The Epsilon System

Principles of Operation

Volume III

Ada Programs and Compilation

Revision 11.1

February 25, 1986

Table of Contents

Ada Programs and Compilation

Revision 11.1
February 25, 1986

This document describes how Ada programs are represented and manipulated in the Epsilon environment. For background, it includes synopses of other parts of the Epsilon object system. This document supersedes previous documents on Diana, Subsystem Compatibility, and Compilation.

1. Overview

1.1. Features

Uniform Use of Views

As long as one's universe view remains constant, one can ignore the existence of multiple versions. Even very low-level operations (*e.g.*, following Diana pointers) go through the view mechanism.

No Hard Pointers Between Ada Units

All references between compilation units are indirect references that use the object table of the enclosing world view and the declaration table of the referenced unit. New versions of Ada objects can be created or existing versions can be modified without invalidating references from other objects.

Transfers Are Efficient

Because there are no hard pointers and because information about the exact address of a referenced node is not scattered through the referencing nodes, transferring Ada units between machines does not require recompilation from source or traversal of Diana trees. Transferred copies of Ada units use no more space than the originals.

When an object is moved, certain values in internal tables must be updated to reflect the new context of the object. The original information in these tables and information to validate other values in the Ada object must be converted to an archive form and saved with the object to be transferred. This additional information is used only to reconstruct the object at the destination and is not retained after the transfer.

Support for Compatible Changes

The system supports various kinds of approximately compatible changes such as addition and deletion of declarations. Actual compatibility is determined on the basis of services that are actually used. High level operations such as a compatible merge of two versions are also supported.

The notion of compatibility is integrated into the basic configuration management, version control, and execution mechanisms.

Compatibility Checking is Fast

Compatibility must be checked before execution begins, even for command execution. This will be fast in common cases.

Graceful Recovery from Lost State

When the disk heads crash (or the state of the disk is lost in some other way), the state of objects and unique number generators may be lost. This can create various kinds of inconsistencies, which we must at least detect, and would like to recover from.

Editor Determines Granularity of Changes

For an arbitrary set of editing operations the editor is able to determine which declarations have been structurally changed. If the editor is conservative (marking unchanged trees as changed) performance will degrade, but correctness will not be affected.

Debugging is Integrated with Compilation System

Debugging and editing of a program may transpire concurrently.

1.2. 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). (A unit still must be *coded* as a whole.) 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. 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.

1.2.1. Exported Declarations

To support the finer granularity of consistency checking and the weaker notion of compatibility (see below), a *declaration number* is assigned to each declaration that can be referenced from another unit. In particular, compilation unit declarations, declarations in package visible parts (including nested visible parts) and declarations in bodies that are visible to subunits or visible to macro-expanded bodies (generics and inlined subprograms) must be assigned declaration numbers. Declarations that are assigned declaration numbers are said to be *exported* by the unit that contains the declaration. A library unit always exports its own declaration. Exported declarations only reference other exported declarations. Exported declarations are also called *external declarations*.

Declaration numbers are assigned in such a way that two declarations in different versions of a compilation unit have the same declaration number if and only if they represent equivalent declarations. Generally speaking, equivalent declarations have the same lexical and syntactical structure, semanticize the same, and are referenced by identical code sequences.

Declaration numbers are allocated globally across all versions of a unit.

The Diana tree for each exported declaration carries its declaration number as a semantic attribute. Furthermore, a *declaration table* in each Ada object maps declaration numbers into the corresponding Diana declaration node in that object.

Declaration numbers are discussed in more detail in Section 2.10.

1.2.2. Compatibility Keys

A *compatibility key* contains the object id of an Ada unit, and a *declaration set*, which is a Boolean array indexed by declaration numbers. An Ada unit ***Foo*** is said to *import* another Ada unit ***Bar*** if ***Foo*** contains a reference to one of ***Bar***'s exported declarations. It also imports the specific declarations that are referenced. For each Ada unit that Ada unit ***Foo*** imports, ***Foo*** has a compatibility key called an *import key*. An entry in the import key's declaration set is true if and only if the corresponding declaration is imported by ***Foo***.

An Ada unit also has a compatibility key called an *export key*. An entry in the declaration set is true if and only if the corresponding declaration is exported by the unit.

1.2.3. 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 world. 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 worlds only during specific, user-initiated operations: when a world is first imported or when that import is subsequently updated (See Section 2.4). Even though two worlds 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.

Two worlds are compatible if all the units in the world are compatible. A unit, ***Client***, is compatible with a unit, ***Server***, that it references if

- ***Client*** has been compiled against some version of ***Server***, and
- ***Server*** provides all of the declarations (subprograms, tasks, constants, variables, exceptions, and types) that ***Client*** needs from ***Server***.

The declarations provided by ***Server*** are codified in its export key; the declarations needed by ***Client*** are codified in its import keys. ***Client*** is compatible with ***Server*** if the import key for ***Server*** in ***Client*** is compatible with the ***Server***'s export key. An import key is compatible with a export key if the object id's in the two keys are identical, and every declaration in the import declaration set is also in the export declaration set. A unit can execute with a set of imported units if each of their export keys is compatible with the corresponding import key of the unit.

By definition, it's always possible to add declarations to compatible units without affecting their compatibility. It's also frequently possible to delete declarations from compatible units without affecting their compatibility. As long as those deleted declarations are not used by the compatible units, the units remain compatible. Because of Ada's rules for the scope of declarations and the resolution of overloaded names, the above declaration changes that did not affect compatibility might violate Ada consistency because the declarations have the same names as some declarations that are required for compatibility. That is, because of name conflicts, attempting to recompile a set of compatible units may produce semantic errors or may change the meaning of the program, even though the set of units executed without errors before the recompilation. In this sense, compatible units are consistent "upto renaming;" if declarations in the compatible units always had unique names, they would remain consistent through all compatible changes.

Adding or deleting declarations that are namesakes of declarations that are required for compatibility could obviously lead to inconsistencies. Note, however, that *any* changed declaration is a potential problem if

1. It appears in the visible part of a package that is **use**'ed by the client; the new name could clash with a declaration in another package that is also **use**'ed by the client,

2. It becomes a derivable subprogram of a visible type; the new name becomes visible in any client that derives from the visible type.

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.

1.2.4. Dependency Database

For greater efficiency, the information kept in Gamma's central dependency database is distributed into three data structures.

A *dependency matrix* in each world records unit-to-unit dependencies within that world in a Boolean matrix that is easily accessed and quickly manipulated. As outlined below, The dependency matrix is used by a number of operations in addition to change analysis. The structure of the dependency matrix is specified in Section 2.5.4.

Specifically for change analysis, each library unit's Ada object contains a *top declaration database*, which keeps track of overloadable declarations in the library unit and its secondary units. Finally, each Ada object contains a *usage map*, which describes how

specific declarations are referenced in that unit. These data structures are defined in Sections 2.6.5 and 2.6.

Section 3.5 discusses how these data structures are used in the computation of semantic dependencies.

1.3. Editing

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." All changes are made to the original tree and displayed in the same window as the rest of the unit.

When an exported declaration is changed or deleted, the editor records that fact for the semanticist, removes the declaration from the declaration table, and removes the declaration number from the declaration node. Traversing a Diana reference to the declaration will fail because of the missing declaration table entry. The editor also records the addition of exported declarations[1].

Changes to statements and local declarations are also recorded by the editor.

Changes to white space and comments that do not cause statements or declarations to be reparsed need not be recorded by the editor since they have no impact on the semantics or coding of the unit.

After declarations or statements in a unit have been edited, the unit is "source" in the sense that it must be re-installed before it can be coded. If the editing has changed or introduced declarations, then units that import the unit (perhaps indirectly) are also considered (and presented to users as) source units. The edited unit must be re-installed before the importing units can be re-installed. The editor, the semanticist, and change analysis (see below) determine what changes have actually occurred and use this information to minimize the work needed to make the units consistent once again.

Changes to local declarations do not impact importers because no external declarations reference a local declaration.

If only statements have been edited, the meaning of importing units is not affected, although they may have to be re-coded because of the use of macro expansions in the target implementation.

If a unit is committed with syntax errors, it cannot be installed until those errors are

---

[1] How editing changes are to be recorded is yet to be determined.

corrected. If the syntax errors occur only in statements or in declarations that cannot affect exported declarations, importing units will not be affected; otherwise, importing units wil be presented as source until the unit is re-installed.

1.4. Installation

Before a unit can be installed, it's sufficient that all units in the closure of its imports be installed. Actually, it's only necessary that each imported unit has been semanticized, is not obsolete, and has no freshly parsed exported declarations that could affect importers.

If the context clause of the unit has not been edited since it was last semanticized, the information in the dependency matrix is valid and can be used to compute the import closure; otherwise the names in the unit's **with**-clauses must be resolved to begin the computation of the import closure. In the computation of the closure, the dependency matrix can be used for any unit whose context clause has not been edited or made obsolete since the last time it was semanticized.

When a unit is installed the system must semanticize the unit and set the compatibility information. If the unit has not been previously semanticized then the semanticist runs as it does in Gamma and computes the values of all semantic attributes. If the unit has been previously installed then the semanticist runs in *verification mode*. In verification mode, each time the semanticist is about to set an attribute, it compares the new value with the old value. If the new value is different, the declaration number of the enclosing declaration is cleared.[2]

After each exported declaration is semanticized, it must be assigned a declaration number so that references to it in the same unit can be filled in correctly as they are semanticized. If the declaration node has a declaration number attribute still then that declaration number is used, otherwise the declaration number is computed as discussed in Section 2.10. When the declaration number has been computed, the declaration table entry for the declaration can be set up and the declaration can be put into the export key for the unit.

Installation also computes the import keys and updates the dependency matrix to reflect the new requirements of the installed unit.

1.5. Change Analysis

The next step of compilation is to compute which dependents of the newly-installed unit may now have obsolete semantic attributes. If the newly-installed unit's export key is the same as the export key it had before the installation, then all dependent unit

[2] If the new attribute is the same as the old one, the attribute is not changed; the page is not dirtied.

semantic attributes are still valid. Otherwise, change analysis is performed on each of the exported declarations that changed. The dependency matrix and information in the referencing units is used by change analysis to determine which units might be affected by the change in exported declarations. The affected units are marked as being potentially obsolete.

Potentially obsolete units are presented to users as source units. Any units whose import closure includes a potentially obsolete unit are also presented as source units. There is enough information in the Dependency Matrix and state information maintained in the world view to rapidly determine the presentation state of a unit without propagating markers to the units themselves. In fact, it will frequently be the case that a unit marked potentially obsolete will not propagate the obsolescence to its importers; obsolescence would be propagated only if exported declarations are themselves affected by the changes that have been made to imported declarations. Obsolescence is propagated only after a unit has been installed and the semanticist has determined that an exported declaration has indeed changed.

Change analysis does not propagate obsolescence across world boundaries. Change analysis is performed on imported units only when they are imported into a world and then only units in the importing world that are not consistent with the new imports will be made obsolete.[3] If only frozen worlds are imported, the change analysis perfomed at time of import is sufficient to guarantee full Ada consistency. If, however, unfrozen worlds are imported, consistency cannot be guaranteed. Programs that once semanticized and now run, may fail to re-semanticize because of changes made in the unfrozen, imported world in the interim.

When a world is imported, the export keys of each unit are compared with the export keys of the corresponding units that had previously been imported. Where the keys differ, change analysis is performed on the changed declarations. The units in the importing world that are impacted by the change (as determined by change analysis) are obsolesced.

1.6. Coding

When a unit is to be coded, the system uses the dependency matrix to compute the order in which to code units and to check that all dependencies can be satisfied. Certain target code generators will introduce *coding dependencies* between units that are not recorded in the dependency matrix. These coding dependencies require that the bodies of certain units already in the import closure of a unit be coded before the unit is coded. A target code generator that supports interunit inlining or uses the macro-expansion model for implementing generics will require such coding dependencies.

---

[3] Two imported worlds could be inconsistent with each other, yet be consistent with the importing world. Is this a problem?

Coding dependencies are not recorded in the dependency matrix. The ordering algorithm used by the code generator must rely on attributes in the visible part of those units whose bodies might have coding dependencies to detect these dependencies and incorporate them in the ordering of coding.

When code generation depends on the body of an imported unit, changes to that body are potentially incompatible changes. Logically speaking, each inlined subprogram body and each macro-expanded generic body should be treated as exported declarations separate from the declarations for their specifications and assigned a declaration number. When changes to the body are significant enough to cause a new declaration number to be assigned, the units that referenced the original declaration number become incompatible by the normal incompatibility mechanism. Using declarations numbers in this fashion is actually impractical because almost any change to a body would probably be viewed as a significant change, and consequently declarations numbers would be consummed at an unacceptable rate.

The proposed mechanism is to assign a time stamp to a macro-expanded body each time it is coded. The time stamp will be stored in each referencing unit. Runtime conformance checking will require that the time stamps in the referencing units equal the time stamp in the referenced unit.[4]

1.7. Loading

1.7.1. Basic Concepts

Loading is the process of creating an object for execution. There are two types of objects that may be loaded: programs and subsystems.

Each program is associated with a *main subprogram*. The main program is identified by a **pragma Main** that is in either the subprogram declaration or body. The program is loaded when the main subprogram body is promoted to coded. A program is executed by calling the main subprogram. This call elaborates all library units in the closure of the main subprogram and then executes the body of the main subprogram.

Each subsystem is associated with a view of a world. The subsystem is loaded by promoting the view to loaded. Some optional information may be provided that controls the loading process: 1) the set of units to be elaborated may be specified, 2) the units exported from the subsystem may be specified, and 3) the complete elaboration order may be specified.

A particular world view may contain multiple loaded programs, but only one loaded subsystem.

---

[4] A similar mechanism might be used with macro-expanded generics, but further study is required on this issue.

When a program or subsystem is loaded several objects are created:

- The *elaboration code segment* contains the code to elaborate the program units that are part of the program or subsystem.
- The *load image* is an object managed by KKOM. The load image contains information concerning all the code segments necessary for execution as well as import information on other programs and subsystems.
- The *load view* is a world view that selects all the Ada units and attribute spaces of program units associated with the load image. The load view maintains references to the information necessary for debugging.

1.7.2. Code Data Base

Associated with every world view is a *code data base* object, which contains all the information necessary for a program or subsystem to be loaded within the world view.[5] The code data base contains information on every coded unit in the world view. For example the code segment names for Ada units and their subunits, elaboration order dependency information, and import information are all stored in the code data base. Also stored in the code data base is dependency information that relates coded Ada units to the loaded programs and subsystems that depend on them.[6] The code data base is updated after every promote or demote of a unit in the world view.

1.7.3. Execution

When execution is to occur, the load image associated with the program or subsystem is passed to KKOM for elaboration. The load view is then open (or otherwise locked) for the duration of the execution, which prevents the Ada units associated with the execution from being deleted. When execution is complete, the load view is closed (or unlocked).

1.7.4. Changing Units in the Program or Subsystem

When changing units that are part of a loaded program or subsystem the environment protects the actual versions of units from being deleted if these versions are needed for debugging currently execution load images. Also, the environment causes the appropriate obsolescence to occur in the current view. There are five activities that affect the state of a previously loaded program or subsystem:

---

[5] Since some of the information in the code data base is target independent (*e.g.*, context dependencies and completeness) it would be better to put this in a target independent data structure in the view.

[6] Since each load image contains information on all the code segments it references it may be possible to compute dependencies on programs by looking up the code segment names in the load images of the current world.

1. Editing a unit causes a new version to be created. The new version is created because the current version is referenced by both the current world view and the load view. If the user explicitly asked for a new version then the new version is pushed onto the front of the retention list. If the user did not ask for a new version then the newly created version replaces the current version on the current view's retention list.

2. Committing a unit causes the obsolescence of any dependent programs or subsystems. This obsolescence has the effect of deleting the load view from the current view. (For executing programs the load view is locked and thus is not expunged). The load image of the program is not deleted at this time. The unit state of the program is then *loaded phase 1*, which means a load image exists but no load view. The load view is deleted as soon as possible so that other changes to units in the current view will not force new versions to be created simply because they are referenced by the load view.

3. Installing the changed unit involves the normal process of semantic analysis, change analysis and compatibility computations.

4. Coding the unit has the effect of generating a new code segment if the old code segment is no longer valid. The code segment would remain valid if the unit was not semantically modified and if all inlined imported subprograms and generic bodies have not been modified. If a new code segment is generated then any dependent load images are deleted.

   Promotion of the changed unit occurs after editing. If actual changes occured in the unit that cause a new code segment to be generated for the unit then the load image of dependent programs is now deleted and the program is no longer in any coded state. If there were no actual changes to the unit (*e.g.*, only comments were added) then the load image is not deleted.

5. Loading the program again will create a new load image if one does not now exist. The load view is then created and the program is promoted to *loaded phase 2*, which means that it can be executed.

1.7.5. Multi-world Programs

Because programs can have units in multiple worlds KKOM supports the notion of *secondary load images*. Each secondary load image contains all the information for the code segments of units in the world of the secondary load image. Additionally, all secondary load images are back linked to a governing primary load image that is associated with the program. Each secondary load image has associated with it a secondary load view that selects all the Ada units and attribute spaces for units in the program that are in that world. Modifying units only causes the obsolescence of load views and load images in the same world.

It should be noted that when a multi-world program is loaded the program is bound to the specific world views that were current when the load occured. This binding is not changed by changing the universe view alone, it can only be changed by reloading the program under a different universe view.

1.7.6. Saving Programs

There exists an operation to *save* a loaded program. Saving a program is useful to prevent a loaded program from being obsolesced by changing the constituent units or for having multiple versions of the same loaded program in the same world view. Saving a program has the following effects:

- A new subprogram declaration object is created. This declaration has the same parameter-result profile as that of the original main subprogram, the name is that of the saved program.

- Copies of the original load image and all secondary load images are created and associated with the new object. Copies of the original load view and all secondary load views are created.

- There are no dependencies from the constituent units to the saved program. Therefore, the saved program will not be obsolesced by any changes to the constituent units. The new load images and load views will exist until the saved program is explicitly deleted.

1.8. 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. 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.

2. Data Structures

2.1. Unique Ids

Several aspects of the design rely on the generation of numbers that are unique across all space and time. For example, each world has a creation stamp, which is a unique id that can be used to precisely identify the world as it changes over time and as it moves from machine to machine. Most numbers used for this purpose are called a *stamp* in this document.

The unique id used for the default creation stamp for all objects consists of a *machine number*, a *boot number* for the machine, and a *sequence number* that counts generated ids since the last boot of the machine. Machine numbers are dispensed by Rational as machines are built. Unique ids are rather large (see Section 2.11), so they should be used only when necessary.

2.2. Objects and Object Ids

Every object on a machine has an *object id*, containing an *object class*, machine number, *world number*, an *object number*, and a hash of its creation stamp. There may be several *versions* of each object, which are objects with the same object id, distinguished by different *version numbers*. An object id and a version number are enough information to open one version of the object, *i.e.*, for object management to find its state stored on disk. While an object is open, it is assigned a memory address, which includes enough information to find its state in virtual memory.

2.2.1. Object Numbers

Every object in a world is assigned a unique object number. All versions of an object have the same object number.

To keep certain tables small, the object numbers of all Ada units in a world are allocated contiguously at the beginning of the space of object numbers. The range of object numbers allocated for Ada units (and therefore the maximum number of units that may be created) is a parameter of each world.

2.3. Worlds

The Epsilon directory system is a heirarchy of objects. *Worlds* and *directories* are objects whose sole purpose is to hold other objects. (In Epsilon, however, any object may have other objects nested within it.) Worlds are distinguished containers in the directory system, much as in Gamma. In Epsilon, however, any world may be a subsystem.

Each world on a machine is identified by a small unique number; its world number. Each defined world is an object of class World. There may be only one version of each world.

Each world contains a number of important nested objects:

link pack A map from a simple Ada name to an Ada library unit that may be `with`'ed by an Ada unit in the world.

world view A map from object numbers of the world to the version of the object that is visible in the view,

universe view A map from a world number into a version of its world view. More than one universe view may be in a world.

2.3.1. Distributed Development

The Epsilon system permits the development of software systems to be distributed over a set of loosely connected machines. The world (subsystem) is the unit of distribution. That is, each subsystem of a system can reside on a different machine, where it can be edited, compiled and debugged by its developers.

At any given time, active development of a subsystem should occur on only one machine in the entire universe, otherwise conflicts in the numberings used in the Ada implementation can arise. These conflicts can be detected, but they cannot be resolved. The development site can be moved from machine to machine as needed, however. Development of a world might be moved when the developers responsible for it start working on a new machine, or when responsibility is transferred to a new group on a new machine. Development is moved by freezing the current development world and then creating an unfrozen copy of the world at the new development site.

A *surrogate world* is used in distributed development to represent on one machine a subsystem that is actively being developed on another machine. A surrogate world has all the data structures of a regular world. It can be a complete, Ada-consistent copy of the world for which it is a surrogate. The surrogate world can be imported by other worlds on the machine as if it were the original. Programs that import it can be loaded and executed. All features of the debugger can operate normally on the surrogate world.[7]

The surrogate world arrives on the machine fully coded. It cannot be demoted or edited. It can be updated by copying into it a set of coded modules from another instance of the world, which is usually, but is not limited to, the active development world.

---

[7]There seem to be good reasons for supporting surrogate worlds that are imperfect copies of the corresponding development world. One form of surrogate might have only exported library unit specifications and a load image, for example. How much mutilation can be sustained and is useful is yet to be determined.

The copies of a world created by the above two copy operations are called *generations* of the world. The special operation of copying an object between generations of its world is called *cloning*. (By extension, *cloning* also denotes the operation that creates a new generation of a world.) The cloning operation is careful to preserve complete Ada-consistency without recompilation of the units in the world. It is this special care that distinguishes clones from other copies of objects. Copying worlds or objects of a world by other means, such as source archive or primitive directory operations will not preserve consistency, and in fact, will actually create new library units unrelated to the units from which they were copied. See Sections 3.6.1 and 3.6.3 for the details of cloning worlds and their objects.

It will be impossible to enforce the rule that there can be only one active development site for a world. Should two active development sites be created for a world they are called *sibling* worlds. The numbering conflict detection mechanisms will work as long as two siblings of a world cannot reside on the same machine.[8] This is easily enforced.

2.4. Universe Views

Each view of the entire system is an object of class universe view. As an object, it has a name, must be opened before it is accessed, has access control, and may be atomically modified under an action. There may be only one version of each universe view.

A universe view contains a *world table*. The world table is an array indexed by world number. Three types of entries are possible:

bound Contains the object id and version number of a world view. This identifies the world view that is to be used for finding objects in the world.

refreshable Contains the object id of a universe view and the object id and version number of a world view. The object id and version number of the world view identifies the world view that is to be used for finding objects in the world. The world view entry may be changed (refreshed) by the user by copying world view information from the universe view specified in the same entry. Consistency checking can be performed when a world view is refreshed.

void This indicates that the world does not appear in this universe view.

[8] The *generation stamp* and *parent stamp* defined in earlier versions of this document, have been dropped for a simpler scheme, which uses the machine number to disambiguate declaration numbers. Hence the insistence on keeping sibling rivalry in check.

2.5. World Views

Each defined view of each world is a version of the world view object in the world. The version to use at any time is determined by the universe view that has been established by the referencing job.

A world view contains the *world state*, an *object table*, a *unit state table*, and a *dependency matrix*.

For fast access, the world state and object table are stored at a fixed offset in the world view. Pointers to the other components of the world view are stored at fixed offsets.

2.5.1. World State

The world state is indicated by the following flags:

loaded The Ada units of the world have been loaded. Demoting any of them from the coded state will cause the load module to be abandoned.

subsystem The world is a subsystem, subject to the more strict import regulations. Worlds that are not subsystems are called *casual* worlds, which are intended for casual uses such as users home directories.

persistent [TBD]

frozen The world cannot be changed when this flag is set.

2.5.2. World Object Table

The world object table is an array indexed by object number. Each entry is a variant record that contains either

- The version number of the version of the object that exists in this view of the world, or
- a null value, which indicates that no version of the object is visible in this view.

2.5.3. Unit State Table

The world unit state table is an array indexed by object number. The index range is constrained to the range of object numbers that correspond to Ada units in the world. Each entry summarizes information about the current state of the corresponding Ada unit in the view.

Each entry contains the following fields:

state A summary of the compilation state of the unit, which may be one of the following:

coded	The unit has been successfully coded and is ready to be loaded.
phase-1 coded	The unit has been attributed by the code generator.
installed	The unit has been successfully semanticized and can be coded.
parsed	The unit is syntactically correct, but contains portions that have not been semanticized.
unparsed	Portions of the unit could not be parsed.

declasse A bit, which when set indicates that forces external to the unit have conspired to cast doubt on the veracity of the current semantic interpretation of the unit; an imported declaration, against which this unit had been installed, has probably changed its meaning in a way that invalidates the current interpretation of the unit's semantics.

modified A bit, which when set indicates that since the last installation of this unit, declarations have been edited that may have changed the meaning of one or more declarations exported by this unit.

2.5.4. Dependency Matrix

The dependency matrix in a world view duplicates information in the Ada objects in the view. It is stored in the world view to accelerate obsolescence propagation, the ordering of unit promotions, and the computation of presentation state.[9]

The dependency matrix, DM, is a rectangular Boolean matrix that summarizes the semantic dependencies between the Ada units of a world and its imports. DM(J,K) is true iff unit J contains a semantic pointer to unit K. (Additional information about the nature of the reference from J to K is stored in the referencing Ada unit, J.)

Each row of DM corresponds to an Ada unit defined in the world. Each column of DM corresponds to an Ada unit that is visible in the world. Some of the columns of DM will

[9]For certain operations the transitive closure of the dependency matrix, called DM^*, is required. As an optimization, DM* could also be stored in the world view.

correspond to units that are imported into the world from other worlds.[10]

The symmetric portion of DM, which records dependencies between local units, is a bit matrix indexed directly by object numbers (one reason for insisting that the object numbers for Ada objects be contiguous and near zero). It will be used for quickly propagating obsolescence within the world view, quickly determining the proper order for installing and coding these units, and quickly computing the presentation state of units.

Note that the propagation and ordering algorithms access the DM along different axes. Record-locking can be used to reduce serializing on access to the DM, but only along one axis. Since record-locking is needed only for update access, each row of DM shall be a record. The dependency matrix is updated by the semanticist.

The asymmetric portion of DM, recording references from local units to imported units, is used only to propagate obsolescence to local units when refreshed imports are inconsistent with their previous versions. (See Section 2.4.)

Because the importing world has no control over the object numbers of the objects that it imports, a map must be used to implement the asymmetric portion of DM.[11] The domain of the map is the set of imported object ids.

The DM records only semantic dependencies. Additional coding dependencies may also exist; for example, when generics are implemented with a macro-expansion model. This information is target-specific and shall be recorded in the Ada unit or its attribute space.

---

[10]Note that references from imported units to local units are *not* recorded in the referenced world. This information would be useful only if the referenced world is unfrozen; it could then be used to obsolesce referencing units in the imported world when changes are made to declarations in the importing world. A number of reasons are offered for *not* doing this:

- Issues regarding access control and resource accounting arise when structures in one world must be changed because of development activity in another world.
- The locality properties of change analysis are considerably worse if several worlds must be consulted during the analysis.
- Developing against unfrozen worlds is inconsistent with the Rational Subsystems(tm) release paradigm.

Developing against unfrozen worlds is allowed, nonetheless, because this will facilitate sharing of programs and macros between the mythical casual users on the system. Run-time compatibility checking will keep such developers out of real harm, but there are sure to be surprises when programs once thought to be working are recompiled against specifications that have changed because they were not frozen.

[11]Because of the difficulty in implementing the map for the asymmetric portion of DM using the record-locking facilities of KKOM, it will *not* be used in the first release of Epsilon. When imports are refreshed, the asymmetric portion will be built as a temporary structure by scanning all units in the world.

2.6. Ada Objects

Each Ada compilation unit is an object of class Ada. As an object, it must be opened before it is accessed, has access control, and may be atomically modified under an action. There may be several versions of an Ada object.

Each version of an Ada object contains the following

Diana nodes — A tree-structured representation of the syntax and semantics of the Ada unit.

string table — Contains symbol representations and comments that appear in the tree.

source modification stamp — A unique id, which is assigned a new value each time the source of the unit is modified.[12]

external node table — An array associated with each declaration that is referenced by node numbers and contains the offset of the referenced node from the root node of its declaration.

declaration table — An array that is indexed by declaration numbers and contains offsets to the external node table for the corresponding declaration. An entry in the declaration table is null when no such declaration exists in the Ada object.

offset array — An array that is indexed by special values in structural links and external node tables and contains the oversized offset that should be the value of that entry. See Section 2.6.3.

unit table — An array indexed by unit numbers. A unit number is a small integer uniquely identifying (to the referencing unit) each imported unit. Any unit referenced by a Diana pointer is included in the imports. Each entry in the unit table includes

- *import key* — A compatibility key that indicates which declaration numbers of the imported unit are referenced. Also includes the object id of the referenced unit.
- *usage map* — A map from each referenced declaration number of the imported unit to a description of the ways the declaration is referenced in the Ada unit.

---

[12] Who uses the source modification stamp?

export key A compatibility key indicating which declaration numbers are defined in this unit.

top declaration database A map from declarations to sets of declarations used by Change Analysis. Not present in secondary Ada units. (See Section 2.6.5.)

2.6.1. Diana Pointers

A **Diana.Tree** value is a memory address, composed of a segment number, and segment offset. A **Diana.Tree** value is an Ada access type and cannot be stored in permanent data structures such as files.

A **Diana.External_Reference** is a record composed of an object id, a declaration number, and a *node number*. A **Diana.External_Reference** can be stored in permanent data structures; however, it includes object ids that are meaningful only with respect to a single machine, and it can only reference selected nodes in exported declarations.

Diana will export functions for translating between **Diana.Tree** values and **Diana.External_Reference** values. In addition, a select set of Diana selector functions will be defined to return **Diana.External_Reference** values. These selectors will be more efficent for applications that need to compare the selector value with one in hand rather than actually following the selector to the node it references.

A **Diana.Version_Reference** is a record composed of an object id, a version number and a segment offset. Values may be stored in permanent data structures, but may access only one version of an Ada object. A value can reference any node in that version, however. Editing an Ada object that is referenced by a **Diana.Version_Reference** pointer may invalidate that pointer.

Diana will export functions for translating between **Diana.Tree** values and **Diana.Version_Reference** values.

2.6.2. Diana Nodes

In addition to making it possible to copy Diana trees without recompilation, the Epsilon design is also focusing on reducing the size of a Diana tree. Gamma trees are ten times their source representation on the average. Epsilon trees should be no more than five times their source representation.

As in Gamma, a Diana node is a variant record discriminated by the **Diana.Kind** of the node. Each variant contains a set of attributes and structural links appropriate to the kind of node. Only the common attributes are allocated in the variant portion of a node. An auxiliary attribute list is used to attach the less-frequently appearing attributes to a

node when needed.[13]

Unlike Gamma, Phase 1 code generator attributes will be implemented the same as semantic attributes. The more common ones (across all targets) will be allocated in the variant record that defines Diana nodes. The others will be attached to the node's attribute list.

Rational-specific Diana nodes will be defined to represent comments and constructs that could not be parsed.

2.6.3. Structural Links

Within a Diana node, a structural link is an offset. The offset can be relative to the node itself, or relative to the start of the segment. Node-relative offsets will use less space, but would have to use an escape mechanism more often. The offset can be node-relative in constructs that are always parsed as a unit, for example within simple declarations and statements. Where subtrees can be parsed independently of the parent, the structural offsets in the parent will have to be segment-relative. (When a component subtree is edited, the new tree may be built an arbitrary distance from the original parent node.)

One bit of each type of structural link field is reserved to signal an overflow. When this bit is set, the rest of the link field is interpreted as an index into the unit's offset array. The indexed position in the offset array contains the actual (segment relative) offset. The offset array is used only when the correct offset is too large to fit in the structural link field of a node. The size of the structural link field is chosen to make the use of the offset table a rare event.

Where possible, structural links will not take up space in a Diana node, at all. For example, since the first child of a **Dn_Var** node is always a **Dn_Id_S**, the **Dn_Id_S** can be incorporated into the **Dn_Var** variant and thus eliminate the need for a **Diana.Child1** pointer. Many node kinds, like **Dn_Var**, have children that must be list headers of a specific kind. The **Diana.Tree** value for an incorporated node will have to be "cons'ed up" from the **Diana.Tree** value for the parent node and the fixed offset to the start of the child node.

So that computation of the parent of a Diana node is blazingly fast, most nodes will have a structural link that references its immediate parent. The node kinds that are incorporated into their parent would not have such a field, of course. Unlike the Gamma implementation, even the nodes on structural lists will have an immediate parent link. Structural lists (**Diana.Seq_Type**) will therefore be the more conventional

[13] Target code generators will have to be able to attach attributes of arbitrary type to this attribute list.

two-cell lisp node. Each cell will contain a segment-relative offset.[14]

2.6.4. External References

Within a Diana node, a **Diana.Tree**-valued attribute is stored (logically) as a variant record. The boolean discriminant indicates whether the pointer is an internal reference or an external reference to a node in an exported declaration. For an internal reference, the rest of the pointer is a segment offset within the same object. For an external reference, it is a triple composed of a unit number, a declaration number, and a node number. Even within the defining unit, all references to exported declarations are represented as external references.

The special requirements for maintaining consistent references to exported declarations are accommodated by the Ada object design through the use of external node tables within an Ada object and the use of node numbers in external references rather than node offsets. After an exported declaration has been semanticized and its declaration number has been assigned, an external node table is built for it. The declaration table entry for the exported declaration points to the external node table.

Each entry in the external node table corresponds to a node of the declaration that can be referenced from outside the Ada unit. Externally referenceable nodes of the declaration are assigned to slots in the node table in the order in which they are visited during a structural walk of the declaration. Thus, identical node numbers will be assigned to corresponding external nodes for any two declarations that are assigned the same declaration number.

One bit of each external node table entry is reserved to signal an overflow. When this bit is set, the rest of the entry is interpreted as an index into the unit's offset array. The indexed position in the offset array contains the actual offset. The offset array is used only when the correct offset is too large to fit in the node table entry for a node. The size of the node table entry is chosen to make the use of the offset array a rare event.

2.6.5. Top Declaration Database

A *top declaration* is a non-overloadable declaration that is hidden by at least one overloadable namesake. A special top declaration, called a *placeholder*, is created when there is no user-defined top declaration and a relation must be recorded.

The **Subordinate_To** relation and the **Sees_Used_Namesake_Via_Use_Clause** (**Sunvuc**) relation are recorded against top declarations. They must be implemented differently in

---

[14]Representations of Diana.Seq_Type are being investigated that will make traversal of lists more efficient. The current proposal is to cdr-code the structural lists and doubly link the blocks of the encoding. The parent link of a list item would reference the list cell, which in turn would reference the Diana list header node.

Epsilon to solve the Gamma object management problems with placeholders. The domain of these relations includes the set of placeholders. Furthermore, these are the only relations registered against placeholders.

In Gamma, one placeholder is used for each unique symrep in the system that needs a placeholder. All units that are related to the placeholder are contained in one set, but Change Analysis (the only user of this dependency information) always and immediately prunes the set of related units to just the units that can see the point of insertion or withdrawal it is analizing. In the worst case, the set of dependents of interest to Change Analysis would be a library unit and all of its secondary units.

Based on this last fact, the **Subordinate_To** and **Sunvuc** relations will be stored in a *top declaration database* contained in each Library unit's Ada object rather than in the dependency matrix or usage map. The top declaration database is created when each version of a library unit is created. The database continues to exist as long as the version of the library unit exists. Relations may be added, deleted or modified whenever the library unit or its secondary units are installed or incrementally modified. When the library unit is demoted to source, the relations can be destroyed, along with the database, if necessary.[15]

Unlike the Gamma implementation, placeholder ids will be created as actual Diana **Def_Ids** in the same space as the top declaration database. The placeholder id is created in the top declaration database in the library unit of the referencing object. When a **Subordinate_To** or **Sunvuc** relation is entered into the dependency database against a declaration, the dependency database manager uses either the dependent object or the declaration (whichever is more convenient) to locate the top declaration relations database into which the relation is to be stored.

2.6.6. Garbage Control

To reduce the amount of garbage generated while semanticizing a **Diana.Tree**, certain nodes that are frequently transformed by the semanticist will be implemented such that they can be transformed in place, by changing a kind field, for example. In addition, the semanticist will be capable of semanticizing transformed trees the same as pristine parse trees. This capability is needed for validation mode and also eliminates the need to keep the discarded parser-generated nodes in the tree.

The change to the semanticist is to rely less on the node's kind to direct its analysis. For example, when it sees a **Dn_Indexed** node it can no longer automatically assume that the

---

[15] Because the top declaration database may be opened for update during the installation of the unit's secondary units, the Diana tree for a library unit specification will not be accessible while its body or one of its subunits is being installed. This will serialize installation and coding within the view, but not significantly. The editor will be able to display the unit because the pretty printed image of the tree is stored in a separate object, which does not need to be accessed by the semanticist.

node is an array reference. It must treat it as an "apply" node and perform overload resolution; changes external to the node since it was last semanticized could easily have converted the reference to be a function call, for example.

Because of the abundance and variety of dynamically growing objects in an Ada object, some form of memory management within the object will be used. Compaction should rarely be needed.

Using memory management the edit-in-place paradigm seems to generate no more garbage than the current incremental compilation. Because no separate spaces are created for insertions, maybe even less garbage is generated.

In the Gamma implementation, all incremental editing occurs in a separate space. When the user promotes that separate space, the parent tree is opened for overwrite, the new parse tree is copied to the end of the parent space, and there it is semanticized in place. In the Epsilon model, the editor always builds new trees from discarded nodes or from fresh space at the end of the (parent) segment and so there is no need to do the copy (or build a separate space) before the additions are semanticized.

2.7. Image Objects and Shape Data

To facilitate the displaying of Ada objects, a separate *image object*, named *object-name*'**Image**, is associated with each Ada object. The image object contains a pretty printed text image of the associated Ada unit organized as a tree of line images. When the Ada unit is to be viewed or edited, the image object is opened and the stored lines of the image are copied to the screen. The Ada object itself must be opened only if the image is modified or if the user makes an object selection on the image. If both objects are opened, they are opened under the same action.[16]

As the user edits the unit, changes are recorded in the image object. When the image is formatted, the lines that have been changed are fed to the parser, which builds, in the Ada object, a new Diana tree for the current content of those lines. If the parse is successful, the new Diana tree is fed to the pretty printer, which generates a new set of lines that contain the pretty printed image of the parse tree. These lines then replace the original lines in the image object.

The editor never parses anything smaller than a statement or declaration. If the user edits just a subcomponent of a declaration (or statement), the region around it is enlarged line-by-line until it includes a full declaration and those lines are passed to the parser.

---

[16]The Ada object and its image object are redundant. If one should get damaged, the other can be used to rebuild it.

To facilitate associating an image selection with a Diana node, the Diana nodes that can be the root of these incrementally parsed subtrees have a *line count* attribute, which indicates how many lines the pretty printed image of the subtree consumes.[17] The editor uses the line counts to quickly locate the statement that contains the selection and then walks the subtree in image order, counting the (non-blank) characters of the image consumed by each node, until the selection is found.[18]

Comments will be attached to Diana nodes as in Gamma. In addition, in Epsilon there is a Diana node kind reserved specifically for representing comments. Because of the separation of the image and Ada objects, the comment images cannot be shared.

2.8. Intersubsystem Visibility

The visibility that one subsystem has to the units of another subsystem is determined by the world *import* and *export* mechanisms of Epsilon.[19] This mechanism is available in both subsystems and casual worlds.

In Epsilon, all views of a subsystem are both Load Views and Spec Views in the Gamma sense; a subsystem can be compiled against any view of another subsystem, and if it is complete, any view can be loaded and executed. The limited intersubsystem visibility provided by Gamma Spec Views is provided by *export objects* in Epsilon. Units from a subsystem are made visible to a client subsystem by *importing* into the client an export object defined in the subsystem. An export object specifies the library packages and subprograms that are visible to the importer. A subsystem may have several export objects, each of which may make visible to an importer a different subset of the world's library units.

When a world is created, a default export object is automatically established, which is used when the importer doesn't name an export object specifically. For subsystems, the default export object is empty. For casual worlds, the default export object specifies all library units. The export object to be used as the default can be changed by the user.

Each export object may have more than one version. The world view specified at the time of the import determines which version is imported.

---

[17] About 10% of the nodes in a Diana tree will carry these line counts. The line count attribute must be large enough to store the size (in number of lines) of the largest possible Ada unit.

[18] In most cases the distance between statements or declarations will be one or two lines, so the time spent computing the offset to the selection will be negligible. However, attempting to make a selection in a large aggregate or long parameter list might take a noticible amount of time. This is a price we are willing to pay.

[19] Don't confuse these subsystem-level notions with the importing and exporting of Ada declarations. The latter notions are not presented to users whereas these are, so there is good reason to reuse the simple terms, even at the risk of confusing readers of this document.

The visibility to library units that is established by the import operation is solely for the purpose of resolving the names that appear in **with**-clauses in the importing subsystem. Once a **with**-clause has been resolved, the semanticist depends solely on the view mechanism to locate Diana trees. Because the semanticist deals with the full **with**-closure of any **with**'ed unit, Diana pointers to units in other subsystems can be established even though the referenced unit has not been imported.

The import operation establishes a correspondence between simple Ada names and Ada objects, not specific versions of Ada objects. The universe and world views established for a compilation determine the versions that will be referenced when the **with**-clauses are resolved.[20]

A subsystem can import an export object in its entirety or just selected units from the export object. Exported units can also be renamed during the import operation. Finally, the import can either be *binding* or *refreshable*. If the import is binding, the set of units made visible by the import is fixed as the set of units specified by the export object at the time of the import operation. If the import is refreshable then each time a view of the imported subsystem is included into the importing subsystem, the set of imported units from that subsystem are updated to reflect the current version of the export object.

2.8.1. Link Packs

A unit that is to be **with**'ed by units in a world must be referenced by the world's link pack regardless of whether the referenced unit is in the world or imported into it. The units are **with**'ed using the simple names in the link pack. The simple names in the link pack must all be distinct. Aliasing or renaming local units is not permitted.

Users will not have direct access to the link pack as they do in Gamma. The principal use of the link pack is to accelerate the resolution of names in a **with** clause. The world import operation and the directory create and destroy operations manipulate the link pack for the user.

A link pack may have several versions. Each world view identifies the version of the link pack to use for its Ada units.

2.8.2. Closed Private Parts

With respect to closed private parts, the user interface will be basically as in Gamma; the default is closed private parts for exported packages, but the subsystem designer can use the **Open_Private_Part** pragma if desired to override the default. In addition, each importer will have the option of importing a unit with a closed private part, regardless of the exporter's designation.

---

[20] The configuration management operations of *release* and *include* build the universe and world views used in this resolution.

Unlike Gamma, whether a package's private part is open or closed has little effect on the compilation of that package. An attribute will be set on each private type to indicate whether its completion is visible to code generation, but otherwise the tree will be semanticized as a standard Ada unit. The attribute will affect the way that the semanticist sets semantic pointers that reference the private type and the code that is generated for references to the private type.

Each private type declaration defines two separate exported declarations; one for the visible type and one for the full type. If the private part is open, most semantic pointers will point to the full type declaration. If the private part is closed, all external semantic pointers will reference the visible declaration only, contrary to the formal definition of Diana.

Similarly, a deferred constant and its corresponding constant declaration have different declaration numbers. If the private part is open, most semantic pointers will point to the constant declaration. If the private part is closed, all external semantic pointers will reference the deferred constant declaration only, contrary to the formal definition of Diana.

Code generators can easily detect when a private type is closed since all type spec-valued attributes will point to the **Dn_Private** node, not the full type as prescribed by Diana.[21]

Since, closed private parts are really only "closed" to the importer and not to the exporter, creation of a view that may be frozen and exported before implementation details have been resolved is a bit more contrived in Epsilon than in Gamma. Since the private part is never ignored when the package spec is compiled, the developer must provide a completion for every private type in the package. If the private part is closed, how the type is completed is important only to the secondary units of the package; so, if no body is present in the view, a **new integer** would work as well as any completion. When the body is added to the view, however, or if the private part is open, then the completion must be correct for the application.

2.9. Ada Attribute Objects

Several object classes will be defined for objects that contain additional attributes of Ada units.[22] Each version of each attribute object is associated with one version of one Ada object. The name of an attribute object is the name of it associated Ada unit, qualified with the object class name, *e.g.*, *object-name*'**Defunct_Cg_Attributes**.

---

[21] At this time, it is not clear how the semanticist will detect that it has a reference to a closed private type.

[22] No such spaces are defined by the current Epsilon design.

An attribute object contains an attribute map and a heap of attribute values. The attribute map is a map from segment offset to segment offset: it maps from the offset of a Diana node in the associated Ada unit to the offset of an attribute value in the attribute value heap.

There are no pointers directly into an attribute space. Any attribute value may be reached via its associated Ada unit only. A reference to an attribute in another unit is stored as a **Diana.External_Reference** to the node corresponding to the desired attribute.

An attribute object may reference the associated Ada unit (or any other Ada unit) using the type **Diana.External_Reference**.

New classes of Ada attributes will be defined, both by us and by customers writing new tools.

2.10. Compatibility Object

Associated with each Ada unit, ***Foo***, is a *compatibility object*, named ***Foo*'Compatibility**, which contains the information required to assign a unique declaration number and *declaration stamp* to each unique exported declaration of the Ada unit. The compatibility object includes a *declaration allocator* and a *declaration map*. There is only one version of a compatibility object.

The declaration allocator consists of the *high declaration number*, an independent *declaration counter*, and a free-list of previously allocated declaration numbers that may be reused. When a declaration number must be assigned, the declaration counter is incremented. If the list of reusable numbers is not empty, one of those is assigned to the declaration and removed from the list. If the list is empty and the high declaration number is less than the maximum allowed declaration number, it is incremented and assigned to the declaration. If the high declaration number is at the maximum, declaration numbers must be reclaimed and one of those is assigned to the declaration. (See Section 3.7.)

At any given time on a single machine, an object's assigned declaration numbers correspond uniquely to the exported declarations of that object that exist in some extant version of the object on the machine. The declaration stamp, together with the object's creation stamp, disambiguates declarations that have been assigned the same declaration number because the declaration number was prematurely reused or because the declarations were created along separate development paths. The declaration stamp consists of the machine number, boot number, and declaration counter values at the time and place the declaration was created. See Section 3.6.1 and Section 3.7 for a complete discussion of the use of the declaration stamp.

The declaration map maps a declaration (**Diana.Tree**) to its assigned declaration

number and stamp. The declaration map is structured as a hash table on the declaration name (or some other simple hash function). Each hash chain is a list of candidate *declaration information records*. Each record contains the declaration number, its declaration stamp and a canonical representation of the declaration (including lexical, structural, and semantic information (as **Diana.External_Reference** values)).

In order to introduce a new declaration into a compatibility object, a job must obtain a write lock on the compatibility object. This provides useful synchronization properties.

2.10.1. Canonical Representation

The following nodes may be the root of an exported declaration:

```
constant
var              (except as a record component)
number
type             (except as a generic parameter)
subtype
subprogram_decl  (except as a generic parameter)
subprogram_body  (with no separate subprogram_decl)
package_decl
task_decl
generic
exception
deferred_constant
```

Exported declarations may not be contained in a **Dn_Generic_Param_S**, a **Dn_Param_S**, a **Dn_Dscrmt_Var_S**, an **Dn_Enum_Literal_S**, a **Dn_Record**, or an **Dn_Inner_Record** thus adding, deleting or modifying a generic formal declaration changes the entire generic declaration. Adding, deleting, or modifying a subprogram parameter declaration changes the entire subprogram declaration. Adding, deleting, or modifying the declaration of a descriminant or component of a record changes the entire record type. Adding, deleting, or modifying an enumeration identifier changes the entire enumeration type.[23]

For declarations of variables the canonical representation includes all nodes of the Diana tree except the optional object def. For declarations of packages, and tasks, only the defining id node is included. (The declarations nested within a package, generic packge or task are recognized as separate declarations.) For subprogram bodies without separate subprogram specifications, all nodes of the header and designator are included, but the **Dn_Block** and **Dn_Stub** nodes are excluded. For generic declarations the id, all nodes of the **Dn_Generic_Param_S**, and the **Dn_Generic_Header** node are included. For all other declarations, all nodes of the declaration are included in the information record.

---

[23] Is this too restrictive? We will probably have an escape mechanism, which allows the user to assign his own declaration numbers in such a way that obsolescence propagation still works.

The canonical ordering of nodes within an information record is the same as the textual ordering of those nodes according to the syntax of Ada. The representation for a node includes only its **Diana.Lx_Symrep** (if any) and **Diana.Sm_Defn** attribute (if any). If a node has neither of these attributes, only its **Diana.Kind** is included.

The information record also includes the canonical representation of any pragmas or representation clauses that apply to the declaration.

For two records to match, the represented nodes must match according to the rules for header-body conformance defined by the LRM.

- A numeric literal can be replaced by a different numeric literal if and only if both have the same value.

- A simple name can be replaced by an expanded name in which this simple name is the selector, if and only if at both places the meaning of the simple name is given by the same declaration.

- A string literal given as an operator symbol can be replaced by a different string literal if and only if they both represent the same operator.

Thus two declarations match if they appear in the same context and, apart from comments and the above allowed variations, both declarations are formed by the same sequence of lexical elements and corresponding lexical elements are given the same meaning by the visibility and overloading rules.

2.10.2. Assigning Declaration Numbers

When a declaration number needs to be assigned to a freshly-semanticized declaration, the declaration is hashed and then the hash chain is searched. A declaration information record is formed for the candidate new declaration and is compared to the information records on the hash chain. (Additional hashings or heuristics can be employed in the search to avoid wasting too much effort building unnecessary information records.) If a match occurs then the new declaration uses the old declaration number. If no match occurs then a new declaration number is assigned.

For library units, at installation all declarations are assigned declaration numbers and entered into the declaration table. For library secondary units, declaration numbers are only assigned where there are subunit stubs or macro-expanded bodies that could possibly introduce external references.

If subunit stubs or macro-expanded bodies are later introduced into a secondary unit, declaration numbers will be assigned to the declarations that are visible at these insertion points. Local references must be converted at this time as well to use these new declaration numbers.

The semanticist assigns all declaration numbers.

2.11. The Size of Things to Come

The sizes of the various components of the Epsilon are summarized here.

OBJECT	Bits	(Range)
World Number		
Object Number		
Version Number		
Class		
Machine Number		
Boot Number		
Unique Id		
Object Id		
Creation Stamp		
Diana.Tree		
Unit Number		
Declaration Number	14	(0..2**14-1)
Node Number		
Declaration Counter		
Declaration Stamp		
Diana.External_Reference		
Diana.Version_Reference		
Semantic Reference		
Node-Relative Structural Link		
Segment-Relative Structural Link		
Line Count		
Modification Stamp		

3. Operations

A number of common Epsilon operations are discussed in this section.

3.1. Opening Objects

In the same way that Diana currently maintains a map from a task to a heap in which to allocate Diana objects for that task, in Epsilon, Diana will also maintain a map from client tasks to action ids. This action id will be used to implicitly open objects used in the decoding of `Diana.External_Reference` values and `Diana.Tree`-valued attributes.

Following an interunit pointer may require opening the referenced unit. Following an interworld pointer may require opening a world view of the referenced world. Each object that Diana opens under a given action id is remembered in an *open object cache* associated with that action id. Before decoding the object id, the open object cache is consulted to see if it is already open, in which case, the segment number of the open Ada object is retrieved from the cache.

Thus, if the desired object has already been opened under the current action, decoding will be fast: it is only neccessary to look up its memory address in the current open object cache. If, however, the object is not open, then it must be opened using the normal KKOM supported open mechanism (relative to the job universe view).

Any object that is opened as a side effect of following a pointer is opened for read access only. Any attempt to modify such an object will fail. In those cases where modifying an object is required (*e.g.*, when promoting an Ada unit), the tools (*e.g.*, the front end) explicitly open the appropriate objects for write access before modifying them.

3.2. Following a Diana Pointer

A `Diana.Tree` is a full memory address (segment number, and segment offset). The value to which it points can be retrieved with minimal effort.

Selecting a lexical, structural or embedded attribute of a Diana node requires adding a field offset to the node address, and reading the field value at that address. Constraint checking is performed to determine that the desired attribute is appropriate for the given node kind.

Selecting an auxiliary attribute of a Diana node requires searching the node's attribute list for a key value associated with the attribute. Associated with the key on the list is the value of the attribute.

For many structural attributes, the field value evaluates to an absolute or relative offset (depending on the node and the attribute) within the same unit. The offset can simply be added to the node or segment address to form the attribute value. An extra level of

indirection is added when the desired offset exceeds the capacity of the field in which it is supposed to be stored. The field actually contains an index into the unit's offset array, which in turn contains the desired segment offset.

Following a **Diana.Tree**-valued attribute when it points to a node in another Ada object is a complex operation that requires microcode assistance for acceptable performance. In this variant, an external reference is composed of a unit number, a declaration number, and a node number. The unit number is used to extract the object id of the referenced unit from the unit table of the referencing object. This object id is decoded using the view mechanism, and the corresponding unit is opened to obtain the segment number for the referenced Ada object.

The declaration number is used to index into the declaration table of the referenced unit, obtaining the segment offset to the node table for the declaration. The node table is indexed by the node number to obtain the segment offset to the desired node. Combining this offset with the segment number completes construction of the **Diana.Tree** value to be returned as the value of the attribute.

Diana.External_Reference values are converted to **Diana.Tree** values in a similar fashion. Since the **Diana.External_Reference** value contains an object id, a unit table is not consulted in the first step. From then on decoding is the same.

3.3. Setting a Diana Pointer

Setting a **Diana.Tree**-valued attribute basically involves reversing the above operation, converting a **Diana.Tree** value into the permanent representation for storing in the Diana node.

If the referenced node is in the same unit and is not an external node, an absolute offset is stored in the referencing node.

If the referenced node is an external node (in the same unit or in another), a (unit number, declaration number, node number) triple is stored in the referencing node.

The unit number is computed by extracting the object id from the root of the referenced unit, and converting the object id to a unit number. The conversion uses a *unit map*, which is the inversion of the unit table. The unit map is constructed during semantic analysis (by Diana) as part of assigning unit numbers. The unit table must also be kept current during semantic analysis so that semantic references can be decoded properly for the semanticist.

For incremental compilation the unit map must be initialized from the unit table.[24]

---

[24] **Should the unit map be a permanent component of the Ada object?**

The unit map is temporary Diana state that is initialized (empty) when setting the current Diana unit (for all allocators and updates).

The declaration number is computed by using **Diana.As_Parent** to reach the enclosing declaration node, which stores the declaration number.

The node number is computed by searching the declaration's node table for an offset equal to the offset in the **Diana.Tree** value.

3.4. Obtaining an Attribute Value in an Attribute Space

The value of an attribute associated with a Diana node is obtained as follows:

1. The associated attribute object is found by resolving its name (*unit-name'attribute-class*).[25]

2. If the attribute object is not currently open, it is opened.

3. The segment offset of the referencing Diana node is fed into the attribute map (stored in the attribute object), which yields the segment offset of the attribute value.

4. The segment offset of the attribute value is combined with the world number and segment number of the attribute object, to yield the desired memory address referencing the attribute value.

3.5. Determining Dependencies

To be documented.

3.6. Archive and Restore

3.6.1. Cloning an Object

Any object may be cloned from one machine to another via any available media such as a magnetic tape, deck of cards, or sophisticated network communication protocol. Worlds may be cloned to any machine, but other kinds of objects may be cloned only to machines that contain a generation of the object's world. That is, objects that are not worlds are cloned to a generation of their world, not to a machine.

The cloning process consists of two major steps: *archive* and *restore*. The archive

[25] An attribute space list is not mainatained in Epsilon.

operation moves the object and associated data off a machine to the exchange media. The restore operation moves the object from the exchange media onto a machine. Although the machine from which an object is archived is usually different from the machine to which it is restored, this is not a requirement. Furthermore, the time an object remains archived on the exchange media is immaterial to the restore operation.

From machine to machine, no attempt is made to synchronize the assignment of world numbers, but across generations of a world, components of the world are numbered consistently except possibly where numbers have been prematurely recycled or development of the world has occurred concurrently on different machines. Thus, when an object is restored to a new environment, each world number in the object must be converted to the world number for the corresponding world in the new environment. Other components of object references, such as object numbers and declaration numbers, do not have to be converted, but because these numbers may have been recycled it must be verified that in the new environment the numbers in the object still refer to the same components they did when the object was archived.[26]

The following data must be archived to clone an object:

- The cloned object.
- The world number and world creation stamp for the world that contains the cloned object and for each world referenced by the cloned object.
- The world number, object number, and object creation stamp for the cloned object and for each object referenced by the cloned object.
- The world number, object number, declaration number, and declaration stamp for each exported declaration defined or referenced by the cloned object.
- The unit state table entry for the object (if any).
- The row of the dependency matrix that corresponds to the object (if any).

If several objects are being archived at the same time, the above information about referenced worlds, objects and declarations can be pooled into one structure for all the objects. For an Ada object, the worlds, objects, and declarations referenced by the object are recorded together in the unit table of the object. To archive other objects, the location of this information must be made known to the archiver. The creation stamp for any object is available from KKOM. The declaration stamps are obtained from the compatibility objects for each declaration. The compatibility objects do not have to be copied; they will be reconstructed as the objects are restored.

---

[26] We are also considering providing the ability to reassign object numbers during the recovery process.

Restoring archived objects onto a machine involves the following steps, all of which are performed under one action:

1. Using the archived world creation stamps and the world creation stamps for worlds on the target machine, a *world number map* is constructed that maps world numbers in the archived objects to the corresponding world numbers for the target machine.[27]

2. Locate an unforzen view for each world that will be modfied. One always has the option of restoring objects into an existing view on the target machine or creating a new one to restore into. If an existing view is used or if only some of the view's objects are restored, the import and export keys of the restored objects must match the corresponding export and imports keys in objects in the view that were not restored. The keys must match exactly, not just be compatible.

3. Using the world number map, locate on the target machine each object referenced by the archived objects and compare creation stamps. They should be the same.

4. Using the world number map, locate on the machine the declaration information record for each declaration defined by the archived objects and compare declaration stamps. They should be the same. If no corresponding declaration exists on the target machine, add it to the appropriate compatibility object.

5. Again, using the world number map, locate on the machine the declaration information record for each declaration referenced by the archived objects and compare declaration stamps. They should be the same.

6. Copy each archived object onto the target machine as a new version of its corresponding object. Create a new object is there is no corresponding object on the target machine. Update the world view to reflect this new version.

7. For each Ada object, copy the archived unit state table entry and dependency matrix row into the updated view.

8. Convert the world numbers in each cloned object according to the world number map.

9. If the new host world is a surrogate world, the objects are frozen.

---

[27] The target machine may maintain a map from creation stamp to world number, or it may be necessary to search all worlds for matching creation stamps.

If there is no world on the target machine to hold an archived object, then the transfer fails: a world must be created before the object can be transferred into it.[28] If there is no world on the target machine that is referenced by an archived object, then a dangling reference is noted for possible future resolution (after the desired world is transferred to this machine).

3.6.2. Cloning a World

A world is cloned to another site by creating a new generation of the world at the new site and then cloning the objects of the world to the new generation.

to be continued...

3.6.3. Moving Active Development

Moving a world involves the following steps:

1. The old world (on the old machine) is frozen.

2. The entire state of the world is cloned to the new machine. New objects, and new versions of existing objects, may be created in the world. Existing objects, however, may not be modified.

Freezing the parent world is important: if development continues in this world, then its numbers will collide with those of its descendants. The tools must enforce the invariant that only frozen worlds may be moved, and that a parent world may not be unfrozen. A tape carrying a moved world should contain some special mark indicating this fact. If a tape is lost (destroyed), it is OK to make a new tape from the frozen parent world, and/or to create a child world on the same machine as the parent, but it is not OK to unfreeze the parent.

Moving worlds may give rise to various inconsistencies, which are detected and corrected as follows:

- Sibling worlds are created, *e.g.*, the tape containing the world state is moved onto two machines. If this happens, and development is done in both siblings, then the numbers allocated in the two siblings may collide. If, at some later date, someone tries to merge objects from both siblings into a single surrogate world, the collision will be detected, and the merge will fail. Detection uses declaration stamps:

Creating sibling worlds is discouraged: development in a world should be done on one

[28] Alternatively, a new generation of the containing world could be created. By searching the machine for old clients of the world, a world number to use might be determined. If that world number has been reused, all old clients would have to be revised.

machine at a time. If siblings are created, then there is a hight probability that the two developments in them cannot be merged: objects or declarations created in different siblings might not merge into a single surrogate world because of numbering conflicts. The move algorithm described above does not create siblings.

Moving a world is a good opportunity to throw out garbage, *i.e.*, old versions, old objects, and old views which are no longer needed. Some judgement is required to decide what is garbage and what is not: the mover of the world should keep in mind that there may be other software somewhere which depends on the objects.

3.6.4. Imperfect Recovery

The state of one or more objects may be stored on tape, *e.g.*, by the system backup utility, and later recovered onto the same or a different machine. The special cases of releasing objects to surrogate worlds, and moving a world from one machine to another, are treated above. Here we consider the case of recovering objects which have not been used in a long time.

An entire world may be archived, or a view (with all the objects in it), or a single object. The information which must be stored on an archive tape is the same as the information required to copy objects between machines. See Section 3.6.1.

Archived objects are recovered as follows:

1. If the world from which the objects were archived still exists, or a generation of that world exists, then it may be possible to recover the objects into that world. First, the object numbers and declaration numbers of the objects on the tape must be examined to see if any of them have been reclaimed. If there are no reclaimed numbers on the archive tape, then the objects can simply be recovered, using the algorithm described in Section 3.6.1.

2. If an archived object has a reclaimed number, *e.g.*, it has the same object number but a different creation stamp than an existing object, then that object cannot simply be recovered. There are several alternatives, each of which makes sense in a different situation. The user of the recovery tool will have a choice:

    - Create a sibling world to contain the archived object. This world will have the same creation stamp as the current world, but a different generation stamp. This alternative preserves the assignment of numbers to objects, which may be useful if there are dependencies thereon (*e.g.*, debugger tables).

    - Assign the recovered object new (currently unused) numbers. Any recovered tables (*e.g.*, view object tables, compatibility keys, etc.) which depend on the numbering system can be permuted to use the

new numbers. This alternative allows archived objects to be merged with current objects.

- Destroy existing objects which use the new numbers, and replace them with archived objects. This is useful for restoring a copy to a surrogate world.

- Don't recover the archived object.

3. If the world from which the objects were archived doesn't exist, and no identifiable descendant of that world exists, then a world must be created to contain the recovered objects. The world is assigned some convenient world number, perhaps chosen because a client already refernces that number. The world and any objects in it may be recovered using the algorithms described in Section 3.6.1. The resulting world is a surrogate world, *i.e.*, before it may be modified it must be assigned a new generation stamp.

3.7. Reclamation

It is desirable to be able to reuse world numbers, object numbers, and declaration numbers. In each case, once the range of numbers has been used, no new entities of the type designated by the number can be created unless values that designate defunct entities can be reclaimed. It is not practical to make the number space arbitrarily large, since there are tables whose size is linear in the size of the number space.

If it were possible to determine unquestionably that a number designates a defunct entity and that no other objects have dangling pointers to that defunct entity, reclaiming the number would be no problem. But, in fact, we cannot reasonably expect to keep track of all copies of an entity and its referencers as they are archived to musty vaults and/or restored to arbitary machines throughout the known universe. Consequently, we will implement reclamation algorithms that will have a low probability of recycling numbers prematurely and we will implement mechanisms to detect when a number has been recycled prematurely.

In all cases, the mechanism associates a unique id with each of the assigned numbers. The unique id is much larger than the assigned number so that its uniqueness can be quaranteed, but it is only stored in a few places, including the object to which the number refers. The numbers are validated at approprate times by comparing these stored unique ids.

3.7.1. Reclaiming Object Numbers

Each object is assigned a creation stamp (unique id) when it is created. The creation stamp is stored in the object. Object creation stamps are unique across all space and time. All versions of an object have the same creation stamp. All clones of the object have the same creation stamp.

Each object id contains a hash of its creation stamp. Whenever an object id is resolved, the referencing creation stamp is compared with the referenced creation stamp for equality. If they are not equal, the reference fails.

An object number may be reclaimed if no object is currently stored at that number. This can be determined by examining the view-independent object table maintained by KKOM for each world. There may be dangling references to the object number in other worlds, or on other machines, or on archive tapes, but they will (probably) be caught when the new object's creation stamp does not match the hash in the referencing object id.

It may be possible to fix a broken reference, if the object with the desired creation stamp exists at some other object number. This situation may arise if an object is destroyed and later (after its original object number has been reclaimed) recreated, either by intermachine transfer or by restoration from archive. The object can be found by searching all object numbers, or by checking a map from creation stamp to object number.

3.7.2. Reclaiming World Numbers

Each world is an object and is thus assigned a creation stamp (unique id) when it is created. The creation stamp is stored in all generations of the world object. World creation stamps are unique across all space and time.

A world number can be reclaimed if no world is currently stored at that number. This can be determined by examining the KKOM world table. There may be dangling references to the world number, in object ids that referenced objects of the world, but these obsolete object ids will be detected when they are resolved as discussed above.

It may be possible to fix a broken reference, if the world with the desired creation stamp exists at some other world number. This situation may arise if a world is destroyed and later (after its original world number has been reclaimed) recreated, either by intermachine transfer or by restoration from archive. The world can be found by searching all world numbers, or by checking a map from creation stamp to world number.

3.7.3. Reclaiming Declaration Numbers

For declaration numbers, storing a unique id (or even a hash of it) with each occurrence of these numbers isn't practical, since there are too many occurrences. Every compatibility key would have to contain the unique id of every decl in its declaration set. Every interunit Diana pointer would have to contain the unique id of the node it pointed to.

Consequently, when an object is copied onto the machine, either from an archive tape or from another machine on the network, the declaration numbers it uses must be

validated. The declaration stamp for each declaration number in the copied object is copied with the object. If this declaration stamp doesn't match the declaration stamp associated with the corresponding declaration number on the target machine, the copy will fail.[29] If the object is an Ada unit, the conflict can be resolved by recompiling the unit from source.

The system supports a local expunge operation, which reclaims all declaration numbers for an object that are not referenced by any object on the machine.[30] Given a methodology which retains every important spec view online, this will provide reliable and efficient operation. Note this does require higher-level procedures not enforced by the environment.

---

[29]The detection scheme assumes that a generation of a world (at least its compatibility objects) will be maintained on a mchine as long as there are referencers of the world on the machine. Declaration stamps are stored with the definition, not with each referencer. If the target machine has no existing generation of the world, the restoration will not fail, but the compatibility keys in existing referencers on the machine might be bogus.

[30]We should also provide a means to preserve declaration numbers even though there are no apparent referencers of the number. In this way, moldy specs do not have to be kept online all the time.

Index