The Epsilon System

Principles of Operation

Volume IV

Configuration Management And Version Control

Revision 0.7

May 21, 1986

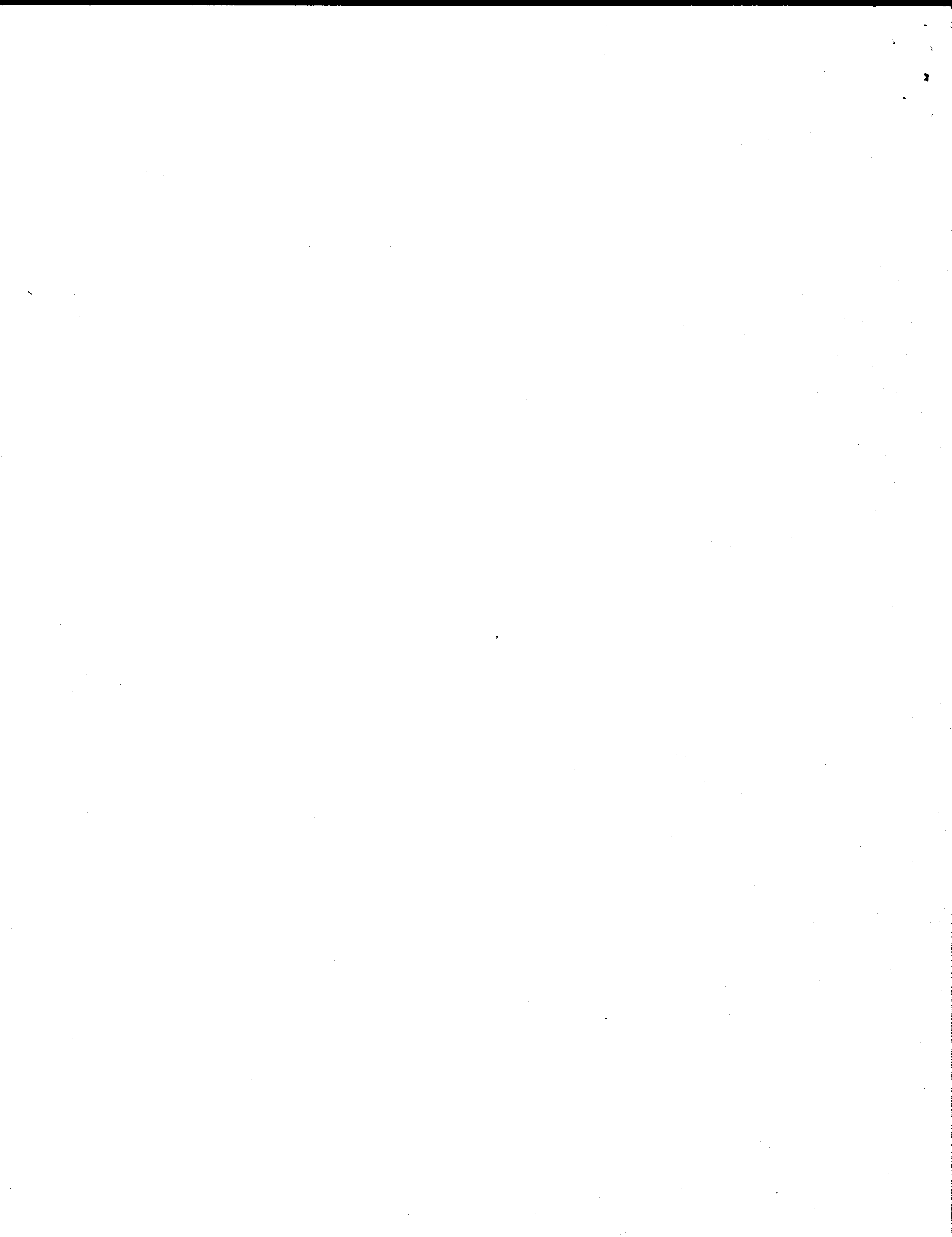


Table of Contents

1. Introduction To CMVC 3
1.1. Introduction 3
1.1.1. Project Goals 3
1.1.2. Some Basic Definitions 3
1.2. The Development Environment - Simple Objects 4
1.2.1. Worlds 4
1.2.2. Views 4
1.2.3. Views And Referencing Other Worlds 5
1.2.4. Views Are Differential 5
1.2.5. Types Of Views 5
1.2.6. The Default View 8
1.2.7. The System View 8
1.2.8. Objects And Names 8
1.2.9. Directories 9
1.2.10. Object And View Interaction 9
1.3. The Development Environment - Complex Items 9
1.3.1. Paths 9
1.3.2. Subpaths 10
1.3.3. Linked Paths 10
1.3.4. Operations On Paths And Subpaths 10
1.4. Selection Of Views 11
1.4.1. The Basic Mechanism 11
1.4.2. Views And Windows 13
1.4.3. The View Stack 13
1.4.3.1. Session, Window, And Job Stacks 13
1.4.3.2. Operations On The Stack 13
1.4.3.3. Default Contents Of The Stack 14
1.5. Naming Details 14
1.5.1. Only A Name 14
1.5.2. Name With View 14
1.5.3. Name With Path 14
1.5.4. Name With Explicit Version 15
1.6. The Import Commands In Managed Views 15
1.6.1. Link Packs, Descriptors, And Export Objects 15
1.6.2. Import 16
1.6.3. Closure 17
1.7. Import And Unmanaged Worlds 18
1.7.1. The Link Pack 18
1.7.2. Closure 18

Rational Proprietary Document

1.7.3. Visibility To Other Than Default Releases Without Importing 19
1.8. The Reservation Model 19
1.8.1. Controlled Objects 19
1.8.2. Check Out 20
1.8.2.1. Policies For Check Out 20
1.8.2.2. History Options At Check Out 21
1.8.3. Check In 21
1.8.3.1. Policies For Check In 21
1.8.3.2. History Options For Check In 22
1.8.4. Options For Annotations 22
1.8.5. Accepting Changes 23
1.8.5.1. Policies For Accept 23
1.8.6. History Options For Accept 23
1.9. Linked Paths 23
1.9.1. Functional Description 24
1.9.2. The Linkage Database 24
1.9.3. Severed Objects 25
1.9.4. Check Out 25
1.9.5. Check In 25
1.9.6. Accept 25
1.9.7. Get 26
1.10. Parallel Paths And Change Propagation 26
1.10.1. Merging 26
1.10.2. Updating The Common Ancestor 28
1.11. Conditional Compilation 29
1.11.1. Basic Features 29
1.11.2. Standards 29
1.11.3. Directives 30
1.11.4. Types 30
1.11.5. Identifiers 30
1.11.6. Switches and Initial Values 30
1.11.7. Expressions 30
1.11.8. Statements 31
1.11.8.1. Ada Utterances 32
1.11.8.2. Assignment 32
1.11.8.3. If 32
2. Implementation Details **33**
2.1. Views 33
2.1.1. What Is A UV 33
2.1.2. Types Of Universe Views 33
2.1.3. A World's Default View 33
2.1.4. The System View 33

2.1.5. A Path's Default View 34
2.1.6. Protected Views 34
2.1.7. Composite Views 34
2.1.8. Other Views 34
2.1.9. Operations On WVs and UVs 34
2.1.10. Read Only Views And Reservations 34
2.2. Paths And Subpaths 35
2.2.1. The Path Operations 35
2.2.2. The Ada Spec For Path 36
2.3. The Linkage Database 36
2.3.1. Link Pack Operations 37
2.4. The Import Operations 37
2.4.1. Checking Compatibility 37
2.4.2. The UV 38
2.4.3. The Link Pack And Change Analysis 38
2.4.4. Refreshable Imports 38
2.4.5. "Import Ada Specs" 38
2.5. The View Stacks 39
2.5.1. View_Stack Ada Specs 39
2.6. Merging 39
2.6.1. Merge_View Ada Specs 39
2.7. Conditional Compilation 39
2.7.1. Grammar 39
2.7.2. Diana 40
2.7.3. Batch Compiler 40
3. Scenarios **41**
3.1. Introduction 41
3.2. Casual Usage 41
3.2.1. Situation Summary 41
3.2.2. Goals 41
3.2.3. Scenario 41
3.2.4. Elaboration 42
3.2.4.1. Setup 42
3.2.4.2. Browsing 43
3.2.4.3. Editting and Compilation 45
3.2.4.4. Execution and Debugging 46
3.3. Simple Subsystems 47
3.3.1. Situation Summary 47
3.3.2. Goals 47
3.3.3. Scenario 48
3.3.4. Elaboration 48
3.3.4.1. Design 48

3.3.4.2. Code 48
3.3.4.3. Subsystem Test 48
3.3.4.4. Integration 49
3.3.4.5. Maintenance 49
3.4. Multiple Users per Subsystem 49
3.4.1. Situation Summary 49
3.4.2. Goals 49
3.4.3. Scenario 49
3.4.4. Elaboration 49
3.5. Multiple Subsystems per User 49
3.5.1. Situation Summary 50
3.5.2. Goals 50
3.5.3. Scenario 50
3.5.4. Elaboration 50
3.6. Multiple Machine Development 50
3.6.1. Situation Summary 50
3.6.2. Goals 51
3.6.3. Scenario 51
3.6.4. Elaboration 51
3.7. Separate Systems 51
3.7.1. Situation Summary 51
3.7.2. Goals 51
3.7.3. Scenario 52
3.7.4. Elaboration 52
3.8. Parallel Development 52
3.8.1. Situation Summary 52
3.8.2. Goals 52
3.8.3. Scenario 52
3.8.4. Elaboration 52
3.9. Simultaneous Development 52
3.9.1. Situation Summary 52
3.9.2. Goals 52
3.9.3. Scenario 53
3.9.4. Elaboration 53
3.10. Simple Subpaths 53
3.10.1. Situation Summary 53
3.10.2. Goals 53
3.10.3. Scenario 53
3.10.4. Elaboration 53
3.11. Subpath Hierarchy 53
3.11.1. Situation Summary 53
3.11.2. Goals 53

3.11.3. Scenario 54
3.11.4. Elaboration 54
3.12. Simple Linked Paths 54
3.12.1. Situation Summary 54
3.12.2. Goals 54
3.12.3. Scenario 54
3.12.4. Elaboration 54
3.13. Conditional Compilation 54
3.13.1. Situation Summary 54
3.13.2. Goals 54
3.13.3. Scenario 55
3.13.4. Elaboration 55
3.14. Target Dependent Units 55
3.14.1. Situation Summary 55
3.14.2. Goals 55
3.14.3. Scenario 55
3.14.4. Elaboration 55
3.15. Target Subsystem Compatibility 55
3.16. Composite 55
3.16.1. Situation Summary 55
3.16.2. Goals 55
3.16.3. Scenario 56
3.16.4. Elaboration 56
Index **57**

Configuration Management
And
Version Control

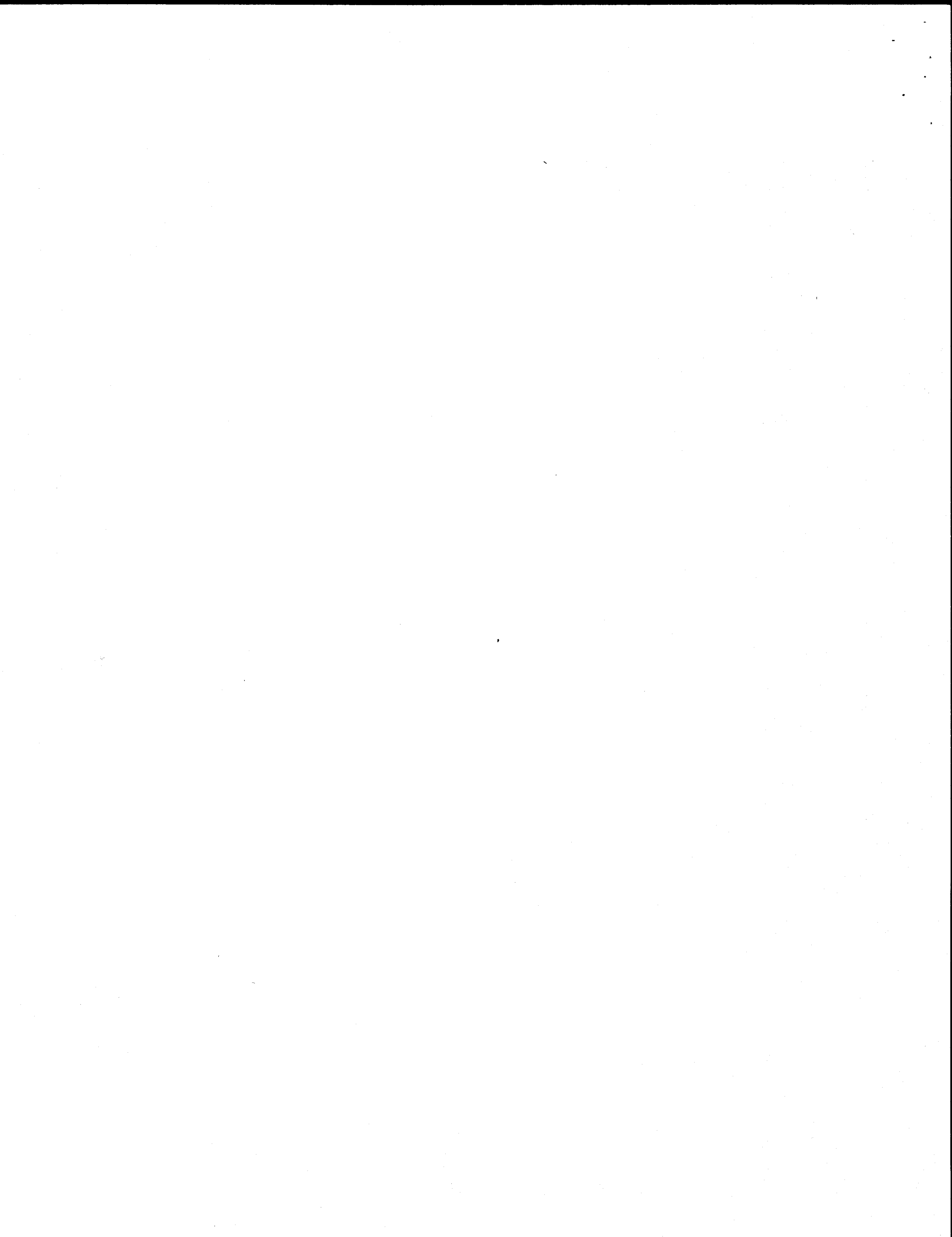
Revision 0.7
May 21, 1986

WORKING DRAFT

This document addresses the Epsilon Configuration Management and Version Control mechanisms (hereafter referred to as CMVC), and the interaction between CMVC, the compilation system, and object management.

This document discusses the mechanism used to implement Rational CMVC. It assumes the use of various facilities provided by the Rational Environment, and does not discuss the design of these facilities.

The document is in three chapters. The first is an introduction. Concepts aand functionality are dicussed from a user perspective. The second chapter is a discussion of the implementation, and requires an understanding of *Volume II, Epsilon Principles Of Operation, KKOM Specifications* and *Volume III, Epsilon Principles Of Operation, Ada Programs And Compilation*. The final chapter takes the reader through several scenarios, showing how CMVC is used.



1. Introduction To CMVC

1.1. Introduction

1.1.1. Project Goals

CMVC is designed to support large software products throughout the product lifecycle. For large software projects, configuration management and version control are tightly coupled with the overall project management process, and with the particular software development methodology employed. A goal of the Rational system is to support a variety of project management and software engineering methodologies.

One methodology that Rational feels is very important is the use of object oriented architectures and hierarchical decomposition. Special attention is given to mechanisms that supports these methods. For the remainder of this document, decomposition methods are referred to whenever the term Rational Subsystems[1] is used.

A second facility felt to be very important is support for rapid prototype and release, with minimal cost in time and space. Special attention is given to mechanisms to support this as well.

Finally, an important goal is to provide an interface to these mechanisms that is comprehensible, well integrated with the Rational Environment, and natural to use.

1.1.2. Some Basic Definitions

A *version* is an single entity. It can be a text file, an Ada program, or what have you. There is one variant extant at any one time; two variants of the same entity are different versions.

An *object* is a set of versions. As such, an object can represent the changes made to an entity over time by containing the versions that represent significant points in the change process.

Configuration Management is defined as the process used to construct, release, and maintain multiple consistent sets of versions. A mechanism is provided to control the various transformations that occur with different projects at different stages in the lifecycle.

Version Control is defined as the process of controlling and tracking changes that occur within a single object. This includes control over which versions can be changed, who can change them, and the recording of what and why the versions were changed.

---

[1]Rational Subsystems is a trademark of Rational

1.2. The Development Environment - Simple Objects

This section gives definitions of the concepts used by CMVC. Key Epsilon architectural concepts are discussed here. Many of these concepts are presented in greater detail in volumes 2 and 3 of the *Epsilon Principles Of Operation*.

1.2.1. Worlds

Software development in the Epsilon system takes place in one or more *worlds*. A world is a collection of related objects, and for Ada units can be thought of as an LRM Ada Library. Worlds have certain properties, and these are:

1. Worlds Contain Objects
   A world contains objects, which can have multiple versions. An object cannot be in more than one world.

2. The Ada Name Space Is Flat
   Within a world, the simple names of Ada compilation units must be unique.

3. There Is No Default Visibility Across Worlds
   An Ada unit can **with** any spec in its own world. However, no specs in other worlds are visible without taking specific action to make them visible.

1.2.2. Views

Since an object within a world can have multiple versions, a mechanism must be provided for selecting which version is to be visible at any one time. A *view* fills this need. For each object in a world, the view selects which version is visible (perhaps none). The view is the basic CMVC mechanism in Epsilon.

A user, working within a world, must have selected a view for that world through the mechanisms explained in section 1.4. Thus the user sees the versions of each object selected by that view. Presumably these versions form a set, related in some fashion. The installed versions of Ada units in a world, selected by a view, are *consistent* with each other, which (loosely) means they are subject to change analysis and obsolescence propagation.

A world can have many views. As such, views can be used to keep variants of a library, to keep releases over time, or both. As said before, views are the mechanism for storing and maintaining various related sets of versions in a world.

A view is an object with one version, and its contents can be retrieved regardless of what other view (if any) is in use. In other words, views do not select versions of views. Views can be *frozen*, which means the view itself cannot be modified to select a new version of an object, and no version selected by the view can be modified in any manner.

1.2.3. Views And Referencing Other Worlds

A view can select views of other worlds. When this done, the view can describe some set of versions across multiple worlds. These views of other worlds are selected via the *import* operation. This operation makes a specific view of some other world visible to the world containing the referencing view. When some other world is imported, the user must specify a view in that other world. The version subset information specified by that view is pointed at (indirectly) by the importing view.

It is important to note that views are associated with worlds. The world containing the view has special significance, as change analysis and obsolescence propagation take place within that world, and between that world and imported worlds, but not between imported worlds.[2] However, in all but one case imported worlds are *compatible*, which means the units can be executed with each other, but not necessarily compiled against each other.[3]

Import is also the command used to make Ada units in other worlds visible to the importing world. See Section 1.6 for details.

1.2.4. Views Are Differential

Views within the same world are *differential*; making a new view from some view does not cause new versions to be created for objects referred to by the original view. The versions are shared between the views. New versions are made only when the user or compiler selects for use a view sharing a version, and then tries to modify that version. When this occurs, a new version is automatically created and replaces the shared version in the selected view, and then the modification proceeds.

The effect of this is twofold. First, making a new view can be very fast, as no versions need to be copied when the view is made. Second, copies are only made when they have to be, saving space. See *Volume II, Epsilon Principles Of Operation, KKOM Specifications* for more detail.

1.2.5. Types Of Views

There are four types of views. These are:

[2] There is a special case where change analysis crosses world boundaries. See Section 1.2.5, Working Views, for details.

[3] For the exception, see Section 1.2.5, Unmanaged Views

1. A *working view* is one where the view and the selected versions in one or more worlds are not frozen, and all other imported views are. A working view is used when the worlds have active development going on within them; the view selects the versions of objects that can be changed. The Epsilon system enforces consistency between the modifiable versions and all of the imported views. The various imported views are compatible with each other.

   When more than one world is not frozen in the working view, these worlds are treated as a unit; operations on one apply to all of the others. For example, if one is released, they all are released. If one is imported into, they all are. In effect, the worlds described in this fashion act as one world. The following rules state these restrictions more formally.

   a. Given a working view in some world W1 that selects unfrozen versions in other worlds W2 through Wn, there exists one and only one working view in each of W2 through Wn that references the same unfrozen versions as the view in W1. These working views are said to be *joined*. The collection of working views joined in this fashion is called the *join set*.

   b. The effects of compilation in any one working view in the join set are immediately and automatically propagated to all other working views in the set.

   c. Any view imported into one working view in the join set is automatically imported into all of the other working views in the set. As such, no working view in a join set can import a different release of some other world than any other working view in the set.

   d. A release of any one working view in the join set causes the release of all working views in the set. If any one of the releases fails, all fail.

   Working views can be joined and split upon demand. A join operation checks that the above invariants will be maintained after the joining, and either fails or fixes the problems, depending on the options.

   Working views are always named `Current`.

2. Release View
   A *Release view* is a frozen working view. The versions selected by such a view are all frozen. It can import views of other worlds, all of which must be frozen. The world containing the view is consistent with all imported views, and the imported views are all compatible with each other.

A release can be converted to or made into one which selects code segments only. Doing so makes an execute only view.

3. Unmanaged View

   The *unmanaged view* is one where no policy or methodology is in use or enforced. Any release or frozen unmanamged view can be imported, and no checks are made. Consistency and change analysis are enforced within the world containing the view. Consistency is enforced between this view and all imported views at the time the compiler is run, but not over time. A user's home world is probably this type of view.

   The release operation can be used to make a frozen copy of an unmanaged view. This copy can be imported by other unmanaged views. It doesn't have any consistency or compatibility properties; thus it cannot be imported into working views.

4. Composite View

   A *composite view* selects zero or more working views and zero or more release views. This type of view allows construction of a view that has no consistency or compatibility properties, and is used for constructing arbitrary views for inclusion into view stacks (see section 1.4.3) and for constructing arbitrary views to be used for execution. As the execution system checks compatibility at run time and the compilation system uses working and unmanaged views, no consistency or compatibility claim need be made for composites.

The first two types of views are referred to as *managed views*. Rational Subsystems are built using managed views. For managed views the compilation system requires the compilation closure for the view be imported before a unit is compiled. The *closure* operation builds the closure and imports views not already imported. See section 1.6 for the description of the closure command. See section 2.4 for more detail on how compatibility is checked between imported views of worlds, and how the compilation closure is computed.[4]

It is important to note that when consistency is enforced, it is between the versions selected by a view and the imported views. Consistency is not enforced between two imported release views, but is enforced between joined working views. However, compatibility is enforced between imported views in working and release views.

---

[4]We need some way to enforce this. Perhaps the compiler clears its view stack and only uses the associated view if the associated view is managed.

1.2.6. The Default View

Every world has an indication of what its *default view* is. When a world is created, this indication is set to the initial unmanaged view for the world. It can be changed at any time to refer to other view; the significance of this operation is explained in the next section.

1.2.7. The System View

The *system view* is a special view, constructed by the environment. It is built by iterating over every world on the system, finding the default view for the world, and inserting the portion of the view that describes its own world into the system view. In other words, it is the union of the default views after any imported worlds have been removed. A change to a default view changes the system view immediately. This view is used in the view stack. See section 1.4.3 for more detail on the view stack. This view has a disk object associated with it, and is named `!System_View`.

The system view mechanism facilitates sharing of information between worlds, such as home worlds, without undue structure. It also facilitates controlling visibility of tools and the like; changing the system view will change visibility for a large class of users.

Changing the default view for a world has immediate effect on the system view.

1.2.8. Objects And Names

Every object in the world has a simple name. In this case, a *simple name* is a string composed from the set of characters making up a valid ada identifier. Remember that all objects exist independent of views; views select versions of objects, not the objects themselves. Thus finding an object by simple name is not dependent on the contents of any views. In other words, resolving a name to an object does not require a view to be selected.

Any object can have child objects, and the names of these child objects are also available without using a view. Children can find their parent object without using a view. Thus, the structural name tree can be built and traversed without using views.

The implementation has at least one interesting property; once an object has been created, and has a version in at least one view, that object is 'stuck' in all other views. This happens because its name is visible, even though its contents might not be. The property doesn't prevent new versions from being created in other views, but it does mean that the new versions must be the same object class. For example, if an old view has some Ada unit named `foo` in some directory `fum`, all other views are stuck with either having an Ada unit or nothing. For example, a file named `foo` cannot be created

in **fum** in some other view, as it would have a different class.[5]

1.2.9. Directories

Directories in Epsilon are objects. This type of object has only one version, which contains no useful information. Objects in a directory are children of the directory object, and naming uses this information. The version is used solely as an indicator, telling KKOM when the directory can be expunged. The directory's version must be selected by at least one view to protect the directory from being expunged from the disk. If a directory's version is not selected by any view, and the directory object has no children, then the directory is expunged.

1.2.10. Object And View Interaction

A name is used to find an object, and the result of this process is an *object handle*. An object handle is a small, convenient, alternate form of the name of the object. This handle is combined with a view to select a specific version of the object.

In addition, a specific version can be named, resulting in an object handle already selecting a version. This form does not need a view. In most cases this form cannot be used to open a version for update.[6]

See Section 1.5 for more details on naming, including syntax.

1.3. The Development Environment - Complex Items

This section describes CMVC items that are composed of the basic items described in section 1.2.

1.3.1. Paths

A *path* is a collection of views. This collection has the following properties:

1. There is at most one, unfrozen, working view.

2. There are potentially many, frozen, release views.

3. The releases are all created from the working view.

---

[5]This is undesirable. It is my understanding an effort will be made to correct this problem as early as possible

[6]It is anticipated that the KKOM policy that allows open for update by version qualification will be set only for objects constrained to have exactly one version.

4. The path is not contained in any other path.

5. Target information is associated with the path.

6. There is a *default release* associated with the path. This view is used if the path name is used in an import operation instead of a view name. It is also used by naming if the path is named as a qualifier and no working view exists in the path.

Thus the path implements a time ordered sequence of releases, ending in the working view. The path itself is a directory, subclass 'path'. Each release in the path represents some point in the development process, and is available to be imported into some other world. The default release allows the administrator to select a release that other worlds are to get when they import a path but do not specify a view.

1.3.2. Subpaths

A *subpath* has the same properties as a path, with two exceptions. It is only found inside a path or subpath, and cannot contain target information. Subpaths are used when a reservation model is desired. Such a model allows the same object to exist in many subpaths, but to be changeable in only one. In addition, the model supports collection of history information, and supports orderly change propagation.

The reservation model is discussed fully in section 1.8.

1.3.3. Linked Paths

These are an extension of the reservation model. This mechanism allows the reservation of an object across multiple targets. See sections 1.8 and 1.9 for a complete discussion of the reservation mechanism as applied to linked paths.

1.3.4. Operations On Paths And Subpaths

The following operations exist for paths and subpaths:

1. Create
   Build a new, initially empty, path. It has an empty working view and no default. By default, the path will be created in the same world as the object being displayed in the window.

2. Fork
   This operation creates a new path as a differential of some release view in the original path. The only relationship between the two paths is that they now have a common starting point for future divergence. This is significant to

the merge operation discussed in section 1.10.1. Only the working view is created from the specified release; no other releases are copied. Furthermore, no deleted versions of objects are referenced by the new working view.

3. Create Subpath

   Create a subpath or a linked path as a differential of the parent path or subpath. The reservation model must be in use to use this operation. This operation copies the working view, not a release. The releases and deleted versions in the working view are not copied.

4. Revert

   Cause the contents of a release view to be copied into the working view. This command is used to recover from a previously known good state, which might be necessary after some catastrophe. This operation takes place between some release and the working view of the path containing the release. This command can also revert individual objects.

5. Destroy

   Destroys a path or subpath. Destroy could cause obsolescence in a lot of other worlds if the path contains releases that have been imported by some other world. There is notification given if a contained release has clients, and the operation fails unless the user explicitly requires the destroy happen anyway.

6. Release

   This operation creates a release view within the path or subpath from the working view. Code only releases can be made.

1.4. Selection Of Views

This section discusses the various mechanisms provided for selecting versions of objects using views, or selecting a view to be used as a default.

1.4.1. The Basic Mechanism

Recall that the name of an object is resolved to an object handle. This handle contains everything required to find an object, in a fast, compact, form. The handle is combined with a view to actually locate a specific version of the object.

The object handle contains, among other things, the following items.

1. The *world number* allows the system to find the world that contains the object. World numbers are unique within one system.

2. An Object Number
   The *object number* is an identifier for the object in the world. No two objects in any one world have the same object number.

3. A Version Number
   The *version number* uniquely selects one version within a world. If a version has already been selected, this field is filled in with the number of the version, otherwise it is nil.

4. A View
   The object handle contains enough information to identify the view to be used when a version is required for this object. This view can be explicit, or can specify that the view stack be used. See section 1.4.3 for details on the second option. If this field identifies a view, the version number field must be nil, and vice versa.

When opening a version using a view (remember, the user can explicitly name a version) the following actions take place:

1. Verify The View References The Object's World
   Check that the view references the world containing the object. The check is made by checking that the object's world is the view's world, or that the object's world has been imported into the view.

2. Check The View For A Version
   The information within a view describing each world is organized as a vector, indexed by object numbers. Thus the system takes the object number from the handle and probes the view vector for the right world. This *slot* contains the list of versions, identified by version number, visible using the view. One of these versions can be (and usually is) marked as the *current version*, and this version is usually the one used. See *Volume II, Epsilon Principles Of Operation, KKOM Specifications* for a discussion of views, slots, retention lists, and open options.

3. Open The Version
   Using the version found above, open that version. If the open is for update and the version is in more than one view, make a copy of the version before it is opened. Fill in the slot in the view with the new version before returning, so the view will refer to the new version.

1.4.2. Views And Windows

If a view is used to select versions of an object, and that object is displayed in a window, then that view becomes associated with that window. This is referred to as the *associated view*. This view is displayed to the user for informational purposes only.

1.4.3. The View Stack

In the case where the object handle selects neither a view nor a version, there must be a mechanism for supplying one on demand. The ***view stack*** serves this purpose. The view stack is a stack of references to views, searched from the top down. For example, if a handle is being used that references object **foo** in world **fum** with no selected view, the view stack is searched from the top down until a view is found that references world **fum**. This view is used to find a version of **foo**.

Note that the search stops at the first view that references the object's world. If this view doesn't select a version, an error is returned. The stack is not searched further for another candidate view.

1.4.3.1. Session, Window, And Job Stacks

The view stack is really three stacks, a session stack, a window stack, and a job stack. The session stack lives from job to job, window to window. The window stack is associated with a window, and lives as long as that window does. The job stack is used by the job as temporary storage. A job can modify any of the stacks.

The window stack initialized with the contents of its parent window, except under some conditions discussed in section 1.5.

The session and window stacks are copied into job state when a job is initiated, and are not changed if other jobs modify these stacks while the job is running. However, modifications to the session and window stacks by the running job are reflected in the those stack immediately, and jobs started after the change see the change, even if the original job is still running. Changes to the window stack are legal only if the job is attached to a window. It should be noted that jobs run with a copy of the stacks. If a job changes the session or window stack, it doesn't see the change unless it changes its own copies as well.

The search order is job stack, window stack, session stack.

1.4.3.2. Operations On The Stack

There are operations to push, pop, and interrogate views on any stack. There is also an operation that, given an object handle, returns the view that would be used to find a version. This operation is used to find and set the associated view.

1.4.3.5. Default Contents Of The Stack

The system pushes the initial system view onto the session stack when the user logs in, and this view cannot be popped off.

1.5. Naming Details

As alluded to in previous sections, there are many ways of naming objects and versions, with different resulting object handles. They all involve a name, with optional qualification. In the examples below, the assumption is made that the object exists and the name is valid.

1.5.1. Only A Name

This method builds an object handle identifying only the object. Specifically, no version is selected; if one is ever needed, the view stack is to be used. The name

```
"!world.directory.object"
```

is an example of this form. This method is expected to be the most commonly used.

1.5.2. Name With View

This method builds an object handle identifying the object and the view to use if versions are needed. No version is selected at this point, and the view stack is not used. The name

```
"!world.directory.object'v(!world.view)"
```

is an example.

If this form is used to create a new window, the window stack is cleared, then the named view is pushed onto the stack.

1.5.3. Name With Path

The object handle built is identical to the 'name with view' form. The difference is the selecting of the view. In this case, the working view of the path is selected. If there is no working view, the default release is selected. The name

```
"!world.directory.object'v(!world.path)"
```

is an example of this form.

1.5.4. Name With Explicit Version

This form doesn't require a view at all. The version to be selected is explicitly named. The syntax is

```
"!world.directory.object'v(34)"
```

where 34 is the version number of the desired version.

1.6. The Import Commands In Managed Views

As stated earlier, the import and closure commands manipulate what other worlds are visible to managed views. The import command also indirectly controls visibility to Ada specs in other worlds. There are five objects of interest to the import command. One is the receiving working view, which controls visibility to views in other worlds. The second is the view in the other world. The third is the *link pack* in the receiving world, which controls visibility to Ada units theoretically made visible by the view in the other world. The fourth is the *import descriptor* object, which contains the information required to build the link pack. The fifth is the *export object* in the other world, which lists which Ada specs can be seen outside of its own world. Views have already been discussed; the two other objects are discussed here.

Note that only release views can be imported by working views.

1.6.1. Link Packs, Descriptors, And Export Objects

In order to be able to `with` an Ada unit, two actions must take place. The exporting world must make the spec visible, and the importing world must ask for it. These actions are accomplished using export objects, link packs and import descriptor objects.

An export object is a list of Ada specs in the same world as the export object that can be imported by other worlds. There can be many export objects, supplying various subsets of specs. All worlds contain a pointer to their default export object. When a world is created, this object is set to export all Ada specs. It can be changed to refer to any export object.

The link pack is the mechanism used to make an Ada name of Ada specs visible. There is one link pack per world, with multiple versions selected by views. The link pack maps simple Ada names to object IDs in the link pack's world and other worlds. All Ada specs in the link pack's world are automatically entered into the link pack. Specs in other worlds are entered using the import operations.

The import descriptor object contains an entry for every imported world. There are multiple versions of this descriptor object as well. Each entry in the descriptor contains:

1. The import descriptor contains the name (in some form) of the export object to use in the imported world. This name is interpreted in the context of the view actually imported, yielding a list of Ada units. If the name of the world is specified instead of an export object, the default export object for the world is used.

2. A List Of Units To Include
   The import descriptor can optionally specify a list of units to import. The user supplies this list. All units specified in this list must also be specified in the export object, but this list can omit unneeded specs supplied by the export object. Doing so allows the user to not import specs that are not needed. If the list is empty, all specs are imported.

3. The Rename List
   The import descriptor can optionally specify Ada names to be used within this world for Ada units from the imported world, which allows the user to remove name conflicts, if needed. If omitted, no renaming is done.

The include list and rename list are actually one list of pairs; an object to import and what to name it.

See *Volume III, Epsilon Principles Of Operation, Ada Programs And Compilation* for more information on links packs, import descriptor objects, and export objects. There will be both an object editor and a programmatic interface available to manipulate the import descriptor object.[7] There is no way to directly manipulate the link pack.

1.6.2. Import

The import command inserts a release view of some other world into a view. In addition, if the import descriptor contains import information for the imported world, the link pack is brought up to date by opening the descriptor object and reading out the entry for the world. The command continues by opening the export object, extracting the unit names, applying the rename and exclusion lists against it, and writing the results back into the link pack. Change analysis is then run to check for consistency in light of the new or changed imports.

Import will accept naming a view using something of the form *import the same view of world X as imported into view Y.* This allows the user to import the same views as

[7] There are differences between what this section says and what the Ada document says. These need to be resolved.

someone else. Import will also accept ***import all the same views of worlds imported into view*** *Y*. Off course, neither of these two operations will replace the view's own world even if view Y imports some other view.

A user can import the default release of a path, or a specific view of a path. If the former is done, the user can later ***refresh*** the import, which causes a re-import of the default if it has changed. The refresh operation can be applied to an entire view. A centralized release methodology can be built using the refreshable form combined with the ***import using other view*** scheme described above.

Import can also specify that the imported view be *protected*, which causes the imported view to be copied, making it more difficult for it to be deleted. It is anticipated that this mechanism will only be used for major releases.

Import checks that any world to be imported is compatible with the worlds already imported. It automatically imports the world into all members of the join set.

1.6.3. Closure

The closure command computes the compilation closure of the imported views, and puts this information into the view. In other words, it automatically imports views of worlds required by the compilation system. The user has some flexibility over which views are imported.

Closure first determines what worlds must be imported. Then for each world in the closure that hasn't already been imported, one of the following actions is taken.

1. If the user supplied a view (as a parameter) to get the required views from, the supplied view is checked to see if it references a compatible view of the desired world. If it does, that view is imported.

2. The view stack is then checked and, if a view of the desired world is in the stack and that view is compatible, that view is imported. The system view is ignored in this check.

3. If a desired world is not on the view stack, closure picks the default release for the path in the desired world that appears most often in the closure computation, if it is compatible.

4. If no path appears more often, closure picks the newest default release that is compatible.

5. If closure cannot find a default release to use, closure picks the compatible view for the world that appears most often in the closure computation.

6. If no compatible view appears in the closure more often than another, closure picks the newest.

7. It is an error if no compatible view can be found.

1.7. Import And Unmanaged Worlds

Unmanaged worlds can and must import Ada specs from other worlds, but need not import specific views of those other worlds. If a view is not imported, the view to use is taken from the view stack. However, taking a view from the view stack has several implications.

1. The view stack can change, causing the view to use to change from compile to compile. If this happens, operations on installed units can fail with potentially obscure messages.

2. Since the view stack can refer to working and and unmanaged views, the unit in the other world can change without obsoleting anything. Again, the result can be obscure error messages.

3. The user doesn't have to be aware of working views, releases, and the like. Changes in unmanaged views can be seen immediately, without any release process. This facilitates sharing between users.

The user can import either release views of frozen unmanaged views into an unmanaged world.

1.7.1. The Link Pack

In order to get visibility to Ada units in other worlds, the user must still use the link pack and the import descriptor. Typically, the export object selected will be the default, but the user can name one. The include list and rename list can be used as well. The major difference between this case and a managed world is that the information is interpreted immediately after the descriptor object is committed, using the view stack or imported views, to fill in the link pack for worlds not explicitly imported. The imported units list is not refreshed until the user asks for it by using the refresh command. The refresh command will also recheck the link pack and propagate obsolescence as needed.

1.7.2. Closure

The closure command does not operate on unmanaged views.

1.7.3. Visibility To Other Than Default Releases Without Importing

The user can push onto the view stack composite views that reference any view desired. The link pack would then be refreshed, which will cause the imported units list to be reinterpreted. The compiler will also see this view when the unit is compiled.

1.8. The Reservation Model

One common style of software project management requires a stylized method for controlling and serializing changes to anything of interest to the project. These include text files, programs, switches, the import descriptor object, and indeed any other kind of object that can be put into a world. The most common implementation for this policy involves reservations; the item is reserved, changes are made, and the item is released. Such a policy is referred to as the *reservation model*. Usually history, user supplied annotations, and 'what changed' information is gathered at the reservation and release points.

In Epsilon, objects are reserved on a path basis. Thus any reservation affects a path and its subpaths. In fact, when the reservation method is in use, subpaths are required. Remember that subpaths are children of paths, and subpaths can have have subpaths as children. Thus the path has an n'ary tree of subpaths below it.

The Rational reservation scheme follows these conventions. Reserving is referred to as *check out*, releasing the reservation is referred to as *check in*.

Various options exist for gathering history, which are discussed below. These options are set on a path basis.

1.8.1. Controlled Objects

CMVC only cares about objects that have been declared *controlled*; these are the objects that reservations are to be applied to. For example, Ada units are normally controlled, output from Text_Io is normally not. The administrator (or user) decalred to CMVC that a particular object is to be controlled, and from that time forward the object is managed.[8]

If there are different versions of an object in more than one subpath within a path, the declaration to make it controlled fails. The other versions must be either deleted or renamed. Enforcing this restriction ensures the basic axiom of CMVC; changes to a controlled object are done serially.

---

[8]It has been suggested that newly created Ada objects default to controlled, while all others default to not controlled. How is this implemented? How are new objects created?

The import descriptor object is a special case. Certain policies (described below) imply an automatic accept of import changes. If these policies are enabled, the system only forces acceptance of the minimum subset of the imports possible. Thus partial changes to the descriptor object can be accepted. However, checking out the descriptor object forces acceptance of all of the imports.

1.8.2. Check Out

Every (controlled object)/path pair has a *token* associated with it. The token forms the basis of the reservation method; possession of the token by a subpath means the object has been checked out to that subpath, and can be modified there. The token can be obtained (the object can be checked out) if some parent subpath or the path has the token. When the object is checked out, the token is moved to the requesting subpath, making it unavailable to any other subpath except children of the requester. The token is implemented using the linkage database. See section 1.9.2 for more detail on this database.

When the reservation model is in use, the path's working view is read only; it cannot be used for making changes. It is used only as a repository for objects that haven't been checked out. Subpaths must be used for actually making changes. However, the path's working view can be compiled.

This scheme does not enforce checking out to a person, project, or purpose. The site can implement whatever scheme it wants by building an appropriate subpath tree.

When the reservation model is being used, the system must be able to determine which path and subpath the user is working with. Doing so is done by looking at the associated view and finding the path or subpath that contains that view. A path is easily found from a subpath. Of course it is possible to directly specify a path or subpath to be used, without relying on a window.

1.8.2.1. Policies For Check Out

Several policies are available, and can be set on a path or subpath basis. These are:

1. The search for the token can be stopped at any parent subpath, which allows the site to implement a policy that the object must be checked out to the parent before a child can check it out.

2. The intermediate (non-leaf) subpaths can be marked as read only, which means the object must be checked out to a leaf to be changed. This policy allows further organizational control, and forces gathering of history if desired. These subpaths can be compiled, however. The path is always marked in this manner.

3. Various history gathering options can be set. See section 1.8.2.2 for details.

4. A path or subpath can be marked to require automatic import of the worlds the checked out unit references. Using this policy requires the unit be in the installed state or better, and may cause obsolescence in the recipient subpath. This policy is set in the parent path/subpath.

These policies are set via switches in the path and subpaths.

1.8.2.2. History Options At Check Out

Some options are available for gathering history information at check out time. These are:

1. Who is checking the object out.

2. The date it was checked out.

3. Any annotations. See section 1.8.4 for details on annotation options.

4. Whether any automatic imports were done.

1.8.3. Check In

The check in operation moves the token to the subpath's parent, and moves the changed object there as well. If the object is an Ada unit, this can cause other units in the parent to be obsolesced.

Note that checking the unit into the parent does not obsolesce any units in any other subpath. Instead, it causes differentiation of the unit in the parent.

One operation can check in all objects checked out to a subpath.

1.8.3.1. Policies For Check In

Again, various policies are available, and can be set on a path or subpath basis. These are:

1. Check in to a parent can cause the unit to be automatically checked into the parent's parent, and so on up to the path.

2. A parent can be marked as requiring **make consistent** be run after the check in, but with no requirement the make succeed.

3. A parent can be marked as requiring **make consistent** run and succeed for the check in to be successful.

4. A parent can require that the worlds in the source subpath's compilation closure be automatically imported in the parent.[9]

5. A parent can require only newly imported worlds be automatically imported.

6. A parent can state that no worlds be automatically imported, which may cause `make consistent` to fail.

1.8.3.2. History Options For Check In

The following history options, which are set on a path basis, are available at check in time:

1. The date of the check in.

2. A log of changes to the source. This log is in the form of a set of differences between the source at check out time *vs.* check in time. Keeping the log allows reconstruction of the source for previous versions.

3. User annotations, describing the changes.

4. What automatic imports were done.

1.8.4. Options For Annotations

User annotations can be structured if desired, which allows keeping of information relevant to the project in a form useful for report generation. The structure is defined by the individual that sets policy for the path. The user builds this structure by declaring *fields*. These fields are named by the user, and can be fixed or variable length. There is no provision for doing validation of data.

If history gathering is on, the system presents the user with a menu of fields and asks that they be filled in. This information is compressed and stored as records. Programmatic interfaces are available for iterating as follows:

1. Over each record for an object.

2. Over each record for every object.

3. Over each record for every object that has a default version in the path.

4. Over the last record of every object.

---

[9]Note that if the import descriptor object is controlled, this and the following policies might not be important.

5. Over the last record for every object that has a default version in the path. This iterator takes into account linked paths. See section 1.9.

6. Over every field of a record.

The default structure is a record with a variable length field named **annotations** and several fixed fields to gather the information, such as dates, described above. These predefined fields have fixed names and formats, to be determined later.

1.8.5. Accepting Changes

After an object is changed and checked back in, the changes can be accepted into other subpaths, which is done using the *accept command*. This command copies the changes into the subpath, then any obsolescence and change analysis is performed. Accepting changes does not give the accepting subpath the right to make further changes.

Changes can only be accepted from a subpath's parent, grandparent, etc.

1.8.5.1. Policies For Accept

Several policies, similar to check in, are available on a path and subpath basis. These are:

1. The worlds in the compilation closure of the parent must be imported automatically into the subpath.

2. Newly imported worlds must be imported.

3. **Make consistent** must succeed.

1.8.6. History Options For Accept

There are only two options available. These are to log that an accept was done, and to note any automatic imports that were done.

1.9. Linked Paths

Linked paths are an extension of the subpath mechanism, allowing changes in target information along with the reservation model. *Target* information is defined as all information required to configure a view for a particular processor, hardware configuration, etc. Conditional compilation switches are not considered target information.

In situations where multiple targets are in use, it is often the case that much of the information in a world is common to the two (or more) paths. However, often some

object is totally different, or at least different enough to make common source inconvenient. Making this object uncontrolled isn't desired, as changes wouldn't be serialized. What is really wanted is to have two or more reservations tokens. Linked paths address these issues.

1.9.1. Functional Description

Linked paths allow the following kinds of sharing between paths:

1. The Object Is Common To All Paths

   In this case, the same source is expected to be used on multiple targets. Conditional compilation might be used. There are no parallel changes, the object is checked out and checked in. There is only one reservation token per object across all of the linked paths.

2. The Object Is Different In Every Path

   The object is entirely different on each path, and can be changed in parallel. Each path has its own reservation token. These are called *severed objects*, as the relationship between paths is broken. The objects are related in name only.

3. Combinations

   Some objects are shared across some paths, but not all paths.

Declarative information is used to keep track of the relationships between paths and objects. Relationships between paths are transitive. The declarative information is kept in the linkage data base.

The subpath structure must be identical in any two paths linked together for certain commmands to work. A policy switch is available that wil cause the creation of a new subpath in one path to cause the creation of the subpath in any linked paths. If any of the creations fail, they will all fail.

1.9.2. The Linkage Database

A *linkage database* describing exactly how objects are shared between paths is built using information provided by the users. This data base contains relations describing which paths are linked, which objects are common between which paths, and which path or subpath currently owns the reservation token for a particular object.

Commands exist to make all objects common between two paths, sever two paths completely, and join and/or sever an object from a set of paths. The join operation can cause changes to be lost if changes were made during the time the paths were

independent. A warning is given if the versions in the paths being joined are different.[10] The sever operation does not automatically accept the latest changes; however, a warning is given if the object is not current.

1.9.3. Severed Objects

If the relationship between paths is severed for an object, each path has a token. Changes can be made independently within each path, with no cross path control. If changes made in one must also be made in the other, the parallel path merge facilities must be used. See section 1.10 for more details on this option.

1.9.4. Check Out

Check out in a linked environment is similar to the non-linked. The only change is the parents in every path defined by the data base to be equivalent for the desired object are searched. Thus the token can move from path to path, but still only parent to child.

If the subpath structure is not identical across the various paths, it is possible that the subpath being checked into doesn't exist on other paths. To account for this possibility, the following search rules are used:

1. Check the subpath's immediate parent. If the token is there, stop.

2. For each linked path, check if an equivalent parent exists by searching down from the path. If there is such an equivalent parent, check it for the token.

3. If the policy on the immediate path allows searching further parents, move up one level in the tree and start again.

1.9.5. Check In

Check in is identical to the non-linked environment. The object is copied to the immediate parent, with all of the same processing. It is specifically not copied to the other paths.

1.9.6. Accept

Accept is also identical to the non-linked environment. The changes can only be accepted from a direct parent. Changes in linked parents cannot be accepted without checking them out, or using the *get* operation described below.

---

[10] This could be the most common case. Is there some way to prevent the data loss?

1.9.7. Get

Get is a new operation that can only be used in linked paths. It causes the token and changes to be moved from the same point on the subpath tree in some other path to this path. The semantics are otherwise identical to check out. This operation is used to avoid changing a parent via check in till the object has been tested in several targets.

Get requires that the subpath structure be identical is as far as the two subpaths are concerned. In other words, the sequence of simple names between the source subpath and its path must be identical to the sequence of simple names between the destination subpath and its path.

1.10. Parallel Paths And Change Propagation

There are cases where multiple paths exist that contain related but not linked objects. These might be differing major releases, totally severed paths because of massive target differences, objects severed on linked paths, or simply test bed views. These are called ***parallel paths***, as there is no synchronization between the paths for at least some of the objects.

One recurring problem occurring when parallel paths are used is change propagation between the paths. For example, assume the existence of a major release of a system that has be shipped, called Gamma, and a development release of the same system, called Delta. All new development is going on in Delta, but occasionally bugs need to be fixed in Gamma. Some method is needed to check if the same bug exists in Delta, and propagate the fix to Delta if it does. Since Delta has had active development going on, the units in Delta could very easily be different than what was left in Gamma. In other words, the two paths have been changed in parallel. What is needed is a method of comparing the units in the two paths, isolating the individual changes, and propagating only the 'right' changes.

1.10.1. Merging

Merging is the process used for comparing a version of an object in one path with a some other version in another path, figuring out what has changed, and making the required changes in a version in a working view. This process works by computing the changes made to the source, changes made to the destination, and checking to make sure the changes were orthogonal. If they are, merging can take place with a reasonable probability of success. A more complete description of the process is:

1. Find The Objects
   Since merging works on versions of an object, only the object or objects must be named. If the reservation model is in force, all objects being looked at must be checked out to the recipient subpath. In this case, the recipient cannot be a path, as it is read only.

2. One of these must be a release view, the other must be a working view. They must be in different paths or subpaths of different paths. A release is made of the working view, and this becomes a source. The working view is the destination.

3. Repeat For Each Object
   The following steps are done for each object. The result is a report and a database of changes.

   a. Find A Common Ancestor
      In order to find the changes between the source and the target, we need to have a common starting point from which the versions diverged. This starting point is called the *common ancestor*, and is usually the point where the paths were forked or the point where the last merge took place.

   b. Compare The Ancestor To The Two Sources
      This compare is done on the text of the versions. Two lists of changes are built, one for each source.

   c. Look For Conflicts
      Conflicts are defined as changes to some text line in the ancestor in both the sources. Additions at the same spot in both sources are deemed conflicts. This information must be reported in the report.

   d. Save The Information
      Save the above information in a form convenient for later perusal and processing.

At the end of the above process, the user has a database of changes that, if done, will merge the specified objects in the two paths. Each slot in the target working view that corresponds to an object with change information in the database is made read only to prevent changes from invalidating the data base.

The user has several options at this point. These steps can be done several times, in any order. They are:

1. Do The Work
   Apply the lists of changes, changing the working view. Produce a report stating exactly what was done. Take the same history information as if this were a check in operation. In addition, if the object is an Ada unit and the

user asks for it, insert information into the unit indicating what was done. See section 2.6 for more detail on what is inserted.

The changes are made into the working view. The user should look at the changed objects and verify the results. If needed, they should be fixed. In addition, since a view was left behind with the old contents, recovery is possible if the merge produces something horrible.

Changes that were successfully done are removed from the database. If all of the changes for an object are removed, the slot is made read/write again. Changes that are in conflict are not attempted, and are left in the database. If there is a problem making a change, these are left in the database as well.

2. Peruse The Change List
   The user can look at the list of changes and approve or reject them. Rejecting a change removes it from the database. The user resolves conficts in this manner, and also can prune the change list to only reflect what operations are actually desired. If the user removes all of the changes for an object, the slot is made read/write again.

   This perusal is done with an object editor that tries to show the user what will happen if a change is done.

3. Abandon The Changes
   This option throws away the database, and can be done on a per object basis. Again, the slot is set read/write if appropriate.

1.10.2. Updating The Common Ancestor

The user must indicate when the common ancestor for some object and path pair must be brought up to date. The user first specifies the path pair, then takes one of two choices:

1. The user indicates that for an object, the release view that was last used to merge the object is now the ancestor. The user normaly uses this method to change the ancestor. If the release view has been deleted, an error message is produced and the ancestor isn't changed. Many objects can be specified. If this is done, each is checked individually, as the ancestor view can be different for each object.

2. The user specifies the version that is to be used as the common ancestor. This operation allows the user to hide changes from the merge process, and should be used with caution. The user can set the ancestor to NIL, which means that no further merges can take place between the two paths on that object.

1.11. Conditional Compilation

Conditional compilation facilities are built into the Epsilon source handling packages. The text editor, CMVC, and Diana all know about these facilities. Conditional compilation is achieved by adding compilation directives to source managed by CMVC. Since the editor is an integral part of CMVC, it knows about these directives. Making Diana know about them is both a standards issue and an efficiency issue. More about this below.

1.11.1. Basic Features

CMVC conditional compilation is built around the syntax of Ada. The system allows the conditional inclusion or exclusion of Ada statements and declarations, according to the value of the conditional compilation switches. Thus, conditional compilation allows the tailoring of source for various targets or configurations.

CMVC conditional compilation cannot be used to change parameters, expressions, and the like. There is also no conditional substitution or other macro processing. There are two reasons for these restrictions. The first is a user interface issue; we want to be able to show easily the various combinations of included source and excluded source with the text editor. Wanting this property argues for a well defined granularity of change so elision and highlight can be used effectively. Having a syntax directed editor rules out macro processing, as the true source wouldn't be known until the macros are substituted.

The second reason is efficiency. The Diana implementation will be much easier and faster if we limit where conditional nodes can land. The changes to semantics and other Diana producer/consumers will also be easier and more robust if we limit the variables.

1.11.2. Standards

One area of great concern is the issue of Ada language extension. If conditional compilation cannot be implemented without extending the language, it will not be implemented. Currently, 'extending the language' is thought to mean changing the compiler in such a way as to allow a non-Ada program (*i.e.*, one with directives in it) to be successfully compiled using the path

```
Text File -> Diana -> Code.
```

We will ensure that this path cannot be taken.

We want to store the conditional compilation directives in Diana as well as in managed source. This desire means we must extend Diana to allow Diana consumers to see either unadulterated Diana or Diana with the directives and skipped source in it. Unadulterated Diana is the default.

If a managed Ada program is opened in such a way as to require its text, such as by

text_io, the correct text of the program must be supplied by default. The correct text is the text with the directives interpreted and removed, with skipped Ada removed. There also must be a way to get at the source with the directives and ignored Ada in it.

1.11.3. Directives

The directives form a small language, whose complete grammar is given in section 2.7.1. The language consists of statements and expressions. Expressions consist of identifiers and operators. All directive have the '@' character in column 1 to distinguish them from Ada.

It is intended that the directive language be a subset of Ada. Some deviations might be required in order to make the parser reasonable.

1.11.4. Types

Quantities represented by identifiers or literals are either integer or string. Strings have some maximum fixed length (how about 100 characters), but the current length is variable. Integers are signed. Literals are written using normal Ada syntax; strings are delimited by '"'s, integers are not delimited. Based integers are accepted. There is no conversion between string and integer. Boolean values **True** and **False** are represented by /= 0 and = 0, respectively.

1.11.5. Identifiers

Identifiers follow Ada syntax. They are not declared explicitly, but are declared implicitly by storing into them. They can represent strings or integers. They must be defined (stored into) before they are used.

1.11.6. Switches and Initial Values

Initial conditional compilation parameters are supplied from a switch file in the path. This file contains (id_name, value) pairs, and is interpreted as a series of assignment statements whenever the source for a unit must be computed. The path and its child subpaths all share the same switch file.

1.11.7. Expressions

Expressions are composed of identifiers separated by operators. The operators allowed only on integers are

+, -, *, /, and, or, not

The operator allowed only on strings is

The operators allowed on either are

```
=, /=, >, >=, <, <=
```

Expressions can be parenthesized.

The expression grammar is a subset of the Ada expression grammar, and is

```
expression  ::= relation [ 'and' relation ] |
                relation 'or' relation

relation    ::= simple_expr [ RELATIONAL_OP simple_expr ]

simple_expr ::= [ '-' ] term |
                simple_expr '+' term |
                simple_expr '-' term |
                simple_expr '&' term

term        ::= factor |
                term '*' factor |
                term '/' factor

factor      ::= [ 'not' ] primary

primary     ::= literal |
                identifier |
                '(' expression ')'
```

1.11.8. Statements

There are three statements. They are the assignment statement, the **if** statement, and the Ada utterance. Statements are always terminated by an EOL (end of line) or by a ';'.

Valid Ada utterances are statements, declarations, and comments. Ada utterances are always terminated by an EOL, as directives must have a '@' in column 1.

```
statements  ::= statement terminator |
                statements statement terminator

statement   ::= assignment |
                if |
                Ada Statement, Declaration, Or Comment

terminator  ::= ';' | EOL
```

1.11.8.1. Ada Utterances

Executing an Ada utterance means 'copying' it to the 'file' that will be compiled. Ada utterances that are not executed are not copied.

1.11.8.2. Assignment

```
assignment  ::= identifier ':=' expression
```

Evaluates the expression and stores the result into 'identifier'.

1.11.8.3. If

```
If          ::= 'if' expression 'then' statements
                [ 'elsif' expression 'then' statements ]
                [ 'else' statements  ]
                'end_if'
```

Evaluates the expression. If true (non-zero) the statements of the 'then' clause are executed. If not, the 'else' or 'elsif' is executed in the normal fashion.

2. Implementation Details

2.1. Views

Views, as discussed in this document, are KKOM universe views. There are no cases where world views are exposed directly to the user. Universe views are hereafter referred to as UVs. World views are referred to as WVs.

2.1.1. What Is A UV

A UV is an array of pairs, 1 pair for each world. The pair is of the form

```
type UV_pair is
    record
        UV_part : KKOM object handle;
        WV_part : KKOM object handle;
    end;
```

If the WV_part is not NIL, that WV is used when the slot is referenced. UV_part can be nil in this case. If the WV_part is NIL, the UV named by UV_part is retrieved, then this process is repeated. If both are NIL, there is an error.

If both UV_part and WV_part are not nil, this is a refreshable view. UV_part names some path's default view. More later.

2.1.2. Types Of Universe Views

There are two types of UVs; complete and stub. A complete UV is one that can reference a WV for any/every world on a system. A stub UV is one that references a WV only for the world containing the UV.

2.1.3. A World's Default View

The default view for a world is a stub UV. This UV is created when the world is created, along with a WV. Changing the world's default view requires changing the WV_part of the stub UV. The UV_part is NIL.

2.1.4. The System View

The system view is a complete UV, built by finding all of the world default UVs and extracting the WV_part. These are inserted into the WV_part of the system view. The UV_part of the system view is set to NIL.

2.1.5. A Path's Default View

The path's default view is also a stub UV. Its WV_part is set to the WV specified by the user as the default. The UV_part is NIL.

2.1.6. Protected Views

Views made by import with protection commands are stub UVs. They are assigned a unique name (by KKOM). The WV part is set to point at the desired WV. KKOM may have to actually make a copy of the WV.

2.1.7. Composite Views

A composite view is a complete UV, and is built as an indirect view. The UV_part points at the included view, and the WV_part is NIL.

2.1.8. Other Views

Unmanaged views, release views, and working views are all complete UVs. None of these are indirect; if the UV_part has something in it, the WV_part does too. Only managed views can have the UV_part filled in, and if it does it refers to a stub UV in some path or subpath.

2.1.9. Operations On WVs and UVs

UVs are created and manipulated using the KKOM view package. The are deleted and expunged using the KKOM object package. There are no operations for directly manipulating WVs.

2.1.10. Read Only Views And Reservations

The KKOM facility provides for marking that the current version of some object (that is, the slot) cannot be modified using the view, even if the view is unfrozen. The reservation model uses this mark to implement the restrictions implied by the token. When this mark is on, any normal open for update fails. There is a form of open that ignores the mark, for use by the environment.

The compilation system always uses the 'ignore the mark' form of open, which allows make consistent to run over objects that are not checked out. Thus the compiler can run over read only views.

When a path is originally created, no slot in the working view is marked. Thus all objects are reserved and changeable in the path. When reservation model is enabled, all slots are marked, since the path is defined to be read only. A new subpath also has all slots marked.

When an object is checked out, the linkage database is checked to see if a parent has the token. If so, that parent's slot is marked, and the receiving subpath's slot is unmarked if it isn't not read only. The linkage database is changed to indicate the new owner of the token. Since new subpaths are created with the mark on for any object under CMVC control, this paradigm ensures that only one slot exists in the path/subpath structure with the mark off.

Check in and out only applies to objects that have been marked in the database as under reservation control. Thus, subpaths can create new objects, but cannot check them in unless and until the object is added to the controlled objects list. Check out will also fail on such an object. If the object is marked as under reservation control, but the subpath has no selected version, the subpath can add the object by using the accept operation.

2.2. Paths And Subpaths

Paths are directories, subclass path. Subpaths are directories, subclass subpath.

2.2.1. The Path Operations

1. Create
   Creating a path involves creating its directory, creating the working view, and the default view. An empty WV is created for the working view and inserted into the UV. It is likely that this WV will have to be filled in with some number of objects, such as the compatibility object, the target information, the path directory, and directories between the path and the world. The default view is then created, with NIL entries.

2. Fork
   Fork creates a subpath directory, a complete UV, and then copies the originating WV into the new WV using KKOM view operations. A copy of the WV is actually made. This copy operation only copies current versions, deleted versions are not copied. There are other actions, such as updating the linkage database, that happen as well.

3. Spawn
   From the KKOM standpoint, this is identical to the fork operation. The linkage database is set appropriately, and the new path is given a target key (somehow).

4. Revert
   If the operation is applied to a view, it sets the appropriate slot in the working UV to point at a copy of the WV named by the release. If the operation is applied to objects, they are copied one at a time to the original WV.

5. The UV is destroyed (expunged). KKOM takes care of the rest, such as deleting the associated WV if needed, and getting rid of versions no longer needed.

6. Release
   Release copies the UV using the KKOM view.copy operation, then freezes it. There will also be some processing to build an import and export keys database to be used by view compatibility checking. It is possible that this database will contain much the same information as the load object, and if this is the case we should find ways of sharing the information if possible.

2.2.2. The Ada Spec For Path

TBD

2.3. The Linkage Database

There is one linkage database per world, with one version. This database applies only to the working views, since the reservation commands can only be applied to such views. The linkage database contains the following information:

1. Path Identifiers
   Each path and subpath in the world is listed here, and given an id number. The Id number is the subscript of the path in the table. Wherever path entries are called for in the linkage database, this path ID is used.[11]

2. The Path Equivalencies
   These describes how paths are linked. These take the form of sets of path IDs.

3. The Object Equivalencies
   Each object in the world has an entry, specifying which paths that share that object. These take the form of sets of path IDs, and are used during checkout to find some other path that might have the object. The first entry in each set is the subpath/path that currently has the object checked out.

---

[11] This might be changed to use the object IDs. This is a space time issue, and will be resolved at implementation time.

4. For each equivalence set, a subpath tree is kept. This is used to find parents during the check out and check in operations. The policies switches are also kept on this tree. For example, an object can be checked out if the subpath retrieved from the object equivalency set is a parent of the requesting object.

2.3.1. Link Pack Operations

Talk about the resolving of the export object against a version of it, the filling of the available units list, and the add and delete operations.

Add takes an export object or an ada object or a world.

Delete takes an export object, and ada object, or a world.

Talk about when refresh occurs, and what information must be available for managed and unmanaged views.

2.4. The Import Operations

Import has several tasks to accomplish. These are to check compatibility of views, to fill in the UV, to fill in the link descriptor object, to resynchronize the link pack, and then to run do change_analysis.

2.4.1. Checking Compatibility

Compatibility checking requires checking a view'w export keys against the import keys of any other view importing the world, and checking the views import keys against every other world it imports. This process uses the ***keys database*** built when the release was made. The keys database contains the export key for every Ada object in the view, plus copies of those object's import keys. The export keys are grouped together and are tagged with and sorted by the Ada unit's object id. The import keys are sorted by world and object ID. Duplicates world/object ID references are removed by 'or'ing them together, leaving an aggregate.

The database is used as follows:

1. Repeat for each view already imported:

2.

   a. Retrieve that view's key database.

b. Check the new view's import requirements against the existing world's export keys. This is done by looking up the aggregate import keys in the new view's key database, and comparing these against the export keys in the existing worlds key database.

c. If the original view references the new world, add the import keys from the database to a set of import keys. This set is organized by object ID; keys with the same object ID are 'or'ed.

3. Check the set of import keys against the export keys in the new view's database.

The compare operation mentioned above is equivalent to **(required_key and new_key) = required_key**. If this isn't true, the views aren't compatible. Note that this only works for installed units, but this seems reasonable as source units in a frozen view aren't much use.

2.4.2. The UV

Filling in the UV in the importer is straight forward. The WV_part is set to refer to the new UVs own WV, found by extracting the WV_part from it's own slot. The a path is being imported, also fill in the UV_part with the name of the default UV from the path, otherwise make it NIL.

2.4.3. The Link Pack And Change Analysis

The link pack for some imported world is refreshed whenever a new view of that world is imported. This is done by opening the new verson of the export object, reading the list of units, and checking these against the existing list of imported units. change analysis must then be run. New imports are added to the list.

2.4.4. Refreshable Imports

The refresh operation traverses the supplied UV, looking for slots that have the UV_part filled in. WHen it finds one, refresh checks the UV pointed at by the UV_part to see if it refers to a different WV then the WV_part. If it does, the new WV is imported by doing a compatibility check, change analysis, and then filling in the WV_part.

2.4.5. "Import Ada Specs"

TBD

2.5. The View Stacks

The view stacks are implemented as stacks of object IDs of UVs. The session stack lives in session state, the context stack lives in core editor window state, and the job stack lives in job state. These stacks are small, perhaps 5 deep. When a job is started, the job stack is built by copying the session and context stacks onto it. The job is then started.

If the job results in opening a new window, the window manager is called with an object handle for the view used to find the version, an object handle returned by naming, and a context stack. The window manager does one of three things:

1. If the view part of the object handle returned by naming is not NIL, the context stack for the new window is cleared, then that view is pushed. The associated view is set to the same view.

2. If the object handle for the view used to find a version is not NIL, the supplied context stack is copied into the new window's context stack. The associated view is set to refer to the view used to find the version.

3. Otherwise the context stack is copied and the associated view is set to NIL.

The version specified by the combination of the two handles is then displayed.

2.5.1. View_Stack Ada Specs

TBD

2.6. Merging

Talk about the least common subsequence, indicating regions of difference, and the common ancestor data base.

2.6.1. Merge_View Ada Specs

TBD

2.7. Conditional Compilation

2.7.1. Grammar

2.7.2. Diana

Talk about the new node types, marking the node as skipped or not, and the implementation of as_list.

2.7.3. Batch Compiler

Talk about the parser not being able to accept programs with cond. comp in them.

3. Scenarios

3.1. Introduction

This section presents a number of development scenarios to illustrate key properties of the CMVC design. Note that these scenarios are entirely illustative and do not represent the only way to use the various CMVC facilities. There are unlimited variations on the strategies shown here, and there are completely different strategies that can be implemented using CMVC.

3.2. Casual Usage

3.2.1. Situation Summary

A novice user wishes to perform simple operations in his home world. The user does not know about CMVC, and for the small programming tasks involved here does not need the power of CMVC.

The user must be able to browse packages provided by others, write small Ada programs using the provided packages, use the interactive compilation facilities, execute and debug his program, and not interfere with more serious development activities on the same machine.

3.2.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. The CMVC facilities are completely transparent to the novice user.

2. The view stack mechanism, combined with other environment facilities, allows more knowledgable personnel to establish the proper environment for a novice.

3. The epsilon paradigm for basic programming tasks (browsing, editting, compiling, executing, and debugging) is easily summarized in simple cases and requires no user understanding of CMVC mechanisms.

3.2.3. Scenario

For this scenario we consider a new user, named Novice, who has recently joined an on-going project. Novice has been through a short training course on the Rational machine, and has some limited experience with Ada.

Before introducing Novice to all of the development and release procedures used on the project, the team leader, named Leader, decides to have Novice write some small test programs. This will allow Novice to become familiar with the project while making an immediate contribution, and will allow Leader to assess Novice's skill level and aptitude.

In addition to Novice's home world, which is where the test programs will be written, Novice will need limited access to three other worlds used by the project -- Regression_Tests, Test_Utilities, and Foo_Subsystem.

Regression_Tests contains documents describing project test procedures and contains a large number of existing test programs which Novice may use as examples to guide him in writing new tests.

Test_Utilities contains the various packages used by all of the regression tests to generate test data, analyze test execution, and report results.

Foo_Subsystem contains software under development by the project team. Leader has instructed Novice to write a regression test package for the Foo_Sorter package in the Foo_Subsystem.

Novice must be able to browse these three worlds, and compile his test programs against the units in these worlds, but he should not be able to modify the contents of these world.

3.2.4. Elaboration

3.2.4.1. Setup

The first step in this scenario is for Leader, who understands the use of CMVC, to set up Novice's account to achieve the properties described above. Leader must create the account, set up the session view stack, update the link pack in Novice's home world, and then establish proper access control. Then Novice can "safely" use the system.

Creating an account is as simple as invoking the **Create_User** command.

If Leader takes no further action, when Novice goes to examine one of the three worlds described above, he will see the view of the world selected by the system view (see 1.2.7).

This project does keep a stable, well-tested release of each world in the system view. However, the Foo_Sorter package is a relatively new package that has been changing frequently. Leader wishes Novice to test the most recent project release, not the old (stale) release selected by the system view.

In this case, the correct release of each of the three worlds is selected by a project release

view, maintained by Leader. In order to make sure Novice sees this view rather than the system default, Leader modifies the login procedure in the newly created user world so that the project view is pushed onto Novice's view stack each time Novice logs in. Then Novice would see the view selected by Leader, and Novice would be unaware of the fact that there are other views of these worlds.

Leader can either push the project view onto the Novice's session stack, or he can set the window view stack of the initial login window on to reference the project view.

In general, the former (using the session stack) should be restricted to use where change is infrequent and multiple views of the selected worlds are not going to be examined. This is particularly appropriate for establishing a tool environment.

Using the window stack allows much more flexibility. However, since the window stack will be used extensively in later scenarios, we will assume that leader elects to use the session stack in this example. This is reasonably approriate, since Novice is no going to want to change this frequently, nor is he going to be examining multiple views.

In order for Novice to compile against specs from the project worlds, the link pack (see section 1.6.1) must be updated. Leader can simply go to the world created for Novice and issue a `Links.Add` command naming each of the three worlds. This will use the default export object in each of the three worlds to determine visibility for compilation purposes.

Finally, Leader must establish proper access control. In this case, Novice will automatically be a member of the `public` group (the group containing all users) and the Novice group (the group consisting of the single user Novice). `Create_User` automatically established the defaults for Novice's home world such that the Novice group has owner and write access to objects, while the Public group has read access to objects.

There is also a group for the project, and all the objects in the project worlds (in particular, the three worlds we have been discussing) are set up to provide read access to the project group and to provide write access and owner access to the specific developers responsible for development in those worlds. Leader can safely add Novice to the project group, giving Novice the ability to read but not modify units in the project worlds.

3.2.4.2. Browsing

Novice must be able to easily browse units in the various project worlds. In particular, Novice will rely heavily upon `Definition` and `Show_Usage`. Both of these operations will rely upon the view stack mechanism.

Let us follow Novice through a sequence of browsing operations. This sequence will primarily illustrate basic environment operation. CMVC is (intentionally) invisible throughout the sequence.

Initially Novice has a single window displaying his home world, with the associated view (see section 1.4.2) being the System View. Throughout all of this section, the window context will be void (see section 1.4.3.1).

Leader gives Novice the name of a test package, Bar_Inverter_Driver, in the Regression_Tests world to use as a model in writing the Foo_Sorter_Driver package.

Novice executes the **Definition** command with the full path name of this model package. As described earlier (see section 1.2.8), a string name resolves to an object and does not require any view information. However, selecting a version of the given object uses the mechanism described in section 1.4. In this example, the project view on the view stack will be used to select the appropriate version of Bar_Inverter_Driver, which is displayed with the project view as the associated view.

If Novice wishes to move to the body of this test package, he simply uses the **Other_Part** command. Again the project view is used to select the correct version (i.e., the one corresponding to the visible part) and the project view is the associated view for the new window.

In the body of this package, Novice notices that there are numerous calls to a Report.Results subprogram. If Novice wishes to see this routine and read the comments describing its use, he simply selects one of the calls and executes the **Definition** command. The visible part of the Report package is brought up in a window with the Results subprogram highlighted. Once again the project view on the view stack is used to ensure that the proper version of the Report package is selected, and the project view is the associated view for the new window.

Novice might then notice that there is another, overloaded, form of Report.Results which looks interesting. Novice would like to look at a using occurence of this form of Report.Results, so he selects the defining occurence in his window and executes **Show_Usage**. This brings up a window with a list of the various packages that contain using occurences. Novice selects one of these and hits the **Definition** key. A new window appears displaying the selected unit with using occurences underlined. Again the project view on the view stack is used to select the proper version of the unit and the project view becomes the associated view.

At this point, Novice receives a mail message from the system adminstrator announcing the installation of a new mathematical library on the system for general use. Novice has a natural interest in such things, and executes the **Definition** command with the full string name of one of the packages described in the mail message. The project view does not select a view of this world; however, the system administrator has updated the system view to include the new math library. Since the system view is in the session view stack, the system default version will be selected and displayed in a window, with the system view as the associated view.

Novice then decides that, while this is a nice math package, he had better start looking at the Foo_Sorter package, so he executes **Definition** with the string name of the Foo_Sorter. Foo_Sorter is in the project view, and once again the system displays the version selected by Leader when he set up the project view, in a window with the project view as the associated view.

Note that we are keeping track of the associated view and the job stack to illustrate the CMVC mechanisms and to prepare for more interesting situations which follow. However, Novice is completely oblivious to CMVC while browsing. Novice does not know about, nor care to know about, view stacks, associated views, or even versions of objects.

3.2.4.3. Editting and Compilation

Eventually, Novice gets to the point where he is ready to start writing some code. By analogy with the structure of the examples he was just browsing, Novice decides to write two packages -- Foo_Sorter_Driver and Foo_Sorter_Tests.

The Foo_Sorter_Driver package will be a test driver with several entries for performing different levels and types of testing. The Foo_Sorter_Tests package provides a set of specific test cases that are invoked by the test driver with different test data. The Foo_Sorter_Driver spec will reference some types and functions in Foo_Sorter_Tests. Both specs reference packages in Test_Utilities and Foo_Subsystem.

Novice first creates Foo_Sorter_Tests with the Create_Package command, creating a new object in his home world and bringing up a window with the system view as the associated view (since that is the first (and only) view on the view stack with an entry for Novice's home world). Novice begins writing the package using the various facilities of the Ada Object Editor.

Novice will occasionally request semantic analysis, using the **Install** key. Since the link pack has been set up to provide visibility to the project worlds, Novice can compile references to units in those worlds. Novice will be compiling against the same versions he was browsing earlier. Thus browsing and compilation reference the same versions in this scenario, and CMVC remains transparent. While correcting semantic errors it is particularly important that Novice see the same versions when browsing that the compiler is seeing when doing semantic analysis.

Note that in Epsilon, unlike Gamma, the user can edit installed units in place without demotion or insertion points. Minimal (declaration level) obsolescence is propogated at the time of installing a modified unit. For this reason the Gamma distinction between **Install** and **Semanticize** does not exist in Epsilon. However, there are still several keys associated with install operations, differing in their willingness to compile the execution or compilation closure of the current unit.

After making some progress with the spec for Foo_Sorter_Tests, Novice creates the spec for Foo_Sorter_Driver. Installing this spec will construct references to Foo_Sorter_Tests. If Foo_Sorter_Tests is not installed when installing Foo_Sorter_Driver, either semantic errors will be reported or the system will automatically install Foo_Sorter_Tests, depending upon which form of **Install** is used and whether or not there are semantic errors in Foo_Sorter_Tests.

When he is reasonably satisfied with the specs (or any time he feels like it) Novice can create skelatal versions of the bodies of these two packages using the **Build_Body** command, and then fill in the two bodies. Using the proper form of the **Code** key will promote all the units in the execution closure to coded in prepartation for execution.

3.2.4.4. Execution and Debugging

Once Novice has his two packages coded, he executes individual tests exported from Foo_Sorter_Tests or executes complete test sets provided by Foo_Sorter_Driver simply by calling those packages from a command window. Novice executes the same versions of all objects as those used during browsing, editting and compiling, avoiding confusion and allowing CMVC to remain transparent.

Novice determines that one of his very first tests, **Foo_Sorter_Tests.T1**, is not working properly. In the window on Foo_Sorter_Tests'Spec, Novice executes the test with the debugger. This brings up a debug window. Novice decides that he wants to set a breakpoint at the first statement of T1. He moves to Foo_Sorter_Tests'Body as usual, selects the first statement, and issues the **Break** command.

Having set the breakpoint, Novice continues execution of his command. The debugger will stop the program at the breakpoint, highlighting the statement where it stopped. In this case, the debugger will highlight the statement in the same window where Novice set the breakpoint, with the system view still being the associated view.

Novice can look at the parameters or locals in T1 (or any other object) by selecting the declartion and executing the **Debug.Put** command. Using these and other facilities Novice can see exactly what is program is doing. Again, the versions executing and being debugged are exactly those editted and compiled.

When Novice finds the problem in T1, he hits the **Edit** key on Foo_Sorter_Tests'Body and makes the necessary changes. The associated view for the window remains as the system view. Novice can then re-execute his test (from the beginning) and see the results of his change.

Finding that there are still problems with T1, Novice sets a breakpoint, executes up to the point of the breakpoint, and examines the state of his program at this point. Once again Novice finds some problems and hits the **Edit** key and makes changes in T1. However, this time he decides to set a breakpoint further down in T1 and then continue execution rather than re-executing the test from the beginning.

Up to this point Novice has not needed to understand that there are multiple versions of objects and multiple views of worlds selecting different versions of objects. However, Novice is now executing a version of Foo_Sorter_Tests which does not contain the changes he just made. When the test comes to the next breakpoint it will display the version that is running. Novice ends up with two windows on Foo_Sorter_Tests, one representing what he is running (associated view is the load image of the command being debugged) and one representing his latest edits (associated view is the system view). At this point it is very obvious that there are two versions. This only occurs because Novice has continued execution after modifying his program.

Note that if Novice does become confused and try to edit the version he is executing when it differs from the edittable version, the system can easily inform him of the problem and can even move him to the appropriate window. Similarly, if Novice is in the edittable version and tries to set a break point or perform some other debugger operation, the system can move him to the window on the executing version of the unit.

In this case the executable versions that differ from the edittable versions will "disappear" when the program completes. The windows will be removed, although those particular versions may remain on the system if

As long as Novice re-executed his program from the beginning after making changes he did not need to understand that there were multiple versions. When Novice does not

3.3. Simple Subsystems

3.3.1. Situation Summary

A system consisting of four subsystems is under development. The development team includes four subsystem owners and one system integrator. Each subsystem is only changed by its owner.

The five developers in this project require support for parallel independent development of each subsystem with incremental system integration using the subsystem paradigm.

3.3.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. The world and export/import mechansims support and enforce subsystem decompostion.

2. The path and release/import mechansims support a methodical release process.

3. Body changes, compatible spec changes and incompatible spec changes are all supported with rapid turnaround, both within a single subsystem and across subsystems.

4. The path and view mechanisms capture declarative information provided by the user (through release/import, etc.) in order to make CMVC transparent to basic operations (edit, compile, execute, debug) in the subsystem case (where the relevant execution closure is included in all views, see section 3.7).

5. The subsystem mechanisms in Expsilon are sufficiently flexible to allow cycles in subsystem spec dependencies.

6. Operations requiring knowledge of the clients of a subsystem (i.e., **show_usage**, reuse of decl numbers, etc.) can be supported.

3.3.3. Scenario

For this scenario we consider the development of a system, SYS, consists of four subsystems SS1, SS2, SS3 and SS4. SS1 provides different facilities to SS2 and SS3. SS4 uses the facilities provided by SS2 and SS3. There are some spec dependencies (in both directions) between SS2 and SS4.

There is a system tests subsystem, SST, which consists of system test programs which exercise the other four subsystems.

Each subsystem belongs to a single user, U1, U2, U3, U4 and UT respectively. Each subsystem is only modified by its owner.

This scenario traces the use of CMVC facilities throughout the life of SYS, starting with preliminary design, through detail design, coding, subsystem test, integration, and maintenance.

3.3.4. Elaboration

3.3.4.1. Design

3.3.4.2. Code

3.3.4.3. Subsystem Test

3.3.4.4. Integration

3.3.4.5. Maintenance

3.4. Multiple Users per Subsystem

3.4.1. Situation Summary

Assume that two developers are working together on one of the subsystems in a system like that described in section 3.3. For whatever reason, the two developers wish to work in a single path, rather than using the parallel path mechanism (section 1.10) or the reservation mechanism (section 1.8).

The developers realize that this approach has severe limitations (i.e., changes by one may obsolesce units the other is using). They are willing to take responsibility for coordinating their own activities; however, they rely upon the basic synchronization facilities of the environment to prevent them from simulatneously changing the same unit or overwriting changes made by each other.

3.4.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. The interaction between views and basic synchronization in the envrionment, including the interaction between retention lists and superscedable read.

2. The limitations of using a single view for several developers.

3.4.3. Scenario

TBD

3.4.4. Elaboration

TBD

3.5. Multiple Subsystems per User

3.5.1. Situation Summary

The scenario of section 3.3 is modified so that two of the subsystems are owned by a single user. This developer wants to be able to make coordinated changes in his two subsystems as if the two subsystems were one. In particular, the developer wants changes in one subsystem that effect the other to immediately (by the next **Install**) obsolesce dependent units in the other subsystem. The developer also wants the changes in one subsystem to be immediately visible to units in the other subsystem.

The user is willing to provide declarative information to describe the desired configuration, but he then wants all of the basic edit/compile/execute/debug operations to behave as if he had one subsystem instead of two.

3.5.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. Composite views support the description of configurations which allow modification in several worlds.

2. Once the composite view has been properly constructed, most environment operations transparently operate as if the user was working in a single world.

3. The full power of the subsystem mechanism is still available when needed.

3.5.3. Scenario

TBD

3.5.4. Elaboration

TBD

3.6. Multiple Machine Development

3.6.1. Situation Summary

Repeat the scenario of section 3.3, except that each subsystem is being developed on a different machine.

3.6.2 Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. The subsystem paradigm extends naturally to a multiple machine environment.

2. The same release procedures used in the single machine case can be used in the multiple machine case, although additional release administration is required.

3. Common user scenarios are identical (from the user's point of view) in the single machine and mulitple machine cases, except for additional latency introduced by the network.

4. Operations requiring knowledge of the clients of a subsystem (i.e., **show_usage**, reuse of decl numbers, etc.) are more interesting in the multiple machine environment.

3.6.3. Scenario

TBD

3.6.4. Elaboration

TBD

3.7. Separate Systems

3.7.1. Situation Summary

Two groups of separate subsystems for two systems, Client and Server. Client wishes to be able to debug Client code, but normally not interested in Server code. Release procedures separated so that client does not require knowledge of execution closure of Server. Violates the "always edit, compile, execute & debug same versions" rule when the user doesn't mind. Must allow full consistent debug when required. (refresh and stack solutions should be illustrated)

3.7.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.7.3. Scenario

TBD

3.7.4. Elaboration

TBD

3.8. Parallel Development

3.8.1. Situation Summary

Two users making changes to same code in parallel, periodically merging.

3.8.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.8.3. Scenario

TBD

3.8.4. Elaboration

TBD

3.9. Simultaneous Development

3.9.1. Situation Summary

Single user with a maintenance path and a development path. Reason about (browse) and make fixes (edit/compile/execute/debug) in both paths at the same time. Propogate changes from maintenance path to development path. Occasionaly fold feature or fix from the development path back to the maintenance path.

3.9.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.9.3. Scenario

TBD

3.9.4. Elaboration

TBD

3.10. Simple Subpaths

3.10.1. Situation Summary

Two users doing concurrent development in the same world, using the reservation facilities.

3.10.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.10.3. Scenario

TBD

3.10.4. Elaboration

TBD

3.11. Subpath Hierarchy

3.11.1. Situation Summary

Two level hierarchy, two teams of two developers with one manager/designer.

3.11.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.11.3. Scenario

TBD

3.11.4. Elaboration

TBD

3.12. Simple Linked Paths

3.12.1. Situation Summary

Same as section 3.11 except R1000, Vax, & 68020 paths. All code is target independent.

3.12.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.12.3. Scenario

TBD

3.12.4. Elaboration

TBD

3.13. Conditional Compilation

3.13.1. Situation Summary

Same as section 3.12 except that target dependencies are represented by conditional compilation.

3.13.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.13.3. Scenario

TBD

3.13.4. Elaboration

TBD

3.14. Target Dependent Units

3.14.1. Situation Summary

Same as section 3.13 except that completely target dependent (severed) units are introduced.

3.14.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.14.3. Scenario

TBD

3.14.4. Elaboration

TBD

3.15. Target Subsystem Compatibility

3.16. Composite

3.16.1. Situation Summary

Combine section 3.14 with 3.6 and with 3.8.

3.16.2. Goals

This scenario is designed to illustrate the following aspects of the CMVC design.

1. TBD

3.16.3. Scenario

TBD

3.16.4. Elaboration

TBD

Index

Accept 25
Accept command 23
Annotations 21, 22
Associated view 13, 20

Check in 19, 21
Check out 19, 25
Closure 7
Closure command 15, 17
Common ancestor 27
Compatible 5
Compilation closure 7, 17, 22, 23
Composite view 7, 34
Conditional compilation 23
Configuration Management 3
Consistent 4
Controlled 19
Current version 12

Default 10
Default release 10, 14
Default view 8
Differential 5, 10
Directories 9

Export object 15

Fields 22
Frozen 4

Get 26

Import 5, 6, 8, 10
Import command 15
Import descriptor 15, 16, 18, 19, 20

Job stack 13
Join set 6, 17
Joined 6

Keys database 36, 37

Link pack 15, 18
Linkage database 20, 24
Linked path 11
Linked paths 10, 23

Managed views 7

Merging 26

Name 9, 11, 14

Object 3, 4, 8
Object handle 9, 11, 13
Object number 12

Parallel paths 26
Path 9, 14, 19
Protected 17

Read only 20, 26
Refresh 17, 18
Release view 6, 9, 10, 11, 27, 34
Reservation model 10, 19, 23, 26, 34

Session stack 13
Severed objects 24
Simple name 8
Slot 12, 27, 34
Subpath 10, 19, 23
System view 8, 14, 17

Target 10, 23
Token 20, 24, 34

Unmanaged view 7, 34

Version 3, 4, 5, 6, 11, 12, 15
Version Control 3
Version number 12, 15
View 4, 5, 9, 11, 12, 14, 33
View stack 12, 13, 17, 18

Window 13
Window stack 13
Working view 6, 9, 10, 11, 14, 20, 26, 27, 34
World number 12
Worlds 4