file-oriented commands (Get_Notes, Create_Empty_Note_Window, Append_Notes, and Put_Notes). These file-oriented commands are most useful for retrieving notes directly into files, although these commands can put notes into special-purpose notes windows.

The window banner for a history image contains the object name followed by a generation attribute (for example `'G(3)`), followed by the attribute `'History`. Furthermore, the window banner contains the string `(cmvc)`. In contrast, the banner of a notes window brought up by the Get_Notes or Create_Empty_Note_Window procedures contains the string `Notes for` followed by the object name. No interactive operations are available from a `Notes for` window.

Parameters

`What_Object : String := "<CURSOR>";`

Specifies the view or object for which to display the history image. A configuration object also can be specified, even if the corresponding view no longer exists. If an object is specified, it must be controlled and it can be checked out.

The default is the object or view on which the cursor is currently located.

```
In_Place :  Boolean := False;
```

Specifies whether the current frame should be used to display the image. The default specifies that the least recently used frame should be used.

---

procedure Put_Notes

```
procedure Put_Notes (From_File   : String := "<WINDOW>";
                     What_Object : String := "<CURSOR>";
                     Response    : String := "<PROFILE>");
```

Description

Replaces the notes for the specified controlled object with the contents of the specified file.

The notes for a controlled object are stored the CMVC database. An object's notes can be used as a scratchpad for arbitrary commentary to be associated with particular generations.

Put_Notes is one of a set of file-oriented commands for managing notes. That is, these commands, including Get_Notes, Create_Empty_Note_Window, and Append_Notes, are most useful for managing notes through files. However, these commands also manage special-purpose notes windows (identified by the `Notes for` string in the banner) in which the Put_Notes command can be used as follows:

- If the Get_Notes procedure has been used to display an object's notes in a notes window, this window can be modified and its contents saved using the Put_Notes procedure. In this case, Put_Notes must be entered (with default parameter values) from a Command window attached to the window that was created by Get_Notes.
- If the Create_Empty_Note_Window procedure has been used to display an empty notes window for an object, Put_Notes can be used to replace the object's existing notes with any text entered in this window. In this case, Put_Notes must be entered (with default parameter values) from a Command window attached to the window that was created by the Create_Empty_Note_Window command.

Note that modified notes windows retain the * symbol in their window banners, even after their contents have been entered in the CMVC database using Append_Notes or Put_Notes. Accordingly, the Quit command reports these windows as changed images when logout is attempted. Because these windows cannot be committed, use the Common.Abandon procedure to remove these windows.

The Notes command provides an interactive alternative to Get_Notes, Put_Notes, and the like. The Notes command displays a history image (identified by `'History` attribute following the object name and generation in the window banner), which allows interactive operations for managing an object's notes.

Parameters

```
From_File : String := "<WINDOW>";
```

Specifies where to find the new notes for the specified object. If this parameter names a file, the contents of that file replace the existing notes for the specified controlled object.

If the default special name "<WINDOW>" is used, it refers to the contents of a notes window created by either the Get_Notes or the Create_Empty_Note_Window command. When the default value is used, Put_Notes must be entered from a Command window attached to the notes window. The first line of the notes window contains the name of the object associated with the notes; therefore, the What_Object parameter is ignored.

```
What_Object : String := "<CURSOR>";
```

Specifies the object whose notes are to be replaced. The specified object must be both controlled and checked out; otherwise, the command quits.

The What_Object parameter is ignored if the From_File parameter's default value is used.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Append_Notes

procedure Create_Empty_Note_Window

procedure Get_Notes

procedure Notes

procedure Release

```
procedure Release
(From_Working_View          : String                    := "<CURSOR>";
 Release_Name               : String                    := "<AUTO_GENERATE>";
 Level                      : Natural                   := 0;
 Views_To_Import            : String                    := "<INHERIT_IMPORTS>";
 Create_Configuration_Only  : Boolean                   := False;
 Compile_The_View           : Boolean                   := True;
 Goal                       : Compilation.Unit_State    := Compilation.Coded;
 Comments                   : String                    := "";
 Work_Order                 : String                    := "<DEFAULT>";
 Volume                     : Natural                   := 0;
 Response                   : String                    := "<PROFILE>");
```

Description

Creates a new released view from each of the specified working views. Releases can be made in subsystems and in systems.

A released view is a frozen copy of the working view and can serve as a baseline for testing and execution.

In addition to creating a new released view, the Release command creates two objects in the directory *(sub)system_name*.Configurations. These objects are:

- A configuration object named *release_name*.
- A state description directory named *release_name*_State. This directory contains several files that store switch values, the names of exported and imported views, the model name, and the like.

If the newly created view is subsequently destroyed to save space, it can be reconstructed from these objects.

If saving space is important, the Release command can be used to create only the configuration object and the state description directory for each specified working view. Full released views can be created subsequently from the configuration object using the Build command. (Note, however, that a configuration object references only the controlled objects in a view; therefore, only the controlled objects can be created by the Build command.) Creating only a configuration is much faster than making a view.

When a released view is created, the controlled objects in it are automatically joined to the corresponding objects in the working view and in the previously released views in the same development path.

Parameters

```
From_Working_View : String := "<CURSOR>";
```

Specifies one or more working views from which released views are to be created. Multiple working views can be in the same or in different (sub)systems. From_Working_View can specify either combined or load views.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Release_Name : String := "<AUTO_GENERATE>";
```

Specifies the simple name of the new released view(s).

The default special name "<AUTO_GENERATE>" allows new release names to be generated automatically. An automatically generated name consists of the path or subpathname (the portion of the view name up to "_Working") followed by "_*n*_*m*", where *n* and *m* represent automatically incremented level numbers. (The Level parameter controls how these numbers are incremented. The number of levels that can be incremented is determined by the Levels file within the model world for the view.)

If the From_Working_View parameter names multiple views and Release_Name has a nondefault value, all of the new released views will have the same simple name. In contrast, if Release_Name has the default value, the name of each new released view is generated individually.

```
Level : Natural := 0;
```

Specifies which level number to increment within each released view's name. The Level parameter is ignored if a nondefault value of the Release_Name parameter is specified.

The default Level value (0) means that the rightmost number is incremented. If Level is 1, the next-to-rightmost number is incremented and so on. Level numbers to the right of the incremented number are reset to 0.

For example, assume that the previously released view was called Rev1_4_2. If a new release is created from Rev1_Working and Level is 0, the name generated for the next release is Rev1_4_3. If a subsequent release is created from Rev1_Working and Level is 1, the name generated for this next release is Rev1_5_0. (Note that the portion of the name up to "_Working" is fixed, so the "1" in "Rev1" is not subject to being incremented.)

The number of levels that can be incremented is determined by the Levels file within the model world for the view. The Release command quits if the value of the Level parameter exceeds the total number of levels specified by the Levels file.

```
Views_To_Import : String := "<INHERIT_IMPORTS>";
```

Specifies one or more spec or combined views for the new releases to import. The default special name "<INHERIT_IMPORTS>" means that each new released view will have the same imports as the working view from which it was released.

Note that if the From_Working_View parameter names multiple combined views among which import relations hold, the imports are automatically adjusted so that the new releases reference each other as appropriate, instead of referencing the working views.

Imports can be changed during the release operation by specifying a nondefault value for Views_To_Import. However, care must be taken to import views that allow the released views to compile.

Views_To_Import can name an activity as an indirect file; if so, the new releases will import the spec view associated with each subsystem, as listed in the activity.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Create_Configuration_Only : Boolean := False;
```

Specifies whether to save space by creating only the configuration object and the associated state description directory for each specified working view.

If true, only the configuration object and directory are created. A full released view is not created at this time; if desired, the view must be built by a subsequent Build operation. (Note, however, that a configuration object references only the controlled objects in a view; therefore, Build can recreate only the controlled objects for which source has been saved.) Creating only a configuration is much faster than making a view.

If false (the default value), a full released view is created in addition to the configuration object and directory.

Whether or not a view is created in addition to the configuration object, the contents of a configuration can be viewed through a configuration image (see the Edit command).

```
Compile_The_View : Boolean := True;
```

Specifies whether to compile all the units in the specified released views before freezing these views.

If true (the default value), an attempt is made to compile the units to the state specified by the Goal parameter. For example, setting Compile_The_View to true recompiles any units that were demoted by changing imports (that is, by specifying a nondefault value for the Views_To_Import parameter). The views are subsequently frozen even if compilation fails.

If false, units remain demoted.

Unless you are making a configuration-only release, it is recommended that this parameter be left as true to guarantee that released views can be executed.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which units are compiled when the Compile_The_View parameter is true. The compilation goal can be any of the enumerations of the Compilation.Unit_State type. By default, the compilation goal is the coded state. To compile units to the installed state, specify Compilation.Installed. If Compilation.Source or Compilation.Archived is specified, all units in the view are put into this state, regardless of the value of the Compile_The_View parameter.

```
Comments : String := "";
```

Specifies a comment to be stored in the CMVC database. In particular, the comment is stored with the notes for the history files that are associated with the specified working views. The history file for each view is called *view_name*.State.Release_History. The notes can be viewed with the Get_Notes command.

The comment is also logged in the work order indicated by the Work_Order parameter. The comment appears with the checkin of the history file.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records when the history files were checked out and in, the name of the new release, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Volume : Natural := 0;
```

Specifies the volume on which to create the new releases. The default value specifies that the releases should be created on the volume with the most free space.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Remove_Import

```
procedure Remove_Import (View        : String := ">>VIEW NAME<<";
                         From_View   : String := "<CURSOR>";
                         Comments    : String := "";
                         Work_Order  : String := "<DEFAULT>";
                         Response    : String := "<PROFILE>");
```

Description

Removes the links that were created when the view specified by the View parameter was imported.

This command does not remove an import if there are units compiled against any of the links it created. However, such an import can be removed if the units are demoted to the source state.

Parameters

`View : String := ">>VIEW NAME<<";`

Specifies one or more views to be removed from the imports of the view specified by the From_View parameter. The default parameter placeholder must be replaced or an error will result.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`From_View : String := "<CURSOR>";`

Specifies one or more views from which the specified imports are to be removed. Imports cannot be removed from code views. The default is the view designated by the cursor.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Remove_Unused_Imports

---

procedure Remove_Unused_Imports

```
procedure Remove_Unused_Imports (From_View   : String := "<CURSOR>";
                                 Comments    : String := "";
                                 Work_Order  : String := "<DEFAULT>";
                                 Response    : String := "<PROFILE>");
```

Description

Removes imports from the specified view or views if none of the links created by those imports are needed for compilation.

Links are removed only on an import-by-import basis. Thus, if any of the links from a given import are needed for compilation, then none of the links created by that import are removed.

A link is needed for compilation if it is referenced in a ***with*** clause in at least one unit that is in the source, installed, or coded state (archived units are ignored). Compare this with the Remove_Import command, which is sensitive only to units that actually are compiled against the link.

Parameters

`From_View : String := "<CURSOR>";`

Specifies one or more views from which unused imports are to be removed. Imports cannot be removed from code views. The default is the view designated by the cursor.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Remove_Import

---

procedure Replace_Model

```
procedure Replace_Model (New_Model   : String := ">>NEW MODEL NAME<<";
                         In_View     : String := "<CURSOR>";
                         Comments    : String := "";
                         Work_Order  : String := "<DEFAULT>";
                         Response    : String := "<PROFILE>");
```

Description

Replaces the model world for the specified view.

A view's model can be changed to:

- Invoke a new switches file for the view.
- Rebuild the view's links.
- Reset the number of levels for automatic name generation for released and spec views. (This affects only future releases.)
- Change the view's target key. However, the change must be to a target key that is compatible with the current target key. For example, a view with target key R1000 cannot change to a model with target key Mc68020_Bare.

Parameters

`New_Model : String := ">>NEW MODEL NAME<<";`

Specifies the name of the world to be used as the model for the view. The context for the resolution of this name is the world !Model, although a model in another world can be specified by using a fully qualified name.

The default parameter placeholder ">>NEW MODEL NAME<<" must be replaced or an error will result.

`In_View : String := "<CURSOR>";`

Specifies the view whose model is to be replaced. The default is the view designated by the cursor.

All units in the view must be in the source state.

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Revert

```
procedure Revert
    (What_Object              : String                    := "<SELECTION>";
     To_Generation            : Integer                   := -1;
     Make_Latest_Generation   : Boolean                   := False;
     Allow_Demotion           : Boolean                   := False;
     Remake_Demoted_Units     : Boolean                   := True;
     Goal                     : Compilation.Unit_State    := Compilation.Coded;
     Comments                 : String                    := "";
     Work_Order               : String                    := "<DEFAULT>";
     Response                 : String                    := "<PROFILE>");
```

Description

Reverts the specified object or objects to the specified generation.

This procedure replaces the contents of each object with the contents of the indicated generation of that object.

The generation to which an object is reverted can be retained as the latest generation if the Make_Latest_Generation parameter is true. Otherwise, the reverted object is updated to the latest generation the next time the object is checked out.

Parameters

```
What_Object :  String := "<SELECTION>";
```

Specifies the object or objects to be reverted. Only controlled and sourced objects can be reverted. (An error is reported if you try to revert an object that was made controlled without saving source.) An object that is currently checked out cannot be reverted, and this is reported in the output log. By default, the selected object is reverted.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
To_Generation : Integer := -1;
```

Specifies the generation to which the specified object is reverted. A positive integer expresses a particular generation number (each generation is numbered, starting from 0). A negative integer expresses a previous generation, counting back from the object's current generation; for example, the default value of −1 indicates the object's previous generation.

If multiple objects are specified and the To_Generation parameter has a positive value, the Revert procedure attempts to change all objects to the same generation. If multiple objects are specified and To_Generation has a negative value, the generation of each object is calculated individually.

```
Make_Latest_Generation : Boolean := False;
```

Specifies whether to retain To_Generation as the latest generation. If true, To_Generation becomes the latest generation, from which subsequent development can proceed. (In this case, the Revert procedure is equivalent to checking out an object, copying the specified generation into the object, and checking it in.)

If false, To_Generation does not become the latest generation. Consequently, a reverted object can be inspected or compiled against other units; however, the next time the object is checked out, it is updated to the latest generation. (In this case, the Revert procedure is equivalent to using the Accept_Changes command to update an object from a configuration containing the specified generation.)

```
Allow_Demotion : Boolean := False;
```

Specifies whether the Revert procedure is allowed to demote Ada units in order to revert the specified objects to the specified generation.

If this parameter is true, the Revert procedure is permitted to demote Ada units if necessary. If it is false, the command proceeds only if no demotion is required; otherwise, an error is reported and the command quits.

```
Remake_Demoted_Units : Boolean := True;
```

Specifies whether to recompile any units that were demoted in the process of reverting the specified objects.

If true (the default value), demoted units are recompiled to the state specified by the Goal parameter. If false, units remain demoted.

```
Goal : Compilation.Unit_State := Compilation.Coded;
```

Specifies the state to which demoted units are recompiled when the Remake_Demoted_Units parameter is true.

The goal can be any of the enumerations of the Compilation.Unit_State type, except Compilation.Archived. By default, the compilation goal is the coded state. To set the compilation goal to the installed state, specify Compilation.Installed. If Compilation.Source is specified, the demoted units are put in the source state, regardless of the value of the Remake_Demoted_Units parameter.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date, the unit reverted, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Sever

```
procedure Sever (What_Objects                : String := "<SELECTION>";
                 New_Reservation_Token_Name  : String := "<AUTO_GENERATE>";
                 Comments                    : String := "";
                 Work_Order                  : String := "<DEFAULT>";
                 Response                    : String := "<PROFILE>");
```

Description

Severs the specified objects from their respective join sets.

When an object is severed, it is given a different reservation token, so that it can be checked out and modified independent of the objects to which it had previously been joined.

Parameters

```
What_Objects : String := "<SELECTION>";
```

Specifies one or more objects to be severed. By default, the selected object is severed. If a view is specified, all the objects in the view are severed.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
New_Reservation_Token_Name : String := "<AUTO_GENERATE>";
```

Specifies the name of the reservation token to be associated with each newly severed object.

The default special name "<AUTO_GENERATE>" means that the reservation token is generated automatically by the Environment. Automatically generated names of reservation tokens are derived from the first portion of the enclosing view name (up to the first underscore character). For example, the severed objects in a view called Rev1_Working would have "Rev1" as the automatically generated name of the reservation token. However, if "Rev1" is currently in use, then "Rev1_1" is generated.

A user-defined token name can be supplied instead to provide subsequent join sets with more meaningful or mnemonic token names.

Note that supplying an existing reservation token name cannot be used to implicitly join the newly controlled objects to any other objects.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. More specifically, the work order records the time and date of checkin, the objects affected, and the username and session in which the command was entered. If the Comments parameter is specified, this comment also is entered in the work order.

The special name "<DEFAULT>" refers to the default work order for the current session.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Show

```
procedure Show (Objects  : String := "<CURSOR>";
                Response : String := "<PROFILE>");
```

Description

Displays checkout and generation information for the specified controlled objects.

In addition, this procedure lists the views containing objects that are joined to each specified object.

The display produced by the Show procedure includes the following fields:

```
 Object Name    Generation     Where      Chkd Out By Whom Expected Check In
=============== ========== ============ ======== ======= =================
UNITS.CMVC_TEX    3 of 3    REV1_WORKING    Yes      SJL     June 15, 1988
```

For each object listed, the fields display the following information:

Object Name	Displays the portion of the object's name that follows the view name.
Generation	Lists a pair of numbers. The first number is the generation of the object in the current view. The second number is the number of generations that exist for that object.
Where	Displays a view name. If the object is currently checked out, this field names the view in which it is checked out. If the object is currently checked in, it names the view that contains the most recent generation.
Chkd Out	Indicates whether the object is currently checked out. If "Yes," the following two fields provide more information.
By Whom	Displays the username of the user who checked out the object.
Expected Check In	Displays the value that was supplied for the Expected_Check-_In_Time parameter of the Check_Out command.

The Show command also displays the names of the views to which the specified objects are joined.

Parameters

```
Objects : String := "<CURSOR>";
```

Specifies the objects for which information is displayed. If a view is specified, information is displayed for the objects in the Units directory as well as for the Release_History file in the State directory.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Show_All_Checked_Out

```
procedure Show_All_Checked_Out (In_View  : String := "<CURSOR>";
                                Response : String := "<PROFILE>");
```

Description

Displays a list of the objects in the specified view that are checked out.

The objects are listed in the same format used by the Show command.

Parameters

`In_View : String := "<CURSOR>";`

Specifies one or more views whose checked-out objects are to be listed.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Show

procedure Show_All_Controlled

```
procedure Show_All_Controlled (In_View  : String := "<CURSOR>";
                               Response : String := "<PROFILE>");
```

Description

Lists the controlled objects in the specified view or views.

The objects are listed in the same format used by the Show command.

Parameters

`In_View : String := "<CURSOR>";`

Specifies one or more views whose controlled objects are to be listed.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Show

procedure Show_All_Uncontrolled

```
procedure Show_All_Uncontrolled (In_View  : String := "<CURSOR>";
                                 Response : String := "<PROFILE>");
```

Description

Lists all uncontrolled objects in the specified views.

Parameters

`In_View : String := "<CURSOR>";`

Specifies one or more views.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Show_Checked_Out_By_User

```
procedure Show_Checked_Out_By_User
                          (In_View   : String := "<CURSOR>";
                           Who       : String := System_Utilities.User_Name;
                           Response  : String := "<PROFILE>");
```

Description

Lists the objects in the specified view(s) that are checked out by the specified user.

The objects are listed in the same format used by the Show command.

Objects are listed even if they are controlled in the specified view but checked out in another view.

Parameters

`In_View : String := "<CURSOR>";`

Specifies one or more views to be searched for objects checked out by the specified user.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Who : String := "System_Utilities.User_Name";`

Specifies the username whose checked-out objects are to be listed.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Show

procedure Show_Checked_Out_In_View

```
procedure Show_Checked_Out_In_View (In_View  : String := "<CURSOR>";
                                    Response : String := "<PROFILE>");
```

Description

Lists the objects that are checked out in the specified view or views, regardless of who checked them out.

The objects are listed in the same format used by the Show command.

Parameters

`In_View : String := "<CURSOR>";`

Specifies one or more views whose checked-out objects are to be listed.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Show

procedure Show_History

```
procedure Show_History (For_Objects             : String  := "<CURSOR>";
                        Display_Change_Regions  : Boolean := True;
                        Starting_Generation     : String  := "<CURSOR>";
                        Ending_Generation       : String  := "";
                        Response                : String  := "<PROFILE>");
```

Description

Displays the history for the specified view or object within a view.

This procedure shows what has changed between two configurations (or two views) on the same path. For example, the Show_History command can be used to display the differences between two released views, between a working view and a previously released view, and the like. It also can be used to display how a particular object has changed from one view or configuration to another.

The Show_History procedure provides the following information for each specified object (if a view is specified, this information is shown for each controlled object in the view):

- The join set name (the name of the reservation token for the joined objects)
- The object's history for the generations that were created between the configurations specified by the Starting_Generation and Ending_Generation parameters

For each of the requested generations of an object, the history includes:

- The time and date of the checkout and checkin that created the generation
- The notes for the object
- The changes that occurred since the previous generation (if requested by the Display_Change_Regions parameter)

Parameters

```
For_Objects : String := "<CURSOR>";
```

Specifies the object or objects whose history is to be displayed. This parameter can specify one or more views or one or more controlled objects within a view. The default is the object on which the cursor is located.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Display_Change_Regions : Boolean := True;
```

Specifies whether to display the differences between a given generation and the one before it.

If true (the default value), the text of the changes is displayed in the same format as that produced by the !Commands.File_Utilities.Difference(Compressed_Output=> True) procedure (see the LM book of the ***Rational Environment Reference Manual***). If false, no changes are displayed.

```
Starting_Generation : String := "<CURSOR>";
```

Specifies the view or configuration that serves as the starting point for the displayed history. The specified view or configuration must contain some generation of each of the objects designated by the For_Objects parameter. The Show_History procedure displays the history for each object, starting with changes to the generation contained in the specified view or configuration.

The default is the view or configuration on which the cursor is located. If the null string ("") is used, the display starts at generation 1.

```
Ending_Generation : String := "";
```

Specifies the view or configuration that serves as the ending point for the displayed history. The specified view or configuration must contain some generation of each of the objects designated by the For_Objects parameter. The history displayed for each object ends with the generation contained in the specified view or configuration.

The default value ("") specifies that history is displayed up to the latest generation.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Show_History_By_Generation

```
procedure Show_History_By_Generation
                        (For_Objects              : String  := "<CURSOR>";
                         Display_Change_Regions   : Boolean := True;
                         Starting_Generation      : Natural := 1
                         Ending_Generation        : Natural := Natural'Last;
                         Response                 : String  := "<PROFILE>");
```

Description

Displays the history for one or more controlled objects across the specified range of generations.

This procedure uses generation numbers to delimit the extent of the displayed history, whereas the Show_History procedure uses views or configurations to delimit the display.

The Show_History_By_Generation procedure provides the following information for each specified object (if a view is specified, this information is shown for each controlled object in the view):

- The join set name (the name of the reservation token for the joined objects)
- The object's history for the generations that were created between the configurations specified by the Starting_Generation and Ending_Generation parameters

For each of the requested generations of an object, the history includes:

- The time and date of the checkout and checkin that created the generation
- The notes for the object
- The changes that occurred since the previous generation (if requested by the Display_Change_Regions parameter)

Parameters

```
For_Objects : String := "<CURSOR>";
```

Specifies the object or objects whose history is to be displayed. This parameter can specify one or more views or one or more controlled objects within a view. The default is the object on which the cursor is located.

Multiple objects can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Display_Change_Regions : Boolean := True;
```

Specifies whether to display the differences between a given generation and the one before it.

If true (the default value), the text of the changes is displayed in the same format as that produced by the !Commands.File_Utilities.Difference(Compressed_Output=> True) procedure (see the LM book of the *Rational Environment Reference Manual*). If false, no changes are displayed.

```
Starting_Generation : Natural := 1;
```

Specifies the number of the generation to serve as the starting point for the displayed history. The default value (1) causes history to be displayed from generation 1 of the specified objects.

If the For_Objects parameter specifies multiple objects, the displayed history of each object begins with the same generation number, as specified by Starting-_Generation.

```
Ending_Generation : Natural := Natural'Last;
```

Specifies the number of the generation to serve as the ending point for the displayed history. The default value (Natural'Last) causes history to be displayed up to the most recent generation of the specified objects.

If the For_Objects parameter specifies multiple objects, the displayed history of each object ends with the same generation number, as specified by Ending_Generation.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Show_Image_Of_Generation

```
procedure Show_Image_Of_Generation
                                    (Object          : String  := "<CURSOR>";
                                     Generation      : Integer := -1;
                                     Output_Goes_To  : String  := "<WINDOW>";
                                     Response        : String  := "<PROFILE>");
```

Description

Reconstructs an image of the specified generation of the designated controlled object.

Successive generations of a controlled object are stored in the CMVC database as a series of changed increments. This command reconstructs a textual image of the specified generation. The reconstructed image is displayed in the output log, unless the Output_Goes_To parameter specifies a file.

Show_Image_Of_Generation is a report-oriented command that is most useful for putting the image of a single generation into a file. As an alternative, the Edit and Def procedures can be used to bring up generation images from which interactive operations can be used to display images of other generations and of the differences between successive generations.

Parameters

```
Object : String := "<CURSOR>";
```

Specifies the object for which a previous generation is displayed. The default is the object on which the cursor is located.

```
Generation : Integer := -1;
```

Specifies the generation of the specified object that is to be reconstructed. The default value specifies the generation before the current generation of Object.

A negative number specifies a previous generation relative to the object's current generation. For example, a value of −3 displays the third generation back from the current one.

A positive number specifies an actual generation number.

```
Output_Goes_To :  String := "<WINDOW>";
```

Specifies where to put the text of the reconstructed generation. If a new filename is specified, a file is created and the text is written into it. If an existing filename is specified, the contents of that file are replaced.

If the default special name "<WINDOW>" is used, the reconstructed generation is displayed in the window containing the output log.

```
Response :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Show_Out_Of_Date_Objects

```
procedure Show_Out_Of_Date_Objects (In_View  : String := "<CURSOR>";
                                    Response : String := "<PROFILE>");
```

Description

Lists the objects in the specified view or views that are not at the most recent generation.

The objects are listed in the same format used by the Show command.

Parameters

```
In_View : String := "<CURSOR>";
```

Specifies one or more views whose out-of-date objects are to be listed.

Multiple views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Show

type System_Object_Enum

```
type System_Object_Enum is (Spec_Load_Subsystem, Combined_Subsystem,
                                                                System);
```

Description

Defines the types of system objects that can be created, where a system object is either a system or a subsystem.

There are two types of subsystems. A subsystem's type determines what kind of views the subsystem can contain—for example, spec/load views or combined views.

A subsystem's type also determines whether hierarchic importing is enforced.

Enumerations

Combined_Subsystem

Defines a type of subsystem that can contain only combined views. Within a Combined subsystem, circular import relations may hold—that is, a view is permitted to be in its own import closure.

Spec_Load_Subsystem

Defines a type of subsystem that can contain spec, load, or combined views. Within a Spec_Load subsystem, all imports must be hierarchic—that is, no view is permitted to be in its own import closure.

System

Defines a system, which is an optional device for creating logical groupings of releases from component subsystems in an application. Operations for systems are in package Cmvc_Hierarchy.

References

procedure Initial

end Cmvc;

package Cmvc_Hierarchy

When an application consists of multiple subsystems, these subsystems optionally can be included in an Environment object called a *system.* Inclusion in a system is a way of identifying particular subsystems as components of a given application or of a major portion of an application. Inclusion in a system also provides an automated means of tracking the latest release from each subsystem and building activities that reference those releases.

A subsystem is included in a system by establishing a parent-child relationship between the system and the subsystem. Therefore, a system does not actually contain its component subsystems in the same way that a subsystem view contains component objects.

Systems have the same internal directory structure as subsystems. Systems contain views called *system views* (not spec/load or combined). As in subsystems, views in systems contain the same subdirectories found in subsystem views (for example, Units) plus an additional subdirectory called Paths.

The initial system view is a working view. Within the State directory of the working system view, you can build a *release activity.* A release activity automatically contains entries that reference the latest release from each child subsystem. After creating a release activity, you can make a release from the working system view to preserve that activity as a frozen object. Every time new releases are made in child subsystems, you can rebuild the release activity and then make a new release of the working system view. You can use the Cmvc.Information command to display the release activity for a given system view.

A system can contain multiple paths that correspond to the paths in the child subsystems. The release activity in each system path references releases from the corresponding paths in the child subsystems.

Setting Up Systems

1. Use the Cmvc.Initial command to create a system. It contains a working system view.
2. Use the Add_Child command to establish the parent-child relationship between the desired subsystems and the system.
3. At each major release point, you can run the Build_Activity command in the working system view to build (or update) a release activity called Release_Activity that references the latest releases from child subsystems. Release_Activity is located in the State directory.
4. Make the release activity the default and execute the application.
5. If desired, you can edit the release activity using Build_Activity to change one of the activity entries (do not use commands from package Activity to modify a release activity).

After a release activity is created, the releases it references cannot be deleted.

Setting Up Paths

You can use the Cmvc.Make_Path command in a system to create multiple paths, one for each path in the component subsystems. Before building a release activity in a given system path, you must explicitly set up the correspondences between that system path and the desired paths from the child subsystems. To do this:

1. Locate the Paths directory in the working view of the system path.
2. In the Paths directory, create a file corresponding to each child subsystem. The name of each file must be the same as the name of the subsystem to which it corresponds.
3. In the file for each subsystem, enter a naming expression that matches the release names in the desired path from that subsystem.
4. When you build a release activity in a given system path, the entry for each subsystem will reference the latest release that matches the naming expression in the Paths file for that subsystem.

For example, assume that a system called Mail_System has a child subsystem called Mail_Utilities and that the child contains two paths whose prefixes are Rev1 and Rev2. Assume further that Mail_System contains a Rev1 path and that this system path is to reference releases from the Rev1 path in Mail_Utilities. To establish the correspondence between the Rev1 system path and the Rev1 subsystem path:

1. Within the Mail_System.Rev1_Working.Paths directory, create a file called Mail_Utilities.
2. Edit the file, entering a naming expression that matches the release names in the Rev1 path of the Mail_Utilities subsystem—for example: `Rev1@`
3. Commit the file.

4. If a Rev2 path is desired in Mail_System, repeat these steps starting in Mail_System.Rev2_Working.Paths and entering a naming expression such as Rev2@.

The naming expression in a Paths file can match releases from more than one path in a given subsystem. In this case, the latest of the releases from among these paths is entered in the release activity.

Releasing System Views

You can use the Cmvc.Release command to make releases of working system views to preserve release activities as frozen objects. When a system view is released, a subdirectory called Release_Information is created within the released system view. The Release_Information directory contains four controlled text files that can be used to rebuild the release activity and the views it references from configuration objects.

The Release_Information directory for a released system view is shown in Figure 12-1.

```
!Users.Rational.Test_System.Rev1_0_1.Release_Information :    Directory;
  Load_Configurations     :    File;
  Load_Views              :    File;
  Spec_Configurations     :    File;
  Spec_Views              :    File;

SYSTEM.REV1_0_1.RELEASE_INFORMATION [Library]          Directory
```

Figure 12-1. The Release_Information Directory

Assume that you have built a release activity in a working system view and made a release of that view. Furthermore, assume that you have destroyed the released system view without deleting its configuration object and then you have destroyed each of the releases that were referenced in the release activity, without deleting their configuration objects. To rebuild the deleted views and release activity:

1. Use the Cmvc.Build command to rebuild the deleted system-view release from its configuration object. The system view will be rebuilt except for the release activity.
2. Locate the Release_Information directory in the rebuilt system view. Using the Load_Configurations and Spec_Configurations files as indirect files, use the Cmvc.Build command to rebuild the views that were referenced in the release activity. (This step assumes that the child subsystems still exist and contain configuration objects for those views.)
3. From the Units directory of the rebuilt system view, enter the Build_Activity command with default parameters to rebuild and then freeze the release activity. The Build_Activity command automatically consults the files in the Release_Information directory.

procedure Add_Child

```
procedure Add_Child (Child       : String := ">>SYSTEM/SUBSYSTEM NAME<<";
                     To_System   : String := "<CURSOR>";
                     Comments    : String := "";
                     Work_Order  : String := "<DEFAULT>";
                     Response    : String := "<PROFILE>");
```

Description

Adds a new child (a subsystem or another system) to the designated system.

A system provides an automated means of tracking the latest release from each child and building activities that reference those releases.

A system cannot directly or indirectly be a child of itself.

Parameters

`Child : String := ">>SYSTEM/SUBSYSTEM NAME<<";`

Specifies one or more systems or subsystems to be added as children of a system.

Multiple systems and subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`To_System : String := "<CURSOR>";`

Specifies the system to which children are to be added.

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

`Work_Order : String := "<DEFAULT>";`

Specifies the work order in which the command's action is recorded. If the Comments parameter is specified, this comment also is entered in the work order.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command.

procedure Build_Activity

```
procedure Build_Activity (Working_System_View  : String  := "<CURSOR>";
                          Views_To_Include     : String  := "<LATEST>";
                          Update_Imports       : Boolean := True;
                          Allow_Code_Views     : Boolean := False;
                          Comments             : String  := "";
                          Work_Order           : String  := "<DEFAULT>";
                          Response             : String  := "<PROFILE>");
```

Description

Builds or updates the release activity in the working system view to include the specified views.

By default, the latest releases of all the children of the system are included in the release activity. Views are included in the release activity only if they have been created after the Build_Activity command was last run on the specified working system view.

Path restrictions can be used to control which releases are included.

By default, the working system view imports spec views from all of the subsystems referenced by the release activity. Updating the system view's imports allows you to execute test programs from the system view, if desired. Note that this importing is subject to the normal compatibility requirements.

By default, code views are overlooked in favor of including the latest load view in the release activity. However, changing the Allow_Code_Views parameter to true allows code views to be included in the release activity.

Parameters

```
Working_System_View : String := "<CURSOR>";
```

Specifies one or more working system views in which release activities are to be built or updated. By default, the working system view designated by the cursor is used.

Multiple systems and subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Views_To_Include : String := "<LATEST>";
```

Specifies one or more views to be included in the release activity. These views must be in subsystems that are children of the system containing the designated system view.

If the default value, "<LATEST>", is specified, then the latest releases of all the children of the system are included in the release activity. Nondefault values for this parameter are especially useful when using the Build_Activity procedure to change entries in an existing release activity.

Multiple systems and subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Update_Imports : Boolean := True;
```

Specifies whether or not the working system view imports spec views from the subsystems referenced by the release activity.

If true, the default, the working system view imports spec views from all of the subsystems referenced by the release activity. Views are imported as specified by the Views_To_Include parameter. Updating the system view's imports allows you to execute test programs from the system view, if desired. Note that this importing is subject to the normal compatibility requirements.

If false, no spec views are imported.

```
Allow_Code_Views : Boolean := False;
```

Specifies whether to include code views in a release activity.

If false, the default, code views are overlooked in favor of including the latest load view in the release activity.

If true, code views are included in the release activity.

```
Comments : String := "";
```

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

```
Work_Order : String := "<DEFAULT>";
```

Specifies the work order in which the command's action is recorded. If the Comments parameter is specified, this comment also is entered in the work order.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

function Children

```
function Children (Of_System : String  := "<CURSOR>";
                   Recursive : Boolean := True;
                   Response  : String  := "<WARN>") return String;
```

Description

Returns a list of designated subsystem's children.

Parameters

`Of_System : String := "<CURSOR>";`

Specifies the system whose children are to be listed.

`Recursive : Boolean := True;`

Specifies whether to list children recursively when the designated system includes other systems as children. If true, the default, child systems are expanded so that their children are listed.

If false, child systems are listed simply as systems.

`Response : String := "<WARN>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is to list errors and warnings but not positive progress messages.

`return String;`

Returns a list of system children.

function Contents

```
function Contents (Of_System_View : String  := "<CURSOR>";
                   Recursive      : Boolean := True;
                   Response       : String  := "<WARN>") return String;
```

Description

Returns the contents of the release activity of the designated system view.

The function returns a string formatted as a naming expression. This naming expression contains the fully qualified name of each view referenced in the release activity. The names are separated by commas and the entire list is enclosed in brackets.

Parameters

`Of_System_View : String := "<CURSOR>";`

Specifies the system view that contains the release activity whose contents are to be displayed. By default, the system view designated by the cursor is used.

`Recursive : Boolean := True;`

Specifies whether to display release activity contents recursively when a release activity includes references to system views. If true, the default value, references to system views are expanded so that the contents of their release activities are returned.

If false, the contents of release activities are not expanded.

`Response : String := "<WARN>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is to list errors and warnings but not positive progress messages.

`return String;`

Returns the contents of the release activity of the designated system view.

procedure Expand_Activity

```
procedure Expand_Activity
                          (New_Activity  : String := ">>NEW ACTIVITY NAME<<";
                           System_View   : String := "<CURSOR>";
                           Response      : String := "<PROFILE>");
```

Description

Makes a dereferenced copy of the release activity in the designated system view.

That is, in the new release activity, the procedure replaces the entries for system views with the entries from the release activities in those system views.

Parameters

`New_Activity : String := ">>NEW ACTIVITY NAME<<";`
Specifies the name for the new release activity.

`System_View : String := "<CURSOR>";`
Specifies the system view whose release activity is to be copied.

`Response : String := "<PROFILE>";`
Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

function Parents

```
function Parents (Of_Subsystem : String  := "<CURSOR>";
                  Recursive    : Boolean := False;
                  Response     : String  := "<WARN>") return String;
```

Description

Returns a list of systems that are parents to the designated subsystem.

Parameters

`Of_Subsystem : String := "<CURSOR>";`

Specifies the subsystem whose parents are to be listed.

`Recursive : Boolean := False;`

Specifies whether to list parent systems recursively. If true, the default, all parents, grandparents, and so on, are listed.

If false, only direct parent systems are listed.

`Response : String := "<WARN>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is to list errors and warnings but not positive progress messages.

`return String;`

Returns a list of systems that are parents to the designated subsystem.

procedure Remove_Child

```
procedure Remove_Child
                    (Child       : String := ">>SYSTEM/SUBSYSTEM NAME<<";
                     From_System : String := "<CURSOR>";
                     Comments    : String := "";
                     Work_Order  : String := "<DEFAULT>";
                     Response    : String := "<PROFILE>");
```

Description

Severs the relationship between a child system or subsystem and its parent.

This procedure is the opposite of the Add_Child procedure.

Parameters

`Child : String := ">>SYSTEM/SUBSYSTEM NAME<<";`

Specifies one or more child systems or subsystems to be removed.

Multiple systems and subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`From_System : String := "<CURSOR>";`

Specifies the system from which the specified children are to be removed. By default, the system designated by the cursor is used.

`Comments : String := "";`

Specifies a comment to be logged in the work order indicated by the Work_Order parameter. If no work order is specified and if there is no default work order, the comment is discarded.

`Work_Order : String := "<DEFAULT>";`

Specifies the work order in which the command's action is recorded. If the Comments parameter is specified, this comment also is entered in the work order.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

```
end Cmvc_Hierarchy;
```

RATIONAL

package Cmvc_Maintenance

Packages Cmvc_Maintenance defines a set of operations for checking and restoring the integrity of the various databases associated with the CMVC system. This package also provides operations for managing ***primary*** and ***secondary*** subsystems (copies of subsystems that support development on multiple R1000s).

Commands Grouped by Topic

The commands in package Cmvc_Maintenance fall into several functional groups. They are listed here by group for your convenience. (Note that the reference entries for these commands are arranged in alphabetical order by command name.)

- Commands for managing the CMVC database:

  Check_Consistency
  Expunge_Database

- Commands for managing the compatibility database (CDB) and multiple-host development:

  Destroy_Cdb
  Display_Cdb
  Make_Primary
  Make_Secondary
  Repair_Cdb
  Update_Cdb

- Commands for managing code views:

  Display_Code_View

procedure Check_Consistency

```
procedure Check_Consistency (Views    : String := "<CURSOR>";
                             Response : String := "<PROFILE>");
```

Description

Checks the consistency of the specified views with respect to the CMVC database and the Environment library system.

In some cases, corrective action is taken. The specified views can be in subsystems or in systems.

The CMVC database and the Environment library system both record various types of information about controlled objects. The Check_Consistency command makes sure that information in the database agrees with the information in the library system. Specifically, Check_Consistency ensures that:

- There is a configuration object in the Configurations directory for every configuration represented in the database. Missing configuration objects are recreated from the database.
- Both the library system and the CMVC database are synchronized with respect to which objects are controlled. If the library system and the CMVC database do not agree, the information in the library system is changed to match the information in the database.
- The text of each object in the view directories matches the text stored in the CMVC database for the appropriate generation. Note that this is a textual comparison, so that differences due to changed pretty-printer switches will be reported. No action is taken by Check_Consistency to reconcile the differences; the Cmvc.Check_Out or Cmvc.Accept_Changes command can be used to get the latest generation from the database.

The Check_Consistency command also checks the library structure independently of the CMVC database. The Check_Consistency command ensures that:

- The directory structure within the specified views or subsystems is complete. Check_Consistency reconstructs deleted directories and/or missing objects such as the Subpath_Name and Last_Release_Name files in the *view*.State directory. (Note that the Last_Release_Name file contains the level numbers of the most recently released view; when Check_Consistency reconstructs this file, all the level numbers are set to 0 and the file must be edited by hand to restore the correct level numbers.)
- The specified views have a model associated with them. Views that reference deleted models lose that reference; the Cmvc.Replace_Model command can be used to provide new models for those views.

The Check_Consistency command verifies that all imported views still exist and ensures that, whenever a view is imported by another view, both views maintain

a record of this relationship. Discrepancies are resolved in favor of the importing view. That is, if View_1 imports View_2, but View_2 does not list View_1 as a referencer, then View_2's list of referencers is updated to include View_1. On the other hand, if View_2 lists View_1 as a referencer, but View_1 does not list View_2 as an import, View_1 is removed from View_2's list of referencers.

Finally, the Check_Consistency command makes sure that the proper links exist for each specified view. Specifically, Check_Consistency examines the *with* clauses within the specified views' Ada units and reports references for which links do not exist. Furthermore, Check_Consistency reports unacceptable links—namely, links that resolve to load views and links that resolve to unimported spec views. (Typically, such reported links result from improperly using the Links.Add command instead of the CMVC importing operations.)

The Check_Consistency command can be used to:

- Reconstruct a configuration object that was deleted by mistake.
- Recover from an attempt to delete a view with commands in package Library or Compilation. Check_Consistency reconstructs enough of the view so that it can be deleted successfully with the Cmvc.Destroy_View command.
- Reconstruct the directory structure within a view after deleting directories or objects on which other CMVC commands may depend (for example, the State, Exports, or Imports directories.)
- Reconcile conflicting reports and error messages—for example, if error messages indicate that an object is already checked out, whereas commands such as Cmvc.Show_All_Checked_Out have indicated that the object is checked in.

Parameters

```
Views : String := "<CURSOR>";
```

Specifies the views whose consistency is to be checked. The default is the view designated by the cursor. If a subsystem or system is specified, all of the views in that (sub)system are checked, along with the directory structure at the (sub)system level.

Multiple views, subsystems, or systems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Convert_Old_Subsystem

```
procedure Convert_Old_Subsystem (Which    : String := "<SELECTION>";
                                 Response : String := "<PROFILE>");
```

Description

Converts the views in one or more subsystems from the Gamma format to the Delta format so that CMVC operations can be used.

This is not applicable to subsystems created on an R1000 that already has been converted from the Gamma release of the Environment to a Delta release.

Parameters

`Which : String := "<SELECTION>";`

Specifies the subsystem whose views are to be converted. The default is the current selection.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Delete_Unreferenced_Leading_Generations

```
procedure Delete_Unreferenced_Leading_Generations
                                    (In_Subsystem : String := "<CURSOR>";
                                     Response     : String := "<PROFILE>");
```

Description

Not yet implemented.

Parameters

```
In_Subsystem : String := "<CURSOR>";
```

Not yet implemented.

```
Response : String := "<PROFILE>";
```

Not yet implemented.

procedure Destroy_Cdb

```
procedure Destroy_Cdb (Subsystem   : String  := "<SELECTION>";
                       Limit       : String  := "<WORLDS>";
                       Effort_Only : Boolean := True;
                       Response    : String  := "<PROFILE>");
```

Description

Destroys the compatibility database for the specified subsystem.

When units are compiled in a subsystem, information from the compatibility database is incorporated into the DIANA representation for those units. Therefore, when a compatibility database is destroyed, all compiled units in the subsystem are demoted to the source state and all code views are deleted.

When the Effort_Only parameter is true, the compatibility database is not actually destroyed; instead, a report is generated listing the units that would be demoted as a result of destroying the database.

The compatibility database for a subsystem is recreated automatically the next time units are compiled in that subsystem. When recreated, however, the compatibility database provides a new subsystem identification number, effectively severing the subsystem from any secondary or primary subsystems with which it was associated. A subsystem is automatically made primary whenever its compatibility database is destroyed and then recreated.

Destroying a compatibility database may be useful in the following cases:

- A compatibility database may need to be destroyed if it is corrupted—for example, if any of the objects in the *subsystem*.State.Compatibility directory are deleted.
- A compatibility database can be destroyed to remove references to any units that were once compiled in the subsystem but are now deleted.

Parameters

```
Subsystem :  String := "<SELECTION>";
```

Specifies one or more subsystems whose compatibility databases are to be destroyed. The default is the selected subsystem.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Limit : String := "<WORLDS>";
```

Specifies which units can be demoted as a side effect of destroying the compatibility database of each specified subsystem.

The default special value "<WORLDS>" means that demotion is limited to the units in the views of the specified subsystems. Other values for this parameter are given as enumerations of the Compilation.Change_Limit subtype. For example, the string "<ALL_WORLDS>" permits the demotion of units in other subsystems in order to demote the units in the specified subsystems.

```
Effort_Only : Boolean := True;
```

Specifies whether to report the effort required without actually destroying any compatibility databases.

When true (the default value), a report is generated listing the units that would be demoted as a result of destroying the compatibility database. The database is not actually destroyed. The effort rating reported is a relative measure of the amount of work involved.

When false, the compatibility database is destroyed.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Display_Cdb

```
procedure Display_Cdb (Subsystem   : String  := "<CURSOR>";
                       Show_Units  : Boolean := False;
                       Response    : String  := "<PROFILE>");
```

Description

Displays information from the compatibility database for each of the specified subsystems.

A subsystem contains a compatibility database only after units have been promoted to the installed or coded state in that subsystem.

The following information is displayed in the output window:

- Whether the subsystem is primary or secondary
- The subsystem identification number
- How many Ada units are represented in the compatibility database

If the Show_Units parameter is true, each unit is listed along with the number of declarations it contains. Note that every unit that was ever compiled in a given subsystem is represented in that subsystem's compatibility database. Therefore, even deleted units appear in the listing.

Parameters

```
Subsystem :  String := "<CURSOR>";
```

Specifies one or more subsystems whose compatibility database information is to be displayed. The default is the subsystem designated by the cursor.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Show_Units :  Boolean := False;
```

Specifies whether to list the units represented in the compatibility database for the specified subsystems. If true, each unit is listed, followed by the number of declarations it contains. If false (the default), only the total number of units is displayed.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Display_Code_View

```
procedure Display_Code_View (View              : String  := "<CURSOR>";
                             Verbose_Unit_Info : Boolean := False;
                             Show_Map_Info     : Boolean := False;
                             Response          : String  := "<PROFILE>");
```

Description

Displays information about the specified code view.

By default, the command displays a list of units in the code view. (Recall that code views are created by the Cmvc.Make_Code_View command.)

If the Verbose_Unit_Info parameter is true, the command displays the *with*ed units and other compiler information for each unit in the code view.

If the Show_Map_Info parameter is true, the command displays a mapping of the code segments and exceptions from the code view to the original view. Since code views do not support source-level debugging, setting Show_Map_Info to true can be used as a debugging aid.

Parameters

`View : String := "<CURSOR>";`

Specifies one or more code views about which to display information. The default is the code view designated by the cursor.

Multiple code views can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`Verbose_Unit_Info : Boolean := False;`

Specifies whether to display a full report for each unit in the specified code views. If true, the command displays the *with*ed units and other compiler information for each unit in the code view. By default, a full report is not displayed.

`Show_Map_Info : Boolean := False;`

Specifies whether to display the code segment mapping between each specified code view and the load view from which it was generated. By default, the mapping is not displayed.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Expunge_Database

```
procedure Expunge_Database (In_Subsystem : String := "<CURSOR>";
                            Response     : String := "<PROFILE>");
```

Description

Expunges the CMVC database, removing stored information and history about unused configurations or objects.

Expunging the database deletes any configuration represented in the database for which there is no corresponding configuration object in the *subsystem*.Configurations directory.

Expunging the database also deletes any join set represented in the CMVC database if no configuration references any object in the set. All generations associated with the join set are deleted, effectively deleting the history for the unused objects from the database.

The Expunge_Database command is useful when attempting to delete a view and then recreate it with the same name. To do this:

1. Enter the Cmvc.Destroy_View command with the Destroy_Configuration_Also parameter set to true. (This destroys the configuration object and the state description directory along with the view.)
2. Enter the Expunge_Database command to remove references to the configuration from the CMVC database.
3. Recreate the view.

Parameters

```
In_Subsystem : String := "<CURSOR>";
```

Specifies the subsystem whose CMVC database is to be expunged. The default is the subsystem designated by the cursor. A system name can be specified for this parameter as well.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Make_Primary

```
procedure Make_Primary (Subsystem       : String  := "<SELECTION>";
                        Moving_Primary  : Boolean := False;
                        Response        : String  := "<PROFILE>");
```

Description

Converts the specified secondary subsystem into a primary subsystem with its own updatable compatibility database.

When development occurs on multiple R1000s, a copy of each subsystem needs to reside on each machine so that the entire application can be executed. One copy of a given subsystem, called the *primary subsystem*, contains an updatable compatibility database and thus supports ongoing development. The other copies, called *secondary subsystems*, have frozen compatibility databases and essentially are local copies for execution and test. Every secondary subsystem is associated with exactly one primary subsystem and shares its subsystem identification number.

Subsystems created by the Cmvc.Initial command are always created as primary subsystems. A subsystem also is made primary whenever its compatibility database is destroyed and then recreated (see the Destroy_Cdb command). By default, subsystems created by the Archive.Copy or Archive.Restore commands are secondary subsystems, even if they were copied from primary subsystems. (Note, however, that the Options parameter in each of these Archive commands can be set to Primary to create primary subsystems.)

The Make_Primary command converts secondary subsystems to primary subsystems and can be used as one step in the process of:

- Creating a a separate, updatable subsystem from an existing subsystem. To create a new primary subsystem that is not associated with any other existing subsystems:
  1. Make a copy of the existing subsystem using the Archive.Copy command. If the default value for the Options parameter is used, a secondary subsystem is created.
  2. Convert the secondary subsystem to a primary subsystem using the Make_Primary command with the Moving_Primary parameter set to false. The converted subsystem is given a unique subsystem identification number and so is no longer associated with any other primary subsystem.
- Relocating a primary subsystem to a different host. To move a primary subsystem to a location currently occupied by an associated secondary subsystem:
  1. Find or create an associated secondary subsystem on the desired host.
  2. Update the compatibility database in the secondary subsystem using the Update_Cdb command.

3. Convert the secondary subsystem to a primary subsystem using the Make_Primary command with the Moving_Primary parameter set to true. This causes the converted subsystem to retain its original subsystem identification number and thus its previous association with other subsystems.
4. Either destroy the original primary subsystem or convert it to a secondary subsystem with the Make_Secondary command. This step must be done to prevent corruption of the compatibility database.

Care must be taken to ensure that the Moving_Primary parameter has the correct value for the desired operation. In particular, the value false assigns the subsystem a new identification number, severing its association from other subsystems, including its original primary subsystem. The new identification number is retained, even if the subsystem is made secondary again.

Parameters

```
Subsystem : String := "<SELECTION>";
```

Specifies one or more secondary subsystems to be converted to primary subsystems. The default is the selected subsystem.

If the specified subsystem is already a primary subsystem, this command has no effect.

If the specified subsystem contains views with target keys other than R1000, the units in these views cannot be in the coded state.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Moving_Primary : Boolean := False;
```

Specifies whether the converted subsystem is to retain its original subsystem identification number.

When false (the default), the converted subsystem is given a new subsystem identification number and so is no longer associated with its original primary subsystem.

When true, the converted subsystem retains its original subsystem identification number and preserves its previous association with other subsystems. This is intended for moving a primary subsystem to a new location.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Restrictions

If the specified subsystem contains views with target keys other than R1000, the units in these views cannot be in the coded state.

References

procedure Destroy_Cdb

procedure Make_Secondary

procedure Update_Cdb

LM, procedure Archive.Copy

procedure Make_Secondary

```
procedure Make_Secondary (Subsystem : String := "<SELECTION>";
                          Response  : String := "<PROFILE>");
```

Description

Converts the specified primary subsystem into a secondary subsystem with a read-only compatibility database.

When development occurs on multiple R1000s, a copy of each subsystem needs to reside on each machine so that the entire application can be executed. One copy of a given subsystem, called the ***primary subsystem***, contains an updatable compatibility database and thus supports ongoing development. The other copies, called *secondary subsystems*, have frozen compatibility databases and essentially are local copies for execution and test. Every secondary subsystem is associated with exactly one primary subsystem and shares its subsystem identification number.

By default, secondary subsystems are created by the Archive.Copy or Archive.Restore commands, even if they were copied from primary subsystems. (Note, however, that the Options parameter in each of these Archive commands can be set to create primary subsystems.)

The Make_Secondary command is used in the last step in the process of moving a primary subsystem:

1. Update the compatibility database in the secondary subsystem using the Update_Cdb command.
2. Convert the secondary subsystem to a primary subsystem using the Make_Primary command with the Moving_Primary parameter set to true. This causes the converted subsystem to retain its original subsystem identification number and thus its previous association with other subsystems.
3. Either destroy the original primary subsystem or convert it to a secondary subsystem with the Make_Secondary command. This step must be done to prevent corruption of the compatibility database.

Parameters

```
Subsystem : String := "<SELECTION>";
```

Specifies one or more primary subsystems to be converted to secondary subsystems. The default is the selected subsystem.

If the specified subsystem is already a secondary subsystem, this command has no effect.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Make_Primary

procedure Update_Cdb

LM, procedure Archive.Copy

LM, procedure Archive.Restore

procedure Repair_Cdb

```
procedure Repair_Cdb (Subsystem      : String  := "<SELECTION>";
                      Verify_Only    : Boolean := True;
                      Delete_Current : Boolean := False;
                      Response       : String  := "<PROFILE>");
```

Description

Verifies that the information in the specified subsystem's compatibility database is consistent with the DIANA representation of the subsystem's compiled units.

When the Verify_Only parameter is false, some or all of the inconsistencies are repaired, depending on the value of the Delete_Current parameter.

When units are compiled in a subsystem, information from the compatibility database is incorporated into the DIANA representation for those units. If the compatibility database is corrupted after units have been compiled, it can be repaired or rebuilt using information from the DIANA representation of the compiled units. For example, if an object in the *subsystem*.State.Compatibility directory is deleted, the Repair_Cdb command can rebuild the object.

Note that Repair_Cdb can rebuild a compatibility database only from the DIANA representation of installed or coded units. Therefore, Repair_Cdb cannot be used to restore a database destroyed by the Destroy_Cdb command, because Destroy_Cdb also deletes the DIANA representation.

As long as there is at least one installed or coded unit in the subsystem, the database can be rebuilt with the same subsystem identification number.

Parameters

```
Subsystem : String := "<SELECTION>";
```

Specifies one or more subsystems whose compatibility databases are to be repaired. The default is the selected subsystem.

Multiple subsystems can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

```
Verify_Only : Boolean := True;
```

Specifies whether to verify the consistency of the compatibility database without actually trying to repair it.

When true (the default value), only a report is generated, and no repair is undertaken.

When false, an attempt is made to repair inconsistencies between the compatibility database and the DIANA representation of compiled units. The extent of the repair depends on the value of the Delete_Current parameter.

```
Delete_Current : Boolean := False;
```

Specifies whether to delete and rebuild the entire existing compatibility database.

If false (the default value), the existing compatibility database is preserved. Existing entries in the compatibility database are verified and missing entries are added.

If true, the entire database is deleted and rebuilt.

Regardless of the value of this parameter, the database can be rebuilt with the same subsystem identification number, provided that at least one installed or coded unit is in the subsystem.

The value of the Delete_Current parameter is ignored if the Verify_Only parameter is true.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Update_Cdb

```
procedure Update_Cdb (From_Subsystem : String := "<ASSOCIATED_PRIMARY>";
                      To_Subsystem   : String := "<SELECTION>";
                      Response       : String := "<PROFILE>");
```

Description

Updates a secondary subsystem's compatibility database by copying the compatibility database from another subsystem.

The two subsystems must have the same subsystem identification number, although they can be on different R1000s.

Typically, a compatibility database is copied from a primary subsystem into an associated secondary subsystem to:

- Compile incremental changes in the secondary subsystem
- Prepare a secondary subsystem to be converted to a primary subsystem (see the Make_Primary command)

Note that the compatibility database is automatically moved whenever Archive.Copy is used to copy views or individual units from one subsystem into another. In contrast, the Update_Cdb command copies only the compatibility database. Thus, using the Update_Cdb command is equivalent to entering the Archive.Copy command with `Options => "cdb"`.

The Update_Cdb command cannot be used to revert a compatibility database to a previous version. See the Revert_Cdb value of the Option parameter of the Archive.Copy command.

Parameters

```
From_Subsystem : String := "<ASSOCIATED_PRIMARY>";
```

Specifies the subsystem whose compatibility database is to be copied. The compatibility database in From_Subsystem must be more recent than the compatibility database in To_Subsystem.

The default special name "<ASSOCIATED_PRIMARY>" designates the primary subsystem associated with the secondary subsystem specified by the To_Subsystem parameter. The default value gets the name of the associated primary subsystem from the *subsystem*.State.Compatibility.State file within the secondary subsystem. Note that if the primary subsystem has been moved, this file may be out of date. If so, the file must be edited to supply the correct pathname for the associated primary subsystem; otherwise, the name of the associated primary must be specified as the value for the From_Subsystem parameter.

```
To_Subsystem : String := "<SELECTION>";
```

Specifies the secondary subsystem whose compatibility database is to be updated. The default is the selected subsystem.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Make_Primary

LM, procedure Archive.Copy

end Cmvc_Maintenance;

RATIONAL

package Work_Order

This package provides operations for creating, viewing, and manipulating work orders, work-order lists, and ventures. These objects can be associated with user sessions (that is, with user IDs and session names) to collect and convey data about project management among team members involved in large-system development using subsystems.

Many characteristics of work orders, work-order lists, and ventures are controllable with session switches, which are described in a following section.

Work orders are designed to communicate details about specific tasks to be accomplished. They may present instructions to a developer and collect project work data to mark ongoing progress. A particular work order may address one or more developers, but typically it is limited in scope; a work order may describe one bug to be fixed, for example.

Groups of related work orders constitute a *work-order list.* For example, a work-order list may relate to a particular module of code or it may be the set of work orders assigned to an individual developer.

A larger component of project management is the *venture.* A venture is a management tool that contains information about groups of work orders and work-order lists and controls their use via *venture policy switches.* Each work order must have a venture that is its "parent."

Ventures, work-order lists, and work orders are library objects. For each project-management object, package Work_Order provides subprograms to:

- Create, display, and edit the object
- View and set the default object for a user or session
- View and set the textual notes within the object

Although there are editing commands in packages within package Work_Order for each object, viewing and editing of work orders, work-order lists, and ventures is perhaps most easily done with commands from package Common. See the following introductions to subpackages Editor, List_Editor, and Venture_Editor for sample displays and specific information about using Common commands for editing.

Session Switches

Many session switches determine how information in work orders, work-order lists, and ventures are formatted. See SJM, Session Switches, for more information on session switches.

The following session switches pertain to project-reporting objects. Unless otherwise specified, the full name for each switch begins with Session. For example, the full name for Cmvc_Break_Long_Lines is Session.Cmvc_Break_Long_Lines.

Cmvc_Break_Long_Lines (default true)

Controls whether lines in work orders, work-order lists, and ventures that exceed the value of the Cmvc_Line_Length switch are broken. User-entered strings are never broken.

Cmvc_Capitalize (default true)

Determines whether words, other than those in user-entered strings, in work orders, work-order lists, and ventures are capitalized.

Cmvc_Comment_Extent (default 4)

Specifies, as an integer value, the number of comments displayed in a work order.

Cmvc_Configuration_Extent (default 0)

Specifies, as an integer value, the number of configurations displayed in a work order.

Cmvc_Field_Extent (default 4)

Specifies, as an integer value, the number of elements of vector fields that are displayed in a work order.

Cmvc_Indentation (default 2)

Specifies, as an integer value, the number of spaces used for indentation in work orders, work-order lists, and ventures.

Cmvc_Line_Length (default 80)

Specifies, as an integer value, how long a line in a work order, work-order list, or venture can be before it is eligible to be broken.

Cmvc_Shorten_Name (default true)

Shows object names in work orders, work-order lists, and ventures in a shortened form.

Cmvc_Shorten_Unit_State (default false)

Shows the state of work orders in a shortened form.

Cmvc_Show_Add_Date (default true)

Displays the date an entry is added to a work order.

Cmvc_Show_Add_Time (default true)

Displays the time an entry is added to a work order.

Cmvc_Show_All_Default_Lists (default false)

Displays only the user's default work-order list in a venture.

Cmvc_Show_All_Default_Orders (default false)

Displays only the user's default work order in a venture.

Cmvc_Show_Deleted_Objects (default false)

Shows deleted work orders or work-order lists in a work-order list. Display of deleted objects is controlled by elision.

Cmvc_Show_Deleted_Versions (default false)

Shows version numbers and information for all versions of a work order or work-order list. Display of deleted versions is controlled by elision.

Cmvc_Show_Display_Position (default false)

Shows display position of user-defined work-order fields.

Cmvc_Show_Edit_Info (default false)

Shows edit information for objects displayed in work orders, work-order lists, or ventures.

Cmvc_Show_Field_Default (default true)

Shows the default value for vector fields. If this switch is true, vector fields will show the default value of all elements that have not been assigned.

Cmvc_Show_Field_Max_Index (default false)

Shows the number of entries in a vector field that have been written.

Cmvc_Show_Field_Type (default false)

Shows the type of field (that is, Boolean, integer, or string) for all scalar and vector fields.

Cmvc_Show_Frozen (default false)

Shows "Frz" for frozen objects displayed in work orders, work-order lists, or ventures.

Cmvc_Show_Hidden_Fields (default false)

Displays hidden fields in a venture.

Cmvc_Show_Retention (default false)

Shows the retention count when displaying objects in work orders, work-order lists, or ventures.

Cmvc_Show_Boolean (default false)

Shows the size of the version, in bytes, when displaying objects in a work order, work-order list, or venture.

Cmvc_Show_Unit_State (default true)

Shows the state of work orders listed in ventures and work-order lists (that is, pending, in progress, closed).

Cmvc_Show_Users (default false)

Shows the list of users in the users field of work orders. Display of users is controlled by elision.

Cmvc_Show_Version_Number (default false)

Shows the version number of objects displayed in work orders, work-order lists, or ventures.

Cmvc_Uppercase (default false)

Determines whether words, other than those in user-entered strings, in work orders, work-order lists, and ventures are displayed in uppercase.

Cmvc_Version_Extent (default 0)

Specifies, as an integer value, the number of versions displayed in a work order.

Default_Venture

Specifies a filename for the default venture for the session. The full switch name is Cmvc.Default_Venture.

procedure Add_To_List

```
procedure Add_To_List (Order_Names : String := "<IMAGE>";
                       List_Name   : String := "<WORK_LIST>";
                       Response    : String := "<PROFILE>");
```

Description

Adds one or more work orders to a work-order list.

Parameters

`Order_Names : String := "<IMAGE>";`

Specifies one or more work orders to be added to the list. The default special name "<IMAGE>" designates the currently selected work order if the cursor is in the selection; otherwise, it designates the work order in the current image.

Multiple work-order names can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`List_Name : String := "<WORK_LIST>";`

Specifies the work-order list to which work orders will be appended. The default special name "<WORK_LIST>" specifies the default work-order list for the current session.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Errors

An error will result if the work order(s) specified by the Order_Names parameter were not created on the same venture as the work-order list specified by the List_Name parameter.

References

procedure Remove_From_List

procedure Close

```
procedure Close (Order_Name : String := "<ORDER>";
                 Response   : String := "<PROFILE>");
```

Description

Sets the status of the specified work order to closed.

Once a work order has been closed, it no longer can be modified.

Parameters

`Order_Name : String := "<ORDER>";`

Specifies the work order to be closed. The default special name `"<ORDER>"` specifies the default work order for the current session. The null string (`""`) is interpreted to mean `"<CURSOR>"`.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Create

```
procedure Create (Order_Name                : String  := ">>OBJECT NAME<<";
                  Notes                     : String  := "";
                  On_List                   : String  := "<WORK_LIST>";
                  On_Venture                : String  := "<VENTURE>";
                  Make_Default_Work_Order   : Boolean := True;
                  Response                  : String  := "<PROFILE>");
```

Description

Creates a work order on the specified venture and adds it to a work-order list.

The new work order is created on the default venture for the current session unless the On_Venture parameter names a venture. The string specified in the Notes parameter is entered into the notes field of the new work order. If the Make_Default_Work_Order parameter is true, the new work order becomes the default work order on the parent venture.

Parameters

`Order_Name : String := ">>OBJECT NAME<<";`

Specifies the name for the new work order. The default parameter placeholder ">>OBJECT NAME<<" must be replaced or an error will result.

`Notes : String := "";`

Specifies a string to be saved in the notes field of the work order. Notes typically are used to provide a brief description of the work order.

`On_List : String := "<WORK_LIST>";`

Specifies a work-order list to which the new work order is appended. The default special name "<WORK_LIST>" specifies the default work-order list for the current session. If the current session has no default work-order list, a warning message appears in the output log. If the value of this parameter is the null string (""), the work order is not added to any work-order list.

`On_Venture : String := "<VENTURE>";`

Specifies the venture for which the work order is created. The default special name "<VENTURE>" specifies the default venture for the current session.

```
Make_Default_Work_Order  :  Boolean := True;
```

Specifies whether to set the new work order as the session default. If true (the default value), the new work order becomes the default work order on the specified venture.

```
Response  :  String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Errors

An error will result if the work-order list specified by the On_List parameter was not created on the venture specified by the On_Venture parameter.

References

function Default

procedure Set_Default

---

procedure Create_Field

```
procedure Create_Field
              (Field_Name        : String  := ">>FIELD NAME<<";
               Field_Type        : String  := ">>BOOLEAN|STRING|INTEGER<<";
               Is_Vector         : Boolean := False;
               Is_Controlled     : Boolean := False;
               Default           : String  := "";
               Display_Position  : Natural := Natural'Last;
               On_Venture        : String  := "<VENTURE>";
               Propagate         : Boolean := True;
               Renumber_Fields   : Boolean := True;
               Response          : String  := "<PROFILE>");
```

Description

Creates a new user-defined field with the designated data type in the specified venture.

This field appears in all work orders subsequently created on this venture. If the Propagate parameter is true, all work orders already created on this venture are updated to contain this field. The new field appears with the initial value specified by the Default parameter.

Parameters

`Field_Name : String := ">>FIELD NAME<<";`

Specifies the name for the user-defined field. The default parameter placeholder ">>FIELD NAME<<" must be replaced or an error will result.

`Field_Type : String := ">>BOOLEAN|STRING|INTEGER<<";`

Specifies the data type for the new field. Fields can be created that contain Boolean, string, or integer data. The default parameter placeholder ">>BOOLEAN|STRING|INTEGER<<" must be replaced or an error will result.

`Is_Vector : Boolean := False;`

Specifies whether the field accepts an array of values or a single value. If false (the default value), the field accepts a single scalar value.

If true, the field accepts an array of values. The range of array indexes is 1..Positive'Last. Because each value of a user-defined field can be modified only once, the field should be created as a vector if its value will need to be updated. Successive elements in the array then can be modified. In this way, a history of changes to this field is provided.

```
Is_Controlled : Boolean := False;
```

Specifies whether the new field is controlled by the Allow_Edit_Of_Work_Orders policy switch.

If true, the field can be edited interactively only if the Allow_Edit_Of_Work_Orders policy switch is true in the venture. If false (the default value), the field can be edited interactively regardless of the value of the Allow_Edit_Of_Work_Orders policy switch.

```
Default : String := "";
```

Specifies an initial value for the field. The initial value will appear on new work orders and in existing work orders if the field is propagated.

```
Display_Position : Natural := Natural'Last;
```

Specifies the display position of the field in the work order. If this is set to 0, the field will not be visible when the work order is displayed with the editor.

```
On_Venture : String := "<VENTURE>";
```

Specifies the name of the venture to which the new field is added. The default special name "<VENTURE>" specifies the default venture for the current session.

```
Propagate : Boolean := True;
```

Specifies, if true, that the field will be added to all existing work orders on the venture.

```
Renumber_Fields : Boolean := True;
```

Specifies, if true, that the fields will be reordered if this is necessary to make the new field have the Display_Position ordinal.

```
Response : String := "<PROFILE>";
```

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

---

procedure Create_List

```
procedure Create_List (List_Name          : String  := ">>OBJECT NAME<<";
                       Notes              : String  := "";
                       On_Venture         : String  := "<VENTURE>";
                       Make_Default_List  : Boolean := True;
                       Response           : String  := "<PROFILE>");
```

Description

Creates a work-order list on the specified venture.

Parameters

`List_Name : String := ">>OBJECT NAME<<";`

Specifies the name of the new work-order list. The default parameter placeholder ">>OBJECT NAME<<" must be replaced or an error will result.

`Notes : String := "";`

Specifies a string to be saved in the notes field of the work-order list. Notes typically are used to provide a brief description of the work-order list.

`On_Venture : String := "<VENTURE>";`

Specifies the name of the venture to which the new work-order list is added. The default special name "<VENTURE>" specifies the default venture for the current session.

`Make_Default_List : Boolean := True;`

Specifies whether to set the new work-order list as the session default. If true (the default value), the new list becomes the new default work-order list in the specified venture.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Set_Default_List

procedure Create_Venture

```
procedure Create_Venture
                    (Venture_Name          : String  := ">>OBJECT NAME<<";
                     Notes                 : String  := "";
                     Make_Default_Venture  : Boolean := True;
                     Response              : String  := "<PROFILE>");
```

Description

Creates a new venture.

Parameters

`Venture_Name : String := ">>OBJECT NAME<<";`

Specifies the name of the new venture. The default parameter placeholder ">>OBJECT NAME<<" must be replaced or an error will result.

`Notes : String := "";`

Specifies a string to be saved in the notes field of the venture. Notes typically are used to provide a brief description of the venture.

`Make_Default_Venture : Boolean := True;`

Specifies whether to set the new venture as the default. If true (the default value), the new venture becomes the default venture for the current session.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Set_Default_Venture

function Default

```
function Default (For_Venture     : String  := "<VENTURE>";
                  For_User        : String  := "<CURRENT_USER>";
                  Ignore_Garbage  : Boolean := True) return String;
```

Description

Returns the name of the user's default work order in the specified venture.

Parameters

`For_Venture : String := "<VENTURE>";`

Specifies the name of the venture to reference. The default special name "<VENTURE>" specifies the default venture for the current session.

`For_User : String := "<CURRENT_USER>";`

Specifies the username for which the default work order is requested. If only a username is supplied, session S_1 is assumed. If the user has multiple sessions with default work orders in the venture, both the username and the session name must be specified when the default work order for a session other than S_1 is desired. The default special name "<CURRENT_USER>" specifies the current session.

`Ignore_Garbage : Boolean := True;`

Specifies how to present results in case the once-valid default work order is missing. If true (the default), the function result is "<>". If false, the contents of the function result are unpredictable. When function results are to be used directly in CMVC commands, it is recommended that Ignore_Garbage is true.

`return String;`

Returns the pathname of the default work order.

Example

The command:

```
Text_Io.Put_Line(Work_Order.Default)
```

displays the name of the default work order for the current user's current session in the default venture.

The command:

```
Text_Io.Put_Line(Work_Order.Default  (For_Venture => "My_Venture",
                                      For_User    => "User1"));
```

displays the name of the default work order for user User1, session S_1, in My_Venture.

The command:

```
Text_Io.Put_Line(Work_Order.Default  (For_Venture => "My_Venture",
                                      For_User    => "User1.Working"));
```

displays the name of the default work order for user User1, session Working, in My_Venture.

References

procedure Set_Default

---

function Default_List

```
function Default_List (For_Venture     : String  := "<VENTURE>";
                       For_User        : String  := "<CURRENT_USER>";
                       Ignore_Garbage  : Boolean := True) return String;
```

Description

Returns the name of the user's default work-order list in the specified venture.

Parameters

`For_Venture : String := "<VENTURE>";`

Specifies the name of the venture to reference. The default special name "<VENTURE>" specifies the default venture for the current session.

`For_User : String := "<CURRENT_USER>";`

Specifies the username for which the default work-order list is requested. If only a username is supplied, session S_1 is assumed. If the user has multiple sessions with default work-order lists in the venture, the username and the session name must be specified when the default work-order list for a session other than S_1 is desired. The default parameter "<CURRENT_USER>" specifies the current session.

`Ignore_Garbage : Boolean := True;`

Specifies how to present results in case the once-valid default work- order list is missing. If true (the default), the function result is "<>". If false, the contents of the function result are unpredictable. When function results are to be used directly in CMVC commands, it is recommended that Ignore_Garbage is true.

`return String;`

Returns the pathname of the default work-order list.

Example

The command:

```
Text_Io.Put_Line(Work_Order.Default_List)
```

displays the name of the default work-order list for the current user's current session in the default venture.

The command:

```
Text_Io.Put_Line(Work_Order.Default_List  (For_Venture => "My_Venture",
                                           For_User    => "User1"));
```

displays the name of the default work-order list for user User1, session S_1, in My_Venture.

The command:

```
Text_Io.Put_Line(Work_Order.Default_List  (For_Venture => "My_Venture",
                                           For_User    =>
                                                        "User1.Working"));
```

displays the name of the default work-order list for user User1, session Working, in My_Venture.

References

procedure Set_Default_List

---

function Default_Venture

```
function Default_Venture (For_User       : String  := "<CURRENT_USER>";
                          Ignore_Garbage : Boolean := True) return String;
```

Description

Returns the pathname of the default venture for a user.

Parameters

`For_User : String := "<CURRENT_USER>";`

Specifies the username for which the default venture is requested. If only a username is used, session S_1 is assumed. If the user has multiple sessions and wants the default venture for a session other than S_1, the session name must be specified. The default special name "<CURRENT_USER>" specifies the current session.

`Ignore_Garbage : Boolean := True;`

Specifies how to present results in case the once-valid default venture is missing. If true (the default), the function result is "<>". If false, the contents of the function result are unpredictable. When function results are to be used directly in CMVC commands, it is recommended that Ignore_Garbage is true.

`return String;`

Returns the pathname of the default venture.

Example

The command:

```
Text_Io.Put_Line(Work_Order.Default_Venture  (For_User => "User1"));
```

displays the name of the default venture for user User1, session S_1.

The command:

```
Text_Io.Put_Line(Work_Order.Default_Venture  (For_User =>
                                              "User1.Working"));
```

displays the name of the default venture for user User1, session Working.

function Default_Venture
package !Commands.Work_Order

References

procedure Set_Default_Venture

---

procedure Delete_Field

```
procedure Delete_Field (Field_Name           : String  := ">>FIELD NAME<<";
                        Venture_Name         : String  := "<VENTURE>";
                        Even_If_Data_Present : Boolean := False;
                        Response             : String  := "<PROFILE>");
```

Description

Deletes the named field from the venture.

Parameters

`Field_Name : String := ">>FIELD NAME<<";`

Specifies the name of the field to be deleted.

`Venture_Name : String := "<VENTURE>";`

Specifies the name of the venture from which the field is deleted. The default parameter "<VENTURE>" specifies the default venture for the current session.

`Even_If_Data_Present : Boolean := False;`

Specifies whether to delete the field despite the presence of data in this field for existing work orders. If false (the default value), the field is not deleted when data would be lost.

If true, the field is deleted from the specified venture and any data in that field of work orders is lost.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Create_Field

procedure Display

```
procedure Display (Order_Name : String := "<ORDER>";
                   Options    : String := "";
                   Response   : String := "<PROFILE>");
```

Description

Formats and displays the contents of the specified work order in the output window.

The format of the display is controlled by the Options parameter. This display cannot be edited; to edit a work order, see the Edit command.

Parameters

`Order_Name : String := "<ORDER>";`

Specifies the name of the work order to be displayed. The default special name "<ORDER>" specifies the default work order for the current session. The null string ("") is interpreted to mean "<CURSOR>".

`Options : String := "";`

Specifies the format of the display. Valid options include names and values for any of the session switches described at the beginning of package Work_Order. When using these switch names as options, omit "Cmvc_" in the switch name.

The following special options exist:

<TERSE> the default, specifies an abbreviated display
<DEFAULT> specifies use of current session-switch values
<VERBOSE> specifies an explanatory display

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Edit

procedure Display_List

```
procedure Display_List (List_Name : String := "<WORK_LIST>";
                        Options   : String := "";
                        Response  : String := "<PROFILE>");
```

Description

Formats and displays the contents of the specified work-order list in the output window.

The format of the display is controlled by the Options parameter. This display cannot be edited; to edit a work-order list, see the Edit_List command.

Parameters

`List_Name : String := "<WORK_LIST>";`

Specifies the name of the work-order list to be displayed. The default special name "<WORK_LIST>" specifies the default work-order list for the session. The null string ("") is interpreted to mean "<CURSOR>".

`Options : String := "";`

Specifies the format of the display. Valid options include names and values for any of the session switches described at the beginning of package Work_Order. When using these switch names as options, omit "Cmvc_" in the switch name.

The following special options exist:

<TERSE> the default, specifies an abbreviated display
<DEFAULT> specifies use of current session-switch values
<VERBOSE> specifies an explanatory display

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Edit_List

procedure Display_Venture

```
procedure Display_Venture (Venture_Name  : String := "<VENTURE>";
                           Options       : String := "";
                           Response      : String := "<PROFILE>");
```

Description

Formats and displays the contents of the specified venture in the output window.

The format of the display is controlled by the Options parameter. This display cannot be edited; to edit a venture, see the Edit_Venture command.

Parameters

`Venture_Name : String := "<VENTURE>";`

Specifies the name of the venture to be displayed. The default special name "<VENTURE>" specifies the default venture for the current session. The null string ("") is interpreted to mean "<CURSOR>".

`Options : String := "";`

Specifies the format of the display. Valid options include names and values for any of the session switches described at the beginning of package Work_Order. When using these switch names as options, omit "Cmvc_" in the switch name.

The following special options exist:

<TERSE> the default, specifies an abbreviated display
<DEFAULT> specifies use of current session-switch values
<VERBOSE> specifies an explanatory display

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

procedure Edit_Venture

procedure Edit

```
procedure Edit (Order_Name : String := "<ORDER>");
```

Description

Edits the designated work order.

The procedure creates a window in which the designated work order is displayed. If a window already exists for that work order, the window is reused. From the window, the work order can be edited with the operations from package !Commands.Common that apply to this class of object.

Parameters

`Order_Name : String := "<ORDER>";`

Specifies the name of the work order to be edited. The default special name "<ORDER>" specifies the default work order for the current session. The null string ("") is interpreted to mean "<CURSOR>".

procedure Edit_List

```
procedure Edit_List (List_Name : String := "<WORK_LIST>");
```

Description

Edits the designated work-order list.

The procedure creates a window in which the designated work-order list is displayed. If a window already exists for that work-order list, the window is reused. From the window, the work-order list can be edited with the operations from package !Commands.Common that apply to this class of object.

Parameters

`List_Name : String := "<WORK_LIST>";`

Specifies the name of the work-order list to be edited. The default special name "<WORK_LIST>" specifies the default work-order list for the current session. The null string ("") is interpreted to mean "<CURSOR>".

References

package Editor

procedure Edit_Venture

```
procedure Edit_Venture (Venture_Name : String := "<VENTURE>");
```

Description

Edits the designated venture.

The procedure creates a window in which the designated venture is displayed. If a window already exists for that venture, the window is reused. From the window, the venture can be edited with the operations from package !Commands.Common that apply to this class of object.

Parameters

`Venture_Name : String := "<VENTURE>";`

Specifies the name of the venture to be edited. The default special name "<VENTURE>" specifies the default venture for the current session. The null string ("") is interpreted to mean "<CURSOR>".

References

package List_Editor

function Notes

```
function Notes (Order_Name : String := "<ORDER>") return String;
```

Description

Returns the notes field of the specified work order.

The notes field typically contains descriptive information about a work order.

Parameters

```
Order_Name : String := "<ORDER>";
```

Specifies the work order whose notes field is to be displayed. The default special name "<ORDER>" specifies the default work order for the current session. The null string ("") is interpreted to mean "<CURSOR>".

```
return String;
```

Returns the notes field of the specified work order.

References

package Venture_Editor

function Notes_List

```
function Notes_List (List_Name : String := "<WORK_LIST>") return String;
```

Description

Returns the notes field of the specified work-order list.

The notes field typically contains descriptive information about a work-order list.

Parameters

```
List_Name : String := "<WORK_LIST>";
```

Specifies the work-order list whose notes field is to be displayed. The default special name "<WORK_LIST>" specifies the default work-order list for the current session. The null string ("") is interpreted to mean "<CURSOR>".

```
return String;
```

Returns the notes field of the specified work-order list.

function Notes_Venture

```
function Notes_Venture (Venture_Name : String := "<VENTURE>")
                                                    return String;
```

Description

Returns the notes field for the specified venture.

The notes field typically contains descriptive information about a venture.

Parameters

`Venture_Name : String := "<VENTURE>";`

Specifies the venture whose notes field is to be displayed. The default special name "<VENTURE>" specifies the default venture for the current session. The null string ("") is interpreted to mean "<CURSOR>".

`return String;`

Returns the notes field for the specified venture.

procedure Remove_From_List

```
procedure Remove_From_List (Order_Names : String := "<IMAGE>";
                            List_Name   : String := "<WORK_LIST>";
                            Response    : String := "<PROFILE>");
```

Description

Removes the entry for the specified work order from a work-order list.

Parameters

`Order_Names : String := "<IMAGE>";`

Specifies one or more work orders to be deleted from the specified work-order list. The default special name "<IMAGE>" designates the currently selected work order if the cursor is in the selection; otherwise, it designates the work order in the current image.

Multiple work-order names can be specified by using wildcards, context characters, special names, set notation, or an indirect file. (For further information, see "Naming" in the Key Concepts in this book.)

`List_Name : String := "<WORK_LIST>";`

Specifies the work-order list from which the work orders are deleted. The default special name "<WORK_LIST>" specifies the default work-order list for the current session.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Set_Default

```
procedure Set_Default (To_Work_Order  : String := "<CURSOR>";
                       For_Venture    : String := "<VENTURE>";
                       For_User       : String := "<CURRENT_USER>";
                       Response       : String := "<PROFILE>");
```

Description

Sets the specified work order to be the default for a given user and session whenever the work order's parent venture is the default.

Each venture contains a list of mappings between user sessions and work orders. When a user sets a venture as the default in a given session, the work order mapped to that session in the venture automatically becomes the user's default work order. This command modifies the venture by adding or changing the mapping from session to work order in the specified venture.

Parameters

`To_Work_Order : String := "<CURSOR>";`

Specifies the new default work order for the specified venture. The default value for this parameter is the work order on which the cursor is located.

Setting the To_Work_Order parameter to either "<>" or "" causes there to be no default work order on the specified venture for the specified user and session.

`For_Venture : String := "<VENTURE>";`

Specifies the venture for which the default work order is to be set. The default special name "<VENTURE>" specifies the default venture for the current session.

`For_User : String := "<CURRENT_USER>";`

Specifies the user and session for which the default work order is to be set. This parameter can be a username (for example, Anderson) or a username and session name (for example, Anderson.S_2). If only a username is supplied, session S_1 is assumed. The default special name "<CURRENT_USER>" specifies the current session for the current user.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

Errors

An error occurs if the To_Work_Order parameter names a work order that was not created from the venture named by the For_Venture parameter.

References

function Default

procedure Set_Default_List

```
procedure Set_Default_List (To_List      : String := "<CURSOR>";
                            For_Venture  : String := "<VENTURE>";
                            For_User     : String := "<CURRENT_USER>";
                            Response     : String := "<PROFILE>");
```

Description

Sets the specified work-order list to be the default for a given user and session whenever the work-order list's parent venture is the default.

Each venture contains a list of mappings between user sessions and work-order lists. When a user sets a venture as the default in a given session, the work-order list mapped to that session in the venture automatically becomes the user's default work-order list. This command modifies the venture by adding or changing the mapping from session to work-order list in the specified venture.

Parameters

`To_List : String := "<CURSOR>";`

Specifies the new default work-order list for the specified venture. The default for this parameter is the work-order list on which the cursor is located.

Setting the To_List parameter to either "<>" or "" causes there to be no default work-order list on the specified venture for the specified user and session.

`For_Venture : String := "<VENTURE>";`

Specifies the venture for which the default work-order list is to be set. The default special name "<VENTURE>" specifies the default venture for the current session.

`For_User : String := "<CURRENT_USER>";`

Specifies the user and session for which the default work-order list is to be set. This parameter can be a username (for example, Anderson) or a username and session name (for example, Anderson.S_2). If only a username is supplied, session S_1 is assumed. The default special name "<CURRENT_USER>" specifies the current session for the current user.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

function Default_List

procedure Set_Default_Venture

```
procedure Set_Default_Venture (To_Venture : String := "<CURSOR>";
                               For_User   : String := "<CURRENT_USER>";
                               Response   : String := "<PROFILE>");
```

Description

Sets the default venture for the specified session.

Setting a venture to be the default automatically sets the default work order and the default work-order list for the current session, if such defaults have been specified for that venture.

Setting a default venture with this command automatically sets the value of the Cmvc.Default_Venture session switch to the specified venture name.

Parameters

`To_Venture : String := "<CURSOR>";`

Specifies the name of the new default venture. The default for this parameter is the venture on which the cursor is located.

Setting the To_Venture parameter to either "<>" or "" causes there to be no default venture for the specified user and session.

`For_User : String := "<CURRENT_USER>";`

Specifies the user and session for which the default venture is to be set. This parameter can be a username (for example, Anderson) or a username and session name (for example, Anderson.S_2). If only a username is supplied, session S_1 is assumed. The default special name "<CURRENT_USER>" specifies the current session for the current user.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

procedure Set_Notes

```
procedure Set_Notes (To_Value    : String := ">>New Notes<<";
                     Order_Name  : String := "<ORDER>";
                     Response    : String := "<PROFILE>");
```

Description

Modifies the notes field for the specified work order.

Any existing notes in the specified work order are replaced by the new notes. Unlike user-defined fields, the notes field can be updated multiple times.

The notes field typically is used to provide a brief description of the work order.

Parameters

`To_Value : String := ">>New Notes<<";`

Specifies the new notes. The default parameter placeholder `">>New Notes<<"` must be replaced or an error will result.

`Order_Name : String := "<ORDER>";`

Specifies the work order whose notes field is to be updated. The default special name `"<ORDER>"` specifies the default work order for the current session. The null string (`""`) is interpreted to mean `"<CURSOR>"`.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

function Notes

procedure Set_Notes_List

```
procedure Set_Notes_List (To_Value  : String := ">>New Notes<<";
                          List_Name : String := "<WORK_LIST>";
                          Response  : String := "<PROFILE>");
```

Description

Modifies the notes field for the specified work-order list.

Any existing notes in the specified work-order list are replaced by the new notes.

The notes field typically is used to provide a brief description of the work-order list.

Parameters

`To_Value : String := ">>New Notes<<";`

Specifies the new notes. The default parameter placeholder `">>New Notes<<"` must be replaced or an error will result.

`List_Name : String := "<WORK_LIST>";`

Specifies the work-order list whose notes field is to be updated. The default special name `"<WORK_LIST>"` specifies the default work-order list for the current session. The null string (`""`) is interpreted to mean `"<CURSOR>"`.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

function Notes_List

procedure Set_Notes_Venture

```
procedure Set_Notes_Venture (To_Value      : String := ">>New Notes<<";
                             Venture_Name  : String := "<VENTURE>";
                             Response      : String := "<PROFILE>");
```

Description

Modifies the notes field for the specified venture.

Any existing notes in the specified venture are replaced by the new notes.

The notes field typically is used to provide a brief description of the venture.

Parameters

`To_Value : String := ">>New Notes<<";`

Specifies the new notes. The default parameter placeholder `">>New Notes<<"` must be replaced or an error will result.

`Venture_Name : String := "<VENTURE>";`

Specifies the venture whose notes field is to be updated. The default special name `"<VENTURE>"` specifies the default venture for the current session. The null string (`""`) is interpreted to mean `"<CURSOR>"`.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

function Notes_Venture

procedure Set_Venture_Policy

```
procedure Set_Venture_Policy
                    (The_Switch    : Venture_Policy_Switch;
                     To_Value      : Boolean;
                     Venture_Name  : String                := "<VENTURE>";
                     Effort_Only   : Boolean               := False;
                     Response      : String                := "<PROFILE>");
```

Description

Sets the specified venture policy switch to the specified value.

This command also can be used to determine the value of a particular switch for a venture that currently is not displayed.

Parameters

`The_Switch : Venture_Policy_Switch;`

Specifies the venture policy switch to be modified or queried. This must be the fully qualified name of the object—for example, Work_Order.Require_Comment_Lines.

`To_Value : Boolean;`

Specifies the new value for the venture policy switch—either true or false.

`Venture_Name : String := "<VENTURE>";`

Specifies the venture whose policy switch is to be modified or queried. The default special name "<VENTURE>" specifies the default venture for the current session. The null string ("") is interpreted to mean "<CURSOR>".

`Effort_Only : Boolean := False;`

Specifies whether to simply determine the value of the specified policy switch, without actually changing that value. If true, the current value of the policy switch is displayed in the output window and the switch is not modified.

`Response : String := "<PROFILE>";`

Specifies how to respond to errors, how to generate logs, and what activities to use during the execution of this command. The default is the job response profile.

References

type Venture_Policy_Switch

---

type Venture_Policy_Switch

```
type Venture_Policy_Switch  is (Require_Current_Work_Order,
                                Require_Comments_At_Check_In,
                                Require_Comment_Lines,
                                Journal_Comment_Lines,
                                Allow_Edit_Of_Work_Orders);
```

Description

Defines the policies that can be enforced for a venture.

When a user has a default venture, the policies on that venture are followed by CMVC commands; errors result if any policies are violated. For example, if the policy Require_Current_Work_Order is enforced for the user's default venture, the user must have a default work order to execute any CMVC commands that would update a work order, such as Cmvc.Check_In and Cmvc.Check_Out.

Using package !Implementation.Work_Order_Implementation, these policies can be interrogated and enforced by other user-defined commands.

Enumerations

`Allow_Edit_Of_Work_Orders`

Defines a policy in which controlled user-defined fields can be modified interactively. User-defined fields that are not controlled can be modified, independent of this policy. Note that all user-defined fields can be modified only once.

`Journal_Comment_Lines`

Defines a policy in which comment strings provided to CMVC commands are recorded in the work-order comments field. It makes no sense to enforce Require_Comments-_At_Check_In or Require_Comment_Lines without enforcing Journal_Comment-_Lines.

`Require_Comment_Lines`

Defines a policy that requires users to provide a string comment to all CMVC commands that have a Comments parameter. The null string will not be accepted.

`Require_Comments_At_Check_In`

Defines a policy that requires users to provide a string to the Comments parameter of the Cmvc.Check_In command. The null string will not be accepted.

Require_Current_Work_Order

Defines a policy that requires users to have a default work order in order to execute any CMVC commands that have a Work_Order parameter.

References

procedure Set_Venture_Policy

RATIONAL

package Editor

The commands in package Work_Order.Editor are used for interactively editing work orders. Generally, users will not enter these commands directly but will invoke them through commands in package !Commands.Common.

The formatted display of a sample work order is shown below. Following this is a field-oriented list of applicable commands from package Common.

```
!Users.Drk.W_1.Order_4  : In_Progress;
-------------------------------------
Notes:
+ "New notes for this work order"

Parent Venture: (!Users.Drk.W_1...)
  ...A_Venture

Status: In_Progress
  Created at 87/04/13 10:40:12 by Drk.S_1

Fields:
  "A Vector String Field" 5 Strings =>
         2 => "number 2"
         3 => "number 3"
         4 => "some more values"
         5 => "another value"
+        0 => "a value which hasn't been saved yet"
    others => "uninitialized"
= "A Controlled Boolean" => False

Comments: 1 + 1
  87/04/27 11:40:03 Drk.S_1 for ">>Element Name<<" => ">>Comment<<"
+ 87/04/28 14:30:23 Drk.S_1 for ">>Element Name<<" =>
+   ">>A New and much longer Comment<<"

Users: 1
  Drk.S_1

Versions: 1 (!Users.Drk...)
  87/04/28 11:38:48 "A_Venture".1  ...W_1

Configurations: 1
  87/04/23 15:18:48 !Machine.Error_Logs
```

In the following list are brief descriptions of the operations affected by commands from package Common for each field in the foregoing work-order display. Commands not listed have no effect or produce results consistent with the descriptions in the EST book.

Field	**Command/Program Action**
Notes	*Delete* reverts to old notes, if any; *Edit/Insert* prompt in a Command window for new notes.
Fields	*Definition* creates minor window to show detailed information about all fields; *Delete* removes a newly inserted field for which no values have been saved; *Edit/Insert* prompt in a Command window for data about a new field; *Expand/Elide* show more/less of field extent; *Explain* shows maximum index, defaults, and type for each field.
Comments	*Definition* creates minor window to show all comments; *Delete* removes newly inserted comment, if it has not been saved; *Edit/Insert* prompt in a Command window for a new comment; *Expand/Elide* show more/less of comment extent; *Explain* shows the date and time comment was added.
Users	*Definition* traverses to session object in user's home directory; *Expand/Elide* show/hide list of users.
Versions	*Definition* creates minor window to show all versions; *Delete* removes newly inserted version data, if it has not been saved; *Edit/Insert* prompt in a Command window for information about a new version; *Expand/Elide* show more/less of version extent; *Explain* shows the date and time version was added.
Configurations	*Definition* creates minor window to show all configurations; *Delete* removes newly inserted configuration, if it has not been saved; *Edit/Insert* prompt in a Command window for information about a new configuration; *Expand/Elide* show more/less about a configuration; *Explain* shows the date and time configuration was added.

When a work order is edited interactively, the object is locked and the # symbol appears in the window banner. Individual changes are marked by a + symbol until they are saved using Common.Commit. Changes can be undone until they are saved.

procedure Add_Comment

```
procedure Add_Comment (The_Comment : String := ">>Comment<<";
                       The_Element : String := ">>Element Name<<";
                       The_User    : String := "<CURRENT_USER>");
```

Description

Adds a comment to those recorded in the work order.

Once a comment has been added, it cannot be removed.

Parameters

`The_Comment : String := ">>Comment<<";`

Specifies the text of the comment to be added. The default parameter placeholder ">>Comment<<" must be replaced or an error will result. This is an Ada string, which cannot span multiple lines.

`The_Element : String := ">>Element Name<<";`

Specifies the name of the object to which the comment applies. The default parameter placeholder ">>Element Name<<" must be replaced or an error will result.

`The_User : String := "<CURRENT_USER>";`

Specifies the name of a user session. If only a username is given, session S_1 is assumed.

Restrictions

This command must be executed in a Command window attached to a work order.

Example

```
Editor.Add_Comment (The_Comment => "This is a comment",
                    The_Element => "An_Element_Name",
                    The_User => "Sue");
```

procedure Add_Configuration

```
procedure Add_Configuration (The_Configuration : String :=
                                                ">>Configuration Name<<");
```

Description

Adds a configuration to those recorded in the work order.

Once a configuration has been added, it cannot be removed.

Parameters

`The_Configuration : String := ">>Configuration Name<<";`

Specifies the pathname of the configuration. The default parameter placeholder `">>Configuration Name<<"` must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a work order.

Example

```
Editor.Add_Configuration (The_Configuration =>
                                    "!Project.User_Interface.Rev1_2_1");
```

procedure Add_User

```
procedure Add_User (The_User : String := "<CURRENT_USER>");
```

Description

Adds a user session to those recorded in the work order.

Once a user has been added, it cannot be removed.

Parameters

```
The_User : String := "<CURRENT_USER>";
```

Specifies the name of a user session. If only a username is given, session S_1 is assumed.

Restrictions

This command must be executed in a Command window attached to a work order.

Example

Example command:

```
Editor.Add_User (The_User => "Bill");
```

Example user field in a work order:

```
Users: 2
  Sue.Devel
  Bill.S_1
```

procedure Add_Version

```
procedure Add_Version
                (The_Configuration : String  := ">>Configuration Name<<";
                 The_Element       : String  := ">>Element Name<<";
                 The_Generation    : Natural := 0);
```

Description

Adds a version to those recorded in the work order.

Once a version has been added, it cannot be removed.

Parameters

`The_Configuration : String := ">>Configuration Name<<";`

Specifies the name of the configuration containing the version. The default parameter placeholder `">>Configuration Name<<"` must be replaced or an error will result.

`The_Element : String := ">>Element Name<<";`

Specifies the name of the object for which a version entry is to be added. The default parameter placeholder `">>Element Name<<"` must be replaced or an error will result.

`The_Generation : Natural := 0;`

Specifies which generation of the object to add.

Restrictions

This command must be executed in a Command window attached to a work order.

Example

```
Editor.Add_Version
     (The_Configuration => "!Project.User_Interface.Rev1_2_1",
      The_Element => "An_Element_Name",
      The_Generation => 0);
```

procedure Set_Field

```
procedure Set_Field (To_Value   : Boolean := False;
                     The_Index  : Natural := Ø;
                     The_Field  : String  := ">>Field Name<<");
```

Description

Sets the Boolean value of the specified work-order field to the specified value.

Once a work-order field has been set, it cannot be modified further.

Parameters

`To_Value : Boolean := False;`
Specifies the Boolean value for the field.

`The_Index : Natural := Ø;`
Specifies which element of a vector field to modify. If the field is a scalar field, this parameter is ignored.

`The_Field : String := ">>Field Name<<";`
Specifies the name of the field to modify. The default parameter placeholder `">>Field Name<<"` must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a work order.

procedure Set_Field

```
procedure Set_Field (To_Value  : Integer := Ø;
                     The_Index : Natural := Ø;
                     The_Field : String  := ">>Field Name<<");
```

Description

Sets the integer value of the specified work-order field to the specified value.

Once a work-order field has been set, it cannot be modified further.

Parameters

`To_Value : Integer := Ø;`
Specifies the integer value for the field.

`The_Index : Natural := Ø;`
Specifies which element of a vector field to modify. If the field is a scalar field, this parameter is ignored.

`The_Field : String := ">>Field Name<<";`
Specifies the name of the field to modify. The default parameter placeholder `">>Field Name<<"` must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a work order.

procedure Set_Field

```
procedure Set_Field (To_Value  : String  := ">>Field Value<<";
                     The_Index : Natural := Ø;
                     The_Field : String  := ">>Field Name<<");
```

Description

Sets the string value of the specified work-order field to the specified value.

Once a work-order field has been set, it cannot be modified further.

Parameters

`To_Value : String := ">>Field Value<<";`

Specifies the string value for the field. The default parameter placeholder `">>Field Value<<"` must be replaced or an error will result.

`The_Index : Natural := Ø;`

Specifies which element of a vector field to modify. If the field is a scalar field, this parameter is ignored.

`The_Field : String := ">>Field Name<<";`

Specifies the name of the field to modify. The default parameter placeholder `">>Field Name<<"` must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a work order.

procedure Set_Notes

```
procedure Set_Notes (Notes : String := ">>New Notes<<");
```

Description

Sets the notes field of the work order to the specified string.

The specified text will replace the existing text.

Parameters

```
Notes :  String := ">>New Notes<<";
```

Specifies the text that will be placed in the notes field of the work order. The default parameter placeholder ">>New Notes<<" must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a work order.

end Editor;

package List_Editor

This package provides operations for adding work orders to work-order lists and setting the notes for a work-order list.

The formatted display of a sample work-order list is shown below. Following this is a field-oriented list of applicable commands from package !Commands.Common.

```
!Users.Drk.W_1.A_List
---------------------
Notes: "Outstanding work orders"

Parent Venture: (!USERS.DRK.W_1...)
  ...A_Venture

Work Orders: (!USERS.DRK.W_1...)
  ...Order_1   : In_Progress;
  ...Order_2   : Pending     ;
  ...Order_3 :               ;
```

In the following list are brief descriptions of the operations affected by commands from package Common for each field in the foregoing display. Commands not listed have no effect or produce results consistent with the descriptions in the EST book.

Field	**Command/Program Action**
Notes	*Delete* reverts to old notes, if any; *Edit/Insert* prompt in a Command window for new notes.
Orders	*Delete* removes unsaved insertions; marks an existing order to be removed from the list.

procedure Add

```
procedure Add (Work_Orders : String := ">>Work Order Names<<");
```

Description

Adds the specified work orders to the local work-order list.

Parameters

```
Work_Orders : String := ">>Work Order Names<<";
```

Specifies which work orders to add to the local work-order list. The default parameter placeholder ">>Work Order Names<<" must be replaced or an error will result.

Wildcards can be used to add multiple work orders with a single command.

Restrictions

This command must be executed in a Command window attached to a work-order list.

procedure Set_Notes

```
procedure Set_Notes (Notes : String := ">>New Notes<<");
```

Description

Sets the notes field of the work-order list to the specified string.

The specified text will replace the existing text.

Parameters

```
Notes :  String := ">>New Notes<<";
```

Specifies the text that will be placed in the notes field of the work-order list. The default parameter placeholder ">>New Notes<<" must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a work-order list.

end List_Editor;

RATIONAL

package Venture_Editor

These commands are intended for use when editing ventures. They will execute only in a Command window attached to a venture; all operations modify that venture. Many commands are bound to keys that, when pressed, prompt the user for parameter completion through a Command window.

The formatted display of a sample venture is shown below. Following this is a field-oriented list of applicable commands from package !Commands.Common.

```
!Users.Drk.W_1.A_Venture
------------------------
Notes: "Notes for this venture"

Policy_Switches:
  Require_Current_Work_Order   => False
  Require_Comment_At_Check_In  => True
  Require_Comment_Lines        => True
  Journal_Comment_Lines        => True
  Allow_Edit_Of_Work_Orders    => False

Fields:
  "A Hidden Field" Integer @ 0 => 0
= "A Controlled Hidden Field" Integers @ 0 => -1
= "A Controlled Boolean" Boolean @ 1 => False
  "A Vector String Field" Strings @ 2 => "uninitialized"

Work_Orders: (!Users.Drk.W_1...)
  ...Order_1   : In_Progress;
  ...Order_2   : Pending     ;
  ...Order_3 :               ;
  ...Order_4   : In_Progress;

Default_Work_Orders: (!Users.Drk.W_1...)
  Drk.S_1 => ...Order_2
  Drk.S_2 => ...Order_3

Work_Order_Lists: (!Users.Drk.W_1...)
  ...A_List

Default_Work_Order_Lists: (!Users.Drk.W_1...)
  Drk.S_2 => ...A_List
```

In the following list are brief descriptions of the operations affected by commands from package Common for each field in the foregoing display. Commands not listed have no effect or produce results consistent with the descriptions in the EST book.

Field	**Command/Program Action**
Notes	*Delete* reverts to old notes, if any; *Edit/Insert* prompt in a Command window for new notes.
Policy	*Delete* sets the policy to false; *Edit* toggles current policy switch; *Insert* prompts in a Command window for new policy value.
Fields	*Delete* sets display to begin with field 0; *Edit* prompts for new type and position on a field display line; *Expand/Elide* show/hide hidden fields; *Explain* shows defaults and type for each field; *Insert* prompts in a Command window for data about a new field.
Orders	*Insert* prompts in a Command window to create a new work order.
Default_Orders	*Delete* sets the default work order to nil; *Edit/Insert* prompt in a Command window for a new default work order; *Expand/Elide* show/hide list of users.
Lists	*Insert* prompts in a Command window to create a new work-order list.
Default_Lists	*Delete* sets the default work-order list to nil; *Edit/Insert* prompt in a Command window for a new default work-order list; *Expand/Elide* show/hide list of users.

procedure Set_Default_List

```
procedure Set_Default_List (New_Default : String := "<SELECTION>";
                            For_User    : String := "<CURRENT_USER>");
```

Description

Sets the default work-order list for a specific user session on the local venture.

Each user session can have a different default work-order list. Several commands reference the default work-order list of the default venture when determining which work-order list to use.

Parameters

`New_Default : String := "<SELECTION>";`

Specifies which work-order list shall be made the default for the local venture. The default special name "<SELECTION>" specifies the currently selected work-order list.

`For_User : String := "<CURRENT_USER>";`

Specifies the user session for which the default is set. If only a username is provided, session S_1 is assumed.

Restrictions

This command must be executed in a Command window attached to a venture.

Example

Assume that a venture has two work-order lists associated with it, as follows:

```
Work_Order_Lists: (!Users.Sue.Development...)
  ...Task_List
  ...New_Tasks
```

The user enters the following command, selecting the work-order list Task_List and specifying the session Sue.S_1:

```
Venture_Editor.Set_Default_List
   (New_Default => "<SELECTION>", For_User => "Sue.S_1");
```

As a result, the following entry appears in the venture's list of default work-order lists:

```
Default_Work_Order_Lists:  (!Users.Sue.Development...)
  Sue.S_1 => ...Task_List
```

procedure Set_Default_Order

```
procedure Set_Default_Order (New_Default : String := "<SELECTION>";
                             For_User    : String := "<CURRENT_USER>");
```

Description

Set the default work order for a specific user session on the local venture.

Each user session may have a different default work order. Several commands reference the default work order of the default venture when determining which work order to use.

Parameters

`New_Default : String := "<SELECTION>";`

Specifies which work order shall be made the default for the local venture. The default special name "<SELECTION>" specifies the currently selected work order.

`For_User : String := "<CURRENT_USER>";`

Specifies the user session for which the default is set. If only a username is provided, session S_1 is assumed.

Restrictions

This command must be executed in a Command window attached to a venture.

Example

Assume that a venture has two work orders associated with it, as follows:

```
Work_Orders: (!Users.Sue.Development...)
  ...Update_L1 : Pending    ;
  ...Update_R1 : Pending    ;
```

The user enters the following command, selecting the work-order name Update_L1 and specifying the session Sue.S_1:

```
Venture_Editor.Set_Default_Order
   (New_Default => "<SELECTION>", For_User => "Sue.S_1");
```

As a result, the following entry appears in the venture's list of default work orders:

```
Default_Work_Orders:  (!Users.Sue.Development...)
  Sue.S_1     => ...Update_L1
```

procedure Set_Field_Info

```
procedure Set_Field_Info (Is_Controlled    : Boolean := False;
                          Display_Position : Natural := 1;
                          The_Field        : String  := ">>Field Name<<");
```

Description

Sets the numeric tag of a user-defined field and specifies whether that field is modifiable.

Numeric tags control the relative display position of the field within the venture.

Parameters

`Is_Controlled : Boolean := False;`

Specifies whether the field should be made controlled. Controlled fields are subject to interactive modification only if the Allow_Edit_Of_Work_Orders venture policy switch is true.

`Display_Position : Natural := 1;`

Specifies the numeric tag of the field.

`The_Field : String := ">>Field Name<<";`

Specifies the name of the field whose numeric tag and controlled flag shall be modified. The default parameter placeholder `">>Field Name<<"` must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a venture.

Example

Given the following user's fields in the venture:

```
Fields:
  "Completion_Date" String @ 5 => ""
  "Problem_Description" String @ 10 => ""
  "Project" String @ 20 => ""
```

the following command will move the "Completion_Date" field between the "Problem_Description" and "Project" fields:

```
Venture_Editor.Set_Field_Info  (Is_Controlled => False,
                                Display_Position => 15,
                                The_Field => "Completion_Date");
```

procedure Set_Notes

```
procedure Set_Notes (Notes : String := ">>New Notes<<");
```

Description

Sets the notes field of the venture to the specified string.

The specified text will replace the existing text.

Parameters

```
Notes :  String := ">>New Notes<<";
```

Specifies the text that will be placed in the notes field of the venture. The default parameter placeholder `">>New Notes<<"` must be replaced or an error will result.

Restrictions

This command must be executed in a Command window attached to a venture.

procedure Set_Policy

```
procedure Set_Policy (To_Value    : Boolean                := False;
                      The_Switch  : Venture_Policy_Switch);
```

Description

Sets the value of the specified venture policy switch to the specified value.

Parameters

`To_Value : Boolean := False;`
Specifies, if true, that the policy shall be enforced by the CMVC system.

`The_Switch : Venture_Policy_Switch;`
Specifies which policy switch to modify.

Restrictions

This command must be executed in a Command window attached to a venture.

References

type Venture_Policy_Switch

procedure Spread_Fields

```
procedure Spread_Fields (Interval : Natural := 10);
```

Description

Renumbers all user-defined fields, assigning new numeric tags using the specified interval.

This command is useful for creating a place to insert a new field between two existing fields that are consecutively numbered.

Parameters

`Interval : Natural := 10;`
Specifies the interval between numeric tags for all fields.

Restrictions

This command must be executed in a Command window attached to a venture.

Example

Given the following fields:

```
Fields:
  "Problem_Description" String @ 1 => ""
  "Completion_Date" String @ 2 => ""
  "Project" String @ 3 => ""
```

the command:

```
Venture_Editor.Spread_Fields (Interval => 5);
```

will result in:

```
Fields:
  "Problem_Description" String @ 5 => ""
  "Completion_Date" String @ 10 => ""
  "Project" String @ 15 => ""
```

---

end Venture_Editor;

```
end Work_Order;
```

RATIONAL

Index

This index contains entries for each unit and its declarations as well as definitions, topical cross-references, exceptions raised, errors, enumerations, pragmas, switches, and the like. The entries for each unit are arranged alphabetically by simple name. An italic page number indicates the primary reference for an entry.

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

V

W

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____

_____

_____

_____

_____

_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____

_____

_____

_____

Indicate any additions or changes you would like to see in the index.

_____

_____

_____

_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____

Company _____

Address _____

City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
3320 Scott Boulevard
Santa Clara, CA 95054-3197

RATIONAL

READER'S COMMENTS

Note: This form is for documentation comments only. You can also submit problem reports and comments electronically by using the SIMS problem-reporting system. If you use SIMS to submit documentation comments, please indicate the manual name, book name, and page number.

Did you find this book understandable, usable, and well organized? Please comment and list any suggestions for improvement.

_____
_____
_____
_____
_____
_____

If you found errors in this book, please specify the error and the page number. If you prefer, attach a photocopy with the error marked.

_____
_____
_____
_____

Indicate any additions or changes you would like to see in the index.

_____
_____
_____
_____

How much experience have you had with the Rational Environment?

6 months or less _____ 1 year _____ 3 years or more _____

How much experience have you had with the Ada programming language?

6 months or less _____ 1 year _____ 3 years or more _____

Name (optional)_____ Date_____
Company _____
Address _____
City _____ State _____ ZIP Code _____

Please return this form to:

Publications Department
Rational
3320 Scott Boulevard
Santa Clara, CA 95054-3197