```
TTTTTTTTTT     EEEEEEEEEE     RRRRRRRR       MM       MM       SSSSSSSS
TTTTTTTTTT     EEEEEEEEEE     RRRRRRRR       MM       MM       SSSSSSSS
    TT         EE             RR     RR      MMMM   MMMM     SS
    TT         EE             RR     RR      MMMM   MMMM     SS
    TT         EE             RR     RR      MM MM MM MM     SS
    TT         EE             RR     RR      MM  MMM  MM     SS
    TT         EEEEEEEE       RRRRRRRR       MM       MM       SSSSSS
    TT         EEEEEEEE       RRRRRRRR       MM       MM       SSSSSS
    TT         EE             RR   RR        MM       MM             SS
    TT         EE             RR   RR        MM       MM             SS
    TT         EE             RR    RR       MM       MM             SS
    TT         EE             RR    RR       MM       MM             SS
    TT         EEEEEEEEEE     RR     RR      MM       MM     SSSSSSSS
    TT         EEEEEEEEEE     RR     RR      MM       MM     SSSSSSSS


TTTTTTTTTT     XX       XX    TTTTTTTTTT                      7777777777
TTTTTTTTTT     XX       XX    TTTTTTTTTT                      7777777777
    TT         XX       XX        TT                                77
    TT         XX       XX        TT                                77
    TT           XX   XX          TT                                77
    TT           XX   XX          TT                                77
    TT             XX             TT                              77
    TT             XX             TT                              77
    TT           XX   XX          TT                            77
    TT           XX   XX          TT                            77
    TT         XX       XX        TT                          77
    TT         XX       XX        TT                          77
    TT         XX       XX        TT                      77
    TT         XX       XX        TT                      77


*START* Job DESIGN Req #976 for EGB    Date 29-Apr-85  9:37:29 Monitor: //, TOPS
File RM:<SYSTEM.SPEC>TERMS.TXT.70, created: 29-Apr-85  9:32:03
        printed: 29-Apr-85  9:39:57
Job parameters: Request created:29-Apr-85  9:37:27   Page limit:261   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASCI
```

# 1. Active Agents.

## 1.1. Tasks.

The Ada Task is the primitive active agent in the environment. Short term scheduling of processor resources occurs at the task level, and supports an implementation of Ada priorities.

## 1.2. Jobs.

A job is a group of one or more tasks performing some user or environment operation. Each command that is executed is a separate job.

From an implementation point of view, each job corresponds to a R1000 Job VPID. Medium term scheduling occurs at the job level and uses a job priority mechanism which supplements the Ada task priority mechanism.

Processor time limits and swapping disk storage limits are enforced at the job level. Resource limits may be set at job creation and changed thereafter. Default limits are used if no explicit limits are provided.

For both processor and disk limits, there is a warning limit and an absolute limit.

When a job exceeds the warning limit, a warning message is sent to the owning session and to the environment log. The job will be suspended by the medium term scheduler. The user may chose to terminate the job, or to examine the suspended job with the debugger, or to resume the job after increasing resource limits or somehow freeing resources. However, the job may continue to consume resources after it has suspended, in which case it may exceed its absolute limits.

When a job exceeds absolute limits, a message is again sent to the system log and to the owning session. Then the job is terminated.

Setting the warning limit at the absolute limit ensures that the job will never be suspended. However, there is no way to gaurantee that a job will never be terminate because of resource constraints. This means that the construction of robust servers must take into account proper management of processor and disk resources.

## 1.3. Session.

A user logging onto the environment interacts with a particular session. A session is the collection of jobs (including editor jobs and command execution jobs) which serve as the active agents on behalf of the user. A session includes certain permanent information (user profile, etc.). (needs work)

# 2. Managed Objects.

## 2.2.1. Managed Types.

The programming environment provides support for a set of types called managed types. These types build upon standard facilities provided by the environment, are registered with the environment,

and follow protocols established by the environment.

The most important managed types include directories, Ada units, views, files, users, and various devices. Objects of these types are managed objects, and are the only permanent objects in the environment.

2.2. Data and Segments.

The virtual memory system provides the primitive storage mechanism for data on the R1000. The segment is the basic unit of storage in virtual memory. A segment stores up to 2**32 bits of data.

Theoretically, the system can store an essentially unlimited number of segments. However, the amount and organization of physical disk storage constrains the number and size of segments stored in the system. For example, the sum of the data in all segments cannot exceed the storage capacity of the disks in the system (note that every existing segment consumes a minimum of one block).

A segment may be permanent, meaning that it will persist across system crash/shutdown. Managed objects are implemented in terms of permanent segments. Temporary segments are used for temporary files and as heaps for segmented heap access types.

The data in a segment consists of a small set of fixed fields common to all segments and a (potentially large) user data area organized as typed Ada data structures. Access to the user data is achieved (after following the protocols discussed below) by constructing an Ada access value of the appropriate type, which provides a typed handle for manipulating the data.

2.3. Objects.

An object is the basic entity in the system. An object is represented in the system as a permanent segment. The legal values for the data in the object, and the operations which may be applied to the object are determined by the type. All objects have a variety of common properties discussed below, as well as properties unique to objects of a particular type.

2.4 Object ids.

Each managed object has an object id that can be used to reference the underlying object.

The object id consists of three components -- the class, the world id, and an object index.

The class is a small integer encoding of the managed type (2.1), and from an implementation point of view determines which object manager is responsible for the object.

The world id identifies the world containing the object (3.7), and from an implementation point of view corresponds to the R1000 VPID.

The object index uniquely identifies the object within the world.

Each object stores its own object id and the object id of its parent (3.1).

2.5. Object Names.

Each object has a full name and a simple name. The simple name is stored in the object. See 3.6.

2.6. Versions.

Each object has a version number. The system supports multiple objects with the same object id and same name, which are distinguished by their version number.

When we refer to a "version of an object" we mean one of the several objects which have the same object id. This usage is somewhat imprecise since the version is an object. In spite of this imprecision, such usage generally allows a clearer description.

Given an object id, a particular object with that object id can be selected with a version specification. A version specification either explicitly provides a version number to select the object, or uses the view mechanism discussed below (4.1).

2.7. Common operations.

There is a set of operations which are defined on all managed objects, called common operations. These include create, delete, copy, etc. Some types may have limited support for particular operations (i.e., copy may not work well for objects of type group). These operations are specified in detail in the package Directory.Operations.

2.8. Open/Close protocol.

In addition to the common operations, managed objects support an open/close protocol for accessing the typed user data stored in the underlying segments.

The open operation takes an object id and version selection information (section 4) as input, and returns a typed handle which references the data in the segment representing the selected object. Synchronization (2.10) and access control (2.11) are associated with the open operation.

Given the handle returned by the open operation, type-specific operations may be used to manipulate the data in the object. For each managed type, there is a small set of packages in the environment which define the type-specific operations (see <System.Dir>).

The close half of the open/close protocol releases access to the data and occurs when the action is committed or abandoned (2.9).

2.9. Atomic Actions.

The programming environment supports simple (non-nested) atomic actions. An atomic action is a sequence of one or more operations where either 1) all of the operations succeed and all of the results are permanently recorded, or 2) none of the operations will have any effect. The environment guarantees the atomicity of such actions, even in the face of system crashes. Atomic actions allow large composite operations to be easily constructed without programming complex error recovery and backout procedures.

In this action paradigm, an agent (Ada task) may obtain an action id, which uniquely identifies the action being performed. All of the common directory operations and all open operations on managed

objects require an action id as a parameter. An agent may perform a large number of operations (limited by contention for a large but fixed pool of system resources required to implement the action paradigm) within one action.

An agent may commit or abandon the action when all operations have been performed.

If an action is committed, then immediately upon return from the call to commit all of the operations performed in that action will have taken permanent effect. The system no longer uses the snapshot mechanism. Permanence is associated with committing each action, there is no waiting for the next snapshot.

If an action is abandoned, then the environment backs out of all the operations performed on behalf of that action, restoring all affected objects to their original state as if no operations had ever occurred.

An action is automatically abandoned when the agent is no longer callable (the Ada T'CALLABLE attribute yields false). This prevents completed, terminated or abnormal tasks from locking resources (2.10). Actions started during the elaboration of a package should be committed or abandoned during that elaboration.

Certain environment operations consist of a large number of smaller operations, all performed as a single large action. The failure of one of the intermediate operations might leave the environment in an inconsistent state if the action were to be committed.

In those cases where a environment operation is performed on behalf of some action and the operation fails in a manner that requires the action to be abandoned, the environment marks the action as uncommittable.

An uncommittable action can be abandoned, but can not be committed. This prevents the client from committing the action and possibly invalidating system invariants. At the same time, this mechanism does not force immediate abandonment of the action, which would close all objects opened by the action and prevent the client from performing reasonable error processing. This facility is used extensively within the environment and is available to users.

## 2.10. Locks and Synchronization.

When performing operations on an object, the operation requests read, update or unsynchronized access to the object. In the first two cases a read or update lock will be obtained on behalf of the specified action. If a read lock is obtained, other readers are allowed, but no updaters. If an update lock is obtained, only operations with the same action id may manipulate the object. Committing or abandoning the action releases all locks obtained by the action. This is the basic environment synchronization mechanism.

Open operations must specify whether the open is for read or for update. If the object is opened for update, the object may be modified. Attempting to modify an object that is only open for read will cause the exception Write_To_Read_Only_Page.

Releasing locks is associated with the commit of the action which obtained the locks. Ability to modify the objects is revoked immediately. Read access (using a handle previously returned by

open) is revoked no later than the next open for update.

For the common operations, the required locks are part of the specification of the operation (for example, copy aquires a read lock on the source and an update lock on the destination).

An agent operating on an object may specify a maximum time that it is willing to wait to obtain a lock on the object. If the requested object is currently locked, the environment will queue the new request until the object becomes available. If the object does not become available within the specified maximum wait time, an error status will be returned and the operation will fail. If no wait time is provided, the standard system default wait time (5 seconds) is used.

The third form of access, unsynchronized access, will obtain no locks and will never queue. Unsynchronized access is inherently usafe, since other agents can modify or delete the object being accessed.

The standard IO packages use update access to implement input_output and output modes. The mode input is implemented with either unsynchronized or read, depending upon user preference (indicated by job switches or the Ada "form" parameter). Using unsynchronized access and opening a file for input does not prevent other agents from writing (or deleting) the same file.

### 2.11. Access Control.

The environment enforces access control at the point of aquiring a lock on an object. The three access rights supported are read, update and owner. Read rights are required to obtain a read lock or unsynchronized access. Update rights are are required to obtain an update lock. Owner rights are required to change the access writes.

The environment stores an access list with each object for each access right. The access lists determine which groups may use the indicated mode to access the object.

The system supports a small set of groups (order 2**8). A group is a set of users. The environment supports adding and removing users from groups. There is a distinguished group (operator) which has access to all objects. There is also a distinguished group (public) to which all users belong.

A session has as session state the list of groups representing the rights for the session. When a session is created, the list is initialized to the set of groups which contain the user who started the session. This list may be modified by the user.

## 3. Directory Structure.

### 3.1. Parent/Child relation.

Every object, C, has a single parent object, P, where P is the only object which has C as one of its children. The only exception to this is the object which is the root of the directory structure, called Universe, which has the nil object as a parent. Only two of the currently supported managed types (object classes) may have children, directories and Ada units. The parent/child relation forms a tree of objects, which is the entire directory system, or universe, for a given R1000.

3.2. Directories.

There is a managed type called directory. Objects of that type are called directories. Directories are the main structuring mechanism in the environment. A directory may have children of any managed type.

There can be only one object with a given object id for directory objects (i.e., multiple versions are not supported for directories).

3.3. Ada Units.

There is a managed type called Ada. Objects of that type are called Ada units. Ada units are primarily for representing Ada programs, and as such are discussed at length in section 5. Here we are only concerned with the structural directory properties of Ada units.

Ada library units appear as children of directories. Ada library unit bodies are children of the associated library unit. Ada library unit bodies can have children which are subunits. Thus the parent of the library unit body is the library unit, and the parent of the subunit is the library unit body. This follows the definitions in the LRM, Chapter 10.

3.4. Object Declarations.

All objects (except attributes, see 3.5) have a declaration in the directory system. For Ada units (one type of managed object) the declaration appears as a unit declaration in a directory or as a subunit stub declaration in an Ada unit. For other managed objects, the declaration appears as an object declaration in a directory.

3.5. Attribute Objects.

Directories and Ada units may have a special kind of child object called an attribute object. This is an object whose parent is the directory or Ada unit, but which is not explicitly declared in the parent. While the normal display of Directories and Ada units will not show these attribute objects, a full display will include them. Attribute objects are named using Ada attribute qualification (3.6).

3.6. Object Names.

Each managed object has a simple name. To form path names, the simple names of objects are combined in sequences, with each simple naming being a child of the preceding simple name. The simple names in the path name are separated by a period (".") where the second of two simple names does not begin with an apostophe Three canonical path names are supported, the simple Ada name of an object, the directory name of an object, and the full name of an object.

3.6.1. Simple Names.

The simple name of an object is limited to 64 characters. A simple name is either an Ada identifier, or an apostrophe (') followed by either an identifier or the Ada reserved word Body.

For Ada library units and subunits, the simple name is the identifier of the unit. If the identifier of the unit is greater than 64 characters, the first 64 characters form the object simple name, which is used for all directory naming operations. Higher level naming facilities may accept the longer names. If two Ada units in the same directory are not unique in the first 64

characters, creating the second one will fail (there are two known solutions which would eliminate this restriction, but it has been decided that it is not worth the trouble at this point).

For Ada library unit bodies, the simple name is 'Body.

An attribute object has as its simple name the identifier which is the name of a user defined attribute, preceded by an apostrophe ('code, 'cg_attrs, etc.).

3.6.2. Simple Ada Names.

Only Ada objects have simple Ada names. The simple Ada name of an Ada unit is always the simple name (3.6.1) except for library unit bodies, where the simple Ada name is the same as the directory name (3.6.3) for the object.

3.6.3. Directory Names.

The directory name of an object is a path name for the object starting at (but not including) the first directory enclosing the object.

3.6.4. Full Names.

The full name of an object is a path name starting with (and including) the Universe.

3.6.5. Naming Example.

Consider a simple example where D is a Directory whose parent is Universe, U is a library unit in D, and U1 is a subunit of U. Assume that each Ada unit involved here as a 'Code attribute.

The full name for U is Universe.D.U. The simple name, simple Ada name, and the directory name are all U.

The full name for the body of U is Universe.D.U'Body. The simple name is 'Body. The simple Ada name and the directory name are U'Body.

The full name for the code for the body of U is Universe.D.U'Body'Code. The simple name is 'Body. The directory name is U'Body'Code. There is no simple Ada name for code.

The full name for U1 is Universe.D.U.U1. The simple name and simple Ada name are U1. The directory name is U.U1.

The full name for the code for U1 is Universe.D.U.U1'Code. The simple name is 'Code. The directory name is U.U1'Code.

3.6.6. Name Resolution.

The system provides facilities for resolving a string name and determining the set of object ids denoted by the name. Depending upon the context and other factors, many names may resolve to a particular object. See 5.4.

3.7. Worlds.

Certain distinguished directories are called World Directories, or simply Worlds. Each world has a unique world id, which from an implementation point of view corresponds to the R1000 VPID. Worlds

are the entities of interest for controlling disk resources, recording history, providing Ada library support, controlling compilation switches and managing configurations. The world manages these for all of the objects within it, including nested directories (that are not worlds) and their contents.

The root of the directory system is a world, each user home directory is a world, and the root directory for each subsystem (section 99) is a world. There is a fixed limit on the number of worlds in the universe (1024 - JobVPs - SystemVPs = approx 750), so their creation must explicitly managed by the user. The fact that worlds are the basis for resource management and configuration management provides additional incentive for users to carefully managed creation of worlds.

3.7.1. Resource Management.

All of the objects in a world are on the same disk volume, which must be specified (explicitly or by default) at the time the World is created. Like a job (1.2), a world has warning and absolute disk resource limits. Exceeding those limits causes a responsible job (that was consuming space in the world) to be terminated. Several jobs may be terminated before the "real" culprit is terminated, since any job allocating space in the world will be terminated when only one might be a "run away" job.

3.7.2. History.

The history mechanism is closely related to version control facilities which must be designed and specified before much can be said about history.

3.7.3. Libraries.

In order to implement the flat name space of Ada units required by the Ada library mechanism, for each world, W, there is a special subdirectory, W'Library, which contains an entry (object of class Alias, see 4.6.) for every Ada unit in the world and every Ada unit imported into the world. This library mechanism is discussed below in the discussion of views and in the discussion of Ada naming.

3.7.4. Switches.

The world contains a switch object which controls compilation, history and certain directory operations which consult the switches to control processing. One of the most important functions controlled is which set of target tools (code generator, etc.) are invoked in compiling Ada units. There may be several different versions of the switches, which appear in (and apply to) different views of the world (4.3).

The switches are related to semantic consistency of Ada units in that all units in the world view must be processed by compatible switches. Changing the switches in an incompatible way will obsolesce compiled units.

3.7.5. Configuration Management.

The role of worlds in configuration management is discusses in section 4.

4. Views.

4.1. Views and Configuration Managment.

The presence of multiple objects with the same object id (multiple versions of an object) requires a mechanism for selecting consistent sets of objects to form a configuration or "view". Constructing, manipulating and maintaining these consistent views is often referred to as configuration management.

There are two kinds of views in the system, universe views and world universe views. A universe view defines a consistent set of world views. A world view defines a consistent set of objects within a world.

4.2. Universe View.

There is a managed type Universe View. A universe view is essentially a map whose domain is world ids and whose range is world views. Thus a universe view selects a particular view of all the worlds in the universe.

The basic operations on universe views are to add world views to the universe view, to remove world views from the universe view, and to query given a particular world id which world view is a member of the universe view.

The world views which make up a universe view must be consistent. The primary consistency requirement is that for each Ada unit imported (4.4) by world views in the universe view, the universe view include a world view which exports a version of the Ada unit which is upward compatible with all of the imported versions. The definition of upward compatibility of Ada units is described in 6.xx.

The environment maintains a single machine-wide default universe view. For each world, one world view may be selected to be in the default universe view.

Each session has a default universe view. Each job has a default universe view. Unless otherwise specified, the job universe view is inherited from the session default universe view.

Any operation which does not specify a particular version when using an object id or object name, will use the job view to select the appropriate world views and through the world views select the appropriate versions of objects.

4.3. World Views.

There is a managed type world view. The world view is essentially a map from object ids to objects.

Recall that an object id consists of a class, a world id and an object index (3.1). The world view has as its domain all the object ids whose world id is the world containing the world view. Stated another way, the world view has as its domain the object indices for objects in the enclosing world.

Recall also that there may be several objects with the same object id (i.e., several versions of an object). The range of the world view map is one of the objects whose object id matches the domain element, or the nil object.

Thus the world view selects at most one object for every object id

in the given world. This provides a consistent "view" of the world. Operating within this "view" frees one from having to explicitly specify version selection information when referencing objects.

The environment maintains structural consistency for the objects in the world view. Structural consistency means that no object in the world can be in the world view (i.e., appear in the range of the object map) unless its parent is also in the world view (the parent object id maps to a non-nil range value). Note that the the directory object which is the root of the world is in every world view for the world and is special in that its parent object id is not in the same world and therefore is outside the domain of the map.

The environment maintains semantic consistency of all installed (see 6.2.) Ada units in a world view. In terms of Diana, the basic invariant is that for any two units with object ids A and B in a world view, if A has semantic attributes which reference a unit with object id C, then any references from B to C reference the same object. Furthermore, the referenced object with id C must appear in the world view. This last point implies that the world view must include units imported from other worlds (see 4.4).

All operations on objects in a world are with respect to a world view. Directory operations (create, delete, etc.) update the world view appropriately to maintain structural consistency, and such operations fail if they violate structural consistency requirements. Similarly, directory operations and compilation operations update the world view appropriately to maintain semantic consistency, and such operations will fail if they violate semantic consistency requirements.

World views are associated with a particular world and cannot appear at arbitrary locations in the directory system. All of the different world views which describe views of a given world, W, appear in the special directory W'Views. Each world view has as its parent a universe view, where the universe view has as its parent the directory W'Views. The universe view always includes the child world view as a member of the universe view. This pair of views (the universe view and the world view) are closely related and are kept consistent by the system. This structure reflects the fact that it actually takes a world view and a universe view to describe a consistent view of a world. (This is described more clearly below, here we are trying to specify the structural aspects of the directory system with respect to world views).

Because of the special nature and contents of the 'View directory, there are certain restrictions on operations there. The common create cannot be used to create objects in the 'View directory. Open and copy operations cannot modify the contents of the 'View directory (although universe views in the 'View directory may appears as the source of a copy). Delete and Destroy operate in the 'View directory, but treat universe/world views as a pair (i.e., destroying one destroys the other also).

The World package provides the basic operations for creating a world with an initial world view, freezing world views, spawning new views of a world, and importing into a world view. These operations properly construct and maintain the contents of the 'View directory.

## 4.4. Creating Worlds

The steps involved in creating a new world are to create the root

directory (on the indicated volume), create the 'Library and 'Views subdirectories, create the initial world view (and its parent universe view), and then update the job universe view (4.2) to include the newly created world.

4.5. Freezing World Views.

A world view may be frozen, in which case none of the objects in the world which appear in the range of the object map can be modified (opened for update, destroyed, etc.).

A world view must be frozen before it can be imported into other world views (4.6). This requirement eliminates the need to record cross world dependencies, and implies that obsolescence processing (6.xx) is always restricted to a single world.

4.6. Aliases and Importing World Views.

The definition of semantic consistency introduced above raises the issue of references to objects in other worlds.

Here we are concerned with importing objects from other worlds so that compiled references can be constructed in an efficient manner while supporting consistency requirements. This is implemented with a special class of object, called an alias. The value of an alias is the object id (and full name) of an object in another world. The contents of the alias is accelarated into the world view so that references to the alias are efficiently mapped to the aliased object.

Note that an alias will map a local object id (same world as that containing the world view) to an object id in some other world; however, an alias does not designate a specific object (since there may be several objects with the same object id).

Dereferencing an alias to get to an object involves a second step using a universe view to interpret this non-local object id. Fully resolving an alias requires extracting from the job universe view the world view corresponding to the world id in the object id that is the value of the alias. This will provide a world view which must either map the object id to an object, or map the object id to nil. In either case, we have completed resolution of the alias.

For compilation and other operations that require strict semantic consistency, the job universe view is set to the universe view associated with the world view of interest. Thus semantic consistenty is defined with respect to the pair of associated views (the universe view and its child world view).

Entries are made into the universe view associated with a world view by importing world views for other worlds. The import operation first adds the world view to the universe view. Then, for each Ada unit in the world described by the imported world view, the import operation creates a alias in the 'Library directory for the importing world (if the alias did not already exist). These aliases appear in the world view, as described above.

It is impossible to construct aliases which reference objects in a world which has not been imported.

4.7. Spawning world views.

4.8. Deleting World Views.

4.9. Destroying World Views.

Destroying a world view destroys the world view object itself, and the associated universe view, but destroys none of the objects in the world view. Destroying a world view may cause objects in the world to no longer be reachable from any world view. The expunge operation (which may be applied to individual objects, or to entire worlds) destroys all objects which are not reachable from an existing world view.

As mentioned earlier (4.2.6), the system relies upon the fact that only frozen world views can be imported to minimize the amount of dependency data recorded and to limit obsolescence processing. Basically, we have shifted cross world obsolescence processing from the demotion of individual units to the destruction of frozen world views.

When a frozen world view is destroyed, the system must check to determine whether that world view has been imported by any other world views. If it has, the destroy fails, producing a list of dependent worlds. The system provides an operation that destroys a world view and the closure of its dependents.

4.10. Expunging Worlds.

5. Naming, Scope Rules and Visibility.

5.1. Ada units.

Ada library units may appear in directories. Within Ada units the environment follows Ada semantics. Library units and library unit bodies are closed scopes and may only reference external units that are imported via WITH clauses. Subunits have visibility to their parents as well as units that are imported via WITH clauses.

A simple name in a WITH clause of an Ada unit is resolved by looking for the simple name in the 'Library directory of the enclosing world. Note that the use of the 'Library directory enforces that all Ada units in a world, including imported units, have unique simple names. Creating Ada units makes an entry in the 'Library, and will fail if the name collides with another entry in the 'Library. Similarly, the import operation discussed above (4.2.10) will make entries in the 'Library for every imported unit. If the name of an imported unit collides with an existing entry in the 'Library, the system chooses a reasonable name and constructs a (renaming) alias in the 'Library and notifies the user.

The WITH clauses on a unit can only denote Ada units. This implies that there are no compiled references to managed objects other than Ada units.

5.3. Command Context.

Commands are given from a command window that is associated with a window on some object in the directory system. Commands are compiled using the 'Library mechanism to provide a context, with an implicit WITH on every unit available in the 'Library. The 'Library for the command is slightly different from that of a normal Ada unit, in that the session includes a search path which designates several worlds.

The units in the 'Library directories of the worlds named in the search path are combined into a single logical 'Library, where units in earlier worlds in the path hide units with the same name from later worlds. The cefault search path is 1) the world containing the object in the window, 2) the user home directory, 3) the environment commands world.

## 5.4. Object Naming.

### 5.4.1. Namingin Context.

The environment resolves a string name to an object id with respect to a particular context. The context may be any object in the directory system. The environment supports a default context on a per job basis. The default job context is the object associated with the window where the command which intiated the job was issued.

### 5.4.2. Name Resolution.

In resolving a name, the first name segment is resolved by determining if the given context (object) has any children (objects) with the given name. If not, move out to the parent and repeat. For worlds, include the contents of the 'Library if no children match. When a match is found, select the child of the matching object whose name matches the second component of the name, and so on.

The 'Body does not affect the meaning of a name, except when appear in the last portion of the name (after the last ".").

Analogous to 'Body, there is a 'Spec which may appear in full names. 'Spec is never a simple name and has no impact on the meaning of a full name for purposes of object name resolution.

### 5.4.3. Version Selection.

Names are resolved with respect to the job universe view, with the defaulting mechanisms as described in 4.3. Specific versions of an object may be named using the version attribute.

Consider the example of a package Foo in a world called Bar. Assume that our session view is based on a universe view called Gamma_1. Assume that Gamma_1 selects a world view for the Bar called B_2. Assume that B_2 selects version 37 of the package Foo. Then simply using the name "Bar.Foo" will resolve to version 37 of the Foo.

If we wish to denote version 31 of Foo, we use the name Bar.Foo'V(31).

If we wish to select the version of the Foo which appeared in the universe view Old_Releases.Gamma_0, we use the name Bar.Foo'V(Old_Releases.Gamma_0).

If we wish to select the version of the Foo which appeared in the B_1 world view of the world Bar, we use the name Bar.Foo'V(B_1).

In the case where the parameter to the version attribute is not a number, the environment first tries to find an object of type universe view with the given name in the current object naming context. If unable to find such a universe view, the environment looks at the world containing the designated object to see if there is a world view with the given simple name. That is why the last example did not require using Bar.B_1 or some other notation.

### 5.4.4. Wildcards.

5.5. Debugging Context. TBD.

# 6. Ada Units

## 6.1. Static Program Representation.

An Ada unit is a type of managed object represented by a program unit stub declaration (rather than an object declaration) in the parent unit or directory. The environment maintains a complete program representation which is made available to all environment facilities and tools. The basic program representation is standard Diana. Additional information is stored in the Ada unit or as object attributes associated with the Ada unit. Section 6.x discusses the interaction between Diana and the view mechanism.

## 6.2. Unit States.

An Ada unit will be in one of the several states, depending upon the phases of compilation which have occurred. Environment operations exist to promote a unit (and its closure) to higher states, and similarly to demote a unit to lower states. Based on user input, these environment facilities will automatically determine the proper closure and compilation ordering, log errors and schedule processing. The following states are supported.

a. Source. A unit which has not been made semantically consistent and cannot be referenced for semantic purposes. When first created (through the editor or batch parser or the copying) units are in the complacent source state.

b. Installed. The unit is semantically consistent, and the environment will maintain that consistency unless requested to return the unit to the source state. A unit is installed with respect to a particular view and target (4.2 and 6.3).

c. Coded. Executable code has been generated for the unit for a particular target.

d. Elaborated. For units in shared libraries (6.4), the unit has been elaborated (in the Ada sense) and a persistent runtime representation exists.

Units may be moved between source, installed and coded in any combination desired by the user, so long as it is consistent with the rules of Ada separate compilation and any constraints imposed by the target code generator (for moving to and from the coded state). The environment will not allow a unit to be demoted from a higher state to a lower state if that would obsolesce other units, unless the request to do so by the user.

## 6.3. Compatibility of Ada units.

A unit, U1, which differs from an earlier version of the unit, U0, only in upward compatible ways is said to be upward compatible with respect to the earlier version. The set of changes which are upward compatible is target dependent. For the P1000, the following changes are upward compatible.

a. Accition and deletion of entire declarations in package specs.
b. Arbitrary changes to private parts designated as "closed".
c. Arbitrary changes to bodies which do not include inlined or

macro-expanded program units.

d. Limited (TBD) changes to bodies which include inlined or macro-expanded program units.

In order for the system tc properly recognize that U1 is upward compatible with U0, U1 must either have been derived from a sequence of incremental changes without demotion to source, or have been constructed by a special tool from the coded U0 and the source for U1.

Each installec Ada unit has a compatibility index stored in it. The index remains the same so long as the unit remains installed and no incompatible changes are made to the unit. Two units with the same compatibility incex are said to be approximately compatible.

While determining that two units are approximately compatible is relatively inexpensive, determining whether U1 is upward compatible with respect to the approximately compatible U0 is more expensive, and involves actually examining attributes on the two Diana trees.

Construction of a universe view enforces that for each Ada unit in the universe, each world view in the universe view selects either no version of that unit, or a version that is approximately compatible with the one in the world view of the world containing (rather than importing) the unit. This level of compatibility does not guarantee complete consistency.

semantic issues.
runtime issues.

Construction of a world view enforces more stringent consistency requirements.

6.4. Program Elaboration and Execution.

There are two fundamental modes of elaboration that the environment supports, shared elaboration and unshared elaboration.

Shared elaboration involves promoting a world to the elaborated state (elaborating all the units in the world in accordance with Ada elaboration semantics) and then sharing that elaboration among one or more clients. Once the user has elaborated the world, all of the units are elaborated and can be called directly from commands. In addition, other shared worlds which reference the first may be elaborated. A shared world must have the pragma SHARED_ELABORATION in its context clause.

Elaborating a shared world involves aquiring a job control point which provides all storage for the elaboration and has all of the properties of any other job (space and time limits, scheduling parameters, etc.).

An important example of shared worlds are the environment packages that are available as part of the R1000. These environment packages have one elaboration that is shared by all users. These packages are in a specially constructed world, which has the semantics of a shared world from the users point of view.

Note that shared worlds which will have multiple concurrent clients must be structured carefully to operate correctly. In particular, one must properly address synchronization and storage management issues recognizing that package state is shared by all clients.

Units from unshared worlds are elaborated as needed. Any time a command is executed, the environment computes (as much as necessary)

the transitive closure of the units referenced by the command. Units from shared worlds are already elaborated and that elaboration is shared by the command being executed, else the command fails with elaboration error (program error exception). If the closure involves unshared units, they will not be in the elaborated state and there will be no elaboration to share. Instead, the command will elaborate it's own copy of the unshared units.

Unshared elaboration basically follows the semantics for Ada main procedures, treating the command as a main procedure. A command which consists of exactly one statement which is a procedure call to an unshared library unit procedure will follow Ada main unit semantics exactly. As an optimization, the user may include a pragma MAIN in a library unit procedure. This causes the environment to save the transitive closure and elaboration information for the procedure so that calls are more efficient. Note that the environment still computes a closure and elaboration order for the command (which may call two main units, for example), but the computation can be faster with main units since the closure information is retained (and obsolesced when appropriate). There is also a facility for computing that information without actually executing the procedure (loading).