The Epsilon System

Principles of Operation

Volume I

User Interface Description

Revision 0.8

April 23, 1986

Table of Contents

1. Overview

1.1. Ada-Orientation

The primary purpose of the environment is to aid in the development of Ada programs. Ada units are represented by their Ada names, structured as Ada subunits, removing the need for implicit translation between file names and program names. Programs are stored with sufficient context to provide assistance in navigation through existing units and in construction of Ada units, statements and declarations.

The command language for the system is Ada. The goal is to provide the power of direct access to Ada constructs without imposing the structure of a compiled, strongly-typed language. Accomplishing this goal requires assistance with both syntax and semantics in a way that allows the required program fragment to be constructed from the user's intention.

1.2. Editor-Based

User interaction with the system and his own programs is through the editor. The users' investment in learning these facilities is repaid in increased functionality and more uniform interface.

The user is primarily interested in manipulating the entities that make up the environment. The user interface is concerned with providing an orderly and convenient method of expressing these manipulations. The user communicates by pressing keys or other input device. Though system entities are often presentable in a readable form, the objects themselves are not made up of the characters used to present them. As a result, the user interface is constructed to interact with the user through character editor and with the entities themselves in terms of their own representation. To accomplish this, the editor is separated into two layers:

1. The visible interface is a multi-window editor that provides a core set of facilities for handling user input, editing and screen management. This is called the *Core Editor*.

2. The type-specific, object-knowledgeable portion of the editor is called the *Object Editor*. Which object editor is used depends on the type of the object (entity). A set of common operations are supported by each of the object editors (to the degree applicable). Specific object types can also provide operations unique the the particular type of object.

Further details on Core Editor and Object Editor functionality are presented in 5.1 and 5.2.

1.3. Extensible

All of the mechanisms that are required to build the interface should be available for users to write similar programs that use all of the features of the R1000 interface. The alternative is to end up writing everything that ever runs on the R1000. Historically, there have been problems with doing this:

1. The interface we need is too complicated for customers

   This is really a stalking horse for the complaint that the only interface is the implementation-level interface, which we intended for internal usage and didn't expect non-compiler-writers (or other discipline) to have to deal with. Lowest level system call interfaces normally have this problem, but there are usually tool interfaces.

   One of the goals of the system call interface has to be providing an interface that provides access to the complete set of facilities in a manner that the appropriate customer personnel can understand. We should also anticipate that this interface will be taken seriously by customers.

2. Making the implementation visible is dangerous

   Very similar to the point above, but somewhat more threatening. With the reduced reliance on segmented heap pointers, this may be less of a risk.

   For new functionality (or new specs for existing functionality), consideration should be given to making it possible to expose the interface without risking system integrity. For the small number of high-frequency paths for which speed is more important than safety, there need to be safer interfaces that provide equivalent functionality.

3. The code isn't structured that way

   A number of parts of the system implementation are designed to provide the functionality that we use and making it possible to do something a different, but functionally equivalent, way can be a lot of work. This is hard to do anything about; resources are finite and complete generality is expensive.

   Separation of policy and mechanism is useful in making it less likely to introduce new instances.

1.4. Integrated

1.4.1. Related Sets of Facilities

There are a number of conceptual levels at which users will need to deal with objects. Some objects will provide all of the various interfaces described.

1. Interactive Editor

   Editor interfaces provide an interactive form of object display and the ability to change characteristics that are presented without interposing a specific naming or command discipline. Basic paradigms are direct modification and select/operate. Section 5.2 describes the *common operations* provided by editor interfaces; each editor can provide *object-specific operations* as required.

   Editor operations are provided for ease of user interaction. Complete system functionality should be available without resorting to editor interfaces.

2. User Commands

   Commands are ada procedures that it is expected will be conveniently entered into command windows by the general user community.

   a. Procedures with **in** parameters that have reasonable defaults and are types with constants (e.g. string, boolean, integer, duration, enumerations) or, less desirably, nullary functions.

   b. Object name parameters deal with wildcards and selections.

   c. String parameters have defaults that are quoted; if there is not reasonable default, the parameter is a recognizably illegal value that briefly describes what successful replacements would be.

   d. Except where the purpose of the command is access files, output is to Current_Output and Current_Error. Do not require user input to clarify options. Use Current_Input if absolutely necessary.

   e. Any command should be able to execute successfully as a background job with the user logged out.

3. Tool Interfaces

   Tool interaces are Ada units for use in programs or advanced command windows; perform I/O only on objects provided as parameters. Collectively, along with the IO packages, provide all of the functionality necessary to implement commands. Commands can be implemented on non-visible interfaces in the interest of efficiency, but there should be understanding of how and why non-exported functionality is required.

4. Input/Output Access of Objects

   Input/Output interfaces are provided for easy programmatic interface to a variety of objects. Two related forms of interface are possible:

   a. Open/Close/Get/Put interface to object images. For any object for which there is an editor-provided image, there should be an interface to read the image (or selection) as text. For object editors that provide Read/Write capability, output should be supported. Users should be able to write applications that read the current image/selection, change it, and write it back.

   b. A uniform Open/Close/Read/Write access should be provided for objects that consist of streams of interesting types (e.g. links, activities in Gamma)[1]. This makes it easy for implementors to write tools or generics built around a standard, familiar interface.

5. Script or Interpreter Interfaces

   A number of parts of the system provide script-based command interpreters or question/answer interfaces. These are provided for low-level interfaces and for interactions where many non-standard operations must be performed in an interaction pattern that is unsuited to an editor interface. These are often the easiest form of interface to build and frequently outlive their originally projected lifetime.

   a. Any response should be recognized if it is a prefix of exactly one of the possible responses. Responses should be chosen so that they are not only meaningful, but tend to be unique in the first few characters.

   b. The input/output streams that provide the script should be done in a manner to allow them to be relocated by appropriate request, *e.g.*, should not be limited to only working on the operations console.

   c. Context-dependent and history dependent prompt defaults should be provided, as in the kernel command interpreter.

   d. Command interpreters should be prepared to accept parameters on the same line as the commands being issued, omitting prompts as the corresponding parameters are supplied ahead of time. Appropriate accomodations for flushing type-ahead in the face of errors is also useful.

   e. Command interpreters should provide a convenient way of accessing functionality, they shouldn't **be** the functionality.

---

[1] This is a special case of uniform vocabulary and a non-trivial design task.

One of the problems in providing a comprehensive facility to be used with the mix of possible execution environments is providing appropriate levels of input for each particular use. One attractive solution is a *progress object*, first introduced in the merge processor. The central idea is to have an object that represents the work done or to be done. The object can be modified equally well by editors or commands. If the appropriate information is captured in the object, it is possible to create the equivalent of a log and it is possible to create an object editor to display the information involved. A description of how such a facility for Ada compilation is described in 3.2.

1.4.2. Uniform Vocabulary and Usage

Ideally, the user should be able to construct a reference to perform a standard operation on a known object type in a known package without actually having to look at the package spec[2]. This is particularly well done in a subset of the abstract types, where an experienced user can predict common operations are done.

One area of change needs to be choice of parameter names. While it is useful to have parameter names that provide useful guides to their meanings, being too specific leads to very similar procedures to have subtly different parameter names. This is a problem when completion provides full name notation and the user wants to convert to a similar procedure, or just another procedure in the same package, to perform a related operation[3].

1.4.3. Ownership of Objects

The user is encouraged to look upon images as representing the underlying objects. It is particularly frustrating to encounter self-locking. If the user can see the object, it should be possible for an invoked tool to read the object. If the user can write the object, tools should be able to. There are obviously restrictions on the number of user jobs that can read/write the object concurrently, but there shouldn't be issues of locks held by the wrong places. Committing an object makes it permanent and makes it potentially visible to other users; it isn't necessary for the user to reference it from commands or tools. Facility is available to specify the last committed version should it be important to see the state of the object as seen by other users.

Looking at an object doesn't keep others from looking at the same version or from creating a new version of the same object. Two users cannot simultaneously create new versions of the same object in the same view; the possibility of doing so is detected when the user first expresses intent to change the object, not when the commit is attempted.

---

[2]The current universe specs bear the mark of their pluralistic origins. Different conventions are used, different names for objects, different names for similar functions, etc. This means that the user is required to know how the particular package is constructed.

[3]Work is required to better specify the vocabulary and usage conventions.

1.5. Maxims and Cliches for Developers

1. Simple Things are Simple

   Providing power is important, but it needs to be layered so that the most common operations can be performed by users who don't really understand Ada, the environment or temporal logic.

2. Customer Programs Run on Target Machines

   Providing a good environment for developing Ada programs natively is a first step, but it doesn't solve the basic user problem. Having integrated facilities for the R1000 target makes it more important to have similarly integrated interfaces for target development. Moving from the 2060-MV based development to the R1000 was a great improvement; customers won't appreciate making the transition in the other direction.

3. We Aren't Representative

   The typical customer is interested in getting programs to work, not in learning how to write compilers or every last detail of the LRM. Because we are very interested in language issues, the local vocabulary and usage is heavily steeped in Ada concepts that the customer won't want to become familiar with. Each customer can be assumed to have a working knowledge of Ada and we don't currently have much choice except to assume knowledge of English; assuming more invites negative customer reaction.

4. Details are Important

   There are a variety of ways in which the product will be judged, including functionality, integration, ease-of-use; each user will view some of these as more important than others. No matter how well the product rates on these criteria, users will expect the system to work. Having additional functionality doesn't buy forbearance for details that don't work; small problems can cast doubt on large areas of functionality.

5. The User Owns the Name Space

   No tool should preempt names in user contexts. There must be a way for the user to choose the name/directory into which state is placed. Reserved, non-Ada names are an option provided they don't appear by default, don't match simple wildcards, and don't prevent the destruction of the world, view, or directory. An extension of this rule requires that tools not dump garbage in the user's home directory.

6. Windows are Expensive

   Placing a window on the screen takes another window off of the screen. If the user asked for the window (*e.g.*, definition), it is reasonable to bring it

up[4]. On the other hand, it is much less likely that the user will welcome the appearance of and unbidden window, even if it promises help or other information. In particular, if the information wasn't really wanted, the loss of the window will far outweigh the good intentions in providing the help[5].

7. Never Force Interaction

Commands that require additional information should be set up to allow this information to be provided by additional parameters or commands. Unless the command provides a general command interpreter or editor interface, it shouldn't expect the user to provide input to answer questions. Not only does this force interaction in undesirable ways, it has unexpected ramifications when commands are run other than interactively. Where additional information is required and it is expected that users may be willing to answer questions, it is possible to have parameter values filled in by functions that request information; this allows user choice as to how information is acquired.

8. Unify Similar Operations

Commands to do the same or similar thing should be unified. To the degree that flexibility requires multiple operations, these should be built using the same paradigm, recognize the same switches, and in general act like they work together. In addition to there being too many commands, there are too many packages.

9. Tailor Commands to User Needs

Commands should be tailored to do things that users want to do, **not** to do things that are easily implemented. Commands should be grouped in terms of operations that the user is likely to want to perform together; this may not create command packages that correspond to the implementation scope of a single group or developer.

Tools packages are expected to supply a complete and consistent set of functionality; commands are expected to provide operations that are commonly executed. As long as commands can be implemented using the tools packages, there is a limit to the degree that all functionality must have a corresponding command.

10. Assume the User is Right

Except for destructive operations, make the assumption that the operation that was requested made some sense. Strict construction of the user's

---

[4]There are issues about window replacement, but that's a different problem.

[5]Menu windows are helpful in some, but not all, of the circumstances when they pop up.

requests will most typically require that they be re-phrased to do what the command could have done originally except that the environment can, presumably, do so faster than the user. Certain operations require some precision is specification because they are deemed to be *destructive*. Care in these cases is considered to be in the user's best interest (whether it is or not).

In addition to the rules for cursors, selections, cursors in selections, etc. If something is selected for an operation, the closest containing object selected set of objects should be operated on. If the object containing the selection doesn't have Diana references, the selection should be interpreted as if the text had been typed into the parameter to operation. Some clarification of the relation between cursors, selections, and images is required.

11. Support Standard Protocols

Operations that *look* like they should work, should work. Whatever user-level mechanisms are supported in the environment should be supported **everywhere** that would make sense to the user. The user is much more likely to view the appropriate function of the system in terms of the way the screen looks and the analogy to how other things work than to act on the basis of specific rules.

12. Succeed Quietly, Fail Loudly

It should be possible to tell that an operation didn't work by its messages. For simple operations, it should be possible to tell that they did work by their completion. Operations should **not** fail quietly. An operation that fails should provide enough information to allow a novice to understand what happened in terms that are appropriate to the problem encountered. Resist the temptation to express the problem in the precise terms that would be used by the implementor; use terms that will make sense to the user.

13. Avoid Directory Nesting

Don't create extra levels of directory to solve name conflicts. Although there seemed to be little choice, the Gamma subsystem tools are particularly bad about this. If nested directories are required, they should be used for information that the user doesn't create and seldom accesses, *i.e.*, it is OK to hide things in out-of-the-way places as long as they aren't frequently named by the user.

14. Avoid Hidden State

It is reasonable to allow the user to specify characteristics of how the system will operate for particular circumstances. It is **not** reasonable to have these characteristics change based on or depend on state that is not readily apparent to the user. Information in banners should explain state. Text banners currently have the bad characteristic of seeming to show a context, but not maintaining it.

15. Pay As You Go

    The work required to perform an operation should be proportional to the work required do what the user thinks was requested. Simple interactive operations should be fast.

16. Provide a Complete Facility

    When introducing an object or capability into the system, provide sufficient facility to meet the spectrum of user needs. This includes tools interfaces, the ability to get back any value that can be set, an object editor, etc[6].

1.6. Error Handling

The following is a list of goals for the Epsilon error detection and reporting scheme.

Flexible — Error messages need to exist separately from the programs. This allows the text of messages to be edited for consistency, wording and language changes.

Specific — Many current error messages include information specific to the particular instance. There needs to be some way to associate this information with the text of the message. Content insertions may not be ordered the same in the final message as in the implementor's initial conception.

Detail — There needs to be a way of associating additional information, LRM references, etc. with error messages.

Separation — Service routines (e.g. directory naming) detect many of the errors that are made. The errors are reported by some higher level. There need to be separate, but compatible, facilities for recording errors and reporting them.

Detection — It should be predictable under what circumstances an error that has been detected will have generated a reasonable error indication.

Efficiency — One program's error is another program's normal processing state. The overhead of reporting errors should be limited to the processing required to capture what happened, not formatting and processing the actual message.

Reporting — Error reporting should take place through some uniform interface that determines what and where to print the message, depending on profile, interactive nature of program, etc.

---

[6]Consider the counter-example of keyboard macros. Keyboard macros are very useful, as is the ability to save them. The can only be saved as a group, there is no way to remove one, view or change its contents, associate a name, comments or documentation with it, there are no tool interfaces, etc.

Use — It should be possible for any implementor to effectively report errors detected at lower levels without major effort. It should be possible to add new errors to the set detected without prior planning and without undo inter-group coordination.

Response — Messages should make it clear whether the error is the result of something the user did wrong or whether the system has malfunctioned. Any mysterious hex numbers should be clearly marked as being intended for Rational personnel to diagnose our problems.

Programs — There should be a programmatic interface that implements all of the appropriate metaphors.

1.6.1. Directory and IO Errors

Most user programs, especially those not intended to tailor the environment to their development style, will tend to use standard IO and directory interfaces. Epsilon will provide better handling for these errors. One method that is being provided is the ability, having received a bad status, to call another interface with some of the same parameters and the offending status to get appropriate details. Chapter 14 IO packages cannot use exactly the same form, since they are constrained to raise for all error cases. It would be possible, however, to attach the error status to the file handle and provide a separate package to give the appropriate detail.

Another improvement is the conversion of the exceptions in IO_Exceptions into flavored exceptions, as with Constraint_Error. This is done by assigning a range of exception values to one IO_Exception exception and having the IO (and other) packages use the specific versions. User programs can still catch the non-specific version, but the name of the exception that come out of the debugger or into the message window can be more precise. More work is required to make the list complete.

1.7. Switches

Switches are used to control the environment in a number of different situations that are often confused with each other. Customer applications should have access to all of these wonderful facilities.

Taste — Different users have different taste about how they want the system to behave. User tastes change relatively slowly.

Parameter — The specific purpose for which this operation is being done requires different behavior than the default.

Declaration — Characteristics of operations that are expected to be constant over the related set of objects, *e.g.*, target key.

- Data_Error
  - Output Value; attempt to write a value not in type
  - Syntax Error; input value had incorrect syntax
  - Type Error; attempt to write an unsafe type
- Device_Error[7]
  - File Gone; reading file with no lock and it disappears
- End_Error
- Layout_Error
- Mode_Error
- Name_Error
  - Ambiguous
  - Directory Not Found
  - Malformed Name
  - Object Not Found
  - Version Not Found
- Status_Error
  - Already Open
  - Not Open
- Use_Error
  - Access Error
  - Check_Out Error; object not checked out[8]
  - Class Error; existing object is different class
  - Frozen
  - Lock Error

Figure 1-1: Flavored IO Exceptions

[7] This could be a very long list, even though the exception is not frequently seen.

[8] Different from Lock Error?

1.7.1. Switch Hierarchy

All of the following are possible switch setting *scopes*:

System	An installation wants different defaults from what we have written into the code.
View	Operations on objects in the view.
Directory	Operations on object in a sub-directory of a view.
User	Specific user tastes.
Account	Specific user taste tailored to a specific purpose.
Session	Changes to the characteristics of the session while it is logged in that are not saved as new Account defaults.
Job	A specific, related set of operations.
Procedure	Passed as a parameter to a particular command or tool procedure.

The scopes are presented in order of increasing specificity. More general scopes override less-specific ones. Procedure, job, session, account, and user are one related group; directory and view another; system is clearly the least specific. Ordering is completed by making user more specific than directory. Not all situations require supporting all scopes, but it needs to be clear which switch value has precedence if there are many to choose from.

Some switches can only be set on a view or directory basis, *e.g.*, target key.

Providing a hierarchy of switches implies a value for each switch corresponding to "don't care". This must be maintained individually for each switch and complicates otherwise simple situations, *e.g.*, Boolean switches.

1.7.2. Switch Parameters

One of the basic problems with switches is that, though they provide flexibility, they also introduce complexity. One of the issues, addressed by the current profile package, is that it isn't reasonable to have one parameter for each possible switch. This is particularly true when commands are composed of other commands or tools with their own switches, possibly requiring the union of the individual switches. This is particularly noticeable because the number of parameters and their types must be pre-specified for Ada procedures.

Some flexibility is added by the ability to push/pop switch state around a particular function, essentially implementing something similar to parameters, but with a different syntax. This is limited by the depth of stack provided, not to mention inconvenience in typing extra calls into command windows.

The issue of how to specify switches is integrally tied up in the challenge of making Ada serve as a convenient command language for normal, trivial user requests. See section 6.3 for more details.

1.7.3. Switch Editor Interface

There a large number of switches in the current system; this number is likely to grow considerably. For users to be able to effectively deal with the switch explosion, some more structure must be imposed. In particular, the switches need to be organized by user-perceived functionality. This impacts two different axes of switch management: switch names/grouping, and the ability to see the switches that a program would see, regardless of the level at which they were specified.

Switches should to be grouped by purpose, rather than by implementation client. Initial display in the switch object editor would show the major categories. This could be kept to a set of manageable size and expanded to provide more detail. All switches should take effect when they are committed. Since the most general form of resolution may be too expensive, there should be some form of notification to persistent environment when switches change.

2. Directory Structure

2.1. Simple Objects

The Epsilon Directory system consists of Objects of distinct classes. Among these classes are the Ada units class, the files class, the directory class, which includes both Worlds and Directories, and the view class. Other classes exist (as in Gamma), but for the purposes of this discussion they have the same properties as files.

The Epsilon Directory system is a hierarchy of Worlds and Directories. Worlds and Directories may contain any object, including other Worlds and Directories. Worlds and Directories have one part. Each object immediately within a Directory or World must have a simple name distinct from the simple names of other objects immediately within the same Directory or World.

2.1.1. Worlds

A World is associated with a disk volume (VPID) and is stored on that unit. Other objects in the directory system are stored on the volume associated with the closest containing World.

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. A world contains objects, each of which can have multiple versions. An object cannot be in more than one world. Within a world, the simple names of Ada compilation units must be unique. An Ada unit can **with** any spec in its own world. However, no units in other worlds are visible without taking specific action.

2.1.2. Views

During the life of a world, the various objects that it contains will change, new objects will appear, objects will disappear. Versions allow the user to capture the state of an individual object at a particular time; multiple versions will exist over time. For most purposes, however, it is most natural to think of a particular version of an object as representing the object. A *view* is a set of versions that, taken together, represent a world at a particular time. For worlds with exactly one view, the result is the same as for a conventional directory system: each object has one version that is easily accessible. For worlds with multiple views, each view gives the illusion of representing the versions of the objects that are accessible.

Though views and worlds are very different from an implementation perspective, a view will appear to be the same thing as its associated world in the same way that a version appears to be the same thing as the object it represents. In normal system interactions, the goal is for the multiplicity of views to impinge on the users' consciousness as seldom

as possible, and only in contexts where it is necessary to understand the relationship between distinct versions of the same object(s).

A user, working within a world, must have selected a view for that world. 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.

A view can select views of other worlds. When this is 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 one world visible to a specific view of another world.

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 the modification proceeds.

2.1.3. Types Of Views

There are four types of views. These are:

1. Release View

   A *release view* is a frozen 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 will select versions of a number of objects related to the versions of Ada units in the view. It is possible to remove various classes of these to save space; doing so remove the ability to do associated operations:

   1. Diana trees. Compilation, definition, tools, and some debugging.
   2. Image. Viewing.
   3. Code segments. Execution.

4. Debug tables. Debugging.
5. Assembly code, listing files, etc.

Removing an image without removing the Diana tree is not supported; similarly, removing code segments while retaining debug tables makes no sense (at least for Native code). The full set of objects associated with various targets cannot be pre-determined, but similar space saving may be required.

It is not possible to delete a frozen view that is currently referenced by another view[9].

2. Working View

A *working view* is one where the view and the selected versions in the world are not frozen, but all imported views are. This is used when the world is supporting active development; the view selects the versions of objects that can be changed. The Epsilon system enforces consistency between the unfrozen view and all of the selected views. The various frozen views are compatible with each other.

3. Composite View

A *composite view* selects zero or more working views and zero or more frozen views. This type of view allows construction of a view that has no consistency or compatibility properties. The selected working views and frozen views are referred to as *original views*.

This type of view allows the user to build a view that operates as if it were a working view on all of the original worlds. All of the operations that operate on working views accept a composite, and *look through* the composite. The goal is to make a composite view work as if all of the versions represented by the unfrozen original views made up a single working view. The process of compiling all of the version in a composite view may cause new releases to be made of some of the original views and imported into others for compilation to succeed. The composite view is never used directly by compilation, so it need not import the compilation closure.

4. Unmanaged View

An *unmanaged view* is one that selects versions of objects, but provides no guarantees about the relationship between the versions in the world and versions in other worlds. Unmanaged views can only be imported by other unmanaged views. All operations on views are available for unmanaged views, but no attempt is made to enforce consistency or compatibility.

---

[9]Seems that there should be some restriction that makes it impossible to delete Diana trees while the view is in the compilation closure of an unfrozen view, but don't know precisely what that restriction is.

This kind of view is used for worlds that are not used for organized software development, *e.g.*, users' home worlds, system log directories, etc. Consistency and change analysis are enforced within the view. A user's home world is probably this type of view.

The first three types of views are referred to as *managed views*.

It is important to note that when consistency is enforced, it is between the versions selected by a view and the imported views. At no time is consistency enforced between two imported views. However, compatibility is enforced between imported views in working and release views.

2.1.4. The Default View

Every world has an unmanaged view created with the world and 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 some release view; the significance of this operation is explained in the next section.

2.1.5. 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. This view is used in the view stack.

This 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.

2.1.6. Objects and Names

Every object in the world has a simple name. In this case, a *simple name* is a string composed from a set of visible characters[10]. Users are encouraged to use Ada simple names for most purposes and required to use Ada naming for Ada units. Various environment functions will construct non-Ada simple names to represent associated objects that are not customarily visible; these names follow a known pattern so that users can avoid conflicts.

---

[10]Clearly have to restrict use of some punctuation and non-visible characters; don't see any reason to enforce all of Ada naming restrictions. Should be possible to express the restriction in terms of character set, not character order

Remember that all objects exist independent of views; views select versions of objects, not the objects themselves. It is, therefore, possible to resolve a name to an object without requiring a view. It is also possible to reference a version of that object without a view, but this requires an explicit version qualification be specified as part of the name. 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. Because objects exist independent of views, version qualification is only necessary (or possible) in reference to the version itself, not its parents in the directory system[11].

The existence of objects independent of views implies that as long as there exists a version for an object, that object logically appears in the name space of all views for that world. This will require care in name resolution specification; users will more often be interested in whether there is a version with a particular name in a view than whether there is an object in the world. Ideally, the implementation should make it possible to ignore the existence of objects for which there are no visible versions. If the class of a version is inherited from its object (*i.e.*, all versions have the same class), this will not be possible, leading to considerable confusion. A more subtle restriction is imposed by the existence of objects that have only one version; their existence would seem to be harder to overcome in another view.

2.1.7. Directories

Directories are objects used by naming to logically structure a name space. They have no affect on Ada naming within a world, but do affect qualified names for objects and versions. Ideally, directories would be entirely a naming construct, with no other side-effects. There is no usefulness in their having versions, but an implementation that prohibits versions of directories may also feel constrained to prohibit re-use of the name of a directory in one view as long as another view of the world still references the directory. This is an unfortunate, but not fatal, property.

2.1.8. Libraries

Worlds consist of objects. Views select versions of objects. A ***library*** is a world seen through a view. As such, a library corresponds to the LRM concept of a library, at least in the relationship of the Ada units represented therein. Although libraries don't directly correspond to any single implementation object[12], they do correspond to what the user actually sees under normal circumstances. Only part of the objects in most worlds are visible (for worlds with a single view, the world and its library are identical). A view

---

[11] There is an issue with specifying the code segment associated with X'V(27). It has to be X'Code'V(F(X'V(27))), instead of X'V(27)'Code; fortunately, this is not the most common form of access

[12] As a result, libraries don't have convenient names.

represents objects in more than one world, so it isn't strictly precise to speak as if views were to worlds as versions are to objects.

2.2. Naming

The syntax for names is assumed to be the same as for Gamma, except where noted. Although the interpretation of names changes and there are some additional features, there don't seem to be any Gamma naming features that become obsolete in Epsilon.

2.2.1. Naming Object and Versions

A name is used to find an object, and the result of this process is an *object handle*[13]. 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.

2.2.2. Object Handles

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. 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.

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. The evaluation context for the view parameter to 'V is the

[13] This seems to be a central user concept with an ugly name. Need to do something about that.

context of the object being named, *i.e.*, the default naming context for selecting the view is the same as for the object[14]. For example,

```
!World.Directory.Object'V(View)
```

is equivalent to

```
!World.Directory.Object'V(!World.Directory.Object.View)
```

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 view is selected. The name

   ```
   !World.Directory.Object'V(!World.Path)
   ```

   is an example of this form. The rules for determining the context of the path parameter to 'V is the same as for views.

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.

2.3. Invisible Objects

There are lots of new objects that will have names. Need places to put them, naming conventions, etc.

Relation between names and when they are seen, *e.g.*, subunits may or may not be visible in LOE, code segments and listings, user-created subordinate objects.

2.4. Questions

Questions about views.

1. What do they really look like?
2. What are the naming conventions?
3. How do I think about them?
4. How do I deal with multiplicity, disambiguation, mismatches?

---

[14]Since the class of objects that can appear in the parameter is limited, it may be appropriate for view names to have the same flat namespace characteristics that Ada units have.

5. How do I deal with invariants; how are they enforced?
6. How to assure that simple things are simple?

3. Ada

3.1. Basic Principles

The Epsilon system supports a more flexible model of compilation, obsolescence and execution than that supported in Gamma. To a large extent, the system's notion of "state" is understood at the granularity of declarations and statements. Within an Ada unit, Diana trees as small as declarations and statements may change state independently. The recognized incremental states are *unparsed* (text), *parsed*, *semanticized*, and *phase-1 coded* (code attributes assigned)[15]. Consequently, the operations of assigning coding attributes, editing, semanticizing, and obsolescing can all be performed on individual declarations and statements.

In contrast, the user deals with full Ada units. The smaller granularity understood by the system is used only to enhance performance. The user edits an Ada unit, but the editor determines which declarations and statements have actually changed. The user promotes a unit, but the semanticist needs to look only at the trees that have changed (and any trees in the scope of changed declarations). Change analysis determines how the changes to declarations in the unit affect referencing statements and declarations outside the unit.

The *presentation state* of a unit, the state of a unit presented to users, is a function of the state of all of its components and the state of all units in its **with**-closure. The possible presentation states are: coded, installed, and source. For example, a coded unit, ***Foo***, will be demoted to source when the user changes a statement or declaration of ***Foo*** (but not white space or a comment), or when one of the units that ***Foo*** **with**'s is edited, or when a unit ***Foo*** **with**'s is potentially obsolete because of changes to units it **with**'s. In the latter cases, the system may be able to re-install ***Foo*** simply by re-installing its **with**'ed units and verifying that changes made to them have no impact on ***Foo*** after all.

Editing a unit involves opening the unit for update, making changes to the unit, and then committing those changes. Note that pieces of the unit are not selected for incremental changes; separate windows are not created. Rather, the unit is edited "in place."

See section 5.2 for more details on object editing and 5.3 for more information on the interactions between editing and promotion.

---

[15] These are the states for the R1000 target; other targets will have corresponding states for the first three, but may have more or fewer coding states. For the R1000 target, a unit must still be *coded* as a whole; this is a characteristic of this code generator implementation, not a fundamental restriction of the model).

3.1.1. Compatibility, Consistency, and Completeness

The term *consistency* refers to Ada naming consistency, and for emphasis is often referred to a *Ada-consistency*. Thus, a set of units are consistent if compiling them together would produce no semantic errors. Naming consistency is constantly enforced within an Ada unit and between Ada units in the same library. A change in an Ada unit that would affect name resolution in that unit or its importing units causes immediate obsolescence and subsequent recompilation of the affected units.

The system enforces naming consistency between different libraries only during specific, user-initiated operations: when a world is first imported or when that import is subsequently updated. Even though two libraries are inconsistent, programs built from them may still be executed if the weaker notion of *compatibility* is satisfied. Compatibility ignores name resolution conflicts that might have been introduced during incremental changes to a unit.

For a unit to be a legal Ada unit (as specified by the LRM) it must be complete as well as consistent. *Completeness* is required only at the time a program is to be loaded. To make incremental compilation a more natural process, the system allows Ada units and libraries to be incomplete. A procedure spec may appear in a package spec even though there is no body for the procedure in the body of the package. There might not even be a body for the package in the library. There is command/editor support for computing the completeness of a unit of set of units. This facility includes the ability to detect non-terminal prompts that would lead to execution errors, even though executing such prompts causes runtime exceptions or debugger action.

3.2. Compilation

One of the areas of continuing interest is how compilation is managed. The goal for Epsilon is to make it possible for compilation to take place in the most convenient manner for the user without making major distinctions between editor and batch compilation facilities. As described in 1.4.1, assume that there is a progress object defined for a view[16]. There is a variety of information that could be of interest from a successful compilation: when compiled, by whom, time required, etc. All of this could be saved in some form, but for the current purpose, the focus is on the work to be done, not what has been done.

To represent the work to be done, the units that need to be compiled are added to the progress object based on the initial compilation request, regardless of source. As each unit is successfully compiled, its entry is updated; either removed or set to the next level of compilation, *e.g.*, code generation. When the entire compilation is complete, the

[16] Issues of concurrency with multiple compilations in the same view and compilations for different purposes.

object represents work remaining. Since error messages from compilation are attached to the Ada object as part of its permanent representation, this object serves as an index to the units with errors. It would be possible from this object and the Ada units it references to generate a conventional batch log; this would be the expected result from reading the object with Text_IO (form parameters or switches could be used to indicate whether this is to be a list of units, a compilation log, or a set of program listings with interlinear errors). Units that are compiled that obsolesce other units can either be added to the set of units to be compiled in this run or added as units requiring processing in a later run.

Separate objects of the type could be created for a particular purpose and act as the specification of work to be completed. This would be particularly useful for creating a display of the objects that were demoted by an incompatible spec change.

3.2.1. Command Windows

Command windows in Epsilon are essentially the same as in Gamma.

Command windows are Ada declare blocks. The Ada library units that are visible to the block are determined by the search list defined for the session. As in Gamma, the *search list* is an ordered list of libraries. The semanticist searches each library in order until an Ada library unit is found whose simple name matches the symbol being resolved. For these purposes, an Ada unit is only "found" if there is a version of the unit visible through the appropriate view of the world. Each entry in the search list indicates whether the link pack usually associated with the library is to be searched for the desired Ada unit.

All names in the command window must be Ada names, unless they are enclosed in string quotes. Command processing, probably in the Command OE, should be made to understand the naming characteristics and provide the appropriate quotes necessary to smooth the interface.

4. Execution

4.1. Accounts and Sessions

Gamma sessions are static objects that represent the both static and dynamic characteristics of user interaction with the system. Gamma enforces that there is at most one active session for each account, but confuses jobs from different logins if they are from the same account. To deal with the problems introduced with this dichotomy, Epsilon supports two related concepts: *account* and *session*.

Account — Users log into accounts in the same way they log into Gamma sessions. Multiple accounts for a user can be used to tailor the environment for a particular purpose and/or to provide differentiated accounting information. A user can log into the same account more than once at the same time.

Session — The name for the active entity that controls user interaction for jobs started between the login and completion of the last job. A session is uniquely attached to a particular account; its lifetime is from login until the termination of the last job started from the session.

The login process consists of entering a user name with optional account and a password[17].

A session can be suspended and later restarted on the same terminal type. A suspended session is disconnected from its terminal port, but is otherwise active, *i.e.*, jobs running on behalf of the session continue to run, can write output to windows, etc. Jobs started by a session can continue to execute after the session has completed[18].

4.2. Jobs

A job is the unit of system execution, corresponding to a command, a system component or a persistently elaborated set of subsystems. A job inherits state from the session that spawns it. Facilities for starting jobs allow specification of context information. Other than this context information, the new job behaves as a child of the spawning session, **not** the specific job doing the spawn.

State for the job consists of two parts, session state and job state. The session state lives from job to job; the job state is used by the job as temporary storage. A job can modify either state; changes to the session state live on after the job, and subsequent jobs see the changes.

---

[17] Seems no need to provide different passwords for different accounts for the same user.

[18] Images and files will have the same representation, so output to a window can have its image made permanent without requiring the session to be active during the output.

The session state is copied into the job state when a job is initiated, and is not changed if other jobs modify the session state while the jobs is running. However, modifications to the session state by the running jobs are reflected in the session state immediately, and jobs started after the change see the change, even if the original job is still running.

Context information consists of:

- Directory context
- View
- Standard Input/Output/Error files[19]
- Switches[20]

The assumption is that successive commands issued without disconnecting will re-use the same job. There should be a way of specifying resource allocation and limits for jobs, either before they are started or while they are running, assuming that the particular parameter can be changed. What volume the job is to use for storage is not changeable once the job starts running. Jobs started with **Run_Job** can specify a volume. There exists a command to discard the current job for future command execution.

4.3. Persistent Elaboration

Persistent elaboration is intended as system-implementation feature and is not an "advertised" feature of the environment. There will be interfaces that we provide that may require persistent elaboration to work (*e.g.*, object editors and target code generators), but the facility is not *per se* presented as an interesting feature. Additional functionality required to support internal uses of persistent elaboration are not discussed.

There is a way of specifying the subsystems to be elaborated, what job they are to be elaborated in, and other important characteristics. There is a way of making it so that a user does or doesn't get a particular instance. For subsystems that are elaborated as part of system initialization, users who do nothing will execute this elaboration. Persistent elaboration is explicitly requested; it is never the side-effect of some other execution.

There is a way to specify that a subsystem must be persistently elaborated to be executed. Users controlling the elaboration process are responsible for maintaining any uniqueness invariants expected, but not enforced, by the subsystem.

An explicit elaboration request must specify which job the elaboration is to take place in. The job specified can already have persistently elaboration state, in which case all of the elaborations share the same lifetime, except that unelaboration can be accomplished

---

[19] Job files are represented by a stack of their own.

[20] These switches override system-wide switch state and must be copied after session state.

according to a stack discipline. Persistent elaboration and normal job execution cannot coexist in the same job.

It is always possible to determine which view and which elaboration of the view would be executed for a particular purpose. Elaboration instances must be named in a manner natural for users.

If debugging is possible in persistently elaborated code, facilities are provided to allow the elaboration being executed to be correctly debugged. If user debugging of such code is not possible, the debugger must know this also.

4.4. Batch

Jobs and the session they are associated with can run independently. This accomplishes one of the purposes of a traditional batch processing system, but not all. A batch system would need to have the basic characteristics of traditional batch system and support for normal environment mechanism:

1. Full Ada Support

   The preparation of the unit to be executed should be done with the standard Ada editing facilities. This means that the batch queue entries have enough information to be executed: code segments, context, etc. All necessary job state is available for execution and user examination (including program text). Runs just like a disconnected job except for queueing characteristics and files.

   One ramification of this is that jobs cannot be submitted to execute programs that cannot be loaded. This can be overcome by submiting a program.run of the string to be executed. Incorporation of a facility to execute strings is a possible extension, but not a possible substitution.

2. Access to Files

   Ability to specify input, output and error files. If not specified, these do **not** default to the current terminal session, but to appropriate log files.

3. Queue Permanence

   As with the print spooler, batch queues must be able to live through crashes and restart effectively. Jobs in progress at the time of a crash have the option of being restarted.

4. Resource Limits

   Ability to limit CPU utilization to keep jobs from running away while unattended. Ability to set execution priority, volume and disk limits.

5. Job Slots

   A batch request that has not yet begun execution must **not** be allocated a job. Doing so places limits on the size of the batch queue and other system resources that are undesirable.

6. Flexible Scheduling

   Scheduling facilities should include those provided by competitive batch systems. These would include:

   1. Before or after a particular time, interval or job.
   2. Periodically.
   3. Equivalence class (priority, volume, resource limit, etc).
   4. Load limit.
   5. Pre-emption characteristics. Allow another job to ***play through***.
   6. Running jobs. Suspend job and place in batch queue.

To the degree possible, there should be a single queueing mechanism that works for printing, system daemons, and user batch jobs. The contents of the queues will be different, but most of the criteria are shared.

4.5. Debugging

Users will be debugging programs that run on the host and programs that execute on targets. The interfaces to accomplish this should be as similar as the implementation allows. This could be accomplished by a general (native) debugger interface and target specific operations packages.

The initiation of execution and debugging need not be coincident. It should always be possible to start debugging a program after it has started running.

Various code generators and other tools will be available that alter the ability of the debugger to provide full service, *e.g.*, no Diana trees or debug tables. Under these circumstances, the debugger must be given enough information to know what is possible for it to do.

The interface for the native debugger and target debuggers should be essentially the same, much the same way editor operations are the same across different object types. This will require provision of a way of selecting target-specific operations based on the target.

4.5.1. Views, Execution, and Changes

When executing a program loaded from a particular view (*i.e.*, calls from command windows to procedures in the world), the effect should be as close as possible to having

the view for execution correspond to that the user had when the program started execution. As long as none of the units has changed, this is straightforward. When a unit is changed, the program that has been modified and the one that is being executed are now different. Since there are two active referencers, there are two distinct versions (the second created by changing the program being executed). Any debugger references to differentiated units will refer to the pre-change version; further changes will have to be made to the changed version. Sufficient information to describe the views involved should retained to support this. Ideally, selections in the changed version would be mapped into the corresponding debug version references, where possible; attempts to edit the debug version should end up in the changed version.

The correspondence between a program that is being debugged and an open view can be of interest even if the connection is not as close as the one described above. A general facility could look on the view stack to find the topmost modifiable view for a world, even if there is a frozen view on the stack above it. This facility removes some of the urgency to completely disambiguate the view being used to reference a version reachable from two different views.

The issue of execution retaining a view so that versions can be retained for debugging is also interesting for code that is being executed on targets. The debugging issues are the same, *i.e.*, access to Diana trees, setting breaks, changing code, etc. There is the additional problem that the notion of when the program is executing is less controlled by the environment, so the creation of temporary view corresponding to those being debugged is less automatic. Exactly how this is handled depends somewhat on the degree of tool integration (source vs. code downloading). The issues are similar to those in doing cross-debugging from the MV with unfrozen worlds.

4.5.2. Execution in Context

Current command windows allow the general execution of Ada programs in static contexts. This facility should be extended to allow execution of similar blocks in dynamic contexts. In addition to the technical issues in making this happen, there are a number of interface issues involving visibility and the level of access to environment functions in dynamic contexts. Consider, for example, the relation between a normal command window and one for executing code in the context of a program running on a different target.

5. Editor Interface

5.1. Core Editor

The Core Editor provides character editing facilities. This section is intended to explain areas of core editor functionality that should be improved. No importance is attached to the order in which the following are presented.

1. Undo/Redo should be implemented in the core editor. Appropriate granularity is line-by-line, with some operations modifying multiple lines. For read/write object editors, this can be accomplished by reporting changes as if the user had gone back and made the changes. No attempt is made to reverse side-effects, *i.e.*, if a change obsolesces units, undoing it doesn't un-obsolesce them, except by running the compiler.

2. Marks should be made to retain their position on the same logical position in the image, rather than the same absolute line/column position.

3. Keyboard macros should be extended to have names, images, help, etc.

4. Key mapping should be implemented to allow any user function to be mapped to a key without running the compiler. Such a facility should provide improved support for changing keymaps.

5. A programmatic interface with access to the appropriate window, cursor, argument, etc. values should be implemented.

6. Window managment should be redesigned without overlapping windows, but with improved real estate management and replacement policies.

7. Elision should be implemented as a core editor notion, allowing object editors that need to do so to deal with complete images. There are a number of design issues with how this can be done to allow appropriate representation of the elided text while retaining core editor control. Some object editors will continue to perform elision as they do now.

8. Object and text selection mechanisms are to be unified so that there is only one form of selection and appropriate interfaces for extracting the appropriate object structure to represent objects or lists of objects where the distinction is important.

5.2. Object Editors

The object editor provides the transformations between the object and its image.

5.2.1. Permanence

An object is represented in the editor by an image. For images that the user can change, the image represents the prospective contents of the object. When the image is committed (or otherwise made permanent), the object and the contents of the image are made to coincide with the image; operations exist for making the image correspond to the previous contents of the object. The explicit commit is desirable and should be observed where possible. The only known exception is libraries, where the image is the table of contents of another object and changes are not made directly. Other editors could have similar models.

5.2.2. Common Operations

One of the useful characteristics of object editors is that there are a common set of supported operations. This is particularly useful if all of them are supported in a similar manner.

The following is an attempt at a standard definition for each. Some of the operations are new; all of the operations currently nested in Object will be directly in Common. The object underlying the image is discussed in terms of the image to reduce confusion with selected subobjects, which are referred to as the selection.

5.2.2.1. Selection Operations

The following operations are used to create or modify the current selection on the basis of object structure characteristics.

Add_Next — Augment the current selection by joining it with the object that would be selected by Next. The position of the cursor is unchanged.

Add_Previous — Augment the current selection by joining it with the object that would be selected by Previous. The position of the cursor is unchanged.

Child — Select the largest strictly contained subobject of the current selection; a selection always results. The position of the cursor is unchanged.

Next — Select the object that is the immediate successor of the current object. If there is no current selection, make a selection using the current cursor position, then apply the rule for Next. The cursor will be on the first position of the new selection.

Parent — Select the next largest selectable object containing the current selection. The selection should grow visibly for each application until the entire image is selected. The position of the cursor is unchanged.

Previous — Select the object that is the immediate predecessor of the current object. If there is no current selection, make a selection using the current cursor position, then apply the rule for Previous. The cursor will be on the first position of the new selection.

5.2.2.2. Cut and Paste

Operations for performing normal cut, paste and relocation operations. For editors where an image represents an object (*e.g.*, Ada or Text), these correspond to comparable Region operations and have the same interactions with the Hold stack, etc. For editors where an image represents a collection of objects (*e.g.*, Library), these operations are convenient shorthand for the corresponding simple operations.

Copy — Copy the selected object(s) to the current cursor position. Selections are interpreted to include any objects for which significant characters are selected.

Delete — Delete the selected object(s).

Move — Move the selected object(s) to the current cursor position. Selections are interpreted to include any objects for which significant characters are selected.

5.2.2.3. Ada Operations

It is possible to reference Ada objects through a number of different object editors. It is desirable to perform basic compilation operations from each of these. As such, the following are "common" in the sense that they are operations that are (at least potentially) provided by all object editors, not because they apply to the range of objects edited. Some additional detail is provided in section 5.3. All of these operations will attempt to format the image if that is necessary for their operation.

Note that there is no Demote operation. The Epsilon obsolescence model and editor changes make it superfluous.

Code — Bring the unit to coded. Includes installing and coding the closure of units that would be required to execute the designated unit, as necessary and possible. Coding of the closure is abandoned if the closure is unable to install.

Install — Bring the unit to installed. Includes installing corresponding visible part, parent units and **with**ed units, as necessary and possible.

Semanticize Determine whether the current image or selection is semantically consistent. Doesn't attempt installation of closure. If successful, the image can still be edited and subsequent installation will require no work if no changes intervene.

Install and code make the current changes to the image permanent, even if the operation is unsuccessful; *i.e.*, they express an intent to make the results of the operation permanent that isn't implied by semanticize.

5.2.2.4. Image/Object Relationship

For most editors, there is an object and an image of the object. These operations control the correspondence between the image, its object, and the locks that are available. These operations are changed from Gamma. Part of the change is due to changes in the underlying implementation; part are an attempt to separate image and window issues. Each of the following has a parameter to indicate the disposition of the window, so key bindings similar to the current ones could be retained.

Close Make the contents of the current image permanent, make the image read-only, and release locks on the object.

Commit Make the contents of the current image permanent, but leave it writeable.

Edit Acquire a write lock on the current image and to the underlying object.

New_Version
The same as edit, except that a new underlying version is forced. If changes have been made since the last commit, they become part of the new version, *i.e.*, the new version is created from the permanent version without changing the image.

Promote The operation corresponding to the most frequently requested major commit/promote/execute operation for the particular object type. For Ada, corresponds to Install or Code, depending on the current state to the object (though the closure promote is the Install closure, regardless).

Revert Make the object and the image consistent by restoring the last permanent version to the editor image.

Write_File Write the image of the object to the named file. The image adopts the name of the file **only** if it was previously unnamed.

5.2.2.5. Traversal

Definition — Move the cursor to the object referenced by the current position of the cursor or the selection, creating an image on the object if not part of the current one. Any new image that results is read only. An editor is obliged to make its best effort to have definition succeed. In general, this means extracting string names from the current image, making use of whatever information is available from the context.

Enclosing — Bring up the image of the object containing the current image; typically the directory or Ada unit containing the current image.

Implementation — The same as definition, but go to the implementation (body) of the object named.

5.2.2.6. Miscellaneous

The remainder of the operations. The careful observer will note the absence of Undo and Redo, which have been moved into the core editor, and Insert, which is no longer required. The core editor will be principally responsible for elision and expansion, but there will continue to be an type-dependent component of these operations.

Complete — Provide assistance in making the current selection or the object associated with the cursor. Provide prompts and/or values required to make it possible for the user to successfully promote the image. As possible, should provide successively more complete information as applied to the same location. An advanced form of completion would provide prompt values that are based on additional information (*e.g.*, function calls referencing user state). Where more than one completion is possible, there should be a mechanism for cycling through the possible values. Implies format.

Command — Create/goto a command window associated with the window associated with the current image. Context of the command is the same as that for the apparent context of the image[21]. Appropriate use clauses are added, as necessary, to provide visibility to object-type-specific operations.

Elide — Reduce the level of displayed detail.

Expand — Reduce the level of displayed detail.

Explain — Provide additional information that may be of interest to the user. Most frequently used to display explanations for error designations. The notion

---

[21] This interacts with how text output windows are presented.

of error designation must be expanded to include unsuccessful completions and designations added by user tools.

Format — Make the image syntactically consistent and pretty.

New_Line — Insert a new line at the current position. Used to allow object editors access to the return key for formatting, etc. If used as a commit key surrogate, should be switch-selectable. The default operation is to insert a line an position cursor on the new (empty) line at the appropriate indentation. For read-only editors, goes to the beginning of the next line of the image.

5.3. Interaction Protocols

The process of editing an image involves a number of transitions for both the image and the object that underlies it. Making these interactions convenient an predictable has material impact on how easy the system is to use. The table below attempts to capture these relationships[22].

Although the new Ada editing model provides considerable additional performance and facility, the additional performance is somewhat unpredictable without experience and an understanding of the changes made; taking a unit from source to coded one time may be much faster than going from source to installed another time, due to the impact of changes made. After the user community has become used to it being much faster, they will begin to be bothered by their inability to predict the performance of operations. Some thought needs to be given to helping providing predictors for the time required.

[22]The table is too complicated, as modify always seems to imply write lock and no-modify implies supersedable read. This will be fixed in a later release of the table.

	Modify		Lock		State	Closure
	work	*fail*	*work*	*fail*	*work*	
Definition	No	No	Read	Read		
Edit	Yes	Yes	Write	Write		
New Version	Yes	Yes	Write	Write		
Close	No	No	none	none		
Commit	Yes	Yes	*note 1*	*note 1*		
Semanticize	*nc*	Yes	*note 1*	Write	*note 2*	none
Install	No	*nc*	Read	*nc*	Installed	**with**
Code	No	*nc*	Read	*nc*	Coded	Execute

Legend

Read	Supersedable read.
nc	Not changed by this operation.
with	Install **with** closure of the unit.
Execute	Code execution closure of the unit.
note 1	Read lock upgraded to Write by modification.
note 2	Unit is installed; Write lock demotes to source.

Table 5-1: Impact of Operations on Locks and Unit State

6. Issues Requiring More Thought

This chapter consists of random notes that are in even more primitive state than the preceding.

6.1. Error handling

Editor puts in message window and/or highlight specific locations. Commands print messages through a standard filter that controls appearance and destination (Log package or similar). Tools return error status and provide a method for acquiring additional detail. IO packages raise standard exceptions, but leave traces in known place for discovering the underlying cause. Any error handling package (e.g. error_reporting) should be available to users. See 1.6.

6.2. Input/Output

6.2.1. Error Messages

There are currently 3 "known reasonable" places to have program output go, all of which have problems:

1. Text window. Uses up a window; takes up screen band-width while running; goes away when the session terminates (causing job to fail if still running).

2. Message window. Limited space. Can be scrolled away without being noticed; not in the line of sight; appears out of context from background job.

3. Log file. Has to have a name; fills up directory; outside of normal consciousness.

In addition, there is the issue of amount of output desired. Current default is clearly excessive; need a simple statement of what "should" be printed by normal commands. Problems include different expectations for large batch jobs and for interactive commands (which aren't easily distinguished), and client differences (I may want to see detail of compilation, but not of moving files). Not enough control over "associated" messages; *e.g.*, I want to see position messages when I get errors, but I don't want position messages for the warnings that I don't see.

The use of progress objects helps this problem by reducing the volume of output that comes from operations that are expected to have interesting error patterns. Not a complete solution.

6.2.2. Interactive IO

The normal Text_IO-style IO provided in an environment window has a number of attractive features, but also some deficiencies relative to conventional systems.

1. Page mode or equivalent.
2. Background jobs and window scrolling.
3. Ability to stop output.
4. Single character input; *i.e.*, without commit.
5. Loss of position due to edit of already processed input.
6. Inability to access (copy, select) output text.
7. Organization/tracking of output from commands.

6.3. Ada Command Interface

There are a number of challenges remaining to make Ada the ideal command interface. Some of the things that make Ada most attractive as a specification and programming language make it cumbersome to type for simple commands. Programs are designed, commented and documented; the contents of a command window usually aren't[23]. On the other hand, many of the problems that the current command interface has are problems with the command interface, not with the command language. Many of our customers believe that command windows provide power and unity to the interface and would like to see their functionality extended (*e.g.*, execution in context)[24].

The future success of commands hinges on general improvements in the performance of what is in Gamma, additional mechanism to provide creature comforts and improve ease of use, and changes to the command set to make it work well with the mechanisms that we do support. Many of the current problems with commands are the result of assumptions that functionality would be available (possible) that isn't.

This section requires additional work. It is organized as a set of problems that need to be solved with notes about possible solutions.

---

[23] Despite problem reports about comments disappearing in command windows.

[24] Significantly fewer resources might be required to provide a simple command interpreter. This doesn't seem to be the right path, however. It would add a new mode to the system that requires documentation, training and explanation, not to mention additional clutter to the interface. It also reduces positive product differentiation. Designing and implementing such an interface is left as an exercise for the reader and is ignored in this discussion.

6.3.1. Overhead

1. Creating Command Windows

   Bringing up a command window is expensive. To create a standard command window, requires approximately 200 characters to be transmitted to the screen; 40 to turn the running flag on and off again, 120 to redraw the banner, and 40 to draw the initial contents of the window. Without allowing for any processing, this means that 0.2 seconds is the fastest that a command window can come up[25]. Deleting a command window takes between 160 (banner and running flag) and 320 (two lines of window) and causes the system to lose the command undo/redo script for the window. Leaving command windows on each object window cost from 10 to 20 percent of the screen and slows inter-window cursor movement.

   Redrawing banners could be fixed by rewriting the terminal controller. Turning the running flag on and off could be reduced by making the indicator shorter, by not having it flicker on and off (which requires that there be no perceptible delay), or by making it a function of the terminal. Command history should be unrelated to whether the command window has been deleted.

2. Using Command Windows

   Committing a command window involves parsing the command, pretty-printing it, highlighting the result, semanticizing the contents and either interpreting or code-generating the result. This takes a fair amount of time. A major part of the time is spent in the pretty-print and redraw, since this typically involves minor spacing, capitalization, and punctuation changes and computing how to redraw them. Using the Epsilon model, where formatting is optional, some of this be reduced. The requirements are that unacceptable commands are retained in the command window and that the text of successful commands be available for modification or re-execution. For most purposes, making completion automatic and formatting optional would make commands more convenient.

3. Syntax

   The typical command is a procedure call with one or more parameters (most of the ones with no parameter are already on keys). Having to type the leading parenthesis and quote adds some psychological cost to the operation. If more than one parameter is required, quotes have to match and commas have to be provided.

---

[25] This may not seem long, but if the 2060 waits that long to return a prompt, people assume it is heavily loaded; it also exceeds customer standards for "trivial interactions".

Part of this will be solved by changes to the parser that make it possible to do better recognition and correction of such simple structures; there are not many (*i.e.*, I know of none) places in Ada where two identifiers appear adjacently. By appropriate assumption, then, a series of tokens without punctuation could be assumed to represent a procedure call and its parameters.

Quoted strings are a little more of a challenge. The most common form of actual provided for string formals is a literal, where leaving off the quotes makes it theoretically difficult to determine whether a literal or an expression was intended. In practice, the number of nullary functions that return strings is small and only infrequently clashes with strings the user wishes to supply[26].

A user typing:

```
copy a b
```

should be able to reasonably expect that it means the same thing as

```
Copy ("A", "B");
```

6.3.2. Names

Providing a complete Ada name for a simple operation generally requires a package.procedure reference. This leads to longer names than those used by conventional systems, but provides a better structuring mechanism and helps to compartmentalize a relatively broad interface. The solution to this is to provide short names for the longer ones. Prefix recognition (or other stylized abbreviation schemes) are usually provided, but most are limited to a known set of names[27].

1. Abbreviations

   Most command shell abbreviation systems are special cases of the word abbreviations provided by Emacs and other editors. Since command entry is done from the editor, we can provide these mechanisms by implementing such a facility and making it sensitive to the command environment that it will be used in. Details that need to be resolved include: trigger key, specification method for abbreviations, context-sensitivity (different abbreviations for different situations, *e.g.*, packages and procedures).

2. Fine-grain completion of names

---

[26] If a string function existed, it should be used; correct entries shouldn't have their semantics changed.

[27] Just try to name a command the prefix of one of AOS's builtin commands.

There needs to be a completion facility that works on the name under the cursor and completes just that name or that name segment. This isn't a replacement for the more comprehensive completion that we now provide, but would be used much more frequently.

3. Completion for String Names
   Whatever fine-grain completion facilities are provided should be able to work on object names as they will appear in commands.

4. Improved recognition for narrow contexts
   Many references in command windows include parameters that are enumeration literals. In these cases, it is almost always that case that the only reasonable context in which to search for the enumeration literals is in the package defining the type. It serves no purpose to verify that the user know how to find the literals (the same would hold for integer parameters with conventional constants, *i.e.*, extensible enuerations). Completion should be able to find the appropriate literals. Similarly, the number of functions returning a package type are usually limited; effort should be made to find the simple name in all of the reasonable places and provide the qualification.

In general, completion needs to be changed to be a program designed to provide useful information on the basis of all available information. Completion and semantics share data structures and Ada, but have very different goals; they need to be separate programs that share service routines.

6.3.3. Command Design

Most commands have some required parameters and optional parameters. Optional parameters overlap in functionality with switches (see Section 1.7). A number of possibilities exist for specifying these options (all of which have been used in the environment). Unfortunately, a good command represents a number of normal Ada procedure calls (as used when programming) and, as a result, tends to have more options[28].

1. Each Option is a Parameter
   This corresponds to the normal Ada programming style, where parameters with default values are used to provide options. Currently, this leads to imposing parameter lists when completion is used. This can be addressed by improving completion to understand required and optional parameters. Using multiple levels of completion, the user need not see all parameters on the first completion. This requires some form of registration of which parameters are to be provided with each level of completion. There would

[28] An unattractive alternative that has been used involves greatly increasing the number of commands.

also have to be support for more flexibility in resolving parameter names and recognizing that the user provided a value for a parameter that appears due to completion. The ability to provide the names for parameter values that can be recognized by their type would also help.

Another drawback of this style is that adding functionality involves a spec change. In concert with the Epsilon view mechanisms, this would work reasonably for commands typed in command windows, but there are no plans to support additional default parameters as an "upward compatible" change.

2. Strings as Switch Parameters

This is the approach described in the LRM for the form parameter. While we will have to support form parameters, making them central to environment functioning is unattractive. Their principal advantage is flexibility; *i.e.*, the Ada spec can remain uncyanged in the face of major functionality changes. First-class support for this sort of facility would require registration of switches, possible values, etc. to provide help and completion. A system-provided switch parsing and recognition package would be required for environment and user programmers to implement the same interface.

A variant of this strategy involves attaching switches to string parameters, typically names. This shares most of the drawbacks above, but doesn't require an additional parameter. This is the closest to conventional switch mechanisms, but requires naming to understand switch syntax.

3. Composite Switch Parameters

This is the approach provided by the **Profile** package in Gamma. It fits into Ada usage patterns and provides compactness, completion support, etc. In Gamma, it is difficult to construct values as part of commands; it is most often used as a switch interface to session state, which is then referenced by the default value.

Better completion facilities could deal with this to a degree. The current completion for a profile parameter is the name of the function that will be invoked to provide the value of the parameter; to be effective at changing that value, the user needs to know what the value is, not how to compute it. Expressing this value is interesting. Ignoring that it is private, it could be displayed as a record aggregate. This would provide all of the values in a form that they could be modified, but shares the problem of the first alternative above: completions could be quite long. For records, selective completion is harder since there is no such thing as a default value for a field in an aggregate. Making the type private makes it possible to add fields without changing the interface, but makes it harder to express the aggregate without adding parameters to the functions that return values of the type.

4. Switches as Separate Procedures

   A common way of dealing with Gamma **Profile**-style switches it to enter a command that changes the value of setting for the job in command(s) preceding the command to be executed. This is cumbersome to type, but relatively easy to understand. In the case of **Profile**, it depends on implicit state in the form of profile switches. A similar method could be used with a switch object, but this would require typing its name twice.

6.3.4. Additional Functionality

In addition to the issues above, there are a number of areas of functionality that are available in one or more other systems that some subset of the user community will miss.

1. Tool Composition

   The ability to compose tools or programs that might work together. Using procedural composition is possible, though it requires that the user type the name of the object being used for the composition in each of the calls that deal with it; this is normal programming practice, but cumbersome for commands. Another form of composition is provided by functions, but without providing command windows that accept (or complete) function calls, this would require each function exist also exist (with the same name and parameters) as the function.

   The most widely copied form of command composition is pipes. The effect of pipes could be modelled by default input and output values that move through a sequence of values with each reference. If each tool opened its input and closed its output, it would be possible to have a conventional value that represented current_input the first time opened and the previous value of current_output for each subsequent usage, etc. This could be made more explicit by procedures whose sole function was supporting such plumbing.

2. File Redirection

   File redirection is available in Gamma, but the full Ada naming used is sufficiently cumbersome that it is seldom used. This could be helped by making the commands to do redirection shorter and featuring them more prominently in the interface. In addition, the environment paradigm (select and operate) strongly suggests that input and output should be redirectable to/from the selected region. Except for issue of naming and the implementation of a selection coupler, this doesn't seem hard. If this facility were available, the ease of file redirection would be more of an issue.

3. General String Processing

   Command languages are usually some form of string-handling language; commands are basically entered as strings. They normally provide easy ways

of doing string substitution for name and parameter abbreviation, facilities for dealing with wildcards, etc. While all of these can be programmed in Ada, none of them are concise enough to use realistically in commands. They can, however, be incorporated into the editor interface and completion.

A possibility for some of these is to provide operations on command windows that specify their input/output, etc. characteristics outside of their parameters. Program.Run_Job provides the ability to set standard_input, etc. outside of the program. Commands could be provided that similarly condition the external environment of a command window, *e.g.*, set standard input to the current selection.

Appendix I
Change History

Reverse chronological list of notable changes in the document. References to section numbers are from the current version.

I.1. Revision 0.8

4.3 Changed persistent elaboration section to reduce user exposure.

Index

List of Figures

List of Tables