```
 SSSS   PPPP    EEEEE   CCCC   III   FFFFF   III    CCCC   AAA   TTTTT   III
S       P   P   E      C        I    F        I    C      A   A    T      I
S       P   P   E      C        I    F        I    C      A   A    T      I
 SSS    PPPP    EEEE   C        I    FFFF     I    C      A   A    T      I
    S   P       E      C        I    F        I    C      AAAAA    T      I
    S   P       E      C        I    F        I    C      A   A    T      I
SSSS    P       EEEEE   CCCC   III   F       III    CCCC  A   A    T     III


DDDD    000    CCCC           1
D   D  0   0  C              11
D   D  0   0  C               1
D   D  0   0  C               1
D   D  0   0  C               1
D   D  0   0  C       ..      1
DDDD    000    CCCC   ..     111


*START* Job SPECIF Req #169 for EGB    Date 28-Sep-82 22:07:43 Monitor: Rational
File RM:<RPE.DOC>SPECIFICATION.DOC.1, created: 29-Jul-82 12:12:57
        printed: 28-Sep-82 22:07:44
Job parameters: Request created:28-Sep-82 22:06:03   Page limit:72   Forms:NORMA
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:
```

1. Introduction

This document contains an outline of the Rational Machines Programming Environment. It is intended as a development aid to help locate unresolved details and as an overview of the structure of the environment.

2. System Structure

2.1. Packages of objects

The system consists of a number of objects. Each object has a type and can be manipulated in certain ways. For the most part, objects behave as objects in an Ada program.

Objects are grouped into packages which are themselves objects. Given a context of a specific package, the names of objects that are visible are defined by Ada scope rules.

2.2. Prototypical structure

The prototype organization of packages on the system is outlined on <rpe.source>system.draft

2.3. Open Items

* Are users in projects? Are projects in users?

* What default structure do we provide to customers?

3. User Interface

3.1. Windows

The user interacts with the system by manipulating objects from a user terminal. The terminal must be a display terminal. The screen is divided into a number of rectangular regions called windows. Windows can be independently manipulated and each window has a context. Thus, one window might be used to edit an Ada package and another might be used to interact with the mail system.

3.2. Screen Management

The user has control over windows on the screen. The following operation can be applied to windows (and, indirectly, the screen):

* Expand: Increase the size of the window
* Shrink: Decrease the size of the window
* Delete: Delete the window
* Move: Move the window to another screen location
* Top: Make the window fully visible
* Scroll: Change the area of the displayed object
* Select: Choose a window to interact with

In addition, depending on the object(s) being displayed in the window and the context of the display, many other operations are available. Many of these are described below.

The user interacts with the system via the editor. There is no separate command interpreter or command language subsystem.

3.3. Open Items

* What are the specific key sequences to control screen management?

* Can one window request transparent mode, thereby disabling the normal keys to move to other windows?

4. Access and Protection

Summary:

* Password or other verification at log on.
* Session state saved between log off and next log on.
* User may change context during a session provided access capabilities permit.
* Capabilities may be increased during a session by providing additional passwords or other verification information.
* Capabilities may be decreased during a session by giving a command.
* During a session, the object names given in a particular context are resolved using Ada visibility and protection rules.
* Users must write abstractions (or use a supplied generic package) to achieve additional protection or more selective access.
* The system provides functions that give unique, unforgeable identification and password checking.

4.1. Log on

In order to use the system, the user must identify him or herself to the system. This process is called logging on. Identity is authenticated by use of a password or other user-defined mechanism.

Default information associated with the user includes the following:

* Last State: the state of the user session when the user last logged off. (This is null if the user has never logged on before, or if the state has been discarded by the system for some reason.)

* Context: The package in which the user will be placed after log on is complete.

* Access: A list of all packages to which the user may legally change context.

4.2. Current context & visibility

After log-on, a window is created with the context of the user's default package environment. Names in commands issued there are resolved in that context. Normal qualification can be used to reference objects in packaged not directly visible, but it is not possible to access an object out of the scope of the particular user package.

4.3. Changing context

The context may be changed by issuing some command. The change is restricted based on the capabilities of the user issuing the change. The context of a window can be changed or a new window created with a different context. There is no Ada analogue for changing the scope of an executing package. Thus, this change operation can be effected at the command level only.

4.4. Open Items

* What is the specific mechanism for establishing a window in a certain context?

* How are passwords stored? (encryption)

* Architectural facilities for encryption or other increased security material?

* Do we REALLY believe that Ada visibility rules are ok for the user environment?

* What other protection types or mechanisms should be provided. (If we say everything is in Ada, then what Ada abstractions or models should we provide?)

5. Online Assistance

Summary:

* Help Facility: Short topics and answers
* Interactive Tutorial: Lessons in using the system

5.1. General Help Facility

The user can request information on a variety of topics including, but not limited to, each of the commands and operations, and procedures. This information is a short description of the topic and a reference to additional information (not all of which may be on-line).

The help information will be kept in a data base that will grow as questions are raised by users. In this way, the help will be more informative than a conventional simple description of each command.

It is possible for the user to add information to the data base both to customize system information for local users and to describe locally produced facilities.

5.2. Self Teach

The self-teaching facility is an interactive dialogue that tutors the user in the use of many of the system functions. This tutorial is intended for a user who has some, but minimal, background in the concepts of the Rational system.

Help information will appear both in the user window or on separate dedicated "help windows". For example, a form of help is to prompt the user with a template containing named parameter notations. Other help requests may point the user to more complete information that can be perused on a separate text window.

6. Basic Editing

6.1. Selecting an object to edit

An editing session begins with the selection by the user of an object to edit. The following steps then occur:

* The system determines if the object is editable. If not, then an error message is displayed.

* An editing window is set up and connected to the object editor for the type of object that was to be edited. If there is no object editor for that type of object, it is not editable and an error message is displayed.

* Some part of the object is displayed in the window and the user begins the interactive edit session.

6.2. Basic editor operations

Editing is accomplished by creating a window that displays the object that is to be edited. The user can then alter the object through a series of editing commands.

The editing process is handled by two system components: the editor, and the object editor for the object that was selected for editing. The editor handles the basic user interface and text and screen manipulation and the object editor handles all object-knowledge based operations. Thus, the object editor is responsible for deciding the storage and display formats of the object, and the legality of specific editing operations.

The editing window is broken into at lease two "panes" or sub-windows. The display pane is used to display the object. A cursor in the display pane shows the part of the object selected for alteration. The result of editing operations are displayed immediately in the display pane.

Editing commands are either single keystrokes which cause specific editing operations to be invoked, or Ada procedure calls to editor or object editor procedures.

The command pane is a part of the window used to enter, edit, and issue procedure-call editing commands. The commands themselves may be edited in this pane. The Ada object editor is "connected" to this area of the window so that the Ada commands can be entered and manipulated with the full power of the Ada program editor.

There are a number of editing operations that are common to all editing windows. These are summarized below:

* Insert: New text (or structured text) is inserted into the object.

* View Control: Control the part and amount of the object that is displayed.

* Select: A part of the object is selected; then, one of the following actions can be performed on the selected part of the object.

* Copy: The selected part of the object is duplicated and the copy inserted at a specified location (which may be in another window).

* Move: Similar to copy, but the selected part of the object is removed from its original location.

* Delete: The selected part of the object is deleted.

Information can be moved from one window to another. This mechanism is used to copy objects and to convert the storage format of objects. If a fragment of Ada program is moved from an Ada editing window to a text editing window, the program code is considered text in the new environment and is stored as text. It may not be possible to paste arbitrary text in an arbitrary window. The object editor for objects in the window must decide the legality of any insertion.

6.3. Open Items

* What is the specific user interface for commands?

7. Ada

7.1. Editing

The editor supports syntax-directed editing of Ada programs. The edit provides structure completion, selections based on Ada objects, and other language-oriented features.

(Need some more details here.)

Searches for structures and text patterns is done by the editor. The result of the searches can form a collection that can be operated on in various ways. (For example: change all occurrences of "A+1" to "A+2", or form a list of all procedures containing a call to procedure P.)

7.2. Execution

Executable Ada entities are procedures, functions, or entries. Procedures and functions can exist by themselves or can be contained within packages. Entries are always contained within tasks. Users can declare procedures, packages, and tasks. (For the remainder of this discussion, "procedures" referes to "procedures" or "functions".)

To execute a procedure, the user issues a command to call the procedure. The procedure is then elaborated, creating any enclosed packages or tasks, and then executed. When the procedure terminates, any enclosed created objects are deallocated. Thus, the procedure is the best way to package an executable entity. Except for objects accessed that are outside the procedure, there are no exclusion problems if the procedure is executed concurrently by different tasks.

In order to execute a procedure contained in a package, the package must first be instantiated. We don't know exactly the procedure for doing this as the lifetime of the elaborated package must be defined. At some point, the user will wish to edit the package. It must be either deallocated at this point, or modified as the user makes changes. The later can be accomplished through re-elaboration, but this is not necessarily what the user wishes.

We may provide an explicit command to kill the instantiation.

Ada programs are invoked by issuing a procedure call to the procedure, or an entry call to the task. The program then executes. The window from which the program is invoked is 'tied up' until the program terminates, although interactive I/O with the program may occur in the window. The user can move to another window and perform other tasks which the program is executing.

While the program is executing, it may be interrupted or aborted by giving a command either in the window where the program was invoked, or in the task status window. Some programs may not be interruptable or abortable. This property depends on the program itself (some may be designed so as to be uninterruptable) and the "owner" of the program. (The debugger may not be invoked after an interruption if the user is not the "owner" of the program.)

7.3. Debugging

Summary:

* Variable examine/modify in any activation.
* Interactive execution pause/continue.
* Breakpoints.
* Variable trace.
* Statement execution trace.
* Execution time profile/statement execution counts.

The debugger provides a set of operations that manipulate running programs. These operations are available when a program is interrupted or may be requested prior to a program's execution.

The simplest debugging facility allows the user to interactively interrupt and continue a task's execution. When interrupted, the debugger will display the area of the program that is executing and allow the user to access and alter values of variables instantiated at that point.

The breakpoint facility provides a means for a user to select points in the program at which the debugger is invoked. The user can then perform examinations as described above. Placement of breakpoints involves making a new version of the program that only that user will use. Thus, if another user executed the same program, the breakpoints would not be encountered.

The tracing and performance monitor facility similarly involves alteration of the program and creation of a new version. In this way, statements to accomplish the trace task are placed in a new version of the program.

An alternative to the modification of the program source is to support some of these functions in the architecture.

It is not clear how time-related information can be gathered. Perhaps another task can be given access to the program counter of the task to be examined, thereby periodically sampling its state to produce an execution profile.

The machine will provide information about cpu and memory usage. The user may intrrogate these parameters for all of his tasks

When the system makes source transformations for debugging the changes in the source are not visible to the programmer. A function or porcedure may be marked as being "traced" (a comment); special command insert and remove tracing.

7.4. Configuration control

Summary:

* Multiple revisions accessed by name and revision qualifier.
* Linear derivation operation to create new revision.
* Split operation to create parallel development derivation strings.
* Merge operation to bring parallel revisions back together.
* Parallel derivation operation to make a change to multiple revisions.
* History/change summary.
* Ability to execute or edit multiple revisions of the same program.

The configuration control facility is designed to support the development and maintenance of programs. Both simple linear development where revisions only of the most recent revision are created and more complex tree and graph development where several revision strings exist are supported.

Revisions are created as follows. A user decides that a new revision is desired. The base revision from which the new one will be created is identified. Then, a series of edit sessions is performed to create the new version. At some point, that revision is "released". This means that it can no longer be modified. Further changes require the creation of a new revision.

New revisions of a program may be created based on any existing revision. In this way, arbitrary tree structures can be created. It is possible for users to create "parallel" revisions based on the same revision. As this may mean that a merger of the changes will be required later, the system provides warnings whenever parallel revisions are created.

The configuration control system provides automated assistance to the user when revisions are to be merged. This works as follows. The dominant revision is identified by the user. This is revision that will serve as the base case into which other revisions will be merged.

The system then locates differences between the revisions being merged, displays similar contexts of both indicating the difference, and provides the user the option of including the change, ignoring the change, marking the change to be examined later, or selecting one of the above options followed by additional editing. A log of the decisions made in the merge is kept for the user to review.

A parallel derivation function is also provided. This allows the user to make a change simultaneously to several versions, producing new versions of each one. This would be used, for example, when a bug is to be corrected that exists in several versions of a program.

Parallel derivation works similarly to the merge. The user chooses a dominant version to modify. For each change made to the dominant revision, corresponding locations in each of the other revisions are displayed and the user is given the options similar to those of the merge. If there is no corresponding location in a revision, this fact is noted and the user is allowed to skip the change, or choose another place to make it. Again, a log is kept of all changes made (or not made) to all affected revisions.

7.5. Open Items - Ada

* Implementation of "deltas" to store differences between revisions.

* How are logs and other information of that sort stored and associated with revisions?

* User interface for editing, debugging, and configuration operations.

* How does support for other processors fit in? (This is major open item as we are selling a software development system.)

8. Program Control

8.1. Task Status

A window can be created that displays the status of a set of tasks that are related to the user or a particular subprogram invocation. The state, priority, and resource utilization of the tasks are displayed.

8.2. Task Control

Editing the task status display causes changes in the status and operation of tasks. The following operations can be performed on tasks:

* Change the priority of a task.

* Change a resource limit for a task. (What resource limits?)

* Abort a task (and its children).

* Delete a task. (This is done only by deleting its declaration. When and how we actually get rid of it are open issues.)

8.3. Open Items

* How do you refer to anonymous tasks (eg, allocated in a collection)?

* How do you know when to reclaim the task? (ie, no more references to it)

* What architecture support is needed (if any) to allow a program to locate and keep track of tasks?

9. Object Management

Summary:

* Dangling references avoided by retaining phantom 'old' version.
* Instantiated objects remain consistent at all times.

9.1. Changing Instantiated Objects

Instantiated objects are changed through use of the re-elaboration mechanism. This involves getting a package to accept a special entry call that causes some new code to be executed as part of the package. The new code may add new objects to the package, or delete or change existing objects. The program segment for the package may also be replaced or altered.

These changes may affect the consistency of related modules. A change of a type definition would affect declared objects of that type. Renaming or deleting a named object would affect any module that referred to the object. Introduction of a new object may create name collisions where none existed before.

Because we are dealing with instantiated environments, these consistency problems must be dealt with without loss of existing information.

9.2. Versions of User Environments

When a change is made to an instantiated package creating a consistency problem with another instantiated package, a new "version" of the altered package is created automatically. References in the "source" that represents an instantiated package is automatically changed to refer to the older version of the altered package. In this way, active references to altered types of objects will maintain their state and operate as though the change was not made.

This implies that, when an instantiated package is changed, only new objects can be created, and the old one must be maintained as long as there are references to it in other instantiated packages. Once there are no more references, the old object can be deleted.

A change to a module containing an un-instantiated package does not require a change to the package. Consistency must be established at package instantiation time, however. The normal Ada compilation rules require that all modules contained in one that is changed be recompiled anyway.

9.3. Open Items

* How do you determine when there are no references to an old object so that it may be deleted?

* How do you re-elaborate tasks?

* What implementation mechanism can be used to update the referencing packages when consistency-maintaining version changes are necessary?

* What is the user interface to say when to instantiate or un-instantiate a package? (This is necessary or else all package would have instantiations.)

* Do we recompile all of a user's programs whenever a "file" or similar object is created or deleted?

10. Archiving

10.1. Summary

* Objects can be symbolically saved and reloaded.
* Symbolic dump for transportation between systems.
* Bulk/incremental dump for system backup/recovery.

10.2. Saving and reloading selected objects

Individual objects or groups of objects can be saved and reloaded. The information saved includes the object's name, type and "value" where the value is converted to some externally representation that can be placed back into the object when it is reloaded.

The type can be saved either symbolically or structurally. If saved symbolically, there must be an equivalently named and structurally compatible type defined in the context in which the object is reloaded. If saved structurally, the symbolic definition of the type is saved with the object. This may create compatibility problems when the object is reloaded, but guarantees that the object can always be reloaded regardless of context.

Normally, objects are saved only in the "external" form. In some cases it is desirable to save the object in an "internal" form. This involves saving the actual memory image of the segments that represent the object. This is useful, for example, in releasing an execute-only version of a program without the source.

When segments are saved in this way, an environment definition is saved with them that supplies the information necessary to "reconnect" the external segment linkages when the object is reloaded. If the reload context is not sufficiently compatible to re-establish the linkages, the object may not be able to be loaded.

10.3. Bulk backup

It is also possible to make bulk backups of the system. This is useful primarily for crash recovery and major archiving. It is not generally possible to reload individual objects from a bulk backup.

The backup can be reloaded to create a "fresh" system.

Bulk backups can be done incrementally so that only segments changed since the last backup need be saved.

10.4. Open Items

* Is it acceptable not to be able to selectively load objects from a bulk backup?

* What information is required to splice a saved segment back into a different system?

* What architecture support is required to splice type links and imports for loading and dumping objects?

11. Documents

Text files are supported at least to the extent of producing documentation for software developmed on the system. Some additional support for general text processing is provided.

11.1. Editing

There is a facility for raw text editing. This facility allows the entry, editing, and display of sequences of ASCII characters. Arbitrary manipulations are allowed.

11.2. Structured Documents

Some support is provided for structured documents. This facility supports the easy creation of structured documents such as letters, memos, reports, manuals, and so on. The editor has knowledge of document structure concepts such as paragraphs, pages, chapters, headers, footnotes, etc.

11.3. Open Items

* What specific features are to be supported?

12. Communicating with Other Users

Summary:

* Terminal-terminal messages.
* Session monitoring/assistance between terminals.
* Mail system.

12.1. Mail System

We will initially provide a basic mail system. Ideally, it would be implemented as an object editor for the editor. This provides an edit-like user interface similar to those of other functions.

The mail system allows the entry, editing, and sending of messages to one or more recipients, and the logging of all outgoing mail.

A directory of incoming messages is kept and displayed for the user as a package containing objects of type message. Individual messages can then be examined, replied to, forwarded, moved to another package, or deleted.

12.2. Terminal Communication

The terminal communication facility allows windows on different terminals to be linked so that both display the same thing. This lets one user monitor transactions that occur in another user's window. It also provides a mechanism for two users to communicate by each monitoring a window that the other is typing into.

The communication mechanism occupies only a single window on the user terminal; other activities can be carried out concurrently in other windows.

12.3. Open Items

* If we make it look like messages are objects in a package, will they really be that? (ie, programs could access them)

* Implementation: Make an abstract type "Message" and define operations on it.

* Should there be a way to have one terminal screen monitor another complete screen?

* What are the restrictions on user terminal communications? How does a user refuse communications?

* How does a user find out that another wants to communicate? (Is there a status window of some sort?)

13. Peripheral control

There are facilities that allow users to manipulate peripherals in various ways. These facilities are provides by calls to visible packages or tasks. Some packages direct access to peripherals; others provide spooled access.

13.1. Printer

The printer is normally accessed via a spooler. Objects to be printed are copied in text form by the spooler. The spooler may understand several types of objects such as program, structured text, and unstructured text. Others may have to be converted to text by system or user supplied conversion functions.

The customer can specify several configuration options to the printer spooler including banner pages/formats, pagenation, carriage control conventions, etc.

Users can inquire about the status of requests to print objects. They can also cancel print requests or (if they have appropriate capabilities) change the order in which objects are printed.

13.2. Tape

Tape is normally accessed directly. There is a package that provides raw tape manipulation functions. The raw access can be combined with other packages to provide storage and retrieval of structured objects.

Some basic functions to "flatten" an object into a binary string, and "raise" it, recreating it from such a string are available for system defined objects. Users can also implement such functions for their own objects.

13.3. Terminals

Terminal access is normally performed by the system. Most terminals communicate to editor tasks that provide the user interface for the system.

It is also possible to designate a terminal line as "raw", allowing use of the line for character-oriented serial communications. This is done by setting the terminal configuration package which defines the way in which terminal lines are controlled. A package is available that provides direct I/O functions on the terminal line.

13.4. Open Items

* How do you go from object to bit string and back?

* IOP implications of user defined terminal handlers?

Other devices to support?

14. Utilities

14.1. Generic file package

(Wolf has this somewhere.)

14.2. Open Items

* What other packages should be provided?

* To what extent should we encourage the use of files, if at all?

Table of Contents