Rational Machines Incorporated

Programming Environment
Reference Manual

DRAFT 21

Table of Contents

Chapter 1
Introduction and Goals

1.1. Primary Goals and Requirements

The R1000 programming environment is guided by the recognition of the importance of program reliability and maintenance; central goals are:

- to reduce software life cycle costs,
- to improve software/system reliability,
- to improve productivity,
- make RMI a very successful company.

Important related concerns are:

- concern for programming as a human activity,
- user friendliness of the system interface,
- tight integration of functions,
- efficiency,
- portability of developed programs to other systems.

The environment design must take into account:

* the R-1000 architecture,

* our reliance on Ada as the definer of our market niche,

* the needs and desires of the user community,

* extensibility, both for adding tools developed at RMI and integrating tools developed elsewhere,

* easing the transition from one phase of software development to another (design, development, maintenance),

* the fact that the environment may be advanced to a point where the user doesn't understand or feel familiar with its operation; the user interface and design must be sensitive to the user's perspective,

* the fact that the environment will be capable of leading the user to learn more about Ada through use of its facilities; everything (almost) learned for the sake of the system is consistent with knowledge needed for Ada,

* helping users (particularly novices) access its component tools in an effective and productive manner,

* the needs of all technical people involved in the software design and development over the life cycle,

* the needs of administrative people that may use the system in a support role,

* the needs of managers and provide tools that will benefit managers.

The user interface must:

* appeal to the management, sophisticated users, and "novice" users. Many of the customer's software people will have little Ada experience, as such they will be apprehensive about Ada.

* be user friendly but enable the sophisticated user to access the full power available with the R-1000.

* help the user develop more sophistication.

1.2. System Design Theme

The underlying themes for the environment design are:

* using Ada concepts and semantics wherever possible. This provides the following benefits:

    1. The system can be explained in terms of Ada semantics; only very few additional concepts are required.
    2. There is a unified notation for system operations whether they occur in programs or user commands and procedures.

* using a visual/editing approach to the user interface. This provides the following benefits:

    1. Editing makes the system easier to use: the basic commands required to manipulate text can be applied to other objects.
    2. It takes less mental effort to point than to describe a

location.
3. A knowledgeable editor can assist the user in entry of all kinds of information, including editor commands.

* providing support for group development of large software projects, including version control, configuration control, documentation management, change tracking, etc.

Chapter 2
Concepts

2.1. Ada Basis of the Environment

1. The system's organization, syntax, and semantics are based on Ada.

2. Review of Ada concepts:

    1. Packages
    2. Data Objects
    3. Types
    4. Declarations
    5. Expressions
    6. Statements
    7. Subprograms
    8. Visibility Rules
    9. Tasks
    10. Exceptions
    11. Program Structure
    12. Attributes

2.2. Objects, Types, and Operations

1. An "Ada Object" (as defined in the Ada Manual) is a variable or a constant which can denote any kind of data element. This includes scalar values, composite values, values in an access type, and values of private types.

2. An "object" (as opposed to an "Ada object") is any one of the following:

    1. A scalar value (comprising enumeration and numeric value)
    2. A composite value (record, variant record, or array)
    3. An access value (which references another object)
    4. A private value (whose properties are defined in some package)
    5. A program unit (either a subprogram, a package, or a task)
    6. An instance of a program unit (a generic instance or

instance of a task type)
7. An entry
8. An exception
9. A type

3. A "declared entity" is an object that is declared in an Ada program unit. Such objects have a lexical name, as opposed to dynamically allocated objects which do not.

4. Each object has a type. (The type of a type is "type".)

5. The type determines what can be done with the object.

6. An "operation" is a procedure, function, or entry. Operations are defined to take and operate on objects of specific types.

7. Implications:

- Users can define new operations by defining new procedures, functions or entries.
- Operations can only be applied to type-compatible objects.
- Overloading is used to create semantically similar operations defined on several (similar) types.

2.3. Storing Objects

1. Objects are grouped into Ada packages, based on normal Ada semantics.

2.3.1. Environment Structure

1. The environment is structured as a single (large) Ada package. System objects and user objects are placed at various points withing this structure.

2. Used in this way, Ada packages function as directories would in conventional systems.

3. The system manager can create packages for users, thereby establishing the structure for the system. Users can create packages within their respective packages, thus grouping their own objects.

4. When users write program units, the program units are usually placed in one of the user's packages. The program unit is structured into packages using the same Ada structuring facilities used by the user in grouping his objects.

5. A hierarchy is formed by containment of packages within other packages.

6. A "static package" is one whose declaration is in the system root package, or in another static package. A package declared inside a subprogram, or in a package contained in a subprogram, is not a static package.

2.3.2. Package and Object State

1. A package consists of "source code" and "data objects". The

source code is the statements and declarations that comprise

the package. The data objects are the actual values that the data declarations denote. The data objects exist only if its declaration is elaborated.

2. A static package, once installed, is generally not withdrawn except on explicit user request (such as when the package is deleted).

3. Some tasks are designated as "permanent". This means that they survive system interruptions. All static packages are permanent. Permanence applies to the elaborated instance of a static package. The source code of a package is permanent regardless of where it is declared.

4. Dynamically allocated objects follow the normal rules of Ada. They are explicitly allocated and are accessible via access objects. If no access object references a dynamically allocated object, then it may be garbage collected.

5. Declared entities not in static packages are created when the package in which they are declared is elaborated as defined by normal Ada rules.

6. Objects may be introduced into or removed from static packages after the package is initially elaborated.

7. Thus, static packages may contain declarations some of which are elaborated, and some of which exist only in source form. Only elaborated declarations may be referenced from other program units.

2.3.3. Object Attributes

1. An "attribute" is a named object that is associated with an object that is a declared entity.

2. Attributes describe some form of supplementary information about the object they are associated with. This information would include, for example, access control information, usage information, etc.

3. Attributes are used to imbed information about special object properties and relationships. The mechanism is quite general and allows construction of a variety of facilities.

2.4. Users and Sessions

1. A "user" is an identity that can access the system.

2. A user has a "home package" associated with him or her.

3. A user has certain resource limits.

4. A "session" is a locus of control that can perform operations in the system. A users logs on and either creates a new session, or connects to an existing session. The session has the identity and capabilities of the users that are connected to it.

5. Additional user identities may be added to and removed from a session.

6. Sessions are suspended or terminated when the user logs off.

2.5. Editing

1. The primary interface to the system is through an editor.

2. The editor manages the screen, breaking it into multiple windows.

3. Each window can display independent information.

4. Objects in the system can be displayed and edited with the editor.

5. The state of the system can also be manipulated with the editor.

6. The editor is described more fully in a later chapter.

2.6. Context and Operation Invocation

1. When a user is interacting with the system at a terminal, we say that the user's session executes operations and manipulates the environment on the user's behalf. Some task running in the session is the actual agent that executes the code to carry out user requests.

2. In addition to manipulating objects with the editor, operations which perform manipulations can be explicitly invoked.

3. A "context" specifies a location within the package structure of the environment. A context specification consists of several parts: the source context, the instance selection, and the version selection.

4. The "source context" is the name of the package, task, or subprogram.

5. The "instance selection" specifies the particular activation of each subprogram component of the source context.

6. The "version selection" specifies the particular version of each component of the source context.

7. Operations are invoked from a specific context. The environment maintains a "current context" from which operations are normally invoked, so that the user need not specify the

context with each invocation.

8. Operations must be visible from the context in which they are invoked, as required by normal Ada visibility rules.

2.7. Active Entities

1. The agents that execute commands (and run edit sessions) are Ada tasks.

2. As such, they can be suspended and aborted.

3. Concurrent execution is achieved by creating multiple tasks and having them each execute commands.

4. There is generally (at least) one task per editor window. This makes each window able to perform concurrently.

Chapter 3
The Environment Model

This chapter describes the structure of the environment. It describes how objects are stored, the type of information associated with objects, and some about how objects are manipulated.

3.1. Packages

1. Users place objects that are long-lived (relative to program invocations) in static packages. Objects in these packages, and the packages themselves, are not deleted except upon

specific user command.

2. A "static object" is a program unit or is an object that is declared in a static package.

3. The user can store any Ada object. Typically, the objects stored will be program units and abstractions for documents or other types of information. The user can also store integers or any other Ada type.

4. User objects are entered into and removed from static packages in any order that the user chooses.

5. Because the user can install and remove objects in any desired order, it is possible to change a set of packages so that they are no longer consistent and would not compile correctly if a recompilation were performed. When this is about to happen, the user warned, and, if the user still wishes to make the change, an old version of the package and/or object will be retained so that existing packages will still be consistent with the old version. The full set of options available during object installation is described in section 6.2.1.

3.2. Object Life Cycle

1. Objects not declared in static packages follow normal Ada rules from their creation at elaboration time to their deallocation.

2. The ability to introduce new objects into static packages complicates the life cycle of such objects.

3. Static objects are declared entities, and, as such, their life begins with the introduction of a declaration into the static package. This declaration may exist only in source form for an arbitrary amount of time.

4. When the user decides the declaration is complete, the "install" operation is applied to the declaration, elaborating

it and creating an object that corresponds to it. Program units may then reference the object.

5. Some attributes of the object may be altered without affecting the object itself.

6. Later, the user may wish to change the declaration. Either a new "version" of the declaration can be created, or the corresponding object must be "withdrawn" before the declaration can be edited. The declaration is then altered, and, when the user desires, reinstalled, creating a new object.

7. When an object is "withdrawn", the elaborated instance is deleted, and the declaration is take out of the system. The declaration is retained in source form.

3.3. Attributes of Objects

1. Each object can have a variety of attributes associated with

it. Some types of information relate to the source of the object (mostly when the object is a program unit) and some relate to the value of the object.

2. Arbitrary ition can be related to a static object. The remainder of this manual assumes that attributes are associated with objects declared in static packages or sources of program units.

3. For each object, there are certain attributes that can be related to it. Not all objects can have the same attributes. Only static objects can have arbitrary attributes.

4. There is some basic information associated with each static object, including:

    1. Object name
    2. Type class
    3. Object state - Not elaborated/elaborated
    4. Object location - where object really is, if elaborated

5. Some categories of attributes are:

    a. Usage information - who is currently using the object and how are they using it. This allows editors to coordinate multiple requests to edit an object.

    b. Concurrency control information - How concurrent edit access requests are to be resolved. (either global lock, or update lock)

    c. Access information - who is allowed to access the

    object. The owner of the object can specify which users have what kinds of access to the object. (See 3.5)

    d. Help information - Used by the help object editor.

    e. Version information - what versions exist and who is using them. This is present if objects are replaced and old versions kept at object installation time. It is also used for more general version control, see 3.4.

    f. Instantiation information - what instantiations exist and where they are. This indicates the location of each version of the object.

    g. Archive information - how should the object be archived and when was it last archived.

    h. Automatic tool invocation information - Who should be notified and what should they be told when certain classes of changes are made to the object.

    i. ...

6. Users can examine the attributes of objects by requesting a view of the object declaration that includes whatever information is desired.

7. Programs can extract object and attribute information by calling appropriate procedures. Attributes defined by Ada may be referenced as defined in the LRM.

8. Modifications to, addition of, and deletion of attributes and objects are permitted only if the user making the change has edit access to the object.

3.4. Version Control

1. The system will maintain multiple versions of declarations and objects in static packages. Each version has the same name but has a different version designation.

2. Each static object has a declaration. The declaration includes a name and type (or specification and body if the declaration is of a program unit). If a declaration is installed, there is an object associated with the declaration.

3. The term "version" applies to the declaration, and, if the declaration is installed, also to the association between the declaration and the object.

4. Versions are named and/or numbered by the user.

5. The version control system keeps track of the order in which versions are created and of the parent(s) of each version.

6. References to objects for which multiple versions exist may specify an explicit version of the object, or may reference the "current version" which is designated by the object owner.

3.5. Access Control

1. Access control restricts the normal Ada operations available on visible Ada objects, and restricts the operations defined by the environment that are not in the domain of discourse of Ada.

2. For access control purposes, a user is represented by his home package. Thus, access is always granted to a set of packages.

3. The scope of a declaration can be decreased by limiting it to specific packages. Also a package can limit the identifiers imported into it. Finally, packages can specify which packages are allowed to manipulate them in non-Ada ways (such as editing, installing, and deleting objects).

4. Each static object can specify access rights for packages in the form of an access list which indicates the kinds of access allowed for each package. Various forms of groups and abbreviations make the specification reasonably compact and efficient.

5. The types of access and further details are described in 5.3.1, after environment manipulating operations are described.

3.6. Backup

1. Objects can be archived individually or by package or other grouping. The system supports attributes that advise the archiving system on how to backup and reload particular objects.

2. Each object can have attributes that specify how the object is to be archived. This information selects from a set of predefined methods or specifies a procedure to be called to perform the archive.

3. Each object can also have attributes specifying the time of the last archive and how often the archive is to be performed.

4. Users can arrange for objects to be archived regularly by either creating a task that traverses their static packages, archiving objects whose attributes indicate a desire to be archived, or by adding attributes indicating that objects are

to be archived by the system master archiver that is run periodically by the system or project administration.

5. Objects can also specify information describing how reload or recovery is to be performed. This typically is a procedure that the reload facility calls when a reload is performed. The procedure decides what to do with the reloaded information, whether the current state should be discarded, and how the reloaded information is to be processed.

3.7. Resource Control and Limits

1. There are several types of resources that are limited:

    1. Disk Pages
    2. CPU time
    3. Connect time
    4. Name usage

2. An "accounting entity" is a static package that is assigned quotas of one or more of the limited resources.

3. Limits for each resource are separately assigned for "permanent" tasks and packages (that survive crashes), and for temporary tasks and packages (that are deleted on crashes).

4. If a package or task is permanent, its resource usage is checked against the permanent allocation, and analogously for temporary packages and tasks.

5. The system administrator can designate packages as accounting entities. Users can designate packages below their main package as accounting entities and assign limits.

6. The assigned limits are deducted from the nearest enclosing accounting entity at the time the assignment is made.

7. Any package or task that is not an accounting entity derives its resource limitations from the nearest enclosing accounting entity.

3.8. Help

1. Users can associate help information with static objects.

2. The user displays and edits the help information associations by selecting a view of the object declaration that includes it.

3. When objects are changed, the user may be requested to inspect the help information to be sure that it is consistent. This feature can be disabled by the user.

4. The help information is structured so as to make replies to help queries work. The detailed structure is described elsewhere (place not yet determined).

Chapter 4
Editor

The editor is the primary interface between the user and all aspects of the environment. The user manipulates the environment by editing text images of objects in the environment, and by entering and editing commands to affect the environment.

This chapter describes the fundamental editor concepts and indicates how the editor is used to manipulate text. A summary of advanced capabilities for editing other types of objects, especially programs, is given with references to sections containing more detail.

4.1. Windows

1. A "window" is a terminal-like display and input path.
2. Windows have an identity, size, content, and state.
3. A window may currently be visible on the screen or it may not.
4. The user can control the visibility of windows.
5. Windows can be created, deleted, and/or removed from the screen.
6. Windows generally display the images of objects in the system.
7. The contents of a window can be edited in ways consistent with the semantics of the displayed object(s).

4.2. Screen Management

1. The screen display (parts of) a number of windows.
2. The user can control the placement of windows on the screen, and the set of visible windows.
3. Programs also create, alter, and delete windows and also control the contents of the screen to a certain extent.
4. A predefined message window is usually visible on the screen. It includes messages from other users, messages such as "you have new mail", messages about tasks that have done something significant, or messages defined by the user.
5. The window directory is an object that lists all currently existing windows; it can be displayed in a window and altered. By editing the window directory, users can make windows visible, delete them, and create new ones.

6. Windows can be made to "pop-up" (automatically appear) when they receive certain signals.

7. Windows can overlap on the screen.

4.3. Objects and Views

1. An "image" is a textual representation of an object.

2. The contents of a window is a (portion of a) structured image of an object.

3. A "view" is a mapping of an object to an object image. Thus, a view of an object determines the exact form of the display of the object. The view controls the amount and type of information displayed, and the format of that information.

4. The view may be under control of the user, or it may be pre-defined by the editor. This depends on the object being edited.

5. In the implementation, there is a central part of the editor called the "core editor", and object-oriented editors called "object editors". The core editor works with an object editor determined by the type of object being displayed and edited.

6. It is possible for the user to write object editors. This allows editing of user-defined objects and/or new editors for existing object types.

7. The system supplies object editors for most Ada-defined objects.

4.4. Detail Control

Detail control is the way in which an object image can be selectively condensed to make better use of limited display space.

1. Increasing the level of detail increases the amount of information and the depth of structure that is displayed. Decreasing the detail level has the opposite effect. For programs, this would roughly correspond to levels of indentation.

2. The level of detail can be set globally to meet an expectation for the use of the entire object or it can be set locally to directly control the appearance of a specific section of the im
3. The user is made aware that detail has been suppressed (by

whichever mechanism) by the presence of "...", an ellipsis, where the suppressed text would have appeared.

4. Detail is locally suppressed by selecting a part of the image

and requesting that it be condensed.

5. Detail is locally augmented by selecting a part of the image and requesting that it be expanded, causing the "..." in the region to be replaced with the text of the image (at some level of detail).

6. Decreasing the global detail level suppresses locally expanded detail only when the containing context of the local object is no longer visible.

7. Local level of detail is a characteristic of the object view and persists when the object leaves a window, but does not affect other windows onto the same object.

4.5. Prompts

1. Prompting is a mechanism where the editor places a highlighted token in the place of a requested input. The user then overtypes the prompt, replacing it with some input.

2. The object editor connected to the window being edited is responsible for placing prompts in the object image.

3. The prompt disappears from the object image as soon as the user begins typing over it.

4. Users can fill in prompts in any order desired, or not at all. (If the input is required for some activity to proceed, the user's failure to fill in a prompt will prevent only that activity from progressing. The user is free to move to another window and continue the session.)

4.6. Name and Command Completion

1. Completion is a facility by which the system provides a complete name (or other entry) from a typed string, normally a prefix of the complete name.

2. Completion can be performed automatically or by user request.

3. Automatic completion is only reasonable where the range of alternatives is small. Examples would be enumeration types, procedure names within a package and other contexts where syntax and semantics limits the possible choices to a very small set.

4. The choice made by a completion is visible as the choice is made.

5. The user is never required to know the exact length of the prefix required for completion. Where completion is provided early, additional characters that could be part of the completed name merely cause the cursor to move across their correspondents in the completed text.

6. For completion purposes, Ada names consist of segments, separated by "." and "_". Completion is available for

individual name segments.

7. There is a "try harder" facility for choosing among completions with the same prefix.

8. Name completion includes as much Ada context information as possible; The "try harder" command may fill in the left context of a name, not just other prefix values.

9. The completion interaction allows feedback as to the types of items to be accepted in the next token. This is especially useful in the context of prompts and automatic completion.

10. The user can define abbreviations that will be treated as names for completion; the result of the completion is the full name.

11. Access to completion does not limit the user's flexibility in choosing entry mechanisms, editing operations or changing contexts.

12. The state of a completion instance includes the current value of the object, how the values have changed, and an indication of the state at the previous attempt.

4.7. Output Control

1. Object editors or other programs can send output to windows to be displayed. If sequential output continues, the window will scroll, causing early information displayed to scroll out of the window.

2. The user can manually stop window(s) scrolling by issuing a command. All windows on the screen will stop scrolling (though output to windows with empty space and activities not involving sequential window output can continue). The user can also continue output by issuing an appropriate command.

3. When output scrolling is suspended, a message indicating this appears somewhere on the screen.

4. Windows can also be placed in modes where they stop scrolling after each window-full of information has been displayed, possibly depending on whether the window is currently visible. A message indicating that output is suspended is displayed in the affected window.

5. The user can directly control the viewing of output independent of the rate at which it is created by moving the viewing cursor back into the output. It is not planned to let the user see output before it is generated.

4.8. User Input

1. User input is based on the depression of keyboard keys, and on other input devices, if available.

2. A key is handled by first mapping it to a "command". The

command is then mapped to an "operation". The operation is then invoked.

3. A "key" is one of the keys on the terminal keyboard. If there are insufficient function keys, then multiple key sequences may be considered a single key. (This is messy; see below)

4. A "command" is a named class of operations. There are no semantics directly associated with commands; instead, each command stands for a class of semantically similar operations.

5. An "operation" is a specific Ada subprogram or entry.

6. The user can control the mapping of keys to commands. There is a default mapping sufficient for most users.

7. The mapping of commands to operations is determined by the context in which the command is issued. This context includes the current window and its associated object editor (which provides the basis for the map). Thus, keys may remain bound to the same command while that command performs different operations in different contexts.

8. Each window can have its own command-operation map. It can also have its own key-command map, though this is not usually the case.

9. Commands may be bound only to procedures declared in static packages. This guarantees that the procedure binding remains as the user context changes. If the procedure that the command is bound to becomes unavailable, then the binding is removed and the command's function becomes undefined.

10. There is also a global key-command map and command-operation map. Keys and commands not bound in editor maps are undefined.

11. One window on the screen is designated the "active" window. This means that input typed on the keyboard is interpreted using that window's command map and key map (augmented by the global maps).

12. Some keys and commands have semi-permanent bindings. Specifically, the help key and the "choose a different window" key always perform their functions regardless of the active window. (It is possible to rebind these keys, but if you do, you get what you deserve.)

13. When multiple key depressions are required to represent a single "key", the first key is called the "prefix" key. Once a prefix key is pressed, the next key(s) are interpreted based on the prefix key.

14. If the user types a prefix key and takes longer than some threshold time to complete a multiple key sequence, then a message is displayed on the screen indicating that the multiple key sequence is in progress. This message is removed immediately when the key sequence is completed.

15. There is an "abort-multiple-key-sequence" key that returns the interpretation of keys to the state where a prefix key was not pressed without taking any other action.

16. Most alpha-numeric keys are bound to the "self-insert" command whose execution results in the insertion of the character before the cursor of the currently active window.

17. To enter and execute a command, see sections 5.1 and 6.4.

4.9. Editing Operations

1. Editing operations are broken into the following groups:

    1. Screen management operations
    2. Intra-window cursor motion
    3. Inter-window cursor motion
    4. View control operations
    5. Basic character operations
    6. Additional editing operations
    7. Context management operations

2. The set of commands in each of these areas is described in following paragraphs. These are operations supported by the core editor for all windows. Object editors may add operations

applicable in windows under their control. Object editors may also remove some of the operations below. This is typically done only if an operation is not meaningful for the object being edited.

3. Screen management operations.

    1. Create_Window
    2. Delete_Window - The window directory cannot be deleted.
    3. Move_Window
    4. Make_Visible
    5. Remove_from_Screen
    6. Enlarge_Window
    7. Contract_Window
    8. Window_Directory
    9. Freeze_Scrolling - Keep things from scrolling out of windows.
    10. Continue_Scrolling
    11. Focus_on_One_Window - Delete most other windows, enlarging this one. Repeated issuing of this command eventually results in only the currently active window occupying the entire screen.

4. Intra-window cursor motion

    1. Cursor_Move(direction, distance)
    2. Search_for_String - Textual search (also regular expressions)
    3. Top_of_Window - Move cursor to upper left of window
    4. Bottom_of_Window - Move cursor to lower left
    5. Left_Margin
    6. Right_Margin
    7. Line_to_Top - Scroll the line containing the cursor to the top of the window.
    8. Line_to_Center/Bottom - Similar to Line_to_Top
    9. Scroll

10. Next_Page
11. Previous_Page
12. Beginning_of_Object
13. End_of_Object
14. Next_Word/Object/Token
15. Previous_Word/Object/Token
16. Next_Prompt
17. Previous_Prompt

5. Inter-window cursor motion

1. Move_to_Window
2. Next_Window
3. Previous_Window
4. Move_to_Last_Active_Window - i.e., pop
5. Mark_Position[_With_Label]

6. Return_to_Mark [name] - Move the cursor to the last [named] mark. This may or may not change windows or cause a window to become visible.

6. Selection operations

1. Select_Object
2. Select_Line
3. Add_to_Selection
4. Unselect_All
5. Unsel_Detail_Level (local/global)
2. Increment_Detail_Level
3. Decrement_Detail_Level
4. Expand
5. Expand_All
6. Elide

8. Basic character operations

1. Insert_Character
2. Replace_Next_Character
3. Join_Next_Line
4. Delete_Next_Character
5. Delete_Previous_Character
6. Delete_Line
7. Delete_Selected_Object
8. Delete_to_End_of_Line
9. Delete_to_Beginning_of_Line
10. Transpose_Characters
11. Capitalize_Token
12. Capitolize_City

9. Additional editing operations

1. Search_and_Replace
2. Query_Search_and_Replace
3. Copy_Selected_Region
4. Move_Selected_Region

10. Context management operations

1. Edit_Object - Create a new window (possierlaying the current one) and set up an object editor for the

designated object. This starts an edit session with a new object.

2. Pop_to_Parent_Context

11. Misc.

1. Keyboard macros
2. Abbreviation mode

4.10. Undo

1. Undo reverses the consequences of one or more user operations.

2. Undo is a meta-operation in the sense that it cannot be undone.

3. Redo meta-operations are available to achieve the effect of undoing an undo.

4. Redo is based on a (modifiable) script of actions to be done.

5. Granularity of undo is dictated by the difficulty for the user of producing the inverse without assistance, i.e. simple cursor motion is easily undone.

6. Undo returns the state of the object to what it was before the major operation being undone, including smaller granularity changes in the interim.

7. Undo sequences that span multiple objects are supported.

8. Undoing only a subset of changed objects is possible, but probably requires editing redo scripts.

Undo strategy is two-level in recognition of the core editor/object editor dichotomy.

1. Intervals of core editor activity that do not involve object editor involvement can be undone independently.

2. Undo across object editor "commits" can only be done in cooperation with the object editor. This leads to some lack of uniformity in how far back the user can undo, but no more so than most conventional systems.

3. The core editor keeps undo information across object editor interactions so the object editor need not keep journals of the changes that it sees, only previous states and enough information to back up to them.

4. Side effects of object editor actions make it impossible to always be able to undo all actions completely. Returning the editor to a previous state should be possible (with a warning) even if side-effects cannot be undone.

Chapter 5
Manipulating the Environment

This chapter describes the ways in which the environment is manipulated, and some additional details about its characteristics.

5.1. Object Editing, Commands, and Command Execution

1. Windows are used for displaying, entering, editing, all objects. Commands are one type of object. The semantics and types of commands available in a particular window may vary somewhat depending on the type of the object that it displays.

2. A window that is used to enter, edit, and request execution of commands is called a "command window".

3. When a session begins, a window displaying the user's root package is displayed on the screen.

5.1.1. Context

1. The "context" of a window is the location of the object that is being displayed. This location includes:

   a. The path name of the object (which specifies its location within the Ada packages).

   b. The version of the object (if more than one exists).

   c. Whether the object is some elaborated instance or source representation of a program unit.

   d. If the object is elaborated, then:

      i. The instance of the object (if more than one instance exists).

      ii. The invocation of the object (if the context is a subprogram invocation within a task).

      iii. The instance of the object (if the object is an elaborated generic instantiation.

2. The context of a window changes as the user moves in and out of objects. The context also changes from elaborated to source when the user makes a change to a program unit.

5.1.2. Naming Contexts

1. Contexts must be named in a command to set the context of an Ada window.

5.1.3. Command Windows

1. At any time, there may be several command windows in a user session; exactly one of them is designated the "current" command window.

2. Part of the current Command Window is usually displayed in the bottom few lines of the screen. It may be displayed elsewhere, be larger, or not be displayed at all.

3. There is a permanently bound key that sets the active window to the current command window. The user uses this key in most cases when commands are to be entered and executed.

4. When this is done, the window in which the command window key was pressed is designated the "last active window".

5. Many commands that can be issued in the command window implicitly refer to the last active window. This includes most editor commands, and some debugger commands, for example.

5.1.4. Command Window Environment

1. The command window has an environment in which Ada sentences are interpreted.

2. The environment consists of several components:

   a. Renames clauses - These provide names in the environment that are synonyms of other object names. Syntax and semantics are as in Ada renames clauses.

   b. Use clauses - These import still more names into the environment. Semantics are as in Ada use clauses.

   c. Context - The location of the environment within the user's program units.

3. Both the user and programs can control the components of the environment by both adding and removing declarations of renames and use clauses, and by changing the context.

4. The user can most easily control the names by editing the declarations that actually appear in the command window.

5. Contexts are specified in one of several ways:

   a. If the context is a static package, then the context is specified by giving the package name. If the package name is not visible from the current context, then a context change to a context where it is visible must be effected first.

   b. If the context is part of a task, then the task must be specified along with the package within the task.

   c. If the context is a procedure frame, then the task, procedure, and instance of the activation must be specified.

   d. If the context is a block, then the task, procedure, block, and instance of the procedure and block must be specified.

   e. If the context is a generic, then the task, package or procedure, and instance must be specified.

5.1.5. Establishing Command Window Context

1. The command window context is set based on how it is entered, and on commands given to explicitly control the context.

2. If the command window is entered by issuing the "command_window" command from another window whose context is elaborated, then the command window context is set to the context of that window. If the context of the other window is not elaborated, then the command window context will be set to the nearest enclosing elaborated package that contains the context of the other window.

3. If the command window is entered by the user requesting that selected Ada statements be executed, then the context of the command window is set to that of the selected Ada statements.

4. The user can also issue the following commands that control the command window context:

   a. Set_Context - Set the command window context to a specified context.

   b. Sync_Context - Set the command window context to that of the last active window.

   c. Fix_Context - Mark the current command window context as "permanent"; that is, subsequent transitions to the command window will not affect its context. Sync_Context removes the sticky nature of the context.

5. The context can be set by the user to any context, subject to the following restrictions:

   a. The context must be elaborated: a static package, an elaborated program package, or an existing procedure frame.

   b. The user must have "Command context" access to the context. (See section 5.3).

5.1.6. Command Execution

1. Recall that

   a. Commands are names that represent classes of operations.

   b. The global and active window command-operation map determine the actual operation executed when a command is entered.

   c. Operations are Ada-invokable entities.

2. Commands are Ada-like in syntax but not necessarily exact Ada calls. In any case, commands are invoked as Ada operation calls.

3. Operations may be invoked directly by typing an Ada subprogram

or entry call and issuing the "execute" command.

4. These operations are compiled in the command window environment and all names given in the operation must make sense in that environment.

5. User programs can also request that statements (given as strings) be executed in a particular context. The context restrictions noted above apply.

6. More details of the command execution process are presented later. (See 6.4).

5.2. Moving Within the Environment

1. The user sets the context of the editor by specifying a package to be edited.

2. This is done either by naming the context, or by moving to the desired context by moving through adjacent contexts.

3. From a given context, the user can request that the context be changed to the parent (enclosing) context.

4. The user can select an object and request that the context be changed to the selected object. This object is typically a program unit.

5. Moving into a task instance or generic instance sets the context to the elaborated context.

6. If the context is set to a generic instance, the display will show the source of the generic, not the code with generic parameters substituted.

7. If the source of a program unit is edited in an elaborated context, the context changes from elaborated back to source.

8. Moving into a data object leaves the command window context at the nearest enclosing package. The data object context is considered elaborated.

9. The user must have edit access to the context to change the source of any objects there. The user must have display access to a context to display the source of any object there.

10. The user can issue the "pop window" command to move from a window to the previous one. (That is, from the command window to the window from which the command window was entered)

5.3. Access Control Issues

1. Access control deals with two issues:

    1. Users performing environment manipulations that are outside the domain of discourse of Ada.
    2. Programs (or users) running statements under Ada

semantics and accessing objects.

2. The access control system can reduce the access normally available via Ada scope rules; it can never grant access that would not have been available under Ada rules.

3. An access list can be associated with any named object except as noted below. The list specifies packages that may refer to the named object, and the type of access that they are allowed.

4. For simplicity, access lists cannot be associated with record fields or parameters.

5. A "User" is a named entity that cgranted access to the system. Each user has a "home package".

6. When a user logs on, a "session" is established through which the user issues commands and manipulates objects. Commands issued from a session have access based on that of the session user's home package. Thus, to grant a user access to an object (independent of that user's current context), one grants access to the user's home package.

7. Access rights are checked at the time that the manipulation begins in the case of meta-Ada environment manipulations, and

at program installation time for normal Ada access by program units.

5.3.1. Types of Access

* There are 6 types of access defined:

1. Ada Reference. Allows the accessor to perform any Ada-defined operation on the object subject to the normal Ada rules.

2. Context. Specifiable for program units only. Context access allows a session evaluation context to be set to the program unit. Then, the Ada Reference access of the program unit and the session user determine the capabilities of statements evaluated in the context.

3. Install. Specifiable for packages only. Allows the accessor to install or remove object declarations from the package.

4. Edit. Allows the accessor to edit the object using an editor defined for that object's type.

5. Display. Allows the accessor to display the value/contents of the object.

6. Owner. Specifiable for packages only. Allows the accessor to change the access control information for the object, and to delete the object and its declaration.

* The above access rights are granted to packages.

* Access is controlled on a per object basis. Each object

specifies the types of access each package is allowed.

5.3.2. Environment Operations and Access

1. Access restriction may prevent elaboration or installation. All objects referenced by a program unit must be both visible and accessible before the unit can be installed.

5.3.3. Ada Program Access

1. In order to reference an object (which includes operations), the object name must be visible (in the Ada sense) from the context of the referencer.

2. Objects can also have additional access restrictions. It may be the case that although a context has Ada visibility to an object, an additional access restriction may prevent the access.

3. An object is said to be "accessible" from a context if it is visible and the package containing the context has access rights (defined by the object's access list) to the object.

5.3.4. Granting and Revoking Access

1. Each object starts with a default access list.

2. The default is controllable by the user, and by various levels of system administrators.

3. Access lists are edited by specifying a view of a static package that includes the access list and editing it. Access lists can also be changed by making appropriate calls to the directory management package.

4. The accessor must have owner access to the object to change its access list.

5. If an object referenced by a program unit is made inaccessible after the unit is elaborated, the access will still be permitted if the unit is a task and will cause an exception when access is attempted otherwise.

6. When an access right is revoked, active threads of control that have access to the object will continue to have the rights specified at the time the program units they are executing were elaborated. Other than that, the revocation is (relatively) immediate. (fuzzy)

7. A user very concerned with revocation of access rights can create a new version of the object and delete the old. This results in immediate termination of any rights that anybody has.

5.3.5. Specification of Import Access Constraints

Any package or subprogram specification with a with clause in its context specification is a "library" unit, meaning that it is a closed scope. Only unit names may appear in the with clause. The only deviation from Ada is the fact that library units appear nest in other

units; however, the enclosing units can be viewed as a hierarchy of libraries.

5.3.6. Specification of Export Access Constraints

The syntax and semantics of Ada are extended as follows:

```
Package_Specification
        ::=     PACKAGE identifier IS
                    {Declarative_Item [export_spec]}
```



```
                [PRIVATE
                    {Declarative_Item}
                    {Representation_Specification}]
                end [identifier];

Export_Spec ::= EXPORT TO Access_List;

Access_List ::= User_List [(Access_Constraint)]
                     {, User_List [Access_Constraint]}

Access_Constraint ::= Retyped_Object_Name | Access_Type_List

Retyped_Object_Name
        ::=     Object_Name [AS function_call]

Access_Type_List
        ::=     WITH Access_Identifier {, Access_Identifier}

Access_Identifier
        ::=     CONTEXT | INSTALL | EDIT | DISPLAY | OWNER

User_List
        ::=     User_Name {, User_Name}

User_Name
        ::=     Package_Name [*] | ALL | Group_Name

Group_Name
        ::=     ( Name )
```

1. The EXPORT part of a declaration lists the set of packages or groups of packages that can access the declaration. A declaration can be made visible to all packages that could normally see it (with no export spec), or to only specific packages or groups of packages that could normally see them. An export can never give access to a package that could not see the object under normal Ada visibility rules.

2. The packages listed in the User_List need not exist at the time of installation of the restricted unit. The check is performed when a referencing unit is installed. If the "*" follows a Package_Name in the User_List, this means that all descendants of that package are also allowed access.

3. Access constraints are applied to objects either to indicate that the object is being exported with a different (more

restrictive) set of Ada operations, or to indicate the kinds of access (context, edit, etc.) for that object. The function call in a retyped_object_name must be a conversion function which converts the type of the object.

4. Access to objects that do not contain an Access_type_list is based on the innermost containing package that does include an Access_type_list part. Thus, the root package of a program can include a specification of who is allowed to edit it. This specification would apply to all contained packages.

5.3.7. User Interface Issues of Access Control

The view control facility normally does not display the export and access part of a package. The access information can be included in the view by explicitly changing the view or by issuing one of a few commands that affect access control parameters.

5.4. Commands for the Command Windows

1. Since edit commands implicitly refer to the last active window, a second command window must be created in order to edit the first.

2. This is done by giving the "command_window" command from a command window.

3. Then, the last active window becomes the first command window, and commands can be issued to edit it.

4. The environment for the new command window is initially the same as the previous one.

5.5. Display, Access, and Update of Objects and Attributes

1. A user with "edit" access to an object can update the object or its attributes.

2. A user must have "install" access to the containing object to install the the modified version.

3. The static package potentially lists many versions of an object, including zero or more installed (and hence elaborated) versions and one or more "paper" versions which either represent current versions, or are being prepared for installation.

4. One of the installed versions is designated as "current" - this is the object that normal users of the package see and can access. If a new object entry or revision is in progress, it is only displayed if a view of the static package that includes objects being modified is requested.

5. A program unit can also reference other than the default version of an object by explicitly specifying the version in

the name reference to the object. Details of this are given in a later chapter.

5.6. Version Control

5.6.1. Naming Versions

1. Extended naming is required to properly reference objects in an environment with multiple versions of objects and declarations.

2. Names within Ada program units are bound to objects when the program unit is installed. This occurs when the user explicitly requests that a program unit be installed as an executable object in the system. The installation process is described in section 6.2.1.

3. A name is said to be "version qualified" if it includes a version designator.

4. A "version designator" is an attribute of the form "VERSION(name)" where name is the name or number of the desired version. Thus, an object named X, version FOO would be denoted "X'VERSION(FOO)".

5. Operator symbols may not be version qualified. (???)

6. If a name is not version qualified, then at installation time the name is bound to the current version of the object. If a name in the declaration of an installed object is not version qualified, then the name references the current version of the object it identifies.

7. It is recommended, when programs are written that explicitly refer to old versions of objects, that the version-qualified name be used only once in a renames declaration and all other references not include version designators.

8. The system may automatically add a version designator to an object name within the declaration of an installed program unit. This keeps object references within the program unit consistent with the objects actually referenced.

5.6.2. Version Freeze and Update

1. A version is "frozen" if changes are not allowed to be made to it.

2. A version is "permanently frozen" when the user requests that it be frozen, or when new versions are created that are based on it. It can never be modified after this point. (New versions based on it can be created.)

3. A version is "temporarily frozen" if it is installed and is being referenced by some other program unit. The freeze may be removed by withdrawing the referencing program unit. This may

require withdrawing the transitive closure of the referencing program unit.

4. When a user tries to edit a frozen declaration, the system gives the user the option of creating a new version or, if the freeze is temporary, breaking the freeze by withdrawing referencing program units.

5. When a new version is installed and designated the current version, references to old versions in existing and installed program units must be qualified to keep them consistent with the objects that they actually reference. A version designator is added to each reference to the old version.

6. When a version is created based on a frozen version containing version designators added by the system, the user is given the option of preserving the version designators in the source of the new version, or of having them not appear in the new version. Individual, groups of, or all version designators can be preserved by issuing an Ada editor command.

7. There is provision for sending mail to the users of an object to inform them of a newly available version. Individual users of the object can then decide whether or not to convert to use it.

8. Finally, versions can be deleted. The object associated with a deleted version is also deleted.

9. Deleting a declaration may delete all versions of it. It is also possible to preserve the versions; they are accessible only by using version-qualified names, or by requesting an editor view that includes deleted object declarations.

5.6.3. Version Operations

The following operations are supported for creating and manipulating versions of declarations:

1. Create new version - Create a new version of a declaration. It is unfrozen and is based on a specified existing version.

2. Freeze version - The specified version is frozen. It can no longer be changed.

3. Merge versions - Create a new version that is the merge of several existing versions. An editing session is setup so that the user can located differences between the parent versions, and choose what is to be placed into the new version.

Chapter 6
Program Preparation and Execution

6.1. Editing Ada Source

6.1.1. Editor Entry Assistance

1. The editor provides syntax-directed completion facilities.

2. Completion works by the user entering the first part of an Ada sentence followed by the "complete" command. The editor then tries to complete the Ada structure that the user began.

3. If the user is unhappy with the editor's completion, then the user can request either that the editor "uncomplete" its completion, restoring the sentence to the state before the user issued the "complete" command, or that the editor try harder and present an alternate completion.

4. Completion can be applied to both Ada syntax constructs (such as packages, procedures, if statements, etc) and names.

5. The editor performs completion of a name by matching the user's prefix of the name with a set of names that can be (legally) used in the position where the user has placed it.

6. The user can negotiate with the editor by using the "uncomplete" and "try harder" command to get the editor to supply the correct name.

7. Repeated use of the "complete" command enlarges the completion, for example by providing additional "when" clauses to a case statement.

8. The following elements in the source are linked. Whenever the opening identifier is changed, the closing identifier is also changed.

   a. Opening and closing identifiers for subprograms, modules, and blocks.

   b. The specification of subprograms that appear in the visible part and body of a module.

   c. Optionally, a name and its references.

9. There are also some editing operations that allow for simple, consistent updates to multiple program units:

   a. Renaming an object and all references.

   b. Update aggregates when a new field is added.

   c. Update calls when a subprogram heading is changed.

6.1.2. Editing a Program Unit

1. There are no restrictions controlling what aspects of a program unit can be changed, in what order changes are made, in what state a program unit is left when the edit session is terminated, or what types of changes are made.

2. When a unit is edited, the system keeps track of the currently installed version and the new one (or new ones) that are being developed.

3. At any point in the edit session, the user can issue the "format" command. This causes the syntax of the program unit

to be checked and the program to be reformatted.

4. At any point in the edit session, the user can issue the "check" command that checks first syntax and then semantics of selected program units. If errors are found, highlighted messages are placed in the program source text.

5. The check operation checks the semantics of user specified program units in the context of other currently installed program units. Where a user specified unit is a new version of an installed unit, the installed unit is ignored.

6. At any point in the session, the user can request that the program unit be installed. The installation procedure is described in 6.2.1.

6.1.3. Views

1. A number of views on the source text are possible. They are:

   a. Version view - Shows what versions of program units exist, which are currently being displayed and edited, and which are designated current.

   b. Configuration View - Shows configuration differences in the unit being displayed. Allows the user to edit all configurations simultaneously. This is discussed further below.

   c. No-comment View - Removes comments from the display.

   d. Compact View - Compacts the program display so as to show the maximum amount of the program. Removes blank lines, places multi-line statements on one line where possible, etc.

   e. History View - Shows the modification history of the source unit.

   f. Documentation View - Shows documents associated with (parts of) then current unit.

   g. Help View - Shows and allows editing of help information associated with (parts of) the current unit.

   h. Access List View - Displays and allows editing of access lists associated with objects.

   i. Accessible View - Lists only those objects to which the user has Ada Reference access.

6.1.4. History Capture

1. As changes are made, the system keeps track of which program units were modified. At points in the edit session, the user may be requested to describe the changes and record reminders to update corresponding documentation.

2. The history capture facility can be configured by the user or project administrator. The frequency and depth of its demands

can be set.

3. The history information is typically collected when the user suspends the edit session, installs the unit, or explicitly updates the history.

6.1.5. Preparation for Execution

1. The "format" command parses and reformats the program. This guarantees at least syntactic correctness.

2. When the user issues the "commit" command and/or the "install" command, the program must be semantically correct.

3. At these points, errors discovered will be reported to the user. The error messages can appear in an error log, and/or in the program source text. There is a command that requests the construction of an error log.

4. Messages are placed in the text associated with the part of the program in which the error occurred, and are highlighted.

5. The user can request that all such errors be removed completely, or that the view be changed so that they are not displayed. On each "check", old messages are first deleted so that only the newly created error messages are displayed.

6. The error messages are associated with the program node to

which they apply. The user can give a command to move to the next such error message. The error message disappears from the view after the node it applied to is edited.

7. Messages placed in error logs name, to the best possible approximation, the innermost program unit containing the error, and the number of the line within that unit. The line number is based on the normal view of the unit (not including special information).

6.2. Object Creation, Deletion, and Alteration

6.2.1. Installation and Elaboration

1. Objects are entered into packages through a process called "installation".

2. Static objects can be installed in static packages. Declarations of program units can be installed in static packages or other program units.

3. In order to be installed, an object definition must be complete and semantically correct in the context of the installation site.

4. The object is installed when the user issues an explicit install command. An error message is returned if the object cannot be installed.

5. Several objects can be installed simultaneously. This is

necessary, for example, to install packages whose bodies reference each other's visible parts.

6. If an exception is propagated out of the unit being installed during elaboration, then the object is not installed. Side effects caused by the the elaboration code of the object up to the point where the exception propagates out remain in effect.

7. Programs can also install objects by making a call to the package manager. The same restrictions apply, and an error is returned if the installation is unsuccessful.

8. Installation requires that the user (or program) have "install" access to the nearest enclosing static package. (See 3.5)

9. Deletion of an object declaration is equivalent to replacing it with nothing; the rules of deletion follow those of replacing an object declaration with a different one.

10. Elaboration refers to the Ada-defined process of "executing" declarations in a declarative part prior to executing the statements.

11. A program unit must be "installed" before it can be invoked, and before a unit that references it can be installed. This is basically the compilation order restriction.

12. The check (used to check semantics of a program unit) operation does not require prior installation.

13. Installation involves performing final semantic checking, code generation, linking of code segments, and elaboration of the new object.

14. Installation of a unit causes any contained subunits to be installed.

15. There are significant implications to the environment when an object is installed. Recompilation of many other program units may be required. The options and restrictions involved are described in more detail in the next section.

6.2.2. Object Installation Options

1. Installation proceeds as follows:

   a. The user edits a text representation of the object into the static package at the point where it is to be installed. Prior to installation, the object is considered to be relatively raw text/diana. Arbitrary manipulations are possible without consequences.

   b. The user selects the text of the new object and gives the install command.

   c. If the object is incomplete or otherwise in error, an error message is returned to the user, diagnostic prompts may be placed in the object, and the installation is not performed.

   d. If all is okay, then the installation proceeds. Several

parameters are supplied: a) Request dependent recompilation, b) Make version, c) automatically update references. These either request re-installation of non-static program units that reference the object, make a new version of the object leaving existing references referring to the old version, or cause all old references to refer to the new version. The later option is available only if the interface to the new object is unchanged.

2. A consequence of installation may be that the environment may not be semantically correct after the installation. This happens if:

1. A new object is introduced that hides an outer identifier from and inner reference,
2. A new object is introduced that causes another to disappear from a scope where it was formerly introduced by a "use" clause,
3. An existing object is replaced with one with a different type or interface,
4. An existing object is deleted.

3. In addition, when an object is replaced with another with the same interface, references to the old version of the object may need to be updated to refer to the newly installed object.

4. An installed program unit is said to be "affected" by an installation if it falls into one of the above categories.

5. If there are no affected program units, then the installation proceeds without further interaction.

6. If there are affected program units, the user is given the option of not proceeding with the installation, or, for each affected program unit, specifying one of:

   a. Update references to refer to the new object. Only allowed if the user has install access to the affected program unit and if the interface to the new object is identical to the old.

   b. Leave references refering to the old object.

   c. Reinstall the unit. This results in the loss of the state of the unit. Not allowed for static packages.

7. Old versions of objects can be explicitly deleted. In this case, references from other objects become invalid and will generate exceptions if access is attempted. In some cases, the referencing object can be withdrawn, making it not runable.

8. If a program attempts to access an object that has been deleted, then the exception "Dead_Object_Error" will be raised in the referencing task.

9. From an object's point of view, when it was installed, it was consistent with the current version of its environment. As the environment changes, the references made by the object are

updated to reference older versions of its environment, or are updated to refer to new versions of objects in its environment if those new versions present the same interface. If an older version becomes unavailable, the object will finally no longer be able to exist and will break.

6.2.3. Changing Data Objects

1. The user can edit data objects (though not Ada program units) in an elaborated package. An object editor for the object's type is used to do this.

2. There are two classes of operations that can be applied to data objects in an elaborated package. These are value-preserving operations and non-value-preserving operations.

3. Value-preserving operations do not result in loss of an object's value. The following are value preserving operations:

   a. Displaying

   b. Renaming

   c. Changing comments attached to an object.

   d. Changing certain attributes of an object.

   e. Moving an object from one package to another in certain cases.

4. The "commit" command is used to incorporate value-preserving changes into an elaborated package. (To reduce the number of commands, perhaps use install with a prompt as to whether new version is desired.)

5. Non-value-preserving operations include creation, deletion, and type change of an object. As with program units, the "install" command is used to place the new version of an object into the elaborated package. The normal options for handling existing references to the object are available.

6.2.4. Pragmatics of Installation

1. While a program unit in an elaborated package is being edited, active entities can reference the existing, elaborated version of the unit. When the edited unit is installed, the user is given the options normally available for installation. (see 6.2.1)

2. There are several cases to consider for editing and installation:

   a. A program unit directly contained in a static package is edited and then installed.

   b. A program unit nested within other units that are not static packages is edited and then installed in its nearest containing module.

c. A program unit nested within other non-static units is edited, and the highest level non-static unit is then installed in its containing static package.

3. These cases correspond to the following situations:

a. Changing and updating a simple function or the top level of a larger program.

b. Changing and updating a running program by replacing a module of it (or a part of a module).

c. Changing and updating a part of a program and installing a complete new version of the entire program.

6.2.4.1. Simple Object Updates

1. This is the case when small subprogram or data objects are changed.

2. New versions of them are installed in static packages and the normal options for doing this are available.

3. Typically, the option chosen would be to update references to refer to the new version of the object (Subject to the normal rules, of course).

6.2.4.2. Updating Running Programs

1. This is the case when a module or procedure of a running program is to be updated.

2. The user makes the editing change and then applies the install operation to the edited object.

3. This results in its placement into the containing unit. The update option would request that references be converted to the new object, subject to the normal rules for doing so.

4. If only subprogram bodies are changed, then existing invocations of those subprograms continue to use the old code while new invocations use the new.

5. If a package body is changed (other than subprogram bodies), then that package must be re-elaborated at installation time. This will result in the loss of the package's current state and the re-running of its initialization (outer block) code.

6. A change to a package visible part is equivalent to changing the enclosing package's body.

6.2.4.5. Updating an Entire Program

1. In this section, "program" refers to the subtree of Ada structures that contain no static packages but whose root is in a static package.

2. Users typically write programs which are invoked only at the root and terminate.

3. When a part of such a program is edited, the user usually wishes to make a new runable version of the entire program and invoke it.

4. This is done by editing the appropriate parts of the program and then installing the root. This installation does appropriate, minimum recompilation as needed, and replaces the program in the static package. The user has the usual update options for dealing with existing references.

6.3. Configuration Management

6.3.1. Introduction and Goals

The purpose of the configuration management system is to help coordinate the production of large software systems by various groups of technical people. Different groups may handle development, quality control, test, and maintenance. Each group may consist of several individuals.

Specific functions that must be supported include:

1. the documentation of the Ada structure of versions of a program,

2. the automatic construction of an executable instance of such a program,

3. the ability to define and instantiate configurations of a program that are composed of various versions of program units,

4. the ability to track system evolution,

5. a convenient way for individuals working on a project to define and instantiate a test environment that includes new versions of their modules without affecting other developers or other versions of a program.

6. managing multiple implementations of a package (several bodies for one visible part),

7. recording (to allow reconstruction) of the current configuration of a program, and

8. the definition and instantiation of a configuration that includes standard instances of most components, and experimental versions of one or more components.

6.3.2. Definition of Terms and Concepts

1. A "version" of a program unit is a declaration of the unit

(which includes its specification and body).

2. Versions are named with strings supplied by the user (or semi-automatically by the system) when the version is created.

3. A "configuration" is an abstract object that describes the structure and content of an Ada program unit. The structure is described by listing the nested unit structure. The content is described by indicating the version of each unit.

4. The structure description in the configuration is redundant with the actual structure of the program unit; that is, each of the actual program units includes, as part of its structure, its subunits. The default version of a unit is used whenever no explicit version qualifier is given.

5. Configurations can have discriminants. The discriminants are used to parameterize the configuration allowing the exact version choices to be based on version actual parameters external to the configuration.

6. An object of type configuration can be instantiated. This results in the creation of the program unit that the configuration describes. The environment treats such instances, called "configuration instances", much as it would an instance of a generic unit.

7. Configurations are "instantiated" by the installation of a declaration whose form is similar to a generic instantiation. The program unit described by the configuration object is instantiated.

8. When a configuration is instantiated, versions of all the program units referenced by the configuration are gathered, checked for consistency, and (if consistent) instantiated. Discriminants of the configuration are set to actual values supplied in the instantiation.

9. A session's context can be set to the instance and normal debugging operations applied.

6.3.3. Structure of Configuration Definitions

A configuration definition consists of nested program unit specifications that follow the Ada structure of the program being described.

The definition specifies, for each program unit, the specific version that is to be part of the configuration. Defaults and parameterizations are also allowed.

The syntactic structure is:

```
Config      ->  CONFIGURATION identifier [discriminant_part] IS
                    unit_selection
                    [unit_selection]
                END identifier;     unit_selection ->
                unit_specification [version_qualifier]
                    [unit_contents];
            ->  CONFIGURATION name [version_qualifier];
```

```
unit_contents ->
                IS (unit_selection) END identifier

unit_specification ->
                subprogram_declaration
             -> PACKAGE [BODY] name
             -> TASK [TYPE | BODY] name
             -> GENERIC subprogram_declaration
             -> GENERIC PACKAGE name

version_qualifier ->
                ' VERSION ( version_expression )

version_expression ->
                version_name
             -> string_expression

version_name -> string_literal
```

6.3.4. Semantics of Configuration Specifications

1. A configuration describes a program unit and its subunits. The top level unit described is called the "program".

2. A program unit is said to be "included" in a configuration if the unit is part of the program.

3. Unit_selections in a configuration specify the specific program units that will be included in an instance of the configuration. The choice of the actual unit to be included is made at the time of instantiation of the configuration. (The "time of instantiation" is the time at which the declaration of the configuration instance is installed.)

4. The components of the configuration are program units that are included in the configuration based on the following rules. When the term "current version" is used, it refers to the current version at the time of instantiation.

   a. The top level unit listed in the unit selection of the Config part of the configuration is included.

   b. For each unit listed (in a unit_selection),

      * if there is no version_qualifier, then the current version of the unit is included,

      * if a version_qualifier is present, then the specified version of the unit is included,

      * if there is no unit_contents part, then the current versions of subunits declared within the unit are included,

      * if there is a unit_selection clause, then the specified versions of subunits listed in the unit_contents, and the current versions of subunits

not listed in the unit_contents are included.

5. If two unit_selections appear in the Config part (the top level unit) then they must be corresponding package visible part and body, task visible part and body, or task type and body.

6. If only a single unit_selection appears in the Config part, then it must list a subprogram, generic subprogram, or package visible part that has no corresponding body.

7. If configurations are nested (which is the case when a unit_specification is of form "CONFIGURATION name") the top level unit(s) and the subunits they contain are included at the position where the nested configuration is listed.

8. A configuration cannot be contained in itself.

9. We say two declarations x and y "match" (Match(x,y)) if:

   a. they are subprograms and have the same parameter and result type profile (LRM 6.6),

   b. they are both package visible parts or both package bodies and have the same name,

   c. they are both task visible parts, both task bodies, or both task types and have the same name, or

   d. they are both generic units and have the same name.

10. The "imports" of a program unit are exactly those declared entities that are inherited from the environment of the declaration of the program unit.

11. The "environment" of a declaration is exactly those declared entities that are visible at that declaration.

12. To be successfully instantiated, a configuration is subject to the following restrictions:

    a. All program units named in the configuration must be visible from the point where the configuration is declared.

    b. If a unit y is listed as a component of unit x, then we must have:

       i. There exists z declared within x such that Match(x.z, y).

       ii. Import(y) is contained in Environment(x.z)

13. These rules have the following implications:

    a. Unrelated program units (from a nesting point of view) may be nested in a configuration.

    b. If a program unit contains a version qualified name that references a version that is not included as part of a given configuration, then an error results if that

configuration is instantiated.

14. The discriminant part, if present, must include only discriminants of type string, Version_Name, and subtypes and derived types of string and Version_Name. (type Version_Name is new string;)

6.3.5. Declaration and Manipulation of Configurations

1. A configuration object is a normal Ada object (LRM 3.2) whose type is Configuration. Configurations are declared in the normal manner (LRM 3.2.1). Configuration is a private type. (Maybe limited???)

2. Configurations can be edited with an object editor designed for configurations.

3. There are also several other operations defined on configurations:

   a. Set configuration. This takes a program unit and a configuration, and initializes the configuration to list

   the current structure of the program unit. All contained units that exist in multiple configurations are listed.

   b. Assignment. Configurations can be copied.

   c. Instantiate. This operation builds an instantiation of the program described by the configuration. (See discussion below)

6.3.6. Instantiation of Configurations

The syntax of a configuration instantiation is:

```
Configuration_Instantiation ->
                [GENERIC] PACKAGE identifier IS
                   NEW configuration_name [config_actual_part];
        ->      [GENERIC] (PROCEDURE | FUNCTION) identifier IS
                   NEW configuration_name [config_actual_part];
        ->      TASK [TYPE] identifier IS
                   NEW configuration_name [config_actual_part];

config_actual_part ->
                config_association {; config_association}

config_association ->
                [config_formal_parameter =>]
                        config_actual_parameter

config_formal_parameter ->
                parameter_simple_name

config_actual_parameter ->
                string_expression
```

1. Configuration instantiation is similar to that for generics.

2. Discriminants must match and are processed as discriminates for a type. (See LRM 3.7.1, 3.7.2) Default values for discriminants are allowed.

3. Configurations can only be instantiated within static packages via the installation of a declaration that is a configuration instantiation.

4. The check operation (ERM 6.1) may also be applied to the declaration of a configuration instantiation. This causes semantic checking of the contents of the configuration as it would be instantiated at the time the check operation is issued.

5. The class of the name in the declaration of the instantiation (PACKAGE, TASK, PROCEDURE, FUNCTION, etc) must be the same as the top level object in the configuration.

6. The object introduced by the configuration instantiation must be semantically legal where it is installed.

7. The object introduced by a configuration instantiation has the type and specification of the top level object in the configuration. This type is used for semantic checks, and for Ada references to the configuration instantiation. Thus, the name of the configuration instance renames the top level object in the configuration.

8. When a configuration is instantiated, the components of the program it describes are gathered and, if (individually and as a group) they pass Ada semantic requirements, instantiated to form an Ada object.

6.3.7. Versions and Configuration

1. Multiple versions of configurations can be created in the normal way for static objects. (In addition, there may be multiple instantiations of any version of a configuration.)

2. When a configuration is changed, if there are instances of it, the user is given the normal options for creating new versions of the configuration when the edited configuration is installed. (ERM 5.6)

3. A version of a configuration is frozen (ERM 5.6.2) if an instance of it exists.

4. Versions of program units referenced in a configuration instance are frozen as long as the instance exists.

6.3.8. Examples

Configurations:

```
configuration Compiler is
    procedure Compile (Unit: Diana.Tree) is
        package Parser;
        package body Parser'Version("3.1");

        package Semantics;
        package body Semantics;

        package Code_Gen'version("1.1");
        package body Code_Gen'version("1.5") is
            package Machine_Defs'Version("VAX");
            package body Machine_Defs'Version("VAX");
        end Code_Gen;
    end Compile;
end Compiler;


configuration Teller (Vendor: Version_Name;
                      Branch_Class: Version_Name) is

    procedure Process_Transaction (x: Terminal_Id) is
        package Command_Interface'version(Vendor);
        package body Command_Interface'version(Vendor);

        package Branch'version(Branch_Class);
        package body Branch'version(Branch_Class);
    end Process_Transaction;

end Teller;
```

Configuration Declarations

```
Development:    Configuration;
Test:           Configuration;
Release:        Configuration;
```

Configuration Instantiations:

```
Compiler is new Development;
Test_Compiler is new Test;
Old_Compiler is new Test'Version("out.of.service");
Special_Compiler is new Test("Dianax");

procedure Handle_Transaction is new Teller("IBM", "Small");
procedure Handle_Transaction_2 is
        new Teller("NCR", "Main_office");
```

The following example relates to the compiler / diana / object editor / machine interface / debugger development structure.

Note: This example is still in progress.

```
package body Ada is

    package Diana is ... end Diana;
    package body Diana is ... end Diana;
    package Machine_Defs is ... end Machine_Defs;
    package body Machine_Defs is ... end Machine_Defs;
    package Debug_Interface is ... end Debug_Interface;
    package body Debug_Interface is ... end Debug_Interface;

    package Compil is ... end Compil;

    package body Compil is

        package Scanner;
        package body Scanner;
        package Parser;
        package body Parser;
        package Semantics;
        package body Semantics;
        package Code;
        package body Code;
        package CodeGen_R1000;
        package body CodeGen_R1000;
        package CodeGen_MV8000;
        package body CodeGen_MV8000;

        procedure Compile (Unit);
        procedure Pretty_Print (Unit);

    end Compil;

    package Debug is ... end Debug;
    package body Debug is ... end Debug;
    package Edit is ... end Edit;
    package body Edit is ... end Edit;

    package Object_Editor is ... end Object_Editor;
    package body Object_Editor is ... end Object_Editor;
Compiler_Config:   configuration;
    Debugger_Config:   configuration;
    Editor_Config:     configuration;

    package Compiler is new Compiler_Config;
    package Debugger is new Debugger_Config;
    package Editor is new Editor_Config;
end Ada;
```

6.3.7. Notes on Configuration Management

1. An alternative facility being considered involves extending generics to allow package parameters.

2. An additional facility for configuration control on a smaller scale remains to be designed. This facilty would allow the substitution of small regions of text within a program unit. This is desirable where there are only very minor differences in a few places between versions of a unit.

6.4. Program Execution

1. The commands requested in each command window are executed in a separate task.

2. Each command window has an associated context that can be changed by user action.

3. Within a command window, execution is sequential.

4. Asynchronous execution is possible by moving to an idle (new) command window.

5. Each command window has an associated command history log.

6.4.1. Program I/O

1. Program-created objects can be monitored on the screen as they are modified. This reduces the need for explicit terminal output.

2. Each command window has its own user interaction (input) window.

3. There is a message output stream associated with each session.

4. The output message stream and default user input are supported by a TTY_IO-like package that provides I/O to implicit (but nameable) windows.

5. Implicit terminal I/O is a short-hand for the standard window I/O routines; any appropriately-typed (i.e. stream of character) window can be made the source/target of terminal I/O.

6. The same window can be used for simultaneous input and output as long as the streams are of the same type, though this is not the default. A window can also appear to have been used for simultaneous input and output by echoing input into the output stream as it is accepted, still retaining a clean copy of the input.

7. Starting new execution in the command area establishes the connection between the task executing the command and the appropriate window(s).

8. Terminal I/O calls determine the appropriate output location from dynamic inheritance, i.e. by looking up the dynamic predecessor chain for I/O position(s).

9. A generic package will be provided that determines the dynamically inherited I/O positions once and directly references the streams for efficiency.

10. The default input and output streams can be passed as parameters to any program expecting a stream of characters.

11. User programs can supply values for window size and placement parameters.

6.4.2. Command Execution Commands and Concurrency

1. Execution of commands is initiated in one of several ways:

   a. The user enters the command text in a command window and issues the "execute" command.

   b. The user selects some Ada text somewhere and enters the "execute" command. The Ada is copied to the command window, and executed.

2. The initiation of the command includes several possible options:

   a. Execute the command and move the cursor to the next input prompt from the command.

   b. Execute the command and have the command window wait for its completion.

   c. Execute the command and move to another command window with the same context.

6.4.3. Control of Executing Commands

1. The user can request a list of tasks running on his behalf.

2. Any of these tasks can be aborted.

3. Any of these can be suspended. The task suspends on the next statement boundary.

4. If the task is in a loop such that it never reaches a statement boundary, then it must be aborted to stop it.

5. Once a task is suspended, debugger operations may be applied.

6. It is also possible to apply certain debugger operations while a task is running.

7. The debugger is described in a later section.

6.4.4. Messages and Task Execution

1. Commands that are invoked with certain options, or that wish to, send a message to the user on their completion. This message appears with the normal set of user messages in the message window.

2. Messages are also sent to the user when a task is aborted or breaks unexpectedly.

3. Tasks requiring user attention (e.g., input) may cause messages to be sent to the user.

4. Usually, the task is deleted when it completes, freeing whatever resources were occupied.

5. The user can place a hold on deallocation so as to inspect the task before it is deallocated. This is equivalent to placing a breakpoint at the final end.

6. An interaction window created by a task will not usually be deallocated when the task terminates. The task can explicitly request deletion of the window.

7. The user can request that the window not be deleted regardless of what the task requests.

8. The interaction window is usually retained so that the user can inspect it, save it, or use it for interactions with other programs, forming a longer term interaction transcript.

6.5. Debugging

1. The debugger provides facilities for interrogating elaborated contexts, for placing breakpoints in programs, for monitoring the values of expressions, for gathering performance analysis information.

2. Performance analysis is discussed below.

3. Debugger functions are callable directly from the user program, and callable from an Ada command window.

4. The debugger is not a subsystem that is entered explicitly;

instead, the debugger receives control and informs the user when certain events occur within user programs.

6.5.1. Summary of Capabilities

1. Placement of breakpoints at any statement
2. Arbitrary (large) number of breakpoints
3. Execution of Ada procedure at breakpoint
4. Boolean condition on breakpoint
5. Insertion/deletion of breakpoints
6. Interactive interruption of running task
7. Break on variable modification
8. Break on procedure call/return
9. Break on exception raise
10. Break on rendezvous
11. Break on violation of invariant condition
12. Evaluation of expressions in context of program break
13. Alteration of simple variables and structures.
14. Procedure call in selected context
15. Single/multiple statement stepping
16. Stepping across subprogram calls

17. Stepping at procedure call/return level
18. Stack traceback for dynamic history
19. Examination/alteration of existing invocations
20. Statement execution counts for performance analysis
21. Execution time profile
22. Display of tasks, including name, state, and PC
23. Ability to suspend/resume tasks
24. Ability to examine entry queues

6.5.2. Selecting a Context

1. User must select task instance to be debugged.

2. This may have to be done prior to issuing the command that initiates the task, as the command is the task to be debugged. The debugger must know in which task to enable breakpoints.

6.5.3. Interrogating a Context

1. The context is examined interactively by issuing "put" procedure calls with variables or expressions of interest. Procedures can also be called from the context, allowing more complex display of structures. The user can also define special "debug procedures" to display items of interest.

2. The dynamic sequence of procedure activations that led to the current state of the program can be displayed. The display includes the called subprogram, the point of call, and parameter values. The user can control the size of the display.

6.5.4. Breakpoints

1. Breakpoints are connections between the user program and the debugger.

2. Breakpoints are placed at specific statements in a task instance. When such a statement is to be executed, the executing task is suspended and the debugger notified. The statement at which the breakpoint was placed will not yet have been executed.

3. The debugger executes whatever action was specified to be done at that breakpoint. This may involve leaving the task suspended and notifying the user, or may involve more complex tests and possible resumption of the broken task.

4. The user can write breakpoint procedures that are executed automatically when a breakpoint is encountered.

5. While suspended, the user can interrogate the context of the task.

6. The following commands are processed by the Ada editor that involve breakpoints:

   a. Set_Breakpoint - The user points at a line in a source window. A breakpoint is set at that line in the "selected" task. The "selected" task is the program

being debugged if there is only one task, a default task among a set of tasks being debugged, or a task explicitly selected by the user.

b. Delete_Breakpoint - Breakpoints can be selectively deleted, or all deleted at once.

c. Display_Breakpoints - All breakpoints for a task can be displayed, along with usage statistics.

d. Execute_Procedure_at_Break - The procedure runs in the context of the statement at which the breakpoint is placed. The procedure may execute the call "BREAK" which causes control to go to the debugger with the program in a broken state, or just return which causes the program being debugged to continue.

e. Display_Breakpoints - Changes view of program to show breakpoint code as in-line with the regular code.

f. Install_Breakpoints - Changes selected or all breakpoint code so as to be permanently part of normal code.

7. Once the debugger receives control after a break and any break

procedure is executed, the user can interrogate the context where the task broke or any other context that the user can access.

8. The program can then be resumed either by allowing it to continue from the statement at which the break occurred, or by raising an exception in the program at that point.

6.5.5. Stepping

1. The debugger can execute the program one statement at a time. This can be done either at a global level where the task breaks to the debugger before each statement, or at a local level where only a single procedure is stepped. In the latter case, calls to other subprograms are executed in a single step.

2. The debugger can also step at the procedure call level. Here, the task breaks at the entry and return of each procedure. The user can, for example, examine the in and out parameters at these points.

3. The debugger can step at the task rendezvous level. The debugger takes control and informs the user each time a rendezvous occurs involving a specific task or group of tasks. Only the accepting task can be stepped at the rendezvous level.

4. The debugger can also be set to receive control at the termination of any task or a gf tasks.

6.5.6. Exception Handling

1. There is a facility for the debugger to receive control at the beginning of the exception handler of each program unit. This occurs even if there is no declared exception handler in a unit.

2. The program breaks in the context of the first statement of an exception handler and this is the context that the user can inspect.

3. Individual exceptions can be set to be handled by the user program without debugger intervention.

6.6. Performance Analysis

1. The debugger provides a facility for producing a statement execution count profile of a program's execution.

2. Counts are associated with the source code and an annotated listing can be produced.

3. Procedure call execution counts can also be produced.

4. Rendezvous log.

Chapter 7
Editing Other Types of Objects

1. There are additional editing operations whose availability depends on the object being edited.

2. When the user requests that an object be edited, an object editor is connected to the window in which the object image will be displayed. Object editor-specific key and command maps are established for the window.

7.1. Text

1. The text object editor provides basic text editing functions.

2. Commands are available to select and manipulate words, sentences, and paragraphs.

3. The Text object editor can be operated in one of several modes corresponding to the level of structure in the document. These levels include raw text, business letters, bug reports, other common forms, and Scribe-like letters, articles, and reports.

4. Specific commands are:

    1. Fill_Paragraph
    2. Justify_Paragraph
    3. Check_Spelling
    4. Set_Margins
    5. Set_View - specifies whether format information should be displayed.

7.2. Help

1. At (almost) any point in a session, the user can request help.

2. There are 5 fundamental questions that can be asked:

    1. How do I do ...?
    2. What does ... do?
    3. What does ... mean?
    4. What state am I in?
    5. What was that error all about?

3. The answer(s) to the question appears in a separate help window, as does any additional dialogue required between the user and the help system.

4. Help information may be associated with (almost) any object in the system, including user defined objects.

5. The help information is added by either editing the "help information" view of the object with which the information is to be associated, or by making appropriate subprogram or entry calls to add the information to the help information data base.

6. The maintainer of an object can edit the help information to keep it consistent and up-to-date.

7. Implementation: There is a help object editor that manages the help window and performs whatever interaction with the user is required in answering questions.

7.3. Mail

1. There is a mail object editor that is used to construct and view mail objects.

2. The user requests mail either by issuing a command or by selecting the mail box in the user's directory and requesting to edit it.

3. The user can create mail to be sent either by issuing a command to create new mail, or by issuing a command that constructs a response to some existing piece of mail.

4. When the active window is managed by the mail object editor, several commands are available designed to manipulate mail:

    1. Reply
    2. Send_New_Mail
    3. File_Message
    4. Search_for_Message_in_File
    5. Forward_Message
    6. Return_to_Sender
    7. Edit_Mailing_List
    8. CC

5. It is also possible to select an arbitrary piece of text (from any type of window) and have it included in a message to be mailed.

7.4. Terminal Communication

1. Users can communicate by sending messages to each other's terminals interactively.

2. Users can send either one-way messages that appear in the receiver's message window, or two or more users can engage in interactive conversations.

3. When a conversation is established and accepted by all users involved, each gets a window containing the interaction that is managed by the communications object editor.

4. In the window, the source of each message is identified, and messages from different users are separated.

5. After the interaction, each user has a transcript in the interaction window that can be saved permanently.

6. Within the interaction window, there are some specific commands:

    1. Message_to_User
    2. Connect_to_User
    3. Disconnect_User
    4. Save_Interaction

7.5. Calendar

1. The calendar facility provides a means for scheduling and planning events. It is useful for reservations, and for personal planning and reminders.

2. There are objects of type Calendar and an object editor to display and edit them.

3. Information placed in calendars are known as events and include the following information:

    1. The name of the event
    2. The duration of the event
    3. A description of the event
    4. Whether, to whom, and how long in advance a reminder of the event should be sent.
    5. How persistent the reminder should be.

4. Reminders are typically messages that go to the message window of the user for which they are destined.

5. Commands are available in the window used to display and edit the calendar object:

    1. Add_Item
    2. Search_for_Item
    3. Remove_Class_of_Items
    4. Regularly_Schedule_Item
    5. etc

7.6. PERT

1. PERT is a project scheduling tool. Users can create PERT structures in objects of type PERT and there is an object

editor to display and edit them.

2. The user enters and edits a structure consisting of nodes of activities. Information on the activities includes a name, duration, dependencies, and description.

3. The user explicitly converts this representation into a graphic form by issuing a command.

4. The PERT system supports tracking changes to schedules and revisions to estimates.

5. Specific commands available in a window managed by a PERT object editor include:

    1. Draw_Graphic_Form
    2. Search
    3. Show_Revisions
    4. etc.

Chapter 8
Ada Tools

* Body Builder

* Parser generator

* Scanner generator

* Object Editor generator

* Call Graph generator

* Cross Reference

* File Transfer

* Download

* Remote Debug

Chapter 9
System Access and Control

9.1. Logon/Logoff

* A "session" is an interaction with the system. The session has a state which consists of the executing commands, the windows and their contents, and the identity of the session and the user(s) logged on to it.

* A user logs on to the system by giving a user name and password. The user may then be connected to an existing session, or create a new one.

* The user logs off by issuing an appropriate command. The session is suspended and remains waiting for the user to log on again.

* Sessions can be explicitly deleted by the user.

* If a session is unattached to any user for a moderate time, the system may delete it or at least force some resources used by the session to be released.

9.2. Current Identity and Capabilities

* The access granted to commands executed by tasks that are part of the session is based on the identity of the session.

* The identity is that of all the users currently logged on to the session.

* This is typically only one user, but it is possible for more than one identity to be associated with a session.

* This allows one user to help another and have access to the union of their accessible objects during the period that both their identities are associated with the session.

9.3. Resource Limits and Allocation

* The system administrator assigns resource limits to users.

* Users may divide their resources among sub-packages that they create and designate as accounting entities.

Chapter 10
Peripheral Access

10.1. Printing

10.2. Tape

10.3. Serial Communication

10.4. Direct Disk

10.5. VCR

I. Glossary

Access control The facility whereby access to objects is limited beyond the normal Ada scope rules.

Access list A specification associated with an object indicating the users that may access it and the type of access that they are allowed.

Accessible An object is accessible from a context if it is visible from that context and the user requesting the access has permission based on the access control information applicable to the object.

Accounting entity A static package that has resource limits associated with it. These limits apply to objects listed within and below the package.

Active Window One window on the screen is designated the currently active window. It is the implicit parameter of several commands, and keyboard input is processed based on its keyboard map.

Command A name representing a class of semantically similar operations. Commands are mapped to operations based on the current window and possible other context-related factors.

Command Window Context Part of the command window environment. It is an

elaborated Ada context which is available to Ada statements entered in the command window.

Configuration — (of an Ada program unit) A member of a group of similar programs that share code and declarations.

Context — See "Object window context", "Command window context"

Elaboration — The Ada-defined processing of declarations when a program unit is invoked. Individual units are also elaborated by the environment when installed.

Image — (of an object) The textual representation of an object The process of pla[illegible] Installation package.

Object Window — A window that displays the image of an object for editing purposes.

Object Window Context — The location of the object being displayed and edited in an object window. This might specify a version, task instance, package, procedure, procedure instance, and/or generic instance.

Operation — An Ada, invokable program unit: a procedure or entry, or a statement.

Owner — (of an object) (Usually) the user that created the object. The owner can set the access list for the object.

Prompt — A highlighted token placed in an object image that the user may replace with some user input.

Resource — (as in resource control) One of disk space, CPU time, connect session time, and segment names. These are in limited supply and users are limited in the amount of such resources that can be used.

Root Directory — (of a user) A directory designated by the system administrator and owned by the user. This is the root of a subtree of packages in which that user stores objects.

Session

Static Object — An object declared in a static package, or any program unit.

Static package — A permanent package that is elaborated only once when it is created. Information in static packages is recovered after a crash.

User — An identity known to the system.

Version — (of an object)

View — (of an object)

Window A rectangular terminal-like display and input facility.

II. Ada Definitions

Declared Entity An object that is declared in an Ada program unit somewhere.

Object (3.2) An entity that has (contains) a value of a given type. Objects can be introduced by object declarations, be components of other objects, be formal parameters of subprograms and generic program units, and be designated by values of an access type.

Program Unit

III. Operations

1. Add_Item
2. Add_to_Selection
3. Beginning_of_Object
4. Begin_Selected_Region
5. Bottom_of_Window
6. Capitalize_Token
7. Check_Spelling
8. Connect_to_User
9. Continue_Scrolling
10. Contract_Window
11. Copy_Selected_Region
12. Create_Window
13. Cursor_Move(direction, distance)
14. Decrement_Detail_Level
15. Delete_Line
16. Delete_Next_Character
17. Delete_Previous_Character
18. Delete_Selected_Object
19. Delete_to_End_of_Line
20. Delete_to_Beginning_of_Line
21. Delete_Window
22. Delete_Breakpoint
23. Disconnect_User
24. Display_Breakpoints
25. Draw_Graphic_Form
26. Edit_Mailing_List
27. Edit_Object
28. End_of_Object
29. End_Selected_Region
30. Enlarge_Selection
31. Enlarge_Window
32. Enter_Context
33. Elide
34. Execute_Procedure_at_Break
35. Expand
36. Expand_All
37. File_Message
38. Fill_Paragraph
39. Fix_Context
40. Focus_on_One_Window
41. Forward_Message
42. Freeze_Scrolling
43. Increment_Detail_Level
44. Insert_Character
45. Install_Breakpoints
46. Join_Next_Line
47. Justify_Paragraph
48. Left_Margin
49. Line_to_Top CENTER, BOTTOM
50. Make_Visible

51. Mark_Position(_With_Label)
52. Message_to_User
53. Move_to_Last_Active_Window
54. Move_to_Window
55. Move_Selected_Region
56. Move_Window
57. Next_Page
58. Next_Prompt
59. Next_Word
60. Next_Object
61. Next_Token
62. Pop_Context
63. Previous_Object
64. Previous_Page
65. Previous_Prompt
66. Previous_Token
67. Previous_Window
68. Previous_Word
69. Query_Search_and_Replace
70. Regularly_Schedule_Item
71. Remove_Class_of_Items
72. Remove_from_Screen
73. Replace_Next_Character
74. Return_to_Mark (name)
75. Return_to_Sender
76. Right_Margin
77. Save_Interaction
78. Scroll
79. Search
80. Search_and_Replace
81. Search_for_Item
82. Search_for_Message_in_File
83. Search_for_String
84. Select_Object
85. Select_Line
86. Send_New_Mail
87. Set_Breakpoint
88. Set_Context
89. Set_Detail_Level (local_global)
90. Set_Margins
91. Set_View
92. Show_Revisions
93. Sync_Context
94. Top_of_Window
95. Transpose_Characters
96. Unselect
97. Unselect_All
98. Window_Directory