```
  SSSSSSSS    PPPPPPPP      EEEEEEEEEE      CCCCCCCC
  SSSSSSSS    PPPPPPPP      EEEEEEEEEE      CCCCCCCC
SS            PP      PP    EE            CC
SS            PP      PP    EE            CC
SS            PP      PP    EE            CC
SS            PP      PP    EE            CC
  SSSSSS      PPPPPPPP      EEEEEEEE      CC
  SSSSSS      PPPPPPPP      EEEEEEEE      CC
        SS    PP            EE            CC
        SS    PP            EE            CC
        SS    PP            EE            CC
        SS    PP            EE            CC
SSSSSSSS      PP            EEEEEEEEEE      CCCCCCCC
SSSSSSSS      PP            EEEEEEEEEE      CCCCCCCC



LL            PPPPPPPP      TTTTTTTTTT                    11
LL            PPPPPPPP      TTTTTTTTTT                    11
LL            PP      PP        TT                      1111
LL            PP      PP        TT                      1111
LL            PP      PP        TT                        11
LL            PP      PP        TT                        11
LL            PPPPPPPP          TT                        11
LL            PPPPPPPP          TT                        11
LL            PP                TT                        11
LL            PP                TT                        11
LL            PP                TT            ....        11
LL            PP                TT            ....        11
LLLLLLLLLL    PP                TT            ....    111111
LLLLLLLLLL    PP                TT            ....    111111



*START* Job SPEC Req #295 for EGB    Date 15-Feb-84 12:20:01 Monitor: //, TOPS-2
File RM:<MTD.ENV>SPEC.LPT.12, created: 30-Jan-84 11:20:16
        printed: 15-Feb-84 12:20:02
Job parameters: Request created:15-Feb-84 11:12:37   Page limit:144   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASC
```

Chapter 1
Introduction and Overview

1.1. Purpose

The purpose of this document is to specify the overall design of the Rational Programming Environment. The primary audience is the software implementation team. The secondary audience is the documentation and technical consulting teams, who may wish to use this material in developing user documentation, training aids, etc.

1.2. Scope

Only the basic environment model, top-level functionality and overall design structure are addressed in this document. Supporting material provides more detail for specific portions of the environment.

1.3. Background

See the overview book for background information.

1.4. Goals Summary

The primary goal of the Rational Programming Environment is to support medium-to-large-scale software development and maintenance in Ada, with improved productivity and improved quality (reliability, maintainability, etc.) of developed software. The basic approach to this goal is to provide a production system which encourages and supports the use of the modern programming techniques (modularity, abstraction, etc.) that underlie the design of Ada, and provides the best characteristics of highly interactive environments such as Smalltalk and Interlisp.

The environment is designed to be a foundation for bringing additional software engineering technology (requirements analysis, project management, documentation, verification and testing, etc.) into production use over time. In the short term, just making Ada and related programming techniques successful in the market place will have a tremendous impact on the industry.

(Should add more specific technical goals, such as those in previous revisions of env spec.)

1.5. Design Principles

Here we review several basic environment design principles which underly the Rational environment. (Comments indicate that this section

needs to be expanded slightly to define terms better and show more logical progression of design principles).

1.5.1. Integrated

The Rational environment is designed to be a highly integrated environment. Rather than being a collection of loosely-coupled tools, the environment is integrated around a small number of basic concepts applied uniformly. The basic facilities of the environment are intended to be used together and composed to perform higher-level operations. Much of the integrated nature of the environment is the direct result of basing the environment on Ada semantics and providing a completely editor-based user interface (see following).

1.5.2. Ada-based Semantic Framework

A consistent semantic framework is essential in an integrated programming environment. In general, it is not possible to hide basic mechanisms from the user. Developing a consistent semantic framework provides a basis for the implementation of the system and provides a foundation for the user. The semantic framework makes it easier to understand the system operation, to compose tools in new ways, and to extend the use of the system to new applications.

The Rational environment is based upon the the semantics of the Ada language. This approach allows the system to be explained largely in terms of Ada concepts and provides a unified notation for system operations regardless of whether they occur in programs or as user commands.

1.5.3. Editor-Based User Interface

From a human engineering point of view, an editor-based user interface is much easier to use than command-oriented alternatives. It is simpler for the user to point than to describe a location. Using editor operations to interact with the system provides a very uniform user interface based on easily understood and very efficient (for the user) operations. The full power of the editor is always available for viewing and manipulating user input and system output.

A problem with basing the environment on Ada is that the verbosity of Ada, while appropriate for documenting a program, is inappropriate for many kinds of user interaction. This is overcome by supporting an editor-based user interface embodying considerable knowledge of Ada syntax and semantics. Such a user interface can exploit its knowledge to allow the user to perform tasks with the minimum number of keystrokes.

1.5.4. Knowledge-Based

Having an integrated environment allows the system to "know" more about what the user is doing. The Rational environment is designed to

provide a framework for building into the system as much knowledge about the software development process as possible. This allows the system to automatically handle many of the clerical and administrative tasks involved in a large development effort.

One example of building knowledge into the system is reflected in the user interface design. The editor system is designed to allow the incorporation of object specific knowledge. In particular, the Ada editor knows Ada syntax and semantics.

Another example is the compilation manager, which (along with other facilities in the environment) embodies extensive knowledge of Ada separate compilation, allowing it to compute compilation orderings and determine minimal incremental recompilation strategies which would be impossible to reliably determine manually. There are many such examples throughout the environment.

1.5.5. Interactive

The Rational environment is designed to be interactive in all phases of development (not just editing). The system is designed to provide interactive assistance and immediate feedback, like that usually found only in interpretive systems. The goal is to replace the edit-compile-load-debug cycle with a much more interactive environment, where users can write small fragments of programs, get rapid feedback on syntactic and semantic errors, and execute those fragments interactively.

Much of the knowledge normally buried in the compiler has been moved into the editor, where it can provide a more interactive environment for syntactic and semantic analysis. The system also supports very incremental program creation and modification, down to the level of individual declarations and statements. Debugging facilities are integrated with (and in many cases indistinguishable from) basic interactive system operations. These facilities are the first steps toward making the entire development cycle more interactive.

1.5.6. Extensible

The Rational Environment is viewed as an extensible foundation both for expanding existing facilities and adding new facilities. All programs in the system are Ada tasks, with little or no distinction between user programs and system programs, allowing easy expansion. Major facilities have been constructed using generic components which make it easy to add additional subsystem which deal with new types provided by users. New systems can easily be constructed by composing various existing facilities.

1.5.7. Maintainable and Modifiable

Given the advanced and somewhat experimental nature of the environment, it has been important to structure the system and its

components to allow for modification and maintenance. During the course of development, many extensive changes have been made. Use of modularity and abstraction in the construction of the system has controlled the impact of changes and allowed the system to evolve as it has been developed. This evolution will continue over the next several years, and the maintainability of the system will be even more important as the system is used in the field.

Chapter 2
Ada Framework

Ada provides most of the structure and the basic semantic framework in the Rational Development Environment. This chapter describes the foundations of the Rational Development Environment that derive directly from Ada or from extensions to Ada semantics. Section 2.1 defines basic concepts that come largely from pure Ada semantics. Section 2.2 discusses declarations in the environment, including meta-operations that allow declarations to be added to and removed from the environment. Section 2.3 describes the package structure of the environment in terms of the concepts introduced in 2.2. The compilation process, which is closely related to the declaration meta-operations, is addressed in section 2.4. Section 2.5 introduces command and program execution within the environment.

2.1. Basic Concepts

Most of the very basic concepts in the environment come directly from the Ada language definition.

2.1.1. Lexical And Syntactic Considerations

Throughout the environment, notation is based upon the use of Ada syntax. Correct input is always lexically and syntactically valid Ada. The editor system provides extensive support for construct correct input with the minimum effort on the part of the user. There are minor extensions to the basic language in the form of special attributes and notation used in name resolution and for separate visible parts. These largely fall within Ada syntax, and are covered later.

2.1.2. Environment Structure

The environment is structured as a hierarchy of Ada packages (LRM 7). A package specifies a group of logically related entities. The root of the package hierarchy is named Universe. Among other things, the package hierarchy serves as a directory system, providing a mechanism (based on Ada semantics) for declaring and naming entities. The package structure is described in more detail in section 5.3.

2.1.3. Entities

Ada defines several kinds of entities (LRM 3.1). Of primary importance in the environment are entities such as types, objects, and program units.

2.1.4. Declarations

Ada entities may be declared (explicitly or implicitly) by declarations (LRM 3.1). Declared entities in the environment are

represented by Ada declarations within the hierarchy of package declarations.

2.1.5. Types

A type (LRM 3.3) is an entity characterized by a set of values and a set of operations. All Ada types are supported by the environment. Users may define additional types, extending the set of types in the environment. Most of the types of interest in the environment are abstract data types, implemented as Ada private types.

2.1.6. Objects

An object (LRM 3.2) is an entity that has a value of a particular type. Objects are created by elaborating an object declaration or by evaluating an allocator. The set of legal values for a object, and the set of operations available on the object are determined by the type of the object.

2.1.7. Managed Types and Objects

A type is by default an unmanaged type. Certain types are managed types. A managed type is registered in the environment, and operates according to a set of conventions, particularly with respect to storage, permanence, and access control. Many of the most important types in the programming environment are managed types. Objects of these types are known to the environment and are treated with special care. See Section 8, System and Managed Types.

2.1.8. Program Units

In Ada there are four kinds of program units -- subprograms (LRM 6), packages (LRM 7), tasks units (LRM 9), and generic units (LRM 12).

2.1.8.1. Subprograms

Subprograms include functions and procedures, and are the primary mechanism for defining operations on objects. Ada defines the semantics for declaring subprograms, calling subprograms, passing parameters, handling and propagating exceptions, visibility, etc. These same semantics apply in the programming environment, where subprograms replace the more traditional notions of commands and programs.

2.1.8.2. Packages

As mentioned earlier, a package specifies a grouping of related entities, and packages are the main structuring mechanism in Ada and the environment. Packages are the foundation for modularity and abstraction in Ada, and are used in that way throughout the environment.

2.1.8.3. Tasks

Task units allow the specification of concurrency and synchronization. Ada Tasking is the only mechanism for concurrency and synchronization in the programming environment.

2.1.8.4. Generics

Generic units allow the specification of parameterized templates that can be used to instantiate packages or subprograms. Much of the environment is constructed out of generic units, many of which are available for use in extending the programming environment and constructing user programs.

2.1.9. Operations

The operations available on a given type include all of the functions, procedures and entries that take parameters (or return results) of the type, including any derived (LRM 3.4) operations.

2.1.10. Names, Expressions and Statements

Names, Expressions and Statements (LRM 4,5) follow Ada semantics exactly within program units and in most other situations within the environment. In the environment there are some issues of context and dynamic binding that do not arise in Ada. These issues are addressed in section 5.5.

2.1.11. Visibility and Scope

Within the environment, the rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the environment follow those of Ada (see LRM 8). The Ada rules impose some ordering restrictions not normally encountered in directory systems. In practice, these restrictions are no more severe than those found in conventional directory systems, except in certain cases involving user defined data types in local scopes (which are not even supported on conventional systems).

2.1.12. Insertion Points

An insertion point may be placed within any Ada unit. The insertion point must appear in a declaration list of a statement list. When displaying the unit, the insertion point appears as a syntactic nonterminal in a special font. The insertion point unambiguously designates a precise location in the package hierarchy, and is required for several of the meta-operations discussed below.

2.2. Declaration Meta-Operations

Since the form and content of the environment is described by Ada declarations, changing the form and content of the environment involves dynamically manipulating declarations. In particular, new declarations must be added, and existing declarations must be modified or deleted. These kinds of operations fall outside of Ada semantics, yet are essential to the operation of the system. The environment provides a set of declaration meta-operations for performing these functions. Each of these meta-operations can be viewed as taking the environment from one semantically consistent state to another semantically consistent state (in accordance with Ada semantics).

2.2.1. Declaration States

In Ada a declaration is elaborated at runtime. Because of the dynamic nature of the programming environment, it is necessary to distinguish three states for declarations -- source, installed, and elaborated.

2.2.1.1. Source Declarations

A source declaration is in "text" form only, need not be semantically correct, is not elaborated, and can not be referenced semantically.

2.2.1.2. Installed Declarations

An installed declaration is semantically consistent, is known to the environment (which will insure that it remains semantically consistent unless explicitly withdrawn), and may be referenced (statically) by other installed declarations.

2.2.1.3. Elaborated Declarations

An installed declaration may be elaborated (LRM 3.1), in which case it has achieved its runtime effect and may be referenced (dynamically) by executing code.

2.2.1.4. States and Ada Semantics

In the most pure view of the environment as an Ada program, only elaborated declarations are "real", since only they are part of the environment as an executing Ada program. From the point of view of static semantic analysis, all installed declarations (which includes all elaborated declarations) are "real" and can be referenced semantically. From a textual point of view, even source declarations are "real", and the user would prefer that the environment treat them as uniformly as possible.

2.2.2. Primitive Declaration Meta-Operations

The declaration meta-operations are basically concerned with moving declarations between the three states described above. The primitive

operations are described here to provide insight into the basic mechanisms involved. Higher-level (and more convenient) composite operations are built upon these primitives.

2.2.2.1. Manipulating Source Declarations

Source declarations are not carefully controlled by the environment (from the point of view of maintaining global consistency), and may be manipulated directly once access has been acquired. Interactively the user may use the full power of the editor system to perform arbitrary transformations on the source. Programatically, any valid operation on the program representation may be used.

2.2.2.2. Installing Declarations

A source declaration may be installed by selecting a position within an installed declarative part and then attempting to install the source declaration at that point.

The first precondition for successful installation is that the declaration to be installed must be semantically correct. The system will perform static semantic analysis in the context of the installation to check this condition. In the event that the installation fails because of semantic errors, those errors are reported.

The second precondition for successful installation is that installing a declaration must obsolesce no other installed declaration. The system performs change analysis to check this second precondition. In the event that the installation fails because it would obsolesce other declarations, the set of affected units is reported. The rules for recompilation are described in 2.4.2.

Installing a declaration installs all of its subcomponents, but does not install separate subunits (only the stubs). A source subunit may be installed separately by associating it with the corresponding stub declaration (rather than an insertion point) and attempting to install it.

Once a declaration has been installed, it is controlled by the system and cannot be manipulated in an unrestricted manner. In fact, an installed declaration may only be modified by use of the meta-operations described here.

2.2.2.3. Withdrawing Installed Declarations

An installed declaration may be withdrawn if the act of withdrawing it would obsolesce no other declarations. For example, a type declaration may not be withdrawn if there are installed object declarations using that type. The source form of the withdrawn declaration is still available in the environment.

2.2.2.4. Deleting Installed Declarations

Deleting an installed declaration is identical to withdrawing it, except that the source form is no longer available.

2.2.2.5. Elaborating a Declaration

In order for the declaration to be elaborated, the declaration and all of its subcomponents must be installed and the parent declarative part must be elaborated. During the elaboration of the declaration, any references to other entities that are not yet elaborated will result in a program_error exception.

Initially, only declarations for program units and managed objects may be elaborated using the environment meta-operations. Elaborated program units may contain other declarations, resulting in elaborated declarations of any kind. However, the environment meta-operations for incremental elaboration and withdrawal only apply to program units and managed objects.

Unhandled exceptions that are propagated out of the elaboration of a declaration will be treated as errors. The elaboration will be abandoned and the declaration will be left installed, but not elaborated. Any side effects from execution during the abandoned elaboration will not be undone. Exceptions that are handled by the elaboration code itself are ignored by (and unknown to) the environment.

2.2.2.6. Withdrawing Elaborated Declarations

Withdrawing an elaborated declaration changes its state (and that of all its components, including separate subunits) from elaborated to simply installed, removing any entities created during the elaboration of the declaration. Any attempts to (dynamically) reference those entities will result in a program_error exception.

2.2.2.7. A Note on Statements

The facilities for installing, withdrawing, and deleting (but not elaborating) declarations apply to statements in installed (but not elaborated) program units. These facilities for manipulating statements provide an incremental compilation facility, but are less fundamental to the environment model. Eventually, there will be support for statement-level operations on elaborated program units.

2.2.3. Composite Declaration Meta-Operations

The primitive meta-operations can be composed to provide higher-level functions. For example, deleting an elaborated declaration can be achieved by withdrawing it (leaving it as an installed declaration) and then deleting it.

The most important composite operations involve situations where a proposed operation would fail because the operation depends on other declarations that are not yet installed or because the operation would obsolesce installed declarations. In these situations the user may specify that the system is to perform any necessary intermediate operations (withdrawing obsolesced declarations, installing source declarations, etc.) to achieve the desired effect.

All declaration meta-operations implicitly involve compilation, and these composite operations depend heavily upon the facilities of the compilation manager to determine the impact of changes, compute minimal recompilation sets, determine compilation order, and schedule the actual compilation. Compilation management is discussed further in 2.4.

2.2.3.1. Composite Installation

The system provides the following composite installation operations.

1. Predict the impact of performing the installation, but do not perform the installation.

2. Only perform the installation if no other declarations need be installed first and the installation would obsolesce nothing else (this is the primitive install).

3. Same as above, except that installed (but not elaborated) declarations may be withdrawn in order to achieve installation.

4. Same as above, except that elaborated declarations may be withdrawn if necessary.

5. For any of the above, optionally specify that the installation applies to all subunits of the designated declaration.

6. For any of the above, perform the installation, installing any other declarations required to make this declaration semantically consistent.

7. For any of the above, optionally specify that the installation applies to all declarations that would need to be elaborated to elaborate this unit.

2.2.3.2. Composite Elaboration

In general, elaborating a declaration may require installation, so all of the various forms of installation are available as composite elaboration commands, with necessary generalizations to deal with elaboration as well as installation.

2.2.3.3. Composite Withdrawal

The system provides the following composite withdrawal operations.

1. Determine the impact of the withdrawal, but do not perform it.

2. Perform the withdrawal only if no declarations would be obsolesced (the primitive withdraw).

3. Perform the withdrawal, withdrawing any other installed (but not elaborated) declarations that are obsolesced by the change (includes subunits of the current unit).

4. Same as above, except that even elaborated declarations may be withdrawn if necessary to complete the operation.

2.2.3.4. Composite Delete

Since deletion generally involves withdrawal, the forms available for withdraw apply to delete.

2.2.4. Synchronization Considerations

The declaration meta-operations, by their very nature, modify the environment, thus potentially modifying the compilation context for other operations in progress. Compilation is a high-frequency operation in a software development environment, and is even more so in the Rational environment where all command execution, name resolution, program initiation, and other declaration meta-operations involve compilation. Therefore, it is unacceptable to serialize updates to a declarative region with all compilation that involves that region as part of the compilation context.

The system is able to impose minimal serialization because of the incremental nature of compilation in the environment. The system already maintains information down to the granularity of individual defining occurrences, thus it is able to serialize at that level. A declaration meta-operation in progress will block compilation that would be dependent upon the exact change in progress, but does not block compilation that only depends upon other declarations in the same declarative part.

2.3. Package Structure

A package is an entity and a package declaration is a declaration like any other. Thus, applying the meta-operations to package entities allows the environment to grow and change shape. Adding new declarations adds new entities. Adding new package declarations allows new groups of entities. These groups of entities can be viewed as corresponding to directories on conventional systems; however, a package is much more general than a traditional directory (in large

part because the notion of an entity in Ada is much more general than the traditional notion of a file).

The declarations in the environment are structured as a tree of packages with the root being an elaborated package. This set of declarations defines the set of objects, types, and operations available to the user, interactively and programmatically. In that sense, this set of declarations is the environment. All of the programming environment software itself appears in this tree of declarations.

Given the rules described above, the full set of elaborated declarations forms a subtree rooted at the root of the environment. Similarly, the set of installed declarations forms a subtree rooted at the root of the environment and covering the subtree of elaborated declarations.

Declarations within an elaborated package must be installed or elaborated. Installed program unit stubs within an elaborated package may have source subunits associated with them. In addition, there may be insertion points with associated source. However, no uninstalled source declarations may appear directly in an elaborated package.

Declarations within an installed (but not elaborated) package must be installed, but cannot be elaborated. As with elaborated packages, there may be source subunits and source associated with insertion points.

Nothing within uninstalled source may be installed or elaborated. Insertion points within uninstalled source may have additional source associated with them, and stubs may have corresponding source subunits.

Usually, each package will be a separate unit (or Ada Unit) in the sense of Ada separate compilation units and in the sense of separate files in a traditional system. As in Ada, packages, subprograms and task bodies may be separate units, with the slight extension that nested visible parts may be separate subunits. An uninstalled (source) unit may contain arbitrary code that need not correspond to an Ada compilation unit. Each Ada Unit is a separate managed object, and is accessed, modified and stored accordingly (see section 8).

The environment is structured as a single package with many nested subunits, rather than as library units. Using only subunits allows a simpler and more uniform environment model and encourages proper grouping of packages. The major problem with using subunits is the lack of visibility control. In particular, the names space in a deeply nested unit becomes somewhat polluted. Eventually, the environment will support mechanisms for better specifying and controlling visibility. Most likely this will take the form of pragmas that indicated that a package defines a closed scope except for specifically imported entities. Warnings would be provided if those stricter visibility restrictions are violated.

The environment provides facilities for traversing the package structure, including facilities to get from a package visible part to its body (and vice versa), to visit every declaration within a package, to visit the parent package of some declaration, and to visit separate subunits. There are facilities for retrieving various attributes associated with each package and each declaration of a managed object in the package structure. These attributes include time of creation, time of last modification, size, etc. These attributes are described in Section 8 (System and Managed Types).

2.4. Compilation Considerations

The complexity of managing compilation of large programs, the computational expense of Ada compilation, the importance of semantic consistency in the environment, and the goal of providing an interactive environment all lead to the need for an automatic, incremental and reliable compilation management system. In constructing large programs the user will require some control over the compilation process and the system must carefully allocate resources. The compilation management facilities to accomplish these goals are covered in this section.

2.4.1. Libraries

Libraries and library units are not obviously consistent with the simplified model of the environment as a single Ada program. The library facilities described here are designed to provide complete compatibility with the language requirements, while integrating libraries and "main programs" into the overall environment model.

2.4.1.1. Library Objects

Libraries are objects of the managed type Library, and are represented as object declarations in the package hierarchy. Once a variable of type library has been elaborated, A user may view the value of the library, which appears very similar to an Ada package body (substituting the word library for package). The contents of the library will appear as a restricted subset of Ada declarations. The legal declarations in a library are program unit stubs, renaming declarations that denote installed program units, use clauses that denote either installed packages or libraries, and pragmas. Only directly within a library may use clauses denote library variables.

2.4.1.2. Library Units

The separate Ada Units contained in the library will have any necessary WITH clauses and are treated as library units in accordance with Ada semantics. The library units may in turn have subunits. Library units may be installed, but can not be elaborated in place.

Units within the library may be named as if the library formed a

package shell, i.e. package P within library X within the the elaborated package D is named D.X.P. This form of name is of somewhat limited use, since by definition it denotes an unelaborated entity. But it is useful in certain applications.

2.4.1.3. Library Context

By nature a library unit is a closed scope with the context limited to other units specifically named in WITH clauses. The environment resolves the simple names in the WITH clauses to entities visible at the end of the library (viewing the library itself as a declarative region, nested at the point of the library variable declaration). This means that WITH clauses may denote any other unit in the library, any unit introduced by a renaming declaration in the library, any unit introduced by a use clause in the library, or any unit directly visible in the environment of the library declaration.

Elaborated packages in the package hierarchy that are visible to library units (either directly, through a rename, or through a use clause) provide linkage between library units and the elaborated environment. This is particularly important in that all system facilities (including Input/Output) are only available through elaborated packages.

Units in other libraries may be made visible to library units through use clauses or renaming declarations. This allows the use of multiple libraries in constructing large systems.

2.4.1.4. Installing Library Units

The program units in the library may be installed, and all of the operations available for installing and withdrawing declarations apply within the library. However, no declaration within the library may be elaborated in place. Within libraries, the declaration meta-operations serve as very efficient facilities for minimal recompilation, but they are not as fundamentally important as they are in the elaborated package hierarchy. The declaration meta-operations would allow declarations to be added to a low-level visible part without causing massive recompilation, but are not essential to properly constructing the library.

2.4.1.5. Main Programs on the R1000

The Ada language definition introduces the concept of main programs as well as libraries. In the elaborated package hierarchy there is no need for a notion of main program, since any procedure or entry can be called once it is elaborated. However, for constructing programs using library units the environment does support a concept of main program.

The system provides a load operation that takes as parameters a location in an elaborated package, the name of the main program to be

constructed, and a subprogram library unit. The load operation constructs an elaborated subprogram at the designated location, with a specification that matches that of the library unit (substituting the user specified name for the new main subprogram). The load operations computes the transitive closure of all units required by the designated main unit, performs any necessary completeness checking, and computes the proper elaboration order. In cases involving multiple libraries, the load operation will provide warnings in the event that the transitive closure includes two units with the same name.

The main subprogram library unit may have parameters; however, the types of the parameters must be types whose declarations are elaborated and visible at the location where the main subprogram is to be elaborated. Once elaborated, the main subprogram may be called like any other subprogram declaration.

The elaborated main subprogram declaration has no body declaration, but the system inserts the pragma LIBRARY_PROGRAM (Library_name, unit_name) immediately after the elaborated declaration. Conceptually, the body of the main program is an invisible system constructed subprogram that elaborates all necessary library units, elaborates the main subprogram, and then calls the main program passing along any parameters. This correctly follows Ada semantics, where all the library units are elaborated on each invocation of the main program.

Once a main program has been installed and elaborated, changes to library units used to construct the main program do not obsolesce the main program. However, debugging facilities may be somewhat restricted in cases where library units have been changed after the main program was replaced. This implies that the system must retain code segments for library units until there are no elaborated main programs which depend upon those code segments.

(issues remain with substituting body only and priority pragma)

2.4.1.6. Target Considerations

While program units in the elaborated package hierarchy necessarily execute on the R1000, library programs may be constructed for execution on other machines. A library may include a TARGET pragma before the first declaration in the library. The TARGET pragma has a single parameter, which is an object of the managed type TARGET. The object of type TARGET provides information used by the compilation system to construct programs for a foreign target machine.

The TARGET specifies an object of type ADA_MANAGER.ID that will be used to obtain the standard package for compilation of all units in the library. Other language required packages (machine code, system, etc.) may be included directly as units in the library, or may be imported by means of a renaming declaration, use clause, or direct visibility.

The compilation system provides a set of couplers for dynamically adding support for different target machines. The TARGET specifies keys that are used for invoking machine dependent processing during semantic analysis, for invoking code generation, and for performing any link/load operations.

2.4.2. Incremental Compilation

The declaration meta-operations (including application to statements) provide the user visible incremental compilation facilities. Essentially, the system supports incremental compilation of individual declarations and statements. Here we cover rules governing when incremental compilation may be applied and the impact (in terms of obsolescing other declarations) of performing incremental compilation.

2.4.2.1. Impact of Installation

When a new declaration is installed, it may hide existing declarations defined in outer scopes. Any units that reference these hidden declarations within the scope of the new declaration will be obsolesced. In addition, the new declaration may overload existing declarations appearing in the same scope as the new declaration or in scopes closely containing the new declaration or closely contained by the scope of the new declaration. References to these overloaded declarations could become ambiguous after the introduction of the new declaration. Units containing such ambiguous references will be obsolesced.

2.4.2.2. Impact of Withdrawal

When a declaration is withdrawn, all units that reference that declaration will be obsolesced. In addition, ambiguous references may be introduced if the withdrawn declaration had previously hidden overloaded declarations. Again, units containing such ambiguous references will be obsolesced.

2.4.2.3. Scope of Impact

For both installation and withdrawal, the scope of a declaration can be extended through the use of expanded (qualified) references and through the use of USE clauses. When determining the set of units to be obsolesed, this extend scope must be fully considered.

2.4.3. Computing Compilation Requirements

In order to support the composite declaration meta-operations defined above, the system must provide support for computing the set of declarations that must be installed before a declaration or subunit can be installed. In addition the system must be able to compute the compilation order required to install a set of obsolete units (and everything they depend on). In general this will require cognizance of source declarations in the environment that must be installed to allow other installations to proceed.

In the most general case, where there are large numbers of uninstalled source units, it is difficult for the system to determine whether a particular unit has semantic errors or is dependent on installation of other source units. To constrain the problem somewhat, and to provide more user control, the system distinguishes between a source unit that is "complacent" and one that is "eager".

Essentially, an eager unit is a syntactically correct compilation unit that is ready to be installed, while a complacent unit is one that is incomplete or requires changes before consideration for installation. A source unit is initially complacent. The user may explicitly indicate that a source unit (or all source units within some unit) should be considered eager (or complacent). An unsuccessful attempt to install a complacent unit will make it eager (if it is syntactically valid). An installed unit that is explicitly withdrawn, but not changed, becomes complacent. Indirectly obsolesced units remain eager. Modifying a unit doesn't change its eagerness, except in the case where an eager unit is made syntactically invalid, becoming complacent.

When computing compilation requirements as the result of a declaration meta-operation, the system will only consider eager source units. Eager units are also candidates for automatic anticipatory compilation, as described in the next section.

2.4.4. Scheduling and Controlling Compilation

Compilation must be scheduled efficiently to optimize use of machine resources and to balance system load. Scheduling must account for all pending activities, must recognize when recent updates change compilation requirements, and must prevent redundant compilations.

Some compilation is closely tied to user interactions. For example, the user will typically view installing an object declaration as an interactive operation. In this case compilation occurs immediately on demand.

The user may request that compilation occur asynchronously. The system provides facilities for the user to monitor the progress of such compilations, including the ability to change priorities, delay compilation, and cancel compilation. The system will then schedule compilation in accordance with user direction, system load, and competing requests.

There is considerable opportunity to perform compilation (both semantic analysis and code generation) in the background in anticipation of user requests. However, lack of experience with system operation, limited heuristics for initiating compilation, and uncertainty about system performance constraints, preclude construction of such mechanisms at this point. Initially, all compilation will be the direct result of user actions.

2.4.5. R1000 Code Generation

R1000 code generation must support incremental compilation to the granularity of individual statements and declarations.

Code generation must be coordinated with activities involved in performing environment meta-operations. In particular, the code generator must cooperate in maintaining consistency between the runtime representation of entities and the various permanent data bases maintained in the environment.

Invocation of the code generator provides control over code generation and optimization parameters, including support for debugging.

Semantic analysis will always occur as the direct result of installing or elaborating some declaration, and is easily controlled by the user; however, code generation is more problematic. In order to provide rapid feedback on semantic errors (and to conserve resources) installation does not result in immediate code generation.

Code generation must occur before elaboration, and part of the elaboration operation involves completing any necessary code generation. However, deferring all code generation until elaboration makes elaboration very expensive. Because code generation can only be applied to installed units, and because it is easier to construct heuristics for invoking code generation, code generation is much more amenable to fully-automatic mechanisms. However, as discussed above, initially all compilation will be the result of explicit user action.

The system provides an operation for explicitly invoking code generation on a set of units. Optionally, the code generation operation may be applied to all subunits of any of the named units, or to all units required to elaborate a particular unit.

2.4.6. Importing Source

The system includes facilities for parsing a text object, or a set of text objects, and inserting the resulting units into a specified library.

2.5. Execution

Within the programming environment, all activity is viewed as the execution of Ada code by some task. In particular, command execution is simply the execution of some statement by a task acting on behalf of a user session (see section xxxxx). Program execution is the same as command execution, where the statement is a procedure call to the desired subprogram. Once execution is initiated, the semantics of execution are essentially those specified by Ada semantics.

2.5.1. Context

Execution in Ada is only meaningful in terms of some particular context.

2.5.1.1. Static Context

Ada requires a static context that is used during compilation to perform static semantic analysis. In particular, the static context provides the environment for resolving names and determining the meaning of expressions. The static context can be viewed as a point within the installed environment. A position within the installed environment determines what entities are visible, and how the meaning of any Ada expression will be resolved.

2.5.1.2. Dynamic Context

The dynamic context corresponds to the actual runtime environment where execution occurs. In simple cases, there is a one-to-one correspondence between the static and dynamic environment, and the dynamic environment can be thought of as a point within the elaborated environment. In general (particularly when debugging) full specification of the dynamic environment must deal with all the complexities of nested recursive calls, dynamically allocated task objects, etc.

In the general case, the dynamic context must specify the runtime environment down to the level of a specific subprogram activation record. A task performing the execution has its runtime environment set up so that execution occurs as if the task (or at least the procedure frame running on the task) is nested in the correct environment. The runtime manager provides these execution facilities, exploiting special facilities in the architecture (i.e., Establish_Frame). (Subcontract additional detail to Phil).

2.5.1.3. Session Context

(This whole section should perhaps move to Chapter 4). Each session has a context defined for command execution. The (dynamic and static) context for a session defaults to the end of the body of the users package (associated with the session). The user may change his context on a session-wide basis. A job executing on the behalf of a session inherits its context from the session context at the time of job invocation.

Frequently used commands and other frequently referenced entities will have renames directly in the outer package of the environment so that they will be visible in every environment (except when there are intervening hiding declarations).

Support for session-wide abbreviations that are visible regardless of the current setting of the session context is provided through a

command context declare block. When the session context is established, it is as if this declare block is nested at that point. Then commands are interpreted as if they occur where the statement list would appear in the declare block. The only declarations allowed within the declare block are renaming declarations and use clauses. The declarations are further restricted to fully qualified names for the renamed or used entities, since these declarations must maintain their meaning when the context is moved. The session context declare block is part of session state, and may be edited by the user. (this facility may not be implemented for some time)

2.5.2. Naming Entities

Entities can be named (from the proper context, in accordance with Ada visibility and scope rules) by using Ada names (LRM 4.1).

In addition to simple Ada naming, the environment supports special attributes that extend Ada naming. In particular, there are attributes for denoting the declaration rather than declared entity, and there are attributes for designating versions. (specify attributes)

String names are also supported by the environment. String names are treated as extended Ada names, which are more flexible with respect to visibility and allow more precise designation of runtime environments. (more precisely ...)

The environment also supports a variety of mechanisms for implicit naming, the most notable being selecting an object with the editor.

2.5.3. Command Execution

Commands are statements that are executed in the context of a particular session. The command may dynamically reference any entity visible in the specified context. In addition, commands often take implicit parameters (currently selected object, current window, etc). These implicit parameters are computed by the called command based on the session and job (see ...). (restrictions etc.)

2.5.4. Program Execution

There is no real distinction between command execution and program execution. Any elaborated subprogram or entry in the environment may be invoked.

(communication, invoking others, composition, process issues)

Chapter 3
System and Managed Types

There are a number of types that that are fundamental to the design, implementation and use of the system. Some of these are primitive, unmanaged types that are used to build the basic environment mechanisms. Others are managed types, built upon the facilities of the object management system.

This section describes the primitive system types, the basic object management facilities and concepts, and certain key managed types.

Managed Objects and Object Ids Actions Object Managers storage, permanence, synch, access control. Users, Groups and Sessions Device managers Address Spaces, Volumes, Segmented Heaps, and Heap Managers Ada Units and Diana Directories (Dependency Data Base Buried in here?) Files Jobs

Chapter 4
User Interface

Chapter 5
Implementation Architecture

Table of Contents