11:01:14, Apr 18, 1984

SPEC.MSS

EGB

```
@Make(Manual)
@Modify (Verbatim, above 1, below 1)
@Modify (Senumerate, above 0)
@Modify (Description, LeftMargin +8, Indent -8)
@String(DocumentTitle=<"..." Spec       >)
@String(Draft=<DRAFT 5>)
@Set(Page=1)
@Comment{
@String(Base="")
@String(Add="")
@String(Major="")
@String(Frill="")
}
@String(Base="[1]")     @Comment(Base system requirements)
@String(Add="[2]")      @Comment(Add-on features that are important and easy)
@String(Major="[3]")    @Comment(New features requiring significant work)
@String(Frill="[4]")    @Comment(Nice to have, but clearly in the future)
@Include(Intro.mss)
@Include(Ada.mss)
@Include(System.mss)
@Include(User.mss)
@Include(Impl.mss)
```

11:01:09, Apr 18, 1984

INTRO.MSS

EGB

@Part(Introduction, root "spec")
@Chapter(Introduction and Overview)

@Section(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.

@Section(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.

@Section(Background)

See the overview book for background information.

@Comment{Include(Background.mss)}

@Section(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.)

@Section(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).

@SubSection(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).

@SubSection(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

11:01:06, Apr 18, 1984

IMPL.MSS

EGB

```
@Part(ImplementationArchitecture, root "spec.mss")
@Chapter(Implementation Architecture)
```

11:01:03, Apr 18, 1984

BACKGROUND.MSS

EGB

@SubSection(Market Need) Rapid advances in hardware technology, resulting in more powerful, inexpensive and reliable computers, have lead to a proliferation of computer-based products and services, involving increasingly complex software. Software costs have increased to the point where they represent the majority of system development and maintenance costs and have become the major constraint on the application of computer technology.

In many of the high-growth sectors of the economy it is recognized that product differentiation often comes from the software component of a product. Companies in these (and other) markets have recognized that a major influence on growth and profitability in the next decade is their ability to bring reliable, maintainable and modifiable software to market in response to rapidly changing market conditions. These companies have begun to invest in improved software technology.

@SubSection(The Software Crisis)
It is generally recognized that there is today a severe software crisis. The symptoms appear in the form of software that is nonresponsive to user needs, unreliable, excessively expensive, untimely, inflexible, difficult to maintain, and not reusable.

Advances in software technology and increasing economic pressure have begun to break down many of the barriers to improved software productivity. The Rational system is designed to remove the remaining barriers by providing an integrated, production system with comprehensive support for the Ada programming language and associated programming technologies.

@SubSection(Software Technology)
Recent advances in software technology provide the foundation for the Rational Development Environment. These advances in methodology, languages, and environments all reflect two fundamental changes in the basic paradigm for programming.

The first change is a shift from concern with relatively local issues to a concern with the issues of programming-in-the-large. While statement level issues (such as using structured flow of control constructs) should not be ignored, in a system with hundreds of thousands of lines of code the important issues involve decomposing the system into levels of abstraction and then into modules, defining abstract interfaces, providing tools to maintain the integrity of the system design in the face of continuous modification, and addressing issues such as concurrency, storage management, and protection.

The second fundamental change in our view of programming is a shift from from an imperative view of programming to a declarative one. Winograd describes this change as follows.

@Begin(Quotation)
We need to shift our attention away from the detailed specification of algorithms, toward the description of the properties of the packages and objects with which we build. A new generation of programming tools will be based on the attitude that what we say in a programming system should be primarily declarative, not imperative. The fundamental use of a programming system is not in creating sequences of instructions for accomplishing tasks, but in expressing and manipulating descriptions of computational processes and the objects on which they are carried out.

@End(Quotation)

@Paragraph(Programming Methodology)
Programming methodology "is concerned with those aspects of the current software problem which result from our human limitations in dealing with complexity."

Two fundamental tools for addressing complexity are modular decomposition and abstraction. Proper decomposition enables one to solve subproblems independently and insures that solving the subproblems solves the original problem. Abstraction serves to reduce the amount of detail that must be comprehended at any one time by providing a mechanism for separating those attributes of an object or event that are relevant in a given context from those that are not.

Traditional software techniques have not adequately supported either modularity or abstraction. In fact, many of the techniques used in large systems have strongly discouraged modularity and abstraction (i.e., functional rather than object-oriented decomposition, using shared data definitions in global data dictionaries (or common blocks) as the approved means of communication between software components, etc.).

Techniques such as stepwise refinement, information hiding, encapsulation and data abstraction are all designed to help manage complexity through application of the basic concepts of modularity and abstraction.

From the management point of view these techniques can provide increased visibility and control over the development process. "Using these tools, large systems may be organized into relatively independent levels of abstraction. Each level of abstraction is a self-contained set of object types along with the operations defined to manipulate those types. Once a system has been defined in such terms, the job of implementing and verifying each level becomes much more limited--and hence more manageable."

@Paragraph(Programming Languages)
It is sometimes argued that modern programming methodologies are independent of the programming languages used for implementation. However, a good language (the central tool in any programming environment) can reinforce and support the use of good programming techniques. "It is obvious that a reasonable language is a prerequisite to communicating something as intangible as an abstraction; it is less obvious, but equally true, that a reasonable language is a prerequisite to the creation of such abstractions." Since programs are not just developed and then abandoned, but rather grow and evolve, it is important that the program source accurately record the refinements, abstractions and decompositions of the design process.

Here we primarily consider the language Ada, since its technical merits and market acceptance have lead to its selection as the foundation of the Rational Development Environment.

Languages such as Ada provide direct support for using and enforcing modularity and abstraction. The package mechanism in Ada supports the construction of modules with a visible specification and a separate body. The language ensures that the module (including any data structures, tasks, etc. in its implementation) can only be accessed through the interface defined in the package specification.

Ada also provides facilities for defining arbitrary abstract data types. The language enforces that only the implementation of the abstraction has access to the concrete operations on objects of the type; users of the type have access only to the abstract operations defined on the type. The rich set of (dynamic) data structures, the strong type checking, and the derived type mechanisms in the language make it possible to construct systems by composing lower-level abstract types to build higher-level types.

The generic facilities in Ada allow construction of reusable software components with parameterized and more completely specified interfaces. Rapid prototyping is supported by building up initial implementations using general algorithms and data structures which are generic packages. Once the system design has stabilized, generic instantiations can be replaced by more specific packages tailored for efficiency in the particular situation.

Ada also includes high level facilities for concurrency and synchronization and for handling exception conditions. These facilities are reasonably well integrated with the facilities for encapsulation and abstraction.

The design of the Ada language, and its use in building systems, reflects the shift toward concern with programming-in-the-large. The main concerns in designing and implementing a large Ada program are determining the proper decomposition into packages, defining the abstract types that represent the objects of interest in the system, identifying reusable components whose construction can simplify the system implementation, defining (generic) packages to encapsulate complex protocols, determining synchronization and storage management properties of the system, and structuring the software to reflect a decomposition into levels of abstraction.

Systems built in Ada also reflect the shift toward an declarative view of programming. A considerable portion of the actual source code in an Ada program is concerned with specifying the system decomposition, the abstract interfaces, and the abstract data types. Ada programs tend to be built in levels of abstraction, providing multiple descriptions of systems and subsystems. The extensive declarative information contained in Ada programs allows the compiler to perform a considerable amount of static consistency checking, detecting many common programming errors at compile time.

@Paragraph(Programming Environments)
The need for improved tools in software development has lead to considerable interest in programming environments...

language based env -- lisp, interlisp, smalltalk, apl.

Goal: env that support modern prog. meth, interactive and self-applicative nature of lisp env., support production programming, rapid prototyping, etc.

@Paragraph(Machine Architectures)
Support for Ada execution plus programming environment needs...

11:00:58, Apr 18, 1984

ADA.MSS

EGB

@Part(AdaFramework, root "spec")
@Chapter(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 @ref(BasicConcepts) defines basic concepts that come largely from pure Ada semantics. Section @ref(MetaOperations) discusses declarations in the environment, including meta-operations that allow declarations to be added to and removed from the environment. Section @ref(PackageStructure) describes the package structure of the environment in terms of the concepts introduced in @ref(MetaOperations). The compilation process, which is closely related to the declaration meta-operations, is addressed in section @ref(compilation). Section @ref(Execution) introduces command and program execution within the environment.

@Section(Basic Concepts)
@Label(BasicConcepts)
Most of the very basic concepts in the environment come directly from the Ada language definition.

@SubSection(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.

@SubSection(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.

@SubSection(Entities)
@Label(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

@SubSection(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).

@Paragraph(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.

@Paragraph(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.

@Paragraph(Tasks)
Task units allow the specification of concurrency and synchronization. Ada Tasking is the only mechanism for concurrency and synchronization in the programming environment.

@Paragraph(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.

@SubSection(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.

@SubSection(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.

@SubSection(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).

@SubSection(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.

@Section(Declaration Meta-Operations)
@Label(MetaOperations)
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).

@SubSection(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.

@Paragraph(Source Declarations)
A source declaration is in "text" form only, need not be semantically correct, is not elaborated, and can not be referenced semantically.

@Paragraph(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.

@Paragraph(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.

@Paragraph(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.

@SubSection(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.

@Paragraph(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.

@Paragraph(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 @ref(Incremental).

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 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.

@Paragraph(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.

@Paragraph(Deleting Installed Declarations)
Deleting an installed declaration is identical to withdrawing it, except that the source form is no longer available.

@Paragraph(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.

@Paragraph(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.

@Paragraph(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.

@SubSection(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 @ref(Compilation).

@Paragraph(Composite Installation)
The system provides the following composite installation operations.

@Begin(Enumerate)
Predict the impact of performing the installation, but do not perform the installation.

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

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

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

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

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

For any of the above, optionally specify that the installation applies to all declarations that would need to be elaborated to elaborate this unit.
@End(Enumerate)

@Paragraph(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.

@Paragraph(Composite Withdrawal)
The system provides the following composite withdrawal operations.

@Begin(Enumerate)
Determine the impact of the withdrawal, but do not perform it.

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

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

Same as above, except that even elaborated declarations may be withdrawn if necessary to complete the operation.
@End(Enumerate)

@Paragraph(Composite Delete)
Since deletion generally involves withdrawal, the forms available for withdraw apply to delete.

@SubSection(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.

@Section(Package Structure)
@Label(PackageStructure)
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. 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. 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.

Nothing within uninstalled source may be installed or elaborated. Stub declarations in a source unit may have source subunits associated with them.

Usually, each package will be a separate 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).

@Section(Compilation Considerations)
@Label(Compilation)

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.

@SubSection(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.

@Paragraph(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.

@Paragraph(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.

@Paragraph(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.

@Paragraph(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.

@Paragraph(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)

@Paragraph(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.

@SubSection(Incremental Compilation)
@Label(Incremental)
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.

@Paragraph(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.

@Paragraph(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.

@Paragraph(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

@SubSection(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.

@Section(Execution)
@Label(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.

@SubSection(Context)
Execution in Ada is only meaningful in terms of some particular context.

@Paragraph(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.

@Paragraph(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).

@Paragraph(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)

@SubSection(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.

@SubSection(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.)

@SubSection(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)

11:00:53, Apr 18, 1984

STRUCTURE.OUTLINE

EGB

Layer
    Subsystem
        Exported Interface (Packages)

Base
    Base

Kernel   ?? see gpa about dividing this up.
    Machine_interface
    Kernel_Debugger
    Device_io
    Virtual_Memory

Abstract_Types
    Abstract_Types

Object Management
    Mechanisms
    Basic_Managers
    Ada_Management
    Runtime_Management

Compilation
    Parser
    Pretty_Printer
    Semantics
    Change_Analysis
    Code_Generation
    Compilation_Management

Directory
    Directory

Core_Editor
    Core_Editor

Object_Editor
    Mechanisms
    Ada_Editor
    Text_Editor

11:00:19, Apr 18, 1984

USER.MSS

EGB

@Part(UserInterface, root "[mtd.env]spec")
@Chapter(User Interface)
User interaction with the system and his own programs is through the editor. The users' investment in learning these facilities is repaid in increased functionality and more uniform interface.

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

@Begin(Enumerate)
The visible interface is a multi-window editor that provides a core set of facilities for handling user input, editing and screen management. This is called the @I(Core Editor).

The type-specific, object-knowledgeable portion of the editor is called the @I(Object Editor). Which object editor is used depends on the type of the object (entity). Although specific object types may require specific operations, there is a common set of operations requiring type knowledge that is provided by all (or most) object editors. These are referred to as object operations.
@End(Enumerate)

@Section(Core Editor Concepts)
The Core Editor provides character editing facilities. This section is an attempt to define and briefly describe these.

@SubSection(Screen Structure)
A @I(Screen) is the entire contents of the display at a particular time. Screens are made up of opaque rectangular areas, called @I(Windows), arranged in a possibly overlapping pattern. More than one screen can be maintained by a session to facilitate changing from one multi-window activity to another (though not initially).

Windows are composed of @I(character positions) and, optionally, @I(borders). Borders are used to visually delineate windows. The character positions represent a bounded rectangular region of the quarter-plane of an @I(Image).

An image is an array (Natural) of lines, each consisting of an array (Natural) or characters. At any time, each image has a specific number of lines, each of which consists of a specific number of characters. Lines beyond the end of the image and characters beyond the end of lines are treated as blanks on the window. A @I(word) is a portion of a line delimited by separator characters. Word boundaries are completely syntactic and are handled by the Core Editor.

An image is the user-readable representation of an entity in the system. One of the functions of the Core Editor-Object Editor combination is to provide mechanisms to reflect changes from the readable to the internal representation and back within the editing paradigm. The image is the Core Editor representation of the object.

@I(Superwindows) are collections of windows that are logically linked and maintained to be physically contiguous. Because of this logical connection, superwindows are commonly referred to as windows composed of windows. The most common example of a superwindow configuration has the following characteristics:
@Begin(SEnumerate)
A window containing the image of an object to be edited. This is called an @I(object window). It normally has top and side borders.

A @I(banner window) that explains the purpose and status of the object windows. Normally presented in a different font than its associated object window, with side borders. Although banners are implemented as windows, no editor operations will be provided initially for their manipulation.

A @I(command window) that is used for entering Ada statements to be compiled and executed to perform actions on the user's behalf. Normally has bottom and side borders.

The appearance of the whole is of a single box, surrounded by borders, with the command window separated from the object window by the banner.

The command window is an object window in its own right.
@End(SEnumerate)

There is a system-managed output window that serves as the destination for general error messages and system output. Its associated banner is used to depict the state of the session.

@SubSection(Cursors)
The physical screen has an apparent @I(cursor), marking the current position of the user's focus of attention (from the editor's point of view). This is called the @I(screen cursor).
If the cursor is within a window, it represents the:
@Begin(SEnumerate)
@I(Image cursor): (line, column) in the image on the window.

@I(Window cursor): (line, column) on the window.
@End(SEnumerate)
If the cursor is not within a window, the image and window cursors, and operations that depend on them, are undefined.

For each type of cursor, there are a variety of operations to specify its position. Changing on the position of one type of cursor often, but not always, changes the position of others.

The window and image cursors are closely linked. When they move in concert, the screen cursor moves across the window; when they move separately, the image scrolls on the window (in addition to possible screen cursor motion). The rest of this section deals with image and screen cursors and their relation to each other, ignoring window cursors to simplify the discussion.

Moving the image cursor causes the screen cursor to move. Moving the image cursor to a position that is currently not on the window causes the window to be scrolled. The screen cursor will not leave the current window because of an image cursor motion.

Moving the screen cursor causes the physical cursor to move without changing the image cursor. Having moved the screen cursor to a position within a window, any operation involving either the image cursor or the underlying image causes the image cursor for this window to be moved to the screen cursor.

Each window has a current image cursor position. Operations that change the focus to a previously visited image (and do not specify a particular position in that image) will place the cursor at the previous image cursor position. Thus, moving away from a window using screen cursors leaves the image cursor at the point of last interest rather than at the exit position.

A @I(mark) is a saved image position. Marks are stored in terms of absolute image positions and do not change to adjust for inserted/deleted lines/characters.

@SubSection(Fonts and Designations)
Each character that appears in a window is displayed in some @I(font). The appearance characteristics of fonts vary from terminal device to terminal

device, but different fonts on the same device commonly differ in boldness, brightness, video presentation (reverse or normal), underlining and blinking. More advanced devices allow traditional font distinctions such as italics. Specific choices are terminal-specific, but banners are typically represented in reverse-video, keywords are underlined or emboldened, etc.

Fonts are used to convey the usage of the characters displayed. In some cases the distinction is for user emphasis (e.g. keywords). More commonly, fonts are used impart a different meaning to the characters displayed. Each window has a default font. Characters that represent themselves and not otherwise special appear in this default font. There are, at least potentially, more different uses for fonts than a particular terminal supports. When this occurs, the same font will be used for more than one meaning, hopefully in a way that is not confusing.

Each non-printing ASCII character can be represented by its traditional position in the Control- sequence. Each of these is printed as a font-changed version of its base character. For example, ASCII.SOH (aka Control-A) might be represented as a reverse-video A.

Many editor operations require one or more implicit operands to accomplish the desired goal. The current cursor is one such implied operand; the current selection is another. Two kinds of selection are available: text and object. Text selections are formed by marking the first and last character positions to be selected, thereby selecting the text in between. Object selections are accomplished by various Object Editor operations. These operations select a region of the image that corresponds to a meaningful portion of the underlying object.

For either form of selection, the region of the image corresponding to the selection is presented in a font to provide visual feedback as to the extent of the selection. It is possible to convert object selections to text selections, so either type is acceptable to text operations. Text selections need not have any relation to object boundaries and are not appropriate for object operations. Even so, the font used for the two types of selection is typically the same, relying on the user to remember how the selection was formed.

A @I(designation) is one of three forms of meta-text that object editors can insert into an image to convey special meaning and support structured text within the editor paradigm. Designations are presented in non-standard fonts.

@I(Elision) is the process of removing detail from an image. The editor supports this by allowing a section of the object to be elided and represented by an @I(Ellipsis) mark (typically "...", but more meaningful phrases are possible). The ellipsis mark is presented in a special font and is treated specially in Core Editor operations. The ellipsis is a placeholder for the elided section of the image. As such, the Core Editor treats the entire ellipsis as a object, rather than as a collection of characters. Specifically, it is not possible to change individual characters. Moving or copying the ellipsis only moves or copies the underlying object if done by object operations.

A @I(Prompt) is a placeholder for an empty place in the object that the user may want or need to fill. The prompt is an extension of the traditional notion of prompt as one or more characters printed at the beginning of a command line to signify readiness and remind the user of the program to which the command will be routed. Prompts are placed wherever the Object Editor expects the user to provide content. The prompt is printed in a distinguished font and disappears when any attempt is made to type over it. As a result, the prompt serves as a reminder and placeholder, but requires no effort to delete.

The contents of a prompt depends on the item to be entered and the amount of information that the underlying Object Editor has about reasonable values. The simplest form of prompt contains the name of the class of object that

needs to be provided. For Ada, this would likely be a nonterminal in the abstract grammar, e.g. expression. In more semantically defined situations, the prompt might contain a reasonable initial value. The default value of a parameter or the default initialization of for a field in an aggregate are examples of prompts that, left alone, become the values provided. An operation is provided to convert the prompt text to plain text, allowing normal edit operations without losing the entire text of the prompt.

An @I(error) is a section of text marked by the object editor to indicate a problem of some sort. An error is treated as a prompt for editing purposes. Correcting the problem detected will cause the error to go away when the object editor re-formats the presentation of the object.

@SubSection(Mechanisms)
The following mechanisms are provided to support editing operations that are not primarily dependent on the apparent objects on the screen.

@Label(Keymap)
A @I(keymap) is a mechanism for binding a key or key-sequence to a specific action. Every key that the user hits is bound by this mechanism to some command. For example, the most common commands are character insertions that are mapped to the key labelled with the character. By changing the keymap, it would be possible to implement a Dvorak keyboard without modifying the terminal. Keys can be mapped to any statement list, but the most common mappings are to specific procedure invocations. Mechanisms are provided to accelerate functions that are mapped to known editor procedures, elaborated procedures that can be invoked independent of context and statement lists that are repeatedly executed in a context that has remained constant. Regardless of the level of acceleration provided, the semantics of a key are defined by the semantics of the Ada statements it maps to.

A @I(macro) is a sequence of saved editor commands that can be invoked together. Macros are appropriate for recording a set of actions for re-use later. It is expected that complicated operations, including those requiring parameter passing, will be done with Ada programs. Facilities are provided for saving macros with a session and for binding them to keys. Macros act like parameterless procedures with no local declarations and no control structures (except those internal to individual commands in the macro). There will, eventually, be a facility to convert macros into equivalent Ada procedures.

A @I(Yank Buffer) is a piece of an image that has been saved for later use, typically by a deletion operation.

A @I(Command Image) is any Ada fragment that is prepared to be compiled and executed on the user's behalf.
The Core Editor has no information about what
each of these commands does, but saves the image in case
the user wishes to repeat the same or similar operation. Command images can reference any of the builtin commands by their Ada names.

A @I(Stack) is a structure for saving a set of objects based on usage patterns. Stacks are used to store marks, windows, images, selections, yank buffer, and command images. The operations described below make it possible to cycle through the previous instances of each type in an orderly manner. The primitive operations are:
@Begin(SEnumerate)
Push. Add/move an item at the top of the stack.

Next. Examine the next item down the stack.

Previous. Examine the previous item up the stack.

Top. Examine the item at the top of the stack.
@End(SEnumerate)
Next (previous) "wraps" to the top (bottom) when applied to the bottom (top).

@Section(Object Editor)
The object editor provides the transformations between the object and its image. This is done by incremental parsing and pretty-printing operations. Four basic operations are supported for viewing and changing objects.
@Begin(Enumerate)
Display. Create the image of an object.

Format. Parse text changes made to the image into the object and update the image to reflect the changes. This provides an opportunity for incremental syntax checking and correction and pretty-printing.

Commit. The object is in a user-desired state. Take the appropriate actions to reflect this intent. For most object types, this means saving the object. For commands, it causes the command to be executed.

Revert. Bring the image back to the state it had following the last commit. This provides a coarse-grain undo facility.
@End(Enumerate)

The object editor provides selection operations that understand the structure of the object being edited. These operations provide the ability to select objects, their parents (the containing object), next and previous brothers, and children. These selected objects serve as operands to move, copy, delete, elide and expand operations, as well as to type-specific tools outside of the editor.

@SubSection(Pointing)
One of the basic notions of the environment is that objects are interconnected and that it is easier for the user to point at an object of interest and request information than it is to formulate a specific procedural request naming the object and the desired information. Having selected an object of interest (either explicitly or by simple cursor placement), at least the following broad categories of information can be requested:
@Begin(Description)
Definition@\Show the definition of this object. For a reference to an Ada object, this move the cursor to the declaration of the object. From the defining occurrence, it moves the cursor to the definition in the body or private part.

Completion@\Provide information about the possible correct completions for the object of interest. Fill out all or part of a name on the basis of a prefix or pattern. Fill out the remainder of a syntactic structure. Provide prompts and/or values based on the type of the object that will make it possible for the user to complete the object. An example of all of these would be entry of the prefix of a procedure name and having it complete to a procedure call with full named-parameter notation for the call prompt-designated presentations of the defaults and nonterminal prompts for parameters without default values. An advanced form of completion is to provide prompt values that are the results of evaluating default value functions. The result of the function will often mean more to the user than the process for determining it. There is an associated ability to cycle through choices be repeatedly evaluating these functions.

Help@\Explain the object. As distinguished from definition, show a description of the object and its use. For an error, show an explanation of what was wrong, associated rules, etc.

Attributes@\Display attributes of the object that are not part of its image. Instances of this sort of information would be modification date, creator, and installation/elaboration status.
@End(Description)

@SubSection(Operations)
Object Editors provide a number of common operations that depend on the form and content of the objects presented. The basic ability is

to read and format the object and, in many cases, take the modified image and convert it back into its object equivalent. In addition, object editors provide movement/selection operations that depend on the structure of the object. The assumption is that the object can be viewed as a tree-structure in which each object has a parent (the object containing it), siblings (objects with a common direct parent), and children (objects that it contains). For Ada programs, these operations follow the logical nesting structure of the language; for text, the correspondance might be sentences, paragraphs, sections, chapters, etc.

@Section(Ada Editor)
While it possible to conceive of object editors for many types, the first and most important is the one for Ada. Because of its interaction with system structure and semantics, the Ada object editor provides operations and, in some cases, imposes restrictions that have no parallel in other objects.

Editing Ada source objects follows the Core Editor-Object Editor paradigm. Changes are made to the source as text. The Ada Object Editor provides syntactic completion, structural motion (parent, child, sibling), etc. based on its knowledge of Ada. Though the object-specific operations differ, there is no conceptual difference between these Ada source objects and text objects. The meaning and variety of operations differ because of the intrinsic differences in the two types of objects, the fundamental reason for the existence of type-specific object editors.

@SubSection(Insertion Points, Installation and Elaboration)
Elaborated and, to a lesser degree, installed objects are fundamentally different from source objects and the editing operations that are appropriate are correspondingly different. Elaborated packages contain Ada declarations that are referenced by other installed or elaborated unit, affecting both their compilation state and any active threads executing in the corresponding code.

To control the changes and make it clear what was intended, operations are provided to explictly withdraw the elaborated version and install/elaborate its replacement. By limiting the scope of what is withdrawn or replaced, it is possible to restrict the impact of the change to the specific objects that were changed. Object deletion provides an unambiguous way of removing a precisely specified set of objects. @I(Insertion points) provide a similarly explicit way of specifying where new objects are to be created. An insertion point is represented to the user as an ellipsis, that when inspected is represented in a source Ada window. The user can then enter the declaration for the object (using Ada source editing). When the source object is installed, it assumes its position at the insertion point, either as an object itself or as a "separate" reference to the newly created separate object.

@SubSection(Directory View and Attributes)
Traditional directory services provide access to a variety of information to help remind the user of what is contained in the directory, when it was created or changed, how large it is, etc. The principal support for this is a procedure that will print a list of objects in a directory accompanied by the appropriate attributes. The list will appear on the screen as an output window; changing the output has no effect on the underlying directory. This will eventually be supplanted by a read-only object editor that provides the same information along with additional control and display facilities. A limited form of writeable object editor could be provided to allow object deletion, changing names, and changing attributes that are user-changeable.

@Section(Program Execution)
User actions are performed by executing Ada statements. These statements can be executed by creating a command window, entering the desired Ada code, and commiting the command window. This causes the statement to be compiled and executed. It is also possible to bind statements to keys in ways that shortcut the compilation without changing the semantics (@ref(keymap)). The sections below describe the context in which statements are executed, the forms of binding and how they interact with execution, and the runtime

environment provided for statements.

@SubSection(Context)
Ada statements are semanticized and executed from a particular context. For user commands, the environment constructs a context that provides convenient access to user, system and object-type-specific objects and procedures.

The context is a declare block at the end of the body of the elaborated package corresponding the current object window. For a command window attached to an elaborated package, this is simply the end of the package body. All other objects are considered to be rooted where they are declared. Dynamically created objects have the context of their creator, e.g. a Text_IO window for an object with no underlying file has the context that was active when the command that created it was started. The initial (and any other for which no predictable dynamic predecessor exists) context for a session is set to the home package of the user. Subsequent session continuations resume the context saved at shutdown.

The declare block that is generated as the default for a particular command window has the following form:
@Begin(Verbatim)
```
    declare
        [global declarations]
    begin
        declare
            [object editor-specific declarations]
        begin
            [statements]
        end;
    end;
```
@End(Verbatim)
The global declarations are typically use clauses and renames that make system and core-editor commands more accessible; the object editor-specific declarations provide the same facility based on the type of the object being editted. The user can change the declarations for a particular execution simply by editing these declarations. Changing the declarations persists with the particular command window, but does not change the underlying defaults. Any legal Ada declaration is possible, but the declaration is elaborated once for each execution, so it is not possible to retain state from one execution to another, only between the statements of the block. The nested declarations are required to allow object-specific operations to hide global operations. The declarations to be used in each context are defined by declarations in the definitions package of the user's home package. Appropriate default values from the system definitions package are used if no user definitions are provided.

@SubSection(Command Windows)
Each object window has (or can have) a command window associated with it, from which it is possible to type Ada statements to be executed. The term, @i(command) is used to mean the set of statements in a command window. For the case where a single statement has been entered, this coincides with the traditional command paradigm; for more complicated command windows, it can be very different. The full facilities of the Ada Object Editor are available to edit commands. The command image initially contains of the context described above. The cursor is positioned on the statement prompt in recognition of the relative frequency of simple statement entry, but it is possible change the declarative part of the block with normal editing. Commands are executed by commiting the current command window. After the command has terminated, the declarative portion of the command image will remain. The statement portion of the window will be converted to a prompt in preparation for new statements.

The standard arrangement for command windows is to have one under each object window (or set of object windows). When a command window is treated as an object window (i.e. the user enters commands to modify it), a new command window is created. This new command window operates on the command window as an object, not on the original object. Its context is rooted in the same

place as the base object window.

Command windows are automatically placed at the bottom of the object window they deal with and are not generally separable from their object windows. Command windows persist, disappearing only when explicitly requested or when their object window is removed or replaced.

A history of command window entries is kept to allow the user to examine and re-use previous commands. The history is retained as a stack of entries that were actually executed from a command window; commands that are directly bound to keys do not appear in this history. One history is kept for all command windows. Consecutive repetitions of commands are reduced to a single instance. The history stack has a fixed (though possibly very large) depth. Later versions will provide facilities for history commands that deal with the history of a particular command window.

@SubSection(Execution and Concurrency)
When the contents of a command window are executed, the editor buffers input until one of following happen:
@Begin(Enumerate)
The command finishes. This is the sequential command execution case. The buffering provides traditional command type-ahead. While the command is executing in this form, the user is said to be @i(connected) to the command.

The command requests input. Input requests from commands are handled by editing into an input window attached to the executing command. If the user is connected to the command, a request for input causes the cursor to be moved in this input window. At this point, the command is waiting (though associated tasks are certainly not stopped) for input. The user can provide that input or perform any other editing operations. When input is provided and commited, the user is again in the connected state with input buffered.

The user disconnects from the command execution. In many cases, the user will not want to wait for the command to complete before going on to do something else. In these cases, it is possible to disconnect from the command, causing it to run asynchronously. Disconnection can occur either before or after the command has started execution. Prior disconnection is possible by issuing the disconnect in conjuction with commiting the command window. This removes any chance of race conditions between the user and the program as to where the cursor ends up or other state transitions. Note that disconnection doesn't inhibit the program from either writing output to a window or requesting input. Input requests no longer automatically move the cursor into the input window. See @ref(ProgramIO) for more details.

The user cancels the command. This causes execution of the command to be terminated. Buffered input is also lost.
@End(Enumerate)

@SubSection(Binding and Builtin Commands)
Two different methods for causing statements to be executed have been discussed: commands bound to keys and execution of the contents of a command window. With the exception of side-effects on various components of editor state, either method results in the same execution. The effect is that of executing the designated Ada code in the proper environment. This does not imply that all execution uses the most general mechanism. Rather, the acceleration mechanisms provided to make builtin commands execute quickly are the result of careful binding of the fixed command set to externally visible procedure instances.

Ada names are bound to internal commands by means of keymaps (@ref(keymaps)). When a key is indexed, the bound command is executed. The process used to determine what to execute depends on the Ada name, the type of the binding, and the ability of the command object editor to detect equivalence to a previously used command.

Builtin commands are Ada procedures that have a fixed location in the

envrionment. Binding keys to these procedures involves the selection of a fixed functionality, independent of the executing context. As a result, once the correspondence has been established, it is possible to shortcut the key to execution process without even calling the indicated procedure. Obviously, if it were possible to change the bodies of the fixed procedures, without changing the internal operation of the procedures, an inconsistency would arise. Similarly, it is always possible to introduce hiding into any Ada scope such that a "fixed" name references a new procedure. Since there is no chance that this would occur inadvertently, no steps are currently envisioned to protect against such a confusion.

Even if a procedure is not one of those implemented inside the editor, it is possible to bind a key to its execution in a way that is context independent. This simply requires that the binding be to a fully-qualified name in a context that is very unlikely to be hidden. This form of binding is treated just like builtin commands, except that a more sophisticated invocation method is required. It suffers from the same unlikely inconsistencies in the face of concerted attempts hide the intial definition.

By binding keys to simple (or not fully qualified) names, it is possible to provide keys that execute different functions depending on the context in which they are invoked. This form of binding is very likely to require semantic analysis and possibly code generation to be successful. Some acceleration is available by recognition of known names from the semanticized name in context or by recognition of previous use of the same procedure in the same context.

One last form of binding allows keys to be bound to commands with the purpose of prompting for the command (placing it in context in the command window), rather than to execute it. This allows keys that provide the command and prompts for the parameters, saving command entry, but still allowing complete parameter flexibility. This mechanism is invoked whenever the Ada that was bound to the keys was incorrect and/or incomplete, allowing the user to see the problem and correct it.

@SubSection(Jobs)
Ada execution takes place within tasks. A single user command, even one that is apparently sequential, may be implemented by more than one task. Jobs (@ref(jobs)) make it possible to treat the command execution as a single entity, without worrying about the precise implementation.

Jobs provide a basic level of execution control. The tasks of the job can be scheduled together or terminated together. The job serves as an identifiable entity for these purposes, where for individual tasks, there is no guarantee that an Ada name exist for the task throughout its execution. The environment also uses jobs as the basis for determining the current user focus: the user is either waiting for the completion of a job (command) or not. The execution priority of the command and the course of the user's interaction with it can be different in the two cases.

Each job has associated state corresponding to traditional program or process state. Although some of this state is system control information, salient pieces are of interest to the user. A good example of this is the standard input and output files defined by Text_IO. The location and status of the windows allocated to these files is part of job state.

A series of user commands executed serially will share (serially) the job and its state. Continuing with the Text_IO example, a series of commands executed will share input and output windows, creating a single script of the various command executions.

Disconnecting from a job creates a new job with its own state. In the Text_IO window example, two asynchronous jobs would update different windows. Once a command is started, its job number doesn't change (though it may create other subordinate jobs). As a result, a job that is started, then disconnected will act upon the inherited state and any new commands that are entered start over

from scratch (in the Text_IO example, the disconnected job uses existing windows and new commands get new windows). Disconnecting a job before it is started causes the newly started job to create its own state and leaves the user attached to the same job as before the command was started.

@Section(Naming Objects)
@Label(Names)
Naming objects in the package directory system follows the naming and visibility structure defined in Ada. The following factors are involved in extending these simple names:
@Begin(Enumerate)
Versions and configurations. Because there are multiple versions of most objects, Ada naming is interpreted within the context of the current configuration. Procedures and functions that provide access to specific versions will do so by explicit version parameters.

Ada program objects. Ada (quite reasonably) provides no way for programs to name their source components. To provide self-reference, attributes have been provided for each program object that allow designation of the principal parts of the object. Names consisting of an Ada name attributed to indicate the part are called @I(source names).

Nascent objects. If the object doesn't exist yet, it can't be named. Strings are used to provide the name the object will take on. Strings are also used as in traditional systems to provide deferred naming within programs.

Convenient aggregation. Users often perform operations on groups of objects whose names are textually related. This is done by providing wildcard characters to be used in conjuction with string names.
@End(Enumerate)

@SubSection(Source Names)
Conceptually, each program object has a visible part and a body. For a object, Ada_Name, the visible part is Ada_Name'Spec and the body is Ada_Name'Body. If the object has only a visible part or only a body, the attributes are interchangeable. For a type completed in the private part, 'Body refers to the completion of the type and 'Spec refers to the incomplete declaration.

Overloading makes procedure and function names without parameter specifications insufficient. For each overloading of a name, a nickname is provided. The nickname is used to index the 'Spec and 'Body attributes. The system assigns numeric values as the nicknames on the basis of their order of occurrence in the visible part and body. The same nickname value is assigned for both the body and visible part of a single object. A facility will be added to allow users to designate nicknames using pragma Nickname (Mumble).

User nicknames define an enumeration type, package_name'Nicknames. As such, the nicknames for the subprograms of a package are unique for the package, not just for the subprogram name that is overloaded.

@SubSection(Strings as Names)
Strings can be resolved as Ada names. Whatever could be provided as an Ada or source name can be placed in a string and resolved as a name. Most procedures and functions for direct user use will provide string names in addition to direct object references. The spirit of the environment is for names to be Ada names, so though it is not possible to keep string names from being interpreted differently, there is considerable advantage to uniformity.

Strings are used to provide the name for new objects. All names are Ada names, so the string must contain a name that will be legal after its introduction into context.

The restrictions on naming Ada objects are not as severe for strings as for direct Ada names. Specifically, program objects (functions, procedures, packages, etc.) need not have 'Spec or 'Body, except as necessary to choose between the two parts. For data objects, 'Spec is required to get the Ada

declaration instead of the object itself. There is no implied evaluation of functions, so the string containing the function name refers to the Ada object rather than some value it might return. In general, the unattributed name refers to both the visible part and the body if both exist.

A string referring to a set of objects (either because of overloading or 'spec/'body ambiguity) refers to all of the objects. If the context requires a unique object reference, a specific, non-surprising interpretation of the multiple name should be chosen.

@SubSection(Context)
As describe in the section on command windows, name resolution depends on the context of the current object. The assumption in that section was that current object is part of the package directory system. This assumption works well until the object under study is the execution instance of a running program. Assume that the user is stopped at an invocation of function F, initially executed from package P. Different contexts are of possible interest:
@Begin(Enumerate)
The initial package context. Objects in this context are available from a command window on the same object as the one from which execution was initiated.

The local execution context. This the context of the command window attached to the point at which execution stopped in F. It corresponds to a debugger context in F. The context is established at a specific point in F and derives its visibility from that instance of F. This will include values local to this instance of F and package-level objects visible from F.

Arbitrary package context. If P is a library package (or part of one), there are objects of possible interest in other packages. If the other packages are in the with list for P, direct reference is available from the local execution context. Otherwise, a different context must be established to access the objects.

Arbitrary active execution context. This is the same as the local execution context above, except for a different stack frame than the one at which execution is halted. Examples would include previous invocations of F (or other units) in the dynamic call history or invocations in a different task thread.
@End(Enumerate)

For each object that currently exists in the program or its environment, there is at least one context from which it is possible to name the object using Ada (source) naming, allowing for the introduction of names for anonymous blocks, etc. There are not always Ada names for the contexts themselves.

A name is used to designate a particular object from a specific context. It carries with it all of the Ada visibility restrictions. A context is a place from which a name can be resolved. As such, the context name is not limited by visibility restrictions. For the following arrangment:
@Begin(Verbatim)
```
    package body X is
        package Y is ... end Y;
        package body Y is package Z is ... end Z; end Y;
    end X;
```
@End(Verbatim)

From outside of X, it is not possible to name objects in X.Y.Z. However, the context X.Y.Z always exists.

The current debugger recognizes a number of different context specifications; some given in specific context instructions, others as part of name specified:
@Begin(Enumerate)
Package directory root. This is specified by "." and indicates that the context provided is an Ada name whose first component is a library unit.

Using full Ada naming, the "." is not strictly necessary, though it can be convenient in bypassing declarations that hide library packages in a local context.

Stack name. A task number or user-specified task name that specifies the execution stack of interest. Most task names or numbers have equivalent to some other (probably longer) form of the name. Ada does allow creation of tasks whose names are lost, however.

Frame within a stack. Specified as @@n, this refers to the subprogram activation at relative position n in the stack of a particular task.
@End(Enumerate)

Unfortunately, this categorization makes context specification seem simpler than the reality of the Ada runtime environment. Specifically, contexts and names must be assembled in arbitrary sequences in order to reach all contexts that can be created.

@SubSection(Context, Creation and Deletion)
Ada makes names more available than typical directory systems, partially because names are only referenced, not created or deleted. All Ada references are also from fixed lexical positions in the program.

Consider the following characteristics of names:
@Begin(SEnumerate)
Simple, qualified.

Local, contextual, absolute.
@End(SEnumerate)

Notes on all of the combinations of these name characteristics.

@Begin(Enumerate)
Simple local names are those available in a closed scope.

There is only one simple absolute name, Universe; all absolute qualified names start with Universe. There is a standard abbreviation, U, introduced by the declaration "package U renames Universe" immediately inside Universe.

Qualified local names start with simple local names.

Contextual names denote objects that are available through Ada visibility that are neither local nor absolute. Examples include names in containing scopes or in library units that are referenced in use clauses.

Object deletion and creation are unsurprising for local or absolute names.

Creation or deletion of objects referenced by contextual introduces the problem of "capturing" unintended objects. This is solved by expanding the name and allowing the user to proceed if that is the intent. Create and Delete will have parameters controlling how automatic capture should be.
@End(Enumerate)

@SubSection(Advanced Topics)
@Comment{%%%%
Reduce this to something closer to a real suggestion.
}
A potential problem is the number of different types of names the user can specify, requiring common operations to be heavily overloaded and potentially leading to user-access inconsistency because not all functions are overloaded on all of the common methods. Note that the concern here is on the form of the name the user enters, not the resolution of the name. The types of naming the user has access to:
@Begin(Enumerate)
Direct object. The name directly specifies a particular object. This is the case where the most Ada type context is available to aid in resolution.

Vector aggregate. The Ada answer to procedures that want multiple objects as arguments. Potential inconveniences are introduced by the difficulties inherent in resolving (A, B, C) to be Array_of_File_of_Integer'(A, B, C) as required by the program. @Value(Add)

Strings.

Wildcard. The user provides a string with wildcard characters that resolves to a selection or a vector. Except for user convenience, string names could be replaced by a function that takes a string and returns a vector of objects. Note that it is not feasible to have an array aggregate whose elements are each strings.

Pattern. The user provides a syntax/semantic pattern for an Ada object with terminals, non-terminals and wildcards. The is the object editor extension of regular expression matching and is very powerful for editing Ada programs.
@Value(Major)
@End(Enumerate)

Immediacy of interpretation is an issue. Simple completion is done by resolving a wildcard to a single name with user interaction to iterate over multiple possibilities. Wildcards will also resolve to vector aggregates in appropriate circumstances @Value(Add). This corresponds to expanding the wildcard for the user prior to executing the command, rather than during command execution.

@Section(Keys and Command Factoring)
The builtin command set of the editor has been factored into a sets of operations and sets of objects. Each group of operations can be applied to a group of types by using the key that specifies the object type followed by the key for the operation. Default object types have been chosen to reduce the frequency of two-key sequences, and since the factoring doesn't occupy all possible keys (especially for terminals with function keys, etc.), it is possible to place commonly used sequences on single keys.

@SubSection(Types)
The set of object types is:
@Begin(SEnumerate)
Character cursor. Character insertion, image position and marks.

Command. Command window and history.

Designation. Elisions, error and prompts.

Macro.

Line.

Screen cursor. Motion on the screen.

Selection. Both object and text selection.

Window.

Yank buffer.
@End(SEnumerate)

@SubSection(Operations)
The following is a brief description of each of the classes of operations and the types that each applies to.
@Begin(Enumerate)
Planar movement (up, down, left, right)
@Begin(SEnumerate)
Cursor. Move user cursor on the image

Screen cursor. Move user cursor on screen.

Selection. Select parent, child or brothers.

Window. Scroll the window over the image.
@End(SEnumerate)

Relative positioning (next, previous, beginning_of, end_of)
@Begin(SEnumerate)
Designation.

Line.

Word.
@End(SEnumerate)

Modification. (copy, delete, insert, move, transpose; capitalize, lower-case, upper-case)
@Begin(SEnumerate)
Character.

Line.

Selection.

Word.
@End(SEnumerate)

Stack. (next, previous, push, top)
@Begin(SEnumerate)
Command. Manipulate history. Push is implied by execution.

Mark.

Selection.

Yank buffer.
@End(SEnumerate)

@End(Enumerate)

More detail, including an initial key assignment for QWERTY-only keyboards is available in [BLS.CE.DOC]R1000_Commands.MSS.

11:00:16, Apr 18, 1984

TYPES.MSS

EGB

@Section(System and Managed Types)

11:00:11, Apr 18, 1984

TODO.MSS

EGB

1. Notes on packages as directories. We want to be able to do the stand sorts of directory operations. {Time, User}X{Create,Use,Update}. Sorted by appropriate fields. Patterns for selection. These can be programs. Do we expect there to be a directory object editor?

2. Installed/Elaborated. What are the operations and how do you really do things. Basically just a description of what we are doing now. Need to look at it to see what it is.

4. Elision. There are multiple levels. Specific interpretation controlled by the object editor. Elision is on an object basis. Initial object display can set to elision level. There will probably be other attributes that control the display (e.g. comments, directory attributes). Would be useful to have elision level of subobjects saved at a new elision; that is, eliding then uneliding a block restores it to the state just before the most recent elision, not to the completely unelided state.

   Documentation/help system will eventually rely on elision to control level of detail and in presenting this information. Limited window size makes one level of elision easy, i.e. anything off the bottom of the window is easy to ignore. Since a crucial part of the documentation will be the comments in the code, is there a way to logically group comments for elision?

5. Job state inheritance, describing jobs and disconnection.

6. Messages to the user. How do you generate messages from below the editor level. A specific example would be completion notices from the compilation manager.

7. Windows. Need to describe the window model. Also need to describe a basic window functionality for implementing it.
   Frames vs. Pop-Up. There are defined frames into which window requests are placed by default. Pop-Up windows are placed relative to the source of the window and other information.
   Current frame vs. LRU frame. Some operations (users) will want to have the next window placed in the current frame or some distant frame. This is probably controlled by a default parameter that can be controlled when the window is desired.
   Frame sizing vs. window sizing. I think that we want to be able to set the current frame size independently of window size, but this is not high priority. There need to be commands that set up the frame configuration and re-map the windows on the screen into the designated frames.
   Splitting and coalescing windows. These are primarily frame operations.
   Focus. Making the current window the one on top, growing it, etc. A high-level version of frame setting and coalescing.
   Relations between windows. Some model of the interaction of windows that facilitates placement decisions. This is an extension of the command-object and input-output pairings.
   Banner state. There will need to be better banner utilizattion to indicate the status of the object.
   Screens. Mechanisms for associating particular contents with particular window configurations for specific functions.

8. Completion and names. There isn't completion for names in strings, but there isn't much support for naming outside of strings. Ideally would like completion on semantic content, i.e. ada.edit (x) ==> ada.edit (x'v). This does raise the names-in-context problem and how to get things out of bodies that aren't visible.

   If we are going to use strings, is it possible to do anything to make it easier for the user to type strings without having to provide quotes.

   One of the things needed for a "normal" command interface is the ability to provide multiple parameters and switches. The most promising Ada

mechanism for this is aggregates. Unfortunately, aggregates are hard to construct without qualification; to be useful, there would need to be completion provided in the form of the aggregate type-specifier.

Switches can be handled by aggregates or name parameters. Would ideally like to have default parameters as prompts and have the user use next prompt to get to them. Would also be good if typing the parameter specifically would override (and cause to disappear) the system-provided default.

Minimizing input syntax (e.g. edit x => Edit_Object (X'V); or Edit ("x"); would also make command entry easier. Expansion of wildcards into aggregates would be useful for multi-object parameter cases.

9. A common concern of users of the machine will be the status of the machine, including load, paging rate, who is logged in, what is the status of tasks running in the background, etc. What displays are going to make sense for the user and where is some of this information going to be available. Need to talk to NCE to find out what MTS will make available.

10. User context. Is there a stack of places that the user has been to recently. Using definition (or whatever) that easy movement to a context; is it easy to return. Similarly, is it possible to remember a place where you have been? This could be a "mark" notion in the CE; would require a facility for naming and marking them. Ideally would be useful to be able to store them "permanently".

11. Keys need to map to more general constructs than an enumeration. How would we see building these more interesting maps. One choice would be to use Ada source fragments or unsemanticized Diana trees. This complicates specification syntax, but otherwise seems explicable and implementable; will need some explanation of binding time. Among the things needed to make this complete is a distinction between binding a key to execute a command and to prompt for it. A major use of stored Diana trees would be as full command prompts for user-supplied procedures that require parameters be filled in.

12. Describe the text I/O paradigm. What happens to input, how to control output. Will need specifics on "standard" output, specific window output, multiple output windows, output that doesn't go through windows. This probably gets into handling the form parameter for text_io. What about elision and ^O of output. Form parameter issues: tape, foreign disk, specific terminals, on window or not, pipes. What is the syntax?

13. Concern over the state of objects within directory packages (and other places), how does the user tell elaborated from installed, etc.

15. State (as described in sessions) doesn't distinguish system/editor or permanent/temporary state issues

16. File types and formats. There need to be developed mechanisms for indicating what is in a file. This seems to require a combination of giving object types to package_directory system and some format indication on the beginning of the file; a sort of file label that is appropriately interpreted. Need to have Text_IO and Sequential_IO define types for the objects that they create. Need to have a low-level I/O routine that can read text or characters interchangeably.

17. "None" access for objects that are being written would allow a form of readonly access to objects in creation. Would require some process (possibly revert) for getting back the most current contents.

11:00:06, Apr 18, 1984

PROCESS.MSS

EGB

@Section(Users, Groups and Sessions)
Users, Groups and Sessions are very simple representations for notions of interest to users of the environment. They all serve to identify tasks and objects on the basis of who created them.

@SubSection(Users)
A user is an object that represents the human user in the system. Its primary purpose is to provide a domain for system access authentication and object access control.

The environment associates information with the user that makes it possible for him to tailor the user interface and resume sessions in a desired state. Specifically, each user is associated with:
@Begin(SEnumerate)
The set of objects and packages that he has created. This includes sessions, files, programs, etc. The user is said to "own" such objects and packages.

A home package in the package directory system. This is the current context for any new sessions (see below) that the user establishes.

A default context clause in which to interpret commands, including specific use and rename clauses to select objects and programs frequently used.
@End(SEnumerate)

@SubSection(Groups)
A group is a set of users. A user may belong to any number of different groups. Groups are used to provide aggregate access control, i.e. access can be granted to a group instead of to each of the individual members of the group.

@SubSection(Session)
Session is a term that is used broadly to represent the contents of an interaction between a user and the environment. While active, the session acts for the user, providing the tasks necessary for user execution. Each session has a unique name, its session_id, that is attached to the base of each of the stacks of all of the tasks making up the session. This common identification is used to provide dyanamic inheritance of state between the tasks of the session.

Sessions provide continuity from one period of interaction to another. When a session is inactivated, the environment saves characteristic information associated with the session_id. When the user resumes the session, this retained information provides continuity with the state prior to suspension.

@SubSection(Login)
Login is how the user acquires a session. For all of the traditional reasons, login validates the user's right to use the system by requesting a password. After validation, the user must establish which session is to be used (either by creating a new one or resuming an old one) and the type of terminal that is being used. Either or both of these could be chosen by apprpriate default.

@Section(Session, Jobs and Tasks)
@Label (Jobs)
Sessions, jobs and tasks form a logical containment hierarchy. Every task is part of a job; every job is part of a session. Tasks are defined by Ada; sessions are defined above.

A job is a logical thread of control as seen by a user. Although the job can contain an arbitrary number of tasks, it represents an autonomous entity started by the user to accomplish a purpose. Jobs form a subdivision of the session name space. This division makes it possible for different logical threads of control to have a common dynamic inheritance that is different from that provided for other jobs in the same session. Information associated with job_id (current source/destination of input/output, storage heap, file naming context, selection, etc.) is more execution-specific than that associated with session_id, but there is no hard distinction.

11:00:02, Apr 18, 1984

OVERVIEW.MSS

EGB

```
@Make(Article)
@Modify (Verbatim, above 1, below 1)
@Modify (Senumerate, above 0)
@Modify (Description, LeftMargin +8, Indent -8)
@String(DocumentTitle=<"..." Overview        >)
@String(Draft=<DRAFT 1>)
@Set(Page=1)
@Include(ada.mss)
@Include(process.mss)
@Include(interface.mss)
@Include(types.mss)
```

11:00:00, Apr 18, 1984

ISSUES.MSS

EGB

Items suitable for group discussion:

1. Jobs, asynchronous execution and commands.
2. Completion, string names, dynamic defaults.
3. High-level directory operations; naming, wildcards, move, copy.
5. Input/output. How these windows are handled.
6. Windows. What they look like, how they are handled. Attributes.
7. Actions; how are they visible to the user?
8. Messages to the user.
9. IO to pseudo-devices, tape, 'disk', pipes.
10. Text_IO; windows, forms.

10:59:56, Apr 18, 1984

IO.MSS

EGB

```
@Part(IO, root "[mtd.env]spec")
@Chapter(Input/Output)
@Label(ProgramIO)
@Section(Sequential_IO, Direct_IO, Text_IO)
@Section(Window I/O)
@Section(Devices)
@Section(Inter-task I/O)

How to do Text_IO-like input and output.
@Begin(Enumerate)
User executes a procedure that does I/O.

Editor focus remains in the command window from which execution starts.

Entry can take place in the command window, but execution from this command
window is interlocked until the procedure completes or is aborted by the
user.

Program requests input or output, causing the appearance of the input and/or
output panes associated with the command window.

The user can also request the input pane in order to type ahead of the
program request for input.

The input window is simply a window of the appropriate type (initially
text, but easily extended to typed data objects in aggregate notation) into
which the user edits values.

More than one input or output window is possible, though this will require
user open and specification of file name.

When a quantum of input is completed, the user formats (enters) the window.
This allows multiple-line inputs at a time, though most users will probably
prefer to have the return key enter the line.

Since the normal state of program input is blocked, waiting for further input
from the user, there is an end-of-file command to signal the user's intent not
to provide additional input.

As characters are read by the program, they are copied to the output window
associated with the input.

It is possible to specify flow control on the output window.  This is done
by stopping output to the window or forbidding the window to be scrolled.

It is possible to remove the input or output windows from the screen
without stopping the execution of the program until all available input is
exhausted.

There is control of the copying of input as read.  The
simplest form is whether or not to echo.  @Value(Add)

It is possible to specify the source of the default input from the command
window invoking the procedure. This allows the association of files or
existing windows with "standard" inputs. @Value(Add)
@End(Enumerate)
```

10:59:52, Apr 18, 1984

INTERFACE.MSS

EGB

```
@Section(Editor-Based User Interface)
@Begin(Itemize)
Editor based.

Object oriented.

Type knowledge.

Simple, yet complex.

Mechanisms
@Begin(SItemize)
Windows.

Completion.

Prompts.

Elision.
@End(SItemize)

Objects.
@Begin(SItemize)
Characters.

Words.

Objects.

Selections.

Cursors.
@End(SItemize)

@End(Itemize)
```

10:59:44, Apr 18, 1984

NAMES.TXT

EGB

1.4.6. Naming Objects

Naming objects in the package directory system follows the naming and visibility structure defined in Ada. The following factors are involved in extending these simple names:

1. Versions and configurations. Because there are multiple versions of most objects, Ada naming is interpreted within the context of the current configuration. Procedures and functions that provide access to specific versions will do so by explicit version parameters.

2. Ada program objects. Ada (quite reasonably) provides no way for programs to name their source components. To provide self-reference, attributes have been provided for each program object that allow designation of the principal parts of the object. Names consisting of an Ada name attributed to indicate the part are called source names.

3. Nascent objects. If the object doesn't exist yet, it can't be named. Strings are used to provide the name the object will take on. Strings are also used as in traditional systems to provide deferred naming within programs.

4. Convenient aggregation. Users often perform operations on groups of objects whose names are textually related. This is done by providing wildcard characters to be used in conjuction with string names.

1.4.7. Source Names

Conceptually, each program object has a visible part and a body. For a object, Ada_Name, the visible part is Ada_Name'Spec and the body is Ada_Name'Body. If the object has only a visible part or only a body, the attributes are interchangeable. For a type completed in the private part, 'Body refers to the completion of the type and 'Spec refers to the incomplete declaration.

Overloading makes procedure and function names without parameter specifications insufficient. For each overloading of a name, a nickname is provided. The nickname is used to index the 'Spec and 'Body attributes. The system assigns numeric values as the nicknames on the basis of their order of occurrence in the visible part and body. The same nickname value is assigned for both the body and visible part of a single object. A facility will be added to allow users to designate nicknames using pragma Nickname (Mumble).

User nicknames define an enumeration type, package_name'Nicknames. As such, the nicknames for the subprograms of a package are unique for the package, not just for the subprogram name that is overloaded.

1.4.8. Strings as Names

Strings can be resolved as Ada names. Whatever could be provided as an Ada or source name can be placed in a string and resolved as a name. Most procedures and functions for direct user use will provide string names in addition to direct object references. The spirit of the environment is for names to be Ada names, so though it is not possible to keep string names from being interpreted differently, there is considerable advantage to uniformity.

Strings are used to provide the name for new objects. All names are Ada names, so the string must contain a name that will be legal after its introduction into context.

The restrictions on naming Ada objects are not as severe for strings as for direct Ada names. Specifically, program objects need not have 'Spec or 'Body unless the operation requires it; the unattributed name refers to both the visible part and the body if both exist.

A string referring to an overloaded set of objects refers to all of the objects. If the context requires a unique object reference, this is an error.

1.4.9. Context

As describe in the section on command windows, name resolution depends on the context of the current object. The assumption in that section was that current object is part of the package directory system. This assumption works well until the object under study is the execution instance of a running program. Assume that the user is stopped at an invocation of function F, initially executed from package P. Different contexts are of possible interest:

1. The initial package context. Objects in this context are available from a command window on the same object as the one from which execution was initiated.

2. The local execution context. This is the context of the current command window. References to objects or procedures local to the current invocation of F are available by the same names as they would be within F, though other procedures or functions in the same package may be available even though their declarations are not visible to F.

3. The global execution context. If F is part of the execution of program created from a library, these are references to objects in other packages in the closure of the packages necessary for the program to execute. For units that are with'ed in the context of F, direct references are available.

4. Dynamic execution context. This is the same as the local execution context above, except it refers to a context other than the current one. Examples would be previous invocations of F (or other units) in the dynamic call history or invocations in a different task thread.

All but the last two of these contexts are available without extension of Ada naming. All of them are available by establishing context at the appropriate context. To do this, there must be a naming convention for contexts. The current debugger uses a preceding "." to indicate global names to be accessed independent of with list and "@n" to indicate relative stack frames. There are also explicit names for execution contexts, e.g. specific tasks instances. The current debugger notation for these is Ada-like. To carry them forward into the environment, it might actually be better if the notation were explicitly non-Ada.

10:59:41, Apr 18, 1984

INTRO.TXT

EGB

The RPE supports design, coding, debugging, and maintenance of medium to large Ada programs while providing the convenience of incremental development of the pieces. In addition to unique native development facilities, host/target tools support simultaneous support for multiple hosts.

The environment is constructed to provide fully interactive development within a universe structured by Ada semantics. This approach provides the advantages of single-user, residential environments for research languages to large-project, real-world development.

Ada-based Semantic Framework

A consistent Ada semantic framework has been used to structure the environment. Permanent objects are stored in an Ada package framework that provides the advantages of traditional tree-structured directories, without introducing the traditional distinction between filesystem and user program. Similarly, all commands are Ada statements with full access to Ada structure and typing, increasing both the power and uniformity of command interaction.

Specific advantages of the Ada framework include:

- Single semantic framework. Programming, user commands, directories and user data are all explicable in Ada terms. There is no duality between the language and the system that supports it.

- Single set of operations. User commands and program procedures are provided uniformly; commands are available to programs and program statements can be tested as commands.

- Extensibility. User programs and system programs share the Ada procedure paradigm and can be called interchangeably. User programs become operationally indistinguishable from basic system functions.

- Semantic richness. Access to full Ada typing and procedural abstraction allows for powerful operations and useful interaction for interesting objects, not just text files.

The primary disadvantage of using Ada consistently for execution and structure derives directly from its strengths: Ada is both syntactically and semantically complex. Realizing the advantages of this power without paying the price with each interaction is accomplished by means of an editor-based, type-knowledgable user interface.

Fully Interactive Interface

All user interaction with the system is managed by a full-screen editor. Within this context, the user edits programs, data objects and commands, responds to program input requests, views program output and manages the process of converting input into executing Ada. Simply managing all interaction from within an editor provides facility and convenience unavailable in simple script-model interactions.

Certainly the most important characteristic of this particular editor interface is that the editor is type-specific. Editing Ada programs and editing text share many purely textual operations, but the do not have the same structure, syntax or semantics. From the type of each object being edited, the environment editor is able to perform type-specific operations. For Ada, these include syntactic and semantic completion of program fragments, early error detection, and operations that deal with the structure of Ada programs.

Full interaction implies a shortening of the edit-compile-debug cycle. Even the fanciest text-editor applied to programs only provides interactive entry; compilation is still a "batch" operation. By integrating Ada knowledge with the editing process, it is possible to perform many of the compilation steps interactively. The specific example of interactive compilation and execution of Ada statements as commands is an example of the process brought full cycle in a very short interaction. Similar facility is provided for manipulating pieces of larger programs.

11:01:18, Apr 18, 1984

SYSTEM.MSS

EGB

@Part(ManagedTypes, root "spec")
@Chapter(System and Managed Types)

There are a number of types that are fundamental to the design, implementation and use of the system. Some of these are primitive Ada Types that are used to build the basic environment mechanisms. Others are Managed Types, built upon the facilities of the object management system. A subset of the managed types are available above the directory layer as Directory Types, which build upon the facilities of the compilation and directory layers.
and (for managed types visible to the user) the directory system. Directory

This section describes the primitive system types, the basic object management facilities and concepts, certain key managed types, the basic facilities of the directory system, and finally some of the

Managed Objects and Object Ids
Actions (action paradigm).
Object Managers
    storage, permanence, synch & queueing, access control, dynamic naming.
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