Epsilon Product Specification

Revision 0.3

January 10, 1991

Draft

Table Of Contents

1. Overview of Object Characteristics 1
1.1. Basic Concepts 1
1.2. Parent/Child Relationships 1
1.3. Object Names 2
1.4. Directory System Structure 2
1.5. Object Classes 3
1.5.1. Structure 4
1.5.2. File 4
1.5.3. Ada 4
1.5.4. Configuration 4
1.5.5. Diana 5
1.5.6. Code 5
1.5.7. Link_Pack 5
1.5.8. Pipe 5
1.5.9. Tape 5
1.5.10. Terminal 5
1.5.11. Null_Device 5
1.6. Object Subclasses 5
1.6.1. Structure Subclasses 5
1.6.1.1. Directory 5
1.6.1.2. Library 6
1.6.1.3. Subsystem 6
1.6.1.4. Project 7
1.6.1.5. Library_Implementation 7
1.6.1.6. Structure_Implementation 7
1.6.2. File Subclasses 7
1.6.3. Ada Subclasses 9
1.6.4. Configuration Subclasses 9
1.6.5. Diana Subclasses 9
1.6.6. Code Subclasses 10
1.6.7. Link_Pack Subclasses 10
1.6.8. Pipe Subclasses 10
1.6.9. Tape Subclasses 10
1.6.10. Terminal Subclasses 10
1.6.11. Null_Device Subclasses 10
1.7. Control Points 10
1.8. Object Attributes 11
1.9. Summary of Object Class Hierarchy 11
2. Libraries, Subsystems, and Configurations 15
2.1. Libraries and Compilation 15
2.2. Library Compatibility 15
2.2.1. Declaration Signatures 15
2.2.2. Compatibility 16
2.3. Library Importing 16
2.3.1. Explict and Implicit Imports 17
2.3.2. Computing Implicit Imports 17
2.3.3. Control Over Visibility 18
2.3.4. Compiler Key 18
2.3.5. Additional Checks 18
2.3.6. Obsolescence Propagation 18

2.3.7. Diagnostic Information 19
2.3.8. Refreshing Imports 19
2.3.9. Example 19
2.3.10. Summary 21
2.3.10.1. Import Inputs 21
2.3.10.2. Import Outputs 21
2.3.10.3. Import Process 22
2.4. Library Execution 22
2.4.1. Library Execution Context 22
2.4.2. Explicit Execution Imports 23
2.4.3. Derived Execution Imports 23
2.4.4. Implicit Execution Imports 23
2.4.5. Execution Checks 24
2.4.6. Diagnostic Information 25
2.4.7. Default Execution Behaviour 25
2.5. Library Structure 25
2.6. Subsystems 26
2.6.1. Spec Releases 27
2.6.2. Support for Multi-User Development 27
2.6.3. Support for Multi-Target Development 27
2.7. Configurations 27
2.7.1. Configuration Entries 28
2.7.1.1. Direct Entries 28
2.7.1.2. Indirect Entries 28
2.7.2. Compatibility and Completeness Checking 28
2.7.3. Configuration Example - Representing a System Release 28
2.7.4. System Configuration 29
2.7.5. Session Configuration 29
3. Access Control 31
3.1. Users and Groups 31
3.2. Access Control and Control Points 31
3.3. Access Permissions 31
3.4. Write Permissions 32
3.5. Combined Permissions 32
3.6. Explicit and Default Permissions 32
3.7. Access Control Lists 33
3.8. Access Control in Independent Directories 33
3.9. Access Control in Independent Libraries 34
3.10. Access Control in Subsystems 35
3.11. Creating New Control Points 38
3.12. Operator Capability 38
3.13. Network Access Control 38
3.14. Implementation Considerations 38

1. Overview of Object Characteristics

This section presents an overview of the objects in the epsilon environment and their general characteristics, including object structuring, classification, and naming. Following sections will consider particular kinds of objects and their operations in more detail.

Readers interested in a user-oriented introduction to the Epsilon system may wish to read *Programming in the Rational Environment* from the *Concepts and Facilities* book.

1.1. Basic Concepts

Objects are the entities stored by the file system. Each object has a *simple name* which identifies it, and a *class* and *subclass* which determine the operations that may be applied to it. Each object may be related to other objects through *parent-child* relationships.

The most fundamental kinds of objects in the system are *files* and *directories*. Files are used for the storage of arbitrary user-data. Directories are used to provide hierarchical structure for the objects in the system. Because directories provide structure we will often refer to the system of objects in the Environment as the *directory system*.

Compilation in the system is accomplished through the use of *Ada units* and *libraries*. Ada units contain the text of Ada source code in an underlying representation distinct from files. Libraries are directory-like structures which provide a compilation context for the Ada units in the library. Directories may be used to provide internal structure to the contents of a library.

Frozen copies of a library, called *releases*, can be created when one wants to capture the state of the library as a backup or when the library is to be made available to clients to compile against.

The Environment also has additional directory-like structuring mechanisms called *subsystems* and *projects* which are used to manage the development of large software systems. Subsystems are useful in developing components of large systems when multiple developers and/or multiple targets are involved. Projects provide methods for managing the architecture, release, and integration of a number of subsystems.

1.2. Parent/Child Relationships

The most basic relationships between objects in the system are *parent/child* relationships. Parent/child relationships provide the basis for *naming* of objects as well as *traversal* among objects. Each object, be it a file, directory, Ada unit, library, or subsystem has a single *parent* object. The sole exception is the directory system root object called "!" (pronounced bang), which is the ultimate parent of all other objects in the directory system. Each object may also be the parent of any number of *child* objects [1].

The purpose of directories is to provide hierarchical structure based on the parent/child relationships between directories and other objects. Other directory-like objects such as libraries and subsystems provide structure along with their other functions of compilation and the management of large system development respectively. We will say that an object is "in" in a directory (or directory-like object) if the object is a child of the directory or the child of some other object "in" the directory.

[1] This differs from Delta in that even objects like files can have children.

1.3. Object Names

Naming is described in detail in another document, below we will describe the basic principles of names that are used in this document.

Each object has a *simple name* which uniquely identifies it with respect to its parent object. Each object also has a *full path name* which uniquely identifies the object within the directory system. The full path name of an object is the full path name of the object's parent followed by the separator symbol '/' and the object's simple name. The full path name of the directory system root is just the root simple name, namely "!".

For example, the the directory `Users` which is a child of "!" has the full path name "`!Users`", and the directory `Tom` nested in `!Users` has the name "`!Users/Tom`". A text file with the simple name "`Memo`" which is in `!Users/Tom` would then have the full path name "`!Users/Tom/Memo`".

In order to support Ada-like naming for objects in the directory system, the period ('.') symbol may be used synonymously with '/'.

Certain objects are designated to be *system objects*. These objects are managed by the system and are usually not displayed automatically in the editor. System objects are known by the form of their simple names which always begin with the period ('.') symbol. Rational reserves all system object names in which an underscore preceeds all alphabetic characters. The number of periods at the beginning of a name will be used by editors to control display of objects at different levels of ellision. For example, `._Switches` and `...._Tree` are the names of two system objects which would be displayed at different levels of ellision.

1.4. Directory System Structure

As we mentioned before, the root of the directory system is the directory called "!" and all other objects have "!" as their ultimate parent. There are other directories built by the system for the storage of system objects, the most notable of which is the directory `!Machine`.

Associated with each user login is a directory called the *home directory* for that user. All home directories are children of the system directory `!Users`.

For example, the home directory of user `Tom` is shown below. Each line of the display contains an object in the home directory and the subclass of the object.

```
!Users/Tom : Directory
----------------------
  Mailbox : Directory
  Memo    : Text
  Tests   : Library
  Tools   : Library
```

The home directory displayed above contains a subdirectory called `Mailbox` and a text file called `Memo`. The home directory also contains two library objects named "`Tests`" and "`Tools`" which the user has created to store and compile different ada objects. The contents of the library `Tools` are shown in the display below. In the display below the library is shown to contain two ada units which correspond to an ada package spec and an ada package body.

```
!Users/Tom/Tools : Library
-------------------------
  Utilities : Pack_Spec
  Utilities : Pack_Body
```

Subsystems are useful in the development large system components because of a subsystems ability to manage multiple libraries. In the example, below the subsystem `Kernel` manages two libraries called `Path1` and `Path2`[2]which represent alternative implementations of the subsystem.

```
!Environment/Kernel : Subsystem
-------------------------------
  Path1           : Library
 ·Path1_Releases  : Directory
  Path2           : Library
```

In addition to the two libraries `Path1` and `Path2` the subsystem also contains two frozen copies of `Path1` which are stored in the `Path1_Releases` directory shown below.

```
!Environment/Kernel/Path1_Releases : Directory
----------------------------------------------
  Rev_0_1 : Library
  Rev_0_2 : Library
```

Since a release is a frozen copy of a library, the release contains all of the objects that were in the library. Thus, the contents of a release above might have the display shown below which would be the same as the if the original library were displayed.

```
!Environment/Kernel/Path1_Releases/Rev_0_2 : Library
----------------------------------------------------
  Object_Management : Package_Spec
  Object_Management : Package_Body
```

1.5. Object Classes

The class of an object determines the basic characteristics of the object. Each object has a particular class which may not be changed during the lifetime of the object. In addition to its class, each object also has a *subclass* and, optionally, a *subclass extension*. The subclass and the subclass extension establish the detailed characteristics of the object. The subclass and extension may be expressed by the notation *Subclass.Extension* [3]. Similarly, full class information may be expressed as *Class.Subclass.Extension* and the class and subclass can be expressed as *Class.Subclass*.

In the preceeding sections we have talked of the system as being composed of files, directories, ada units, libraries, and subsystems. In reality the system is composed of a richer set of objects, each of which has a class, subclass, and an optional subclass extension, which together determine the properties of the object. Files, directories, ada units, libraries, and subsystems are particular classes and subclasses which play prominent roles in the system.

[2]Libraries in subsystem are often called *development paths*, hence the names `Path1` and `Path2`.
[3]This notation may be used in `Class` attributes.

Below are listed the complete set of objects classes and an overall description of the class. Objects that are used primarily for implementation purposes are listed for completeness. A later section will list the subclasses and subclass extensions associated with each class.

1.5.1. Structure

Structure class objects (also known as *structural* objects or *directory-like* objects) are used to:

- Organize the objects in a hierarchical manner.
- Provide a compilation context and storage for program units.
- Manage and coordinate large scale development.

This class includes *directories*, *libraries*, *subsystems*, and *projects*. Structural objects have no user-data in the sense that files contain user-data, rather structural objects are important because of their ability to organize other objects.

Structural objects may only be created as children of other structural objects (subject to constraints outlined later). Structural objects may not be children of non-structural objects.

Some structural objects objects are designated as *control points* which allows disk space and access control characteristics to be controlled at these nodes in the directory hierarchy. The characteristics of control point will be discussed later in more detail.

1.5.2. File

Files are used to store arbitrary user-data. The format of the data and the access methods to the data are dependent on the particular subclass of the file object. For example, files may contain ascii text or may contain data in arbitrary binary formats.

1.5.3. Ada

Ada objects are used for the storage and compilation of ada program text and correspond to *compilation units* as defined by the Ada Language Reference Manual (LRM). Each ada object is nested (perhaps indirectly) in some library which provides the compilation context for the ada unit. The status of an ada object in its compilation context is defined by the *compilation unit state* of the object which keeps track of the current level of compilation the object in its library.

1.5.4. Configuration

Configuration objects are used to group together related libraries [4]. Configurations play a key role in compilation, execution, and the management of large systems. Configurations are also used to establish system-wide and session-wide defaults for command execution and name resolution. Later sections will describe in detail how configurations are used and manipulated.

[4]Configurations are similar to Activites in Delta.

1.5.5. Diana

Diana objects are used by the system to store compiler oriented information about the internal representation of programs. Users neither create nor modify Diana objects explicitly, rather these objects are created and modified by the system when compilation occurs.

1.5.6. Code

Code objects are used to store the executable form of programs. As with Diana objects, Code objects are created as a side effect of compilation operations.

1.5.7. Link_Pack

Link_Pack objects are used by the compilation system to manage the visibility of names in libraries.

1.5.8. Pipe

To be described.

1.5.9. Tape

Tape objects are used to represent physical tape devices.

1.5.10. Terminal

Terminal objects are used to represent physical terminal devices and the windows of users sessions.

1.5.11. Null_Device

This class is used to represent the *null device* which is used mainly as a means of discarding output.

1.6. Object Subclasses

The subclass of an object establishes the detailed characteristics of the object. Each subclass is associated with a particular class. Each subclass may also have a set of subclass extensions.

1.6.1. Structure Subclasses

Below are presented the structure subclasses and a brief description of their characteristics.

1.6.1.1. Directory

Directory objects are the most basic structural object in the system. Directories are used to produce arbitrary hierarchical structuring in the system. There are two kinds of directories with extensions *independent* and *dependent*.

- **Independent**

  A directory is independent if is a control point as defined above. Independent directories, like other control points, can be used to control disk space utilization and access control characteristics. Examples of independent directories are the file system root directory '!' and all home directories.

  Independent directories are not allowed in libraries, subsystems or projects.

- **Dependent**

  Dependent directories are directories which are not control points and thus are dependent on their enclosing control point for disk space limits and access control restrictions. Dependent directories are allowed in libraries, subsystems, and projects.

1.6.1.2. Library

Library objects are used for the storage and compilation of ada program units. Each library defines a compilation context in the sense of the Ada LRM. All program units must be stored in a library or in a release of a library as shown below.

Libraries may not be nested, even indirectly, within other libraries. However, libraries may be arbitrarily nested within directories as long as this does not violate the prohibition of libraries nested within libraries.

- **Independent**

  An independent library is a non-frozen library that is not nested in a subsystem. An independent library is a control point.

- **Working**

  A working library is a non-frozen library that is nested within a subsystem.

- **Release**

  A release is a frozen copy of a library. A release may be used by clients to compile against or to execute against. Thus, a release captures the state of library for internal backup purposes or for external distribution. The use of releases in multi-library compilation will be discussed later.

  Releases follow the same rules as libraries for nesting within other structural objects.

- **Spec_Release**

  A spec_release is a release which only contains Ada spec units. Spec_releases may be used for compilation but not for execution. Spec_releases are particularly useful for emphasizing the distinction between specification and implementation in software development.

- **Code_Release**

  A code_release is a release which contains no ada units but does not contain executable code segments. Code_releases are useful when it is necessary to execute code while preventing any access to the source.

1.6.1.3. Subsystem

A subsystem is a structure that is used to manage and coordinate development in a set of related libraries. Subsystems may be nested within other structural objects other than libraries. In particular, a subsystem may nested within another subsystem as long as it is not nested within a library of the enclosing subsystem. Subsystems are described in detail in a later section.

1.6.1.4. Project

A project is a structure that is used to manage and coordinate development in a set of related subsystems.

1.6.1.5. Library_Implementation

Associated with each independent library is an object of subclass *Library_Implementation* which is used by the system to implement the independent library. A library_implementation object is created automatically whenever an independent library is created.

For example, consider an unmanged library `Tools` in a users home directory. The library would have an *implementation directory* associated with it as shown below. The implementation directory is a system object and so would not normally be displayed by the editor.

```
!Users/Tom : Directory
----------------------
  Tools                       : Library
  ...._Tools_Implementation : Directory
```

The implementation directory for the library would then contain the library_implementation object for the library as shown below.

```
!Users/Tom/...._Tools_Implementation : Directory
------------------------------------------------
  Tools : Library_Implementation
```

1.6.1.6. Structure_Implementation

Objects of subclass *structure_implementation* are used for the implementation of other structural objects. Structure_Implementation objects are always system objects. In particular each independent directory, subsystem and project contains a subobject called `...._State` of subclass Structure_Implementation.

```
!Users/Tom : Directory
----------------------
    ...._State : Structure_Implementation
    Mailbox    : Directory;
    Memo       : Text
    Tests      : Library
    Tools      : Library
```

1.6.2. File Subclasses

File subclasses are used for variety of purposes from the repesentation of text files to a number of special purpose databases.

- **Binary**

  Binary files may be used to contain data in any arbitrary binary format.

- **Text**

  Text files are used to contain structured text information. In particular the images associated with Ada units are represented by Text files.

- **Ascii**

  Ascii files are used to contain sequences of ascii characters.

- **Switch_Definition**

  Switch_Definition files are used to contain the machine-wide definitions of available switches.

- **Directory_Switch**

  Directory_Switch files contain the switch values associated with a particular directory or directory-like structure such as a library.

- **Session_Switch**

  Session_Switch files contain the switch values associated with a particular user session.

- **User_Profiles**

  User_Profile files contain the profiles associated with user accounts.

- **World_Database**

  World_Database files are used by the operating system for implementation purposes.

- **Search_List**

  Search_List files contain the search lists associated with user sessions.

- **Cmvc_Database**

  Cmvc_Database files contain the historical information associate with development activities.

- **Object_Set**

  Object_Set files contain sets of objects.

- **Dictionary**

  Dictionary files contain dictionary information used by text processing tools.

- **Postscript**

  Postscript files contain document formatter output in postscript notation.

- **Temp_Heap**

  Temp_Heap files are used for implementation purposes by various parts of the environment.

- **Import_Database**

  Import_Database files contain information that is used to manage inter-library compilation. These objects are described in detail in a later section.

- **Domain_Errors**

  Domain_Error files contain information for the generation of system error messages.

- **Master_Errors**

  Master_Error files contain information for the generation of system error messages.

- **Document_Database**

  Document_Database files are used by the *Rational Design Facility*.

- **Element_Cache**

  Element_Cache files are used by the *Rational Design Facility*.

- **Markup**

  Markup files contain text formatter markup input.

- **Menu**

  Menu files contain state saved by various environment menus.

1.6.3. Ada Subclasses

Ada subclasses generally correspond to the different types of ada compilation units. Familiar examples, of Ada subclasses are *Package_Spec* and *Package_Body*. Additional subclasses exist for use by the compilation system to handle objects that are not yet fully formed such as the *Compilation_Unit* subclass.

1.6.4. Configuration Subclasses

A configuration is a set of related libraries. A configuration is restricted to contain at most one library or release from a given subsystem, and at most one release from a given independent library. The different configuration subclasses express different constraints on the relationships between the libraries and releases in the configuration.

- **General**

  A General configuration may contain arbitrary libraries. There are no constraints on the libraries that may appear in such a configuration.

- **Compatible**

  In a *compatible* configuration all libraries must be compatible in the sense defined in the next section of this document.

- **Complete**

  In a *complete* configuration all libraries must be compatible and complete in the sense of Ada program libraries.

1.6.5. Diana Subclasses

Each Diana object is associated with a particular Ada object and contains information generate by the compiler when the Ada object was compiled. The different subclasses of Diana represent different kinds of compiler information.

- **Tree**

  Objects of this subclass contain the abstract syntax tree (AST) associated with a particular compilation unit.

- **Top_Decl_Database**

  Objects of the subclass contain symbol table information used by the compiler during name resolution.

- **Cg_Attr**

  Objects of the subclass contain code generation information for the associated ada unit.

1.6.6. Code Subclasses

- **Relocatable**

  Objects of this subclass contain *relocatable* code.

- **Executable**

  Objects of this subclass contain *executable* code.

1.6.7. Link_Pack Subclasses

- **Visible_Names**

  Objects in this subclass contain the set of program units which are visible in the current compliation context.

1.6.8. Pipe Subclasses

There is a single Pipe subclass named Nil. All Pipe class objects are of this subclass.

1.6.9. Tape Subclasses

There is a single Tape subclass named Nil. All Tape class objects are of this subclass.

1.6.10. Terminal Subclasses

- **Physical**

  Objects of this subclass represent physical terminals.

- **Message_Window**

  Objects of this subclass represent the message window in an environment session.

- **Io_Window**

  Objects of this subclass represent i/o windows in environment sessions.

1.6.11. Null_Device Subclasses

There is a single Null_Device subclass called Nil. All Null_Device class objects are of this subclass.

1.7. Control Points

Certain structural objects in the system are designated to be *control points*. Control points are used to control certain system characteristics such as access control information and the Unix paritition on which the control point is mounted. Control points include:

- All subsystem and project objects.
- All independent libraries.
- All independent directories.

Every object in the directory system is either a control point or is associated with a control point. If an object is in a subsystem or project then the associated control point is the enclosing subsystem or project. If the object is in an independent library or a release of an independent library then the associated control point is the independent library. If the object is not in a subsystem, project, or independent library (or release thereof) then the associated control point is the enclosing control point directory.

Control points are the focus for the control of certain properties including:

- Control points are associated with particular Unix disk partitions.
- Control points are the focus for access control.
- Control points may be moved within the directory system without changing the characteristics of the objects within the control point.

1.8. Object Attributes

Associated with each object in the system are a number of *attributes* which capture various characteristics about the current state of the object. Attributes include things like the last update time of the object, size of the object, or the last release that was created from a library.

Attributes are either *scalar valued* or *object valued*. Scalar valued objects, such as the last update time of an object, can be displayed by the editor, or can be be used to filter objects in naming expressions [5]. Object valued attributes can be used in naming expressions as alternative ways to name the object that is the value of the attribute.

1.9. Summary of Object Class Hierarchy

Below we present a summary of the object class hierarchy for classes and subclasses that are directly manipulated by the user. Omitted are classes and subclasses that used just for implementation purposes.

Each class, subclass, and optional subclass extension is presented along with comments.

[5] In general this is done via the `'If` attribute.

Figure 1.1 - Object Class Hierarchy

Class	Subclass	Extension	Comments
Structure			
	Directory		
		Independent	*control point*
		Dependent	
	Library		
		Independent	*control point*
		Working	
		Release	
		Spec_Release	
		Code_Release	
	Subsystem		*control point*
	Project		*control point*
	Library_Implementation		
	Structure_Implementation		
File			
	Binary		
	Text		
	Ascii		
	Switch_Definition		
	Directory_Switch		
	Session_Switch		
	User_Profiles		
	World_Database		
	Search_List		
	Cmvc_Database		
	Object_Set		
	Dictionary		
	Postscript		
	Temp_Heap		
	Import_Database		
	Domain_Errors		
	Master_Errors		
	Document_Database		
	Element_Cache		

Figure 1.1 - Object Class Hierarchy (continued)

Class	Subclass	Extension	Comments
	Markup		
	Menu		
Ada			
	Compilation_Unit		
	Package_Spec		
	Package_Body		
	...		
Configuration			
	General		
	Compatible		
	Complete		
Diana			
	Tree		
	Top_Decl_Database		
	Cg_Attr		
Code			
	Relocatable		
	Executable		
Link_Pack			
	Visible_Names		
Pipe			
	Nil		
Tape			
	Nil		
Terminal			
	Physical		
	Message_Window		
	Io_Window		
Null_Device			
	Nil		

2. Libraries, Subsystems, and Configurations

Libraries, subsystems, and configurations are key objects in the Epsilon Environment and are the building blocks for user development methodologies. In this section we'll examine these objects in detail. We will pay particular attention to the integration between these objects and other fundamental building blocks of the environment such as the compilation system. The integration between libraries, subsystems, and the compilation system is particularly important because the level of integration in Epsilon will make it possible to support development methodologies that were prohibitively expensive in Delta.

2.1. Libraries and Compilation

All compilation occurs in the context of some library. Libraries provide both a location for the storage of program units in the directory system hierarchy, and a context for the compilation of those units [6]. Each library manages the visibility between local program units and units in other libraries. A library also manages all compilation dependencies between units and all internal compiler state. A frozen copy of a library, called a release, can be created from any library. A release captures both the contents of the program units in the library and the compilation state of those units.

The model for compilation within a single library is described in the *Epsilon Compilation Document*. In there following sections we will be concerned with the inter-library relationships that support compilation involving multiple libraries.

2.2. Library Compatibility

Compatibility is a central notion of inter-library compilation. Compatibility information is used to determine which libraries and releases are suitable to be used together. Compatibility information is also used to limit obsolescence and recompilation when inter-library relationships change.

2.2.1. Declaration Signatures

An Ada object is composed of a set of *declarations* the form of which is specifed by the LRM. Declarations may reference other declarations either in the same object or in different Ada objects in accordance with Ada visibility rules.

Each declaration in a compiled Ada object has a unique *declaration signature* that is based on the syntax of the declaration and the signatures of all referenced declarations [7]. The declaration signature includes the simple name of the control point containing the Ada object, however the declaration signature does not contain the name of the enclosing library. Thus, equivalent declarations in different libraries of the same control point have identical declaration signatures. Intuitively, declarations in different libraries have the same declaration signature if they really are the "same" in the sense that they have the same meaning.

A declaration is *exported* if that declaration may be referenced by another Ada object based on Ada visibility rules. Each compiled Ada object has an *export signature* which is the set of all the declaration signatures of the exported declarations. Each compiled Ada object also has an *import signature* which is the set of declaration signatures of all of the declarations that are referenced by code in the Ada object.

[6] For Ada units, the library is a *program library* in the sense defined by the Ada LRM.

[7] The declaration signature is actually computed via a hash function on the syntax and the referenced declarations.

In an intuitive sense the export signature of unit is just the set of declarations that are exported by the unit. Conversely, the import signature of a unit is just the set of declarations from other units which the unit depends upon.

The export signature of a library is then the set of all export signatures for the units in the library and the import signature of a library is the set of the import signatures for all of the units in the library.

2.2.2. Compatibility

Compatibility is a condition that holds between *supplier* and *client* libraries. A supplier library provides the declarations that are required by a client library. If the export signature of a supplier library includes all the declarations specified in the import signature of the client library (for the control point of the supplier) then the client is *compatible* with respect to the supplier, otherwise the client and supplier are *incompatible*. We call the compatibility relationship between a single supplier and a single client *pair-wise compatibility*.

Compatibility for an entire configuration of libraries [8]can be based on the pair-wise compatibility of the library in the configuration. A configuration is compatible if all the libraries in the configuration are pair-wise compatible, and if no library has an import signature that contains a reference to a control point that is not represented in the configuration.

The compatibility of a configuration guarantees that any reference to a name that is used in a library may be resolved to some declaration in some library of the configuration. Not all configurations are compatible. The construction of compatible configurations is one of the chief tasks of the importing operations described below.

2.3. Library Importing

In this section we describe the *importing* process which allows program units in one library to be compiled against program units in another library. The importing process involves a client library (called the the *importer* or *importing* library) and a set of supplier libraries (called the *imports*).

The import process has a several goals:

- Construct a compatible configuration, called the *compilation configuration* which references every imported library. No other libraries will participate in compilation in the importing library
- Establish the set of units from the imported libraries which will be visible in the importing library.
- Propagate obsolescence into the importer library based on the contents of the imported libraries.
- Construct a configuration, called the *execution configuration*, that references the libraries which will be used when a program is *linked* in the importing library. The contents of the execution configuration may be the same or different than the contents of the compilation configuration.

Although we describe the imports as being other "libraries", in truth the imports are almost always releases or spec_releases of libraries. Only under special circumstances, described below, can a library import another non-release library.

All importing information and control over the importing process is centralized in the *import database* of a library. An editor is provided for manipulating the database and for querying diagnostic information about the current state

[8]Recall that a configuration is a set of libraries.

of the database. Examples in later sections will present the form of the import database.

2.3.1. Explict and Implicit Imports

Imported libraries may be either *explicit* or *implicit*. Explicit imports are specified directly by the user. Implicit imports, on the other hand, are not directly specified, rather they are computed based on the implicit imports to satisfy the requirements of the importing process.

The importer library may have visibility to any units in the explicit imports (subject to user policies described later). However, the importer has no visibility to any units in the implicitly imported libraries.

The complete set of explicit and implicit imports make up the *compilation configuration* of the importer. The reason for including implicit imports is that the compilation configuration must be compatible which may not be true of the explicit imports alone.

The compilation configuration and the visible names make up the complete context for compilation in the importer library. This means that compilation in the importer will rely on the program units in the imports, but not any other characteristics of the imported libraries (and in particular not on their imports). This self-sufficiency of libraries allows more flexibility in the import process than was possible in earlier environment releases. This self-sufficiency is also the basis for the compatibility condition on the compilation configuration, since all declarations in the imported libraries must be resolvable within the libraries of the configuration.

2.3.2. Computing Implicit Imports

We noted above that along with the explicit imports, implicit imports are required in order to satisfy the compatibility of the compilation configuration. Implicit imports are not specified directly by the user, but rather are computed by the importing process.

The user does have control over how the implicit imports will be computed. The implicit imports may be computed from the compilation configurations of the explicit imports alone. Alternatively, the implicit imports may computed from the explicit imports and from a configuration from which the implicit imports may be taken. In particular, the importing mechanisms support two policies for the computation of implicit imports:

- **COMPUTE_FROM_IMPORTS**

  When this policy is applied, the implicit imports are computed from the compilation configurations of the explicitly imported libraries. In effect, the compilation configurations of all explicit imports are merged together to form a new compilation configuration. Whenever there are multiple libraries for the control point, the latest library is chosen to go into the final configuration. The resulting configuration is required to be compatible.

  This policy is the default.

- **COMPUTE_FROM_CONFIGURATION**

  When this policy is applied, the implicit imports are taken from a named configuration. The control points that must be represented in the compilation configuration are computed by determining which control points are represented in the compilation configurations of the explicit imports. A library for each such control point is then selected from the specified configuration and placed in the compilation configuration. The result compilation configuration must be compatible.

  The configuration that is specified may be an explicit user configuration or may be the users *session configuration*. Details of the session configuraton will be described later.

2.3.3. Control Over Visibility

All of the objects in the explicitly imported libraries are potentially visible. Users have control over which objects are actually visible. Visibility is controlled by a filter, which is a set of naming expressions. The naming expressions are composed of wildcard names which describe the units to include or exclude, as well as other expressions that describe units to rename. A wildcard name in the set may include the symbols '#' and '@'. In particular the filter will contain entries of the form:

- *<wildcard name>*

  Specifies that all units in the imported library whose simple name matches *<wildcard name>* are to be include.

- *~<wildcard name>*

  Specifies that all units in the imported library whose simple name matches *<wildcard name>* are to be excluded.

- *<simple name> => <simple rename>*

  Specifies that the unit whose name matches *<simple name>* is to be included and will be referred to by the name *<simple rename>*.

For example, the filter `[@, ~foo, abc=>xyz]` would specify that all units except `Foo` are to be included, and the unit `Abc` is to be renamed as `Xyz`.

2.3.4. Compiler Key

The compiler key of a library specifies the characteristics of the compiler that will be used in the library. The compiler key specifies the front end, back end, and policy checking that will be used to compile units in the library. A library and all of its imports (explicit and implicit) must have keys that agree with respect to the front end and back end components of the compiler key.

2.3.5. Additional Checks

Generally, only releases may be imported. When it is necessary to import non-release libraries (ie, with subclass Library) additional conditions must be satisfied. In particular if a library L_1 needs to import library L_2 then:

1. The importing must be explicit. L_2 must be an explicit import of L_1.

2. The importing must be mutual. L_2 must also import L_1.

3. The compilation configurations must be identical. The compilation configurations of L_1 and L_2 must have the same contents.

4. Releases in the compilation configuration may not depend on suppliers in the configuration that are not releases. If R is a release in the compilation configuration of L_1 then R must not have a compilation configuration that includes a library from the control point of L_2.

2.3.6. Obsolescence Propagation

When the imports of a library are changed, obsolescence is propagated to the library on the basis of the changes.

2.3.7. Diagnostic Information

The import database, which manages the import process, keeps diagnostic information in a permanent form. This information is available for the users to review at their convenience. Below are listed various problems for which diagnostic information is kept:

- Problems encountering in resolving the names of libraries.
- Problems due the deletion or inaccesibility (eg, due to network failures) of libraries that are currently imported.
- Problems in computing the implicit imports,
- Problems due to incompatibilities between the imports. When incompatibilities occur, the libraries involved and information about the missing declarations are available.

2.3.8. Refreshing Imports

Each explicit import may have associated with it an additional configuration through which the import is *refreshed*. When import refreshing is requested, the library which the configuration references is used to update the explicit import.

For example, each working library has associated with it a configuration in the release directory which references the latest release. This configuration is called `Latest` and provides an easy way to keep imports up-to-date.

2.3.9. Example

In this section we present an example of the importing process. Typical screen images from the import database of library `!Env.Kernel.Alpha` will be used to illustrate the process. We start with figure 2.1 which displays the initial imports of the library. Those imports include libraries from the `!Implementation`, `!Io`, and `!Lrm` subsystems.

```
!Implementation Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Io             Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Lrm            Alpha_Releases.Rev_0_0_2 Alpha'Latest
```

Figure 2.1 - Initial Imports

The goal of the import process is to add an additional import from the `!Env.Utilities` subsystem. This import is entered and the resultant display is shown in figure 2.2.

```
#     !Env.Utilities  Alpha_Releases.Rev_0_0_4 Alpha'Latest
      !Implementation Alpha_Releases.Rev_0_0_1 Alpha'Latest
      !Io             Alpha_Releases.Rev_0_0_1 Alpha'Latest
      !Lrm            Alpha_Releases.Rev_0_0_2 Alpha'Latest
```

Figure 2.2 - New Import Entered

The line for the new import is preceeded by the character '#' to denote that the new import has been entered, but has not yet been *promoted*. Promoting the imports is required in order to actually change the compilation configuration and establish visibility to objects in the imported libraries.

Therefore, the next step is to promote the imports. However, this results in a compatibility problem because the new import `!Env.Utilities.Alpha.Rev_0_0_4` depends on a declaration defined in the subsystem `!Implementation` which does not exist in library `Implementation.Alpha.Rev_0_0_1`. Figure 2.3 displays the diagnosis of the compatibility problem.

```
!Env.Utilities    Alpha_Releases.Rev_0_0_4 Alpha'Latest
!Implementation   Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Io               Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Lrm              Alpha_Releases.Rev_0_0_2 Alpha'Latest

Diagnosis =>
  *** The compilation configuration of !Env.Kernel.Alpha
      is not compatible.

Attempted_Imports =>
      !Env.Utilities.Alpha_Releases.Rev_0_0_4
      !Implementation.Alpha_Releases.Rev_0_0_1
      !Io.Alpha_Releases.Rev_0_0_1
      !Lrm.Alpha_Releases.Rev_0_0_1
```

Figure 2.3 - Compatibility Diagnosis

The diagnosis of the compatibility problem also includes a list of the attempted imports and underlines the import whose requirements were not met in the compilation configuration. Visiting this library from the import editor will underline a specific declaration usage that is incompatible.

The problem is then fixed by refreshing the import from `!Implementation` to be the latest release from from `Alpha` library. Figure 2.4 illustrates the import database image after the imports have been updated and successfully promoted.

```
!Env.Utilities    Alpha_Releases.Rev_0_0_4 Alpha'Latest
!Implementation   Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Io               Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Lrm              Alpha_Releases.Rev_0_0_2 Alpha'Latest

Diagnosis =>
  +++ Import operation succeeded
```

Figure 2.4 - Import Succeeded

Because `!Env.Utilities.Alpha_Releases.Rev_0_0_4` imports a release from another subsystem, namely `!Env.Abstract_Types.Alpha_Releases.Rev_0_0_2` there is an implicit import added. This implicit import can be displayed by expanding the import database display as shown in figure 2.5.

```
!Env.Utilities   Alpha_Releases.Rev_0_0_4 Alpha'Latest
!Implementation  Alpha_Releases.Rev_0_0_4 Alpha'Latest
!Io              Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Lrm             Alpha_Releases.Rev_0_0_2 Alpha'Latest

Implicit_Imports =>
  !Env.Abstract_Types.Alpha_Releases.Rev_0_0_2

Diagnosis =>
  +++ Import operation succeeded
```

Figure 2.5 - Implicit Import

2.3.10. Summary

Below we summarize the import process:

2.3.10.1. Import Inputs

Below are listed the inputs to import process.

- **Explicit Imports**

  Specifies the libraries that are to be explicitly imported and which contain the objects that may be visible in the importer library.

- **Compiler Key**

  Specifies the compiler to be used in the library.

- **Implicit Import Policy**

  Specifies how the implicit imports are to be computed.

- **Visibility Filters**

  Specifies which units in the explicit imports should be visible and what their local names should be.

2.3.10.2. Import Outputs

Below are listed the state objects that are affected by changes to the imports.

- **Visible Names**

  Names of units which may be compiled against.

- **Compilation Configuration**

  The configuration which, along with the visible names, forms the complete context for compilation in the library.

- **Compilation State of Units**

  The compilation state of the units in the importer is changed to be consistent with the new imports.

2.3.10.3. Import Process

Below are listed the major steps in the import process:

1. **Validate Explicit Imports**

   Verify that all objects exist.

2. **Compute Implicit Imports**

   Compute the implicit imports based on the explicit imports and the chosen policy.

3. **Check Compatibility**

   Check the compatibility of all explicit and implicit imports.

4. **Perform Additional Checks**

   Check that compiler key components match and make any additional checks that are required if unfrozen library are being imported.

5. **Establish Compilation Context**

   Set new values for the visible names and the compilation context.

6. **Propagate Obsolescence**

   Propagate obsolescence in the library based on the new imports.

2.4. Library Execution

Up to this point we have considered the import mechanisms from the standpoint of establishing the compilation context. However, the import mechanisms are also used to establish the *execution context* of a library. The following sections will a library's execution context and show that it's management is analogous to the management of the compilation context.

2.4.1. Library Execution Context

The execution context of a library is the set of libraries that will be included in any programs that are *linked* in the importing library. The execution context may include libraries from the following sources:

- Libraries that have been explicitly designated to be part of the execution context.
- Libraries that have been explicitly imported to be part of the compilation context and which are also executable.
- Libraries which are part of a designated configuration, possibly the user's session configuration.

The following sections will describe how libraries from these sources are included in the execution context.

The execution context of a library is represented by the *execution configuration* of the library. The goal of the importing mechanisms with respect to execution is the construction of the execution configuration.

Besides explicit linking of programs in the library, the execution context also plays a role in the implicit linking that occurs during *command execution* when the units in the library are invoked as part of a command. In a later section we will discuss how these mechansisms are used to compile and execute commands.

2.4.2. Explicit Execution Imports

Libraries that are designated by name to be part of the execution context are called *explicit execution imports*. Explicit execution imports may be specified in addition to the explicit imports that are specified for compilation. When an explicit execution import is specified it follows the explicit import entry in the import editor display.

For example, in the import editor display fragment below a spec_release is specified for compilation and a full release named `!Env.Utilities.Alpha_Releases.Rev_0_0_1` is specified for execution.

```
!Env.Utilities  Alpha_Releases.Spec_0_0_4 Alpha'Latest_Spec
       Execution => Alpha_Releases.Rev_0_0_1
```

Figure 2.6 - Explicitly Specified Library for Execution

By specifying an explicit executable import for a subsystem the user can guarantee that a particular release will be linked into any programs.

2.4.3. Derived Execution Imports

When an executable release is specified as an explicit import for compilation purposes, that release will also be included in the execution context unless there is an explicit execution import that overrides it. Such an import is called a *derived execution import* because in such a case the entry in the execution context is *derived* from the compilation context.

For example, consider the import editor fragment below in which the full release is imported for compilation purposes. Because there is no explicit execution import specified (ie, the value is `<nil>`), the import will be used in both the compilation context and the execution context.

```
!Env.Utilities  Alpha_Releases.Rev_0_0_1 Alpha'Latest
       Execution => <nil>
```

Figure 2.7 - Library for Execution Derived from Compilation Context

The handling of derived execution imports in Epsilon is similar to the handling of combined view imports in Delta. In Delta a combined view import is compiled against and executed regardless of the contents of the current activity.

2.4.4. Implicit Execution Imports

So far all entries into the execution context have been explicitly named by the user either as explicit execution imports or as explicit compilation imports which may be executed. However, the execution context may need to contain many other libraries which are not named but are rather computed.

There are two cases in which entries in the execution context may need to be computed. First, an explicit import may be a spec_release and there is no explicitly specified executable import for that subsystem. In this case, the system must compute the library to execute. Second, a library in the execution context may reference units in other subsystems for which there are no specified executable imports. In this case also the system must compute the libraries which need to be included in the execution context. The libraries that are included in the execution context for either of these reasons are called *implicit execution imports*.

Users are provided with a means of controlling the computation of the implicit execution imports by specify the *execution policy* for the library. There are three policies available:

- **USE_CONFIGURATION**

  This policy specifies that a configuration will be supplied which references all necessary implicit execution imports. The configuration may be either explicity named or may be the user's current session configuration.

  This policy also specifies that the implicit execution imports are not computed at the time the imports are changed but rather are computed when a link operation occurs. Therfore, the contents of the configuration may change causing different libraries to be included in the linked program.

  This policy is the default and by default the specified configuration is the user's current session configuration. These defaults lead to behavior that is very similar to Delta behavior in that the program closure is based on current session information (ie, the current activity in the case of Delta).

- **COMPUTE_FROM_IMPORTS**

  This policy specifies that additional libraries for the implicit execution imports will be computed from the explict imports. More precisely, the implicit execution imports will be computed from the implicit execution imports of the explicit execution imports and any derived execution imports.

  The computation occurs at the time of the import operation. This policy is analagous to the COMPUTE_FROM_IMPORTS policy for the compilation context.

  In effect this policy causes the execution contexts of all explicit execution imports to be merged together. Conflict are resolved in favor of the latest library.

- **COMPUTE_FROM_CONFIGURATION**

  This policy specifies that the implicit execution imports are to be computed from a configuration which may be either named or be the user's current sessions configuration. Unlike the USE_CONFIG policy this causes the execution context to be computed at the time of the import operation. The execution context is not affected by any later changes to other configurations.

By selecting different policies users have control over how the execution context is constructed and the binding time of the execution context. The default behavior has been set up to resemble Delta behavior.

2.4.5. Execution Checks

When the execution context is computed at "import time" (ie, the execution policy is COMPUTE_FROM_IMPORTS or COMPUTE_FROM_CONFIG) the user may request additional checks to be made on the resulting execution configuration. When a check is requested the import operation will fail if the check does not pass. The use of checks also restricts the libraries that may be in an execution context to be releases which are frozen. The types of checks are listed below:

- **NONE**

  Specifies that no checks are made on execution context. This is the default.

- **COMPATIBLE**

  Specifies that the execution context and thus the resulting execution configuration must be *compatible*. This is the same level of checking that is always made for compilation configurations.

- **COMPLETE**

  Specifies that the execution context must be compatible and in addition all required bodies must be present and all units must be coded. Linking in a library with a *complete* execution configuration will always succeed.

Using the checks allows users to check for possible "link time" errors at "import time". In particular, the COMPLETE check guarantees that no link time errors will occur. However, it is not the case that errors will inevitably occur in an execution context that is not COMPLETE and or even COMPATIBLE since a given linked program may not run into any of these problems.

2.4.6. Diagnostic Information

The import database maintains diagnostic information for execution imports that that is identical to that maintained for compilation imports.

2.4.7. Default Execution Behaviour

Below we present an image of and import editor showing a number of explicit imports and all policy information at their default settings. In addition there have been no explicit execution imports set.

Most of the explicit imports are spec_releases and therefore executable libraries for these subsystems must be computed implicitly. A single import (ie, `!Lrm.Alpha_Releases.Rev_0_0_2` is a full release and thus will be included in the execution context as a derived execution import.

```
!Env.Utilities   Alpha_Releases.Spec_0_0_4 Alpha'Latest_Spec
!Implementation  Alpha_Releases.Spec_0_0_1 Alpha'Latest_Spec
!Io              Alpha_Releases.Spec_0_0_1 Alpha'Latest_Spec
!Lrm             Alpha_Releases.Rev_0_0_2  Alpha'Latest_Spec

Compilation_Policy => COMPUTE_FROM_IMPORTS

Execution_Policy => USE_CONFIGURATION
Execution_Config => <SESSION_CONFIGURATION>
Execution_Checks => NONE
```

Figure 2.8 - Default Import Behavior

The compilation policy specifies that the compilation context will be computed from the compilation contexts of the explicit imports.

The execution policy information specifies that the users session configuration will be used to compute the implicit execution imports that will be part of the execution context and no checking will be performed on this context. These default settings produce behavior similar to Delta in that session information is used to compute execution closures at "link time". Futhermore, as in Delta, link errors may occur either because of incompatibilities or because required bodies are not present or because units are not coded.

2.5. Library Structure

Along with the program units in a library there are certain predefined system objects that are used to manage the import and compilation processes.

Below is the display for a typical library which contains two ada units and a number of system objects.

```
!Users/Tom/Tools : Library
--------------------------
  ._Imports          : Import_Database
    ._Compilation    : Compatible
    ._Execution      : General
    ...._Errors      : Binary
  ._Names            : Visible_Names
  ._Release_Names    : Text
  ._Switches         : Directory_Switch
  Utilities          : Pack_Spec
  Utilities          : Pack_Body
```

The system objects in the library have the following meaning:

- `._Imports`

  The import database object for the containing library. This object contains all of the information involving the import process. Changes to the imports are accomplished by editing this object.

- `._Imports/._Compilation`

  The compilation configuration for the containing library. These are the libraries and releases involved in compilation. This configuration is managed by the import database and its contents may be examined only through the import database.

- `._Imports/._Execution`

  The execution configuration for the containing library. This configuration references all libraries and releases involved in linking programs in the context of the containing library. This configuration is managed by the import database and may be examined only through the import database.

- `._Imports/...._Errors`

  A file of error and warning messages associated with the last operations performed on the import database. This file is in a binary format. The error messages are displayed through the import object editor.

- `._Names`

  A link_pack object containing the names that are visible in the compilation context of the library. This includes the names of local objects and the names of local objects that have been imported. This object is not editable, all modifications are a side-effect of operations on the import database.

- `._Release_Names`

  This file is used for the automatic generation of release names.

- `._Switches`

  This file contains the switches that can be set to control compiler options and various other characteristics.

2.6. Subsystems

Subsystems play a central role in the Environment's support for large system development. Large systems may be decomposed into a number of component subsystems each of which may be developed and released independently.

Interfaces between the component subsystems may be strictly controlled to conform to the overall system architecture.

Actual development of a subsystem proceeds in the libraries of the subsystem. Multiple libraries in a subsystem can be used to support both multi-user and multi-target development. Different releases of libraries in the subsystem are used to keep track of alternative implementations of the subsystem, and in the case of multi-target development these alternative implementations capture the variations required for each target. Different spec releases can be used to track alternative exported interfaces of the subsystem.

2.6.1. Spec Releases

Spec releases are frozen copies of libraries with all units other than the exported specs removed [9]. Spec releases serve as the "visible" part of subsystems and are analagous to Ada spec units. Spec releases reenforce the distinction between specification and implementation by providing distinct objects to encapsulate the exported interfaces of the subsystem.

2.6.2. Support for Multi-User Development

As we said above, development in a subsystem occurs in the libraries of the subsystem. When multiple developers need to work in a subsystem simultaneously each developer may have a personal subsystem library in which to work. Personal libraries provide stable areas in which a developer can work free of interference from other developers. Finally, when the work of individual developers needs to be integrated together, source control tools can be used to perform the integration. The source control tools will be described in a later section.

2.6.3. Support for Multi-Target Development

The multiple libraries of a subsystem also support multi-target development. Multi-target development typically requires that some source code differ from target to target while most source code stays the same.

2.7. Configurations

Configurations group together related libraries. For example, a set of libraries in a configuration may represent:

- The set of libraries that together make up either the compilation or execution context of a library.
- A set of subsystem releases that together make up a system release and which must be used together because of various functional dependencies.
- A set of default libraries for a user's session.

A configuration can also be thought of as a *mapping* from a control point to a library of the control point. Considered as a mapping, a configuration provides a way of selecting of library for any control point represented in the configuration.

For example, consider the naming expression `!Environment.Abstract_Types'Library.Map_Generic` where `!Environment.Abstract_Types` is a subsystem and `Map_Generic` is an ada unit that is present in all libraries of the subsystem. The use of the attribute `'Library` specifies that the user's session configuration will

[9]As in Delta, the exported specs must be "complete" in the sense that all suppliers of an exported spec must also be exported.

be used to select the library of `!Environment.Abstract_Types` in which to find the ada unit. Thus, the session configuration provides a mapping from a subsystem to a particular library of the subsystem.

The following sections will present detailed characteristics of configurations and present some specific configurations that are particularly prominent in the functioning of the system.

2.7.1. Configuration Entries

A configuration is composed of a set of *configuration entries*. Each entry is associated with a specific control point and contains information that is used to select a library for that control point. The information in the entry is determined by the kind of the entry.

2.7.1.1. Direct Entries

A direct entry contains a reference to a specific library for the associated control point. When the containing configuration is used to select a library for the associated control point, the directly referenced library is selected.

A direct entry may also contain a *refresh reference* to another configuration (called the *refresh configuration*) which can be used to update the library that is referenced by the entry. When the entry is *refreshed* the library is changed to whatever library is currently selected by the refresh configuration.

2.7.1.2. Indirect Entries

An indirect entry contains a reference to another configuration called the entries *indirect configuration*. When the entry is used to select a library for the associated control point, the library selected by the indirect configuration is used.

There may be a single indirect entry that is used for all control points that have no specific entries. Such an entry is called the *default entry*. When a configuration is used to select a library for a control point and there is no direct or indirect entry associated with the control point, then the configuration referenced by the default entry is used.

2.7.2. Compatibility and Completeness Checking

It is useful in some development methodologies to ensure that certain configurations are *compatible* and/or *complete*. A compatible configuration is one in which all libraries in the configuration may be used togther (ie, all suppliers provide what is required by their clients). Futhermore, a complete configuration is one that is compatible and also is executable because all units are coded and all required bodies are provided. Compatibility and completeness conditions have already been described for compilation and execution configurations. In general, a user can request that any configuration be constrained to be compatible or complete.

2.7.3. Configuration Example - Representing a System Release

In the figure below, we display the contents of a configuration that is used to represent a system release. All of the entries of this configuration are direct and refreshable. In other words each entry refers to a specific library (actually a release in this case) and may be refreshed to the latest release of the specified working library when requested.

```
!Env.Utilities       Alpha_Releases.Rev_0_0_4 Alpha'Latest
!Env.Abstract_Types  Alpha_Releases.Rev_0_0_1 Alpha'Latest
!Env.Kernel          Alpha_Releases.Rev_0_0_1 Alpha'Latest

Checks => COMPLETE
```

Figure 2.9 - Configuration for a System Release

The configuration also specifies that a completeness check is also to be made each time the configuration is changed. This check can ensure that the releases in the configuration can always be executed in the same program.

2.7.4. System Configuration

Each file system has a predefined configuration named `!Machine.System_Configuration`. This configuration contains defaults that are to be system-wide. Authorized users are free to change the contents of this configuration.

2.7.5. Session Configuration

Each session has a session configuration that establishes the defaults for the session. The user is free to set the session configuration to any values that are needed.

The session configuration resides in the users session library and has the name `Session_Configuration`. When a session configuration is created the default indirect entry is always set to reference the system configuration. If the session has no session configuration then the system configuration is used as if it were the session configuration.

The figure below presents a typical session configuration. The user has established two direct entries that refer to to specific working libraries in which he develops program units. Futhermore, there is an indirect entry to a configuration representing a system release that specifies the system release will be used to select a library for the specified subsystem. Finally, for any other control points the system configuration will be used.

```
!Env.Abstract_Types Alpha
!Env.Kernel         Alpha

Indirect =>
  !Env.Utilities !Env.System_Release_0_1_3
  Others         !Machine.System_Configuration

Checks => NONE
```

Figure 2.9 - Session Configuration

3. Access Control

Access control provides a means of specifying which users have the capability to perform certain classes of operations on the objects in the system.

3.1. Users and Groups

Access control in Epsilon is defined in terms of groups of users. Both users and groups are created and managed by the Unix host system. We will say no more here about the creation or management of users by Unix.

When a user is running an environment session the user will run under his Unix user identity. The Unix groups that contain this user identity will be used to control access to the objects in the environment file system.

3.2. Access Control and Control Points

Environment control points are the focus of access control. Control points include independent directories, independent libraries, subsystems, and projects (note that independent libraries are libraries that are not contained in a subsystem or project, and independent directories are those which are not contained in a library, subsystem or project). All access control information for a control point and the objects in it are centralized in the control point and manipulated through the control point. In particular, each control point has an *access control database* which describes the access control characteristics of the control point and the objects within it.

3.3. Access Permissions

The access permissions of a control point determine the groups that may access the control point and its subobjects in various ways. Access permissions may be granted to some set of groups and/or to all users by using the psuedo-group "All_Users". The number of different groups that may be granted a particular permission for an object is a property of the host Unix system.

There are three basic types of permissions available:

- *Owner* permissions determine which users may change the access control information of the control point, delete the control point, or move the control point. In particular, each control point has a CONTROL_POINT_OWNER permission associated with it that determines which users may change the permissions, delete the control point, or move it.

- *Reader* permissions determine which users may resolve names within a structural object (like a directory) or read the contents of an object (like a file). For example, an independent directory has a CONTROL_POINT_READER permission that determines which groups may resolve names in the directory, while the subobjects in the directory have OBJECT_READER permissions which determine the groups that can read the contents of the subobjects.

- *Writer* permissions determine which users may create and delete objects within a structural object, or may modify the contents of an object. For example, an independent directory has CONTROL_POINT_WRITER permissions which determine the groups that may create and delete objects within the directory, and the subobjects in an independent directory have OBJECT_WRITER permissions that determines the groups that may modify the contents of the subobjects.

As an example of access control settings consider a home directory !Users/Tom. The permissions for this directory might be

```
CONTROL_POINT_OWNER  => Tom
CONTROL_POINT_READER => All_Users
CONTROL_POINT_WRITER => Tom
```

which allows the group containing the user "Tom" to change permissions and modify the contents of the directory, and allows all users (specified by the default setting "All_Users") to read the contents of the directory.

If the directory above contains a file called `!Users/Tom/Memo` this file might have the permissions

```
OBJECT_READER => All_Users
OBJECT_WRITER => Tom
```

which would allow only Tom to modify the file,, but would allow all other users to write it.

The actual permissions available within a control point depend on the type of control point, however, all are derived from the basic kinds of permissions listed above.

3.4. Write Permissions

Unlike, Delta and Unix, any modification of objects requires both Read and Write permissions for the user performing the modification. However, this need not be provided by the same group in both cases. For instance, in the examples above the user Tom has OBJECT_WRITER access to the file and since OBJECT_READER access is specified for All_Users, Tom is also allowed to read the file. Thus modifications of the file by Tom will succeed because he has both read and write permission.

3.5. Combined Permissions

In the examples of access permissions above, it was shown that within an independent directory there can be separate permissions for controlling access to directory structure (eg, CONTROL_POINT_READER and CONTROL_POINT_WRITER) and for controlling access to each of the objects in those directories (eg, OBJECT_READER and OBJECT_WRITER). However, because of the complex operations in some structures it is not possible to have separate permissions in the way that they are available in independent directories. In particular, within libraries there are only two permissions, called LIBRARY_READER and LIBRARY_WRITER, which control access to both the structure of the library and the objects within it. Library permissions are organized this way in order to support the compilation capabilities that are central to library management. For example, providing LIBRARY_READER access to a library will allow names to be resolved in the library, the content of objects in the library to be read, and ada units in the library to be compiled against. LIBRARY_WRITER permission allows program units to be create, modified, deleted and also compiled.

3.6. Explicit and Default Permissions

For some objects, such as files in independent directories (like a users home directory) permissions can be either "explicit" or "default". Explicit permissions describe the file by name and give the groups that are allowed to read and write that file. Default permissions apply to all files that are not named.

For example, a fragment of the access control information for a home directory might look like

```
Default permissions for control point subobjects:
        OBJECT_READER => All_Users
        OBJECT_WRITER => Tom

Individual permissions for control point subobjects:
    Foo, Bar
        OBJECT_READER => Tom
        OBJECT_WRITER => Tom
    Secret
        OBJECT_READER => Tom, Jim
        OBJECT_WRITER => <None>
```

In the description, the files "Foo" and Bar have explicit permissions that allow reading and writing only by "Tom", file "Secret" can only be read by users "Tom" and "Jim" and cannot be written by anyone, other files in the same control point can be read by anyone and written by "Tom".

3.7. Access Control Lists

The access control information associated with a control point is composed of a list of access control entries. Where each entry contains a permission, whether the permission is a default or applies to a specific object, and the groups that are granted that permission.

The access control list (ACL) for a control point is a list of all the access control entries for the control point.

In the next sections we will look at the ACLs associated with different kinds of control points.

3.8. Access Control in Independent Directories

In an independent directory, the directory has permissions that are similar, but not identical to those of unix directories or delta worlds and the objects within them.

The permissions are listed below:

- CONTROL_POINT_OWNER
  - Allows the listed groups to change the permissions for the directory and objects within the directory.
  - Allows the listed groups to delete or move the directory.
- CONTROL_POINT_READER
  - Allows the listed groups to resolve names within the directory.
- CONTROL_POINT_WRITER
  - Allows the listed groups to create and delete objects within the directory.

The objects within a control point (eg. files) all have their own permissions that are similar to unix or delta permissions. The object permissions are listed below:

- **OBJECT_READER**
  - Allows the listed groups to read the contents of the object.
- **OBJECT_WRITER**
  - Allows the listed groups to modify the contents of the object.

The file access permissions can be set up to apply to all files by default and can be customized for individual files. Unlike delta the default settings apply to all objects that have not been customized are not simply used to establish initial values. This allows users to easily modify acls in a global way by changing the default.

As an example, consider the access control settings for the home directory below:

```
!Users/Tom : Directory
    Status : Text
    Review : Text
```

The home directory contains a single file "Review" that is to be completely protected, while other files can be examined by anyone but only changed by the owner. The control point acls be displayed as shown below:

```
Directory !Users/Tom
        CONTROL_POINT_OWNER  => Tom
        CONTROL_POINT_READER => All_Users
        CONTROL_POINT_WRITER => Tom

Default permissions for control point subobjects:
        OBJECT_READER        => All_Users
        OBJECT_WRITER        => Tom

Individual permissions for control point subobjects:
    Review
        OBJECT_READER        => Tom
        OBJECT_WRITER        => Tom
```

In particular, note that the file "status" is covered by the file default acls because there is no explicit acl for it, but the file "review" is covered by its own permissions.

3.9. Access Control in Independent Libraries

Because of the requirements of the compilation system access control in independent libraries is treated a little differently. There is no explicit access control for the objects in independent libraries (or releases created from them), rather the access control is coordinated completely at the library level. The permissions for independent libraries are listed below:

- **CONTROL_POINT_OWNER**
  - Allows the listed groups to change the permissions for the library.
  - Allows the listed groups to delete or move the library and its associated releases.
- **LIBRARY_READER**

- Allows the listed groups to resolve names in library.
- Allows the listed groups to read the contents of objects in the library and releases.
- Allows the listed groups to compile against the program units in the library.

• **LIBRARY_WRITER**

- Allows the listed groups to create/and delete objects in the library.
- Allows the listed groups to modify objects in the library.
- Allows the listed groups to compile the program units in the library.

There is no access control associated with specific objects within an independent library or its releases, rather all access is centralized in the library.

Consider as an example the access control information for an independent library in a users home world:

```
Library !Users/Tom/Tools
    CONTROL_POINT_OWNER => Tom
    LIBRARY_READER      => All_Users
    LIBRARY_WRITER      => Tom
```

The owner of the library has the ability to create, delete, and edit objects in the library. All other users have permission to browse through the library and examine objects.

3.10. Access Control in Subsystems

A subsystem contains libraries which are treated similar to independent libraries, and objects outside of libraries which are treated similarly to objects in indendent directories. In particur, the permissions associated with a subsystem are:

• **CONTROL_POINT_OWNER**

- Allows the listed groups to change the permissions for the subsystem.
- Allows the listed groups to delete or move the subsystem.

• **CONTROL_POINT_READER**

- Allows the listed groups to resolve names within the subsystem.

• **CONTROL_POINT_WRITER**

- Allows the listed groups to create or delete objects within the subsystem.

• **LIBRARY_READER**

- Allows the listed groups to resolve names in any library or release in the subsystem.
- Allows the listed groups to read the contents of any objects in the listed libraries.

  - Allows the listed groups to compile against the listed libraries.

- **LIBRARY_WRITER**
  - Allows the listed groups to create or delete objects within a library.
  - Allows the listed groups to modify objects within a library.
  - Allows the program units in the listed libraries to be compiled.

As in an independent directory, objects which are not in a library have their own permissions which can by apply to all files by default or can be customized for individual objects.

- **OBJECT_READER**
  - The listed groups are allowed to read the contents of the object.
- **OBJECT_WRITER**
  - The listed groups are allowed to change the contents of the object.

Consider as an example the subsystem below:

```
!Env/Kernel : Subsystem
    Alpha                 : Library
    Alpha_Releases        : Directory
    Alpha_Tom             : Library
    Beta                  : Library
    Beta_Releases         : Directory
    Memo                  : Text
    Release_Information   : Directory
```

The subsystem might have the following access control display:

```
Subsystem !Env/Kernel
        CONTROL_POINT_OWNER  => Gpa
        CONTROL_POINT_READER => Env_Group
        CONTROL_POINT_WRITER => Kernel_Group

Default permissions for subsystem libraries:
        LIBRARY_READER       => Env_Group
        LIBRARY_WRITER       => Kernel_Group

Individual permissions for subsystem working libraries:
    Alpha
        LIBRARY_WRITER       => Gpa
    Alpha_Tom
        LIBRARY_WRITER       => Tom
    Beta
        LIBRARY_WRITER       => Marlin

Default permissions for subsystem subojects:
        OBJECT_READER        => Kernel_Group
        OBJECT_WRITER        => Kernel_Group

Individual permissions for subsystem subobjects:
        <None>
```

and the groups are defined as

- Gpa, Marlin, and Tom are groups containing the users with that name.
- Kernel_Group is a group containing Gpa, Marlin and Tom
- Env_Group is a group containing Gpa, Marlin, Tom, and all other developers of the system, especially those that might need to import releases from the Kernel subsystem.

Note the following things:

- Anyone in the env_group can read things in subsystem or libraries of the subsystem.
- Only the kernel_group can modify any objects in the subsystem
- Each member of the kernel group may only modify and create objects in a single library.
- Files like the Memo file may are covered by the default file permissions for the subsystem.

3.11. Creating New Control Points

Creation of a new control point requires CONTROL_POINT_WRITER and CONTROL_POINT_READER permission to the enclosing control point.

Permissions for a new control point are inherited from the permissions of the parent control point.

3.12. Operator Capability

If the host Unix system has defined a group called "Rational_Operator" then the members of this group are said to have *operator capability*. Operator capability gives a user the ability to change the access permissions of a control point without CONTROL_POINT_OWNER access to the control point.

Any users with Unix super-user privileges are also granted operator capability.

3.13. Network Access Control

In a network environment, access control privileges are determined by the *remote identity*. One goal is this proposal is to be compatible with remote access protocols and procedures available on the Unix host. Therefore, as in NFS, we consider three possible mode establishing remote identity.

1. In this first mode, the user has no specific identity on the remote machine. Access is provided only through the All_Users psuedo-group.

2. In the second mode, we assume that Unix user and group ids are managed in a coordinated way on all unix hosts. Thus, the remote identity is based on the remote use of the original user id on the originating machine.

3. In the final mode, users specify, a remote user name and password (in a suitably encrypted form) for remote access. Access restrictions are then applied to the remote identity.

3.14. Implementation Considerations

The access control design is heavily influenced by the mechanisms available in unix and by the way kkom (the epsilon environment object management system) is built on top of unix.

The environment file system is implemented in terms of unix directories and files. Environment control points are represented by unix directories. All the other objects in a control point including libraries, ada units, and text files are represented by unix files. To be even more precise, each version of an environment object is represented by a unix file.

In the user model, corresponding objects in different libraries of the same subsystem are different objects. In the kkom implementation, those objects may actually be implemented by references to the same unix file. This allows space to be shared by the libraries and releases of a given subsystem, and makes it possible to create releases very quickly. Methodologies for epsilon will rely heavily on these characteristics.

Because environment versions are represented by unix files, it is always possible for someone to circumvent the environment file system and manipulate the unix files directly. Thus the environment access control mechanisms must protect the files from illegal access through the environment and through unix. At the same time during crash recovery (and in a few other instances) special environment processes must be able to access the unix files in arbitrary ways. The only way to accomplish all of this seems to be to make unix acls the basis for environment acls and have the environment mechanisms coordinate the unix acls in the same way that cmvc_access_control in delta coordinates the low level acls.

Basic unix acls have read, write, and execute access for the owner, one group, and others. AIX (luckily) supports additional groups. In order to accomplish these goals we will have one special user identity which is called "Tippy" (the name has an obscure history) to represent the environment. Tippy is the owner of all unix files and directories that are used to implement the environment file system. Crash recovery, for instance, runs with the Tippy identity and thus is able to freely reconstruct the file system.

User access control is implemented by the group and default fields of the unix acls. Access by users either through the environment or through unix will need to pass checks based on the unix acls.

In delta, there are distinct cmvc_access_control settings for each view of a subsystem, however in epsilon this is not possible because the underlying unix files (upon which the checks are based) are shared between the different libraries and releases of a subsystem. Thus, Read access to the objects in subsystem libraries must be the same for all libraries. However, for write access things are a little different. When an object is written by kkom, a new unix file is created to contain the changes, since this file has only a single usage (while the writes are going on) it is possible for it to have a special set of permissions. Therefore, we allow each library to have its own set of write permissions that determine the settings when the new file is being written. After the writes are committed the permissions are set back to the subsystem-wide read permissions.

Note that the steady state unix permissions do not allow writing by any users other than tippy. This should actually help preserve the environment file system from accidental or malicious destruction by unix programs.