Technical Report
CMU/SEI-88-TR-021
ESD-TR-88-022

Experiment Transcripts for the Evaluation of the Rational Environment

Grace Downey
Mitchell Bassman, Computer Sciences Corporation
Carl Dahlke, Computer Sciences Corporation

September 1988

Technical Report

CMU/SEI-88-TR-021
ESD-TR-88-022
September 1988

Experiment Transcripts for the Evaluation of the Rational Environment

Grace Downey

Evaluation of Environments Project

Mitchell Bassman
Carl Dahlke

Computer Sciences Corporation

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl Shingler
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Experiment Transcripts for the Evaluation of the Rational Environment

Abstract: This report supplements the SEI report *Evaluation of the Rational* Environment (CMU/SEI-88-TR-15) by Peter Feiler, Susan Dart, and Grace Downey. It contains the instantiation of the experiments presented in the *Evaluation of Ada Environments* by Nelson Weiderman, et al. (see [7]). Overall conclusions and analysis of the Rational Environment can be found in *Evaluation of the Rational Environment*.

1. Introduction

1.1. Scope

The evaluation of the Rational Environment and R1000 computer was initially conducted by Computer Sciences Corporation (CSC) using the Gamma Release of the Rational Environment. Authors of an internal CSC report released in January of 1987 were Mitchell J. Bassman and Carl Dahlke (see [3]). The instantiation of the experiments on the Gamma Release of the Rational Environment were written and performed by Carl Dahlke, Diane E. Odiorne, and Katherine E. Stanton at CSC's Star* Lab. This report contains the results of repeating the experiments on the Delta Release of the Rational Environment at the Software Engineering Institute (SEI). The Delta Release of the Rational Environment provides the initial release of configuration management and version control tools. This report contains the results of instantiating the Configuration Management/Version Control Experiments from the *Evaluation of Ada Environments*. Grace Downey repeated the experiments according to the CSC transcripts, adjusted the transcripts for the Delta Release, and instantiated and performed the Configuration Management/Version Control Experiments. Analysis of the Rational Environment and analysis of the experiment results are provided by Peter Feiler, Susan Dart and Grace Downey in *Evaluation of the Rational* Environment (SEI-88-TR-15).

1.2. Evaluation Experiments Performed

Of the six categories of experiments presented in *the Evaluation Method,* five were performed on the Delta Release of the Rational Environment. A brief description of each experiment follows.

1.2.1. Configuration Management/Version Control Experiments

This group of experiments provides an evaluation of an environment's version control capabilities (i.e., support of successive versions, variant versions, file checkin/checkout, etc.) as well as its configuration management capabilities (i.e., support of system construction and reconstruction, baselining, release management, history collection, etc.). The first of three experiments simulates the system integration and testing phase of the software development cycle. The second experiment assumes the first has been conducted and investigates system construction, reconstruction of previously baselined systems, and a combination of elements from old and new

systems. Chapter 2 contains the instantiation of the first two experiments. The third experiment investigates software management activities, including release control and release history. Rational expects the user to tailor commands from work order, configuration management, and version control commands to enforce a software management policy. The third experiment was not performed because it does not specify a software management policy nor does the Rational Environment dictate one.

1.2.2. System Management Experiments

Four aspects of system management are separately evaluated: Ada Programming Support Environment (APSE) installation, user account management, environment maintenance and support for machine usage accounting.

Rational technicians perform all APSE installation and updating, and the procedures used are not described in the documentation for system managers; therefore, the APSE installation experiment was not performed.

User account management is largely concerned with controlling access to user accounts. The Gamma Release of the Rational Environment did not provide any access control capabilities except for requiring an operator password to execute certain environment commands. The Delta Release, however, does provide access control functionality. Therefore, the experiment was re-instantiated and includes the steps which exercise the access control facilities.

The environment maintenance experiment consists of a series of questions, which were answered.

Little of the Environment Maintenance and Support for the Machine Usage Accounting Experiment had to be instantiated for the Rational Environment, as most of the functionality requested already exists in commands provided by the Environment. The experiment is presented through descriptions of how to access and use the environment-supplied accounting information.

1.2.3. Design and Development Experiment

The Rational Environment provides most of the capabilities evaluated by the design and development experiment category. Some steps of the experiment were omitted because the Rational Environment does not supply any graphical design interface tools. The experiment consists of developing a small system consisting of several Ada units. Creating procedures and packages, compiling, and changing the procedures and packages constitute many of the steps of the experiment.

1.2.4. Unit Testing and Debugging Experiment

The Rational Environment provides most capabilities evaluated by the Unit Test and Debug Experiment category. Again, some steps of the experiment were omitted because the Rational Environment provides no supported static or dynamic analysis tools. The experiment consists of testing a small system of Ada units, debugging and changing the units, and regression testing.

1.2.5. The Project Management Experiment

The Project Management Experiment (see [5]), by Peter H. Feiler and Roger Smeaton, evaluates an environment's ability to support four categories of project management: project plan management, plan instantiation, project execution, and product management. The level of support that the Rational Environment provides in these areas is tested in the Configuration Management/Version Control Experiments; hence, this experiment is not instantiated for the Rational Environment.

1.2.6. Prototype Ada Compiler Evaluation Capability (ACEC)

The prototype Ada Compiler Evaluation Capability (ACEC) test suite was run on the Rational Environment. The Institute for Defense Analysis (IDA) developed the test suite from public domain software, and the suite and its use is described in IDA PAPER P-1879, *User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1*, by Audrey A. Hook, Gregory A. Riccardi, Michael Vilot, and Stephen Welke (see [1]). Numeric results from running the ACEC suite, problems with the suite detected by the Rational Environment, and some tailoring required for the Rational Environment are reported.

1.2.7. Appendices

The appendices to this document consist of code developed specifically for the Rational Environment to perform measurements for some of the experiments. The Prototype ACEC suite requires that some environment-specific code be developed; this is presented as well.

1.3. Environment Version and Hardware Evaluated

The evaluation of the Rational Environment was performed with the following hardware configuration and software configuration. The hardware configuration is a R1000 Model 200-20 with the following components:

- 32 Mb of primary memory.
- Approximately 2,010 Mb of unformatted disk storage (3 disks with approximately 670 Mb capacity each).
- Tape drive PE/GCR 75 ips streaming tape.
- Ethernet connection.
- 8 Rational terminals connected to the R1000 over the Ethernet via a DECserver.

The software configuration is release *D_9_25_1* or *Delta0* of the Rational Environment. The environment evaluated was the base environment, which comes as one package and includes the following:

- Basic operating system functionality, such as file and directory system, process management, access control, etc.
- A tiled window system for character terminals.
- An Ada command processor.
- An editor and browser sensitive to Ada syntax and semantics.

- An incremental Ada compilation system.
- A debugging system with extensive coverage of the Ada language.
- Programming-in-the-large support in the form of the subsystem concept.
- Configuration management and version control support.
- Workorder management support.

Unsupported tools contributed by users include a reusable component library, metric collection tools, and browsing tools. These packages were not included in the evaluated environment because they either are unsupported tools or they were not available to the evaluators at the time of evaluation.

1.4. Report Structure

Chapters 2 through 6 each represent one experiment category of the SEI evaluation method as it was performed on the Rational Environment.

- Chapter 2 - Configuration Management/Version Control Experiments
- Chapter 3 - System Management Experiments
- Chapter 4 - Design and Development Experiments
- Chapter 5 - Unit Testing and Debugging Experiments
- Chapter 6 - Prototype ACEC

An introduction in each chapter outlines the experiment and explains the notation used in the experiment transcripts. Some categories contain multiple experiments. The experiment transcripts are then presented, followed by a checklist summarizing the functionality provided or not provided by the Rational Environment. Questions associated with each of the experiments are answered. A brief conclusion to each experiment discusses the Environment's capabilities in terms of functionality, performance, user interface, and system interface. Any problems encountered in instantiating the experiment are discussed in the functionality subsections of these conclusions.

Previous SEI environment evaluations included command files that instantiate the experiments. Because the Rational Environment is intended for use as an interactive environment, each experiment step is listed, followed by a "script" of Rational commands necessary to perform the step. The transcripts of the Configuration Management/Version Control Experiments and System Management Experiments explore the Environment commands more closely by detailing the commands' program interface. The transcripts of the Design and Development and Unit Testing and Debugging Experiments are more terse because many of the Environment commands are bound to program function keys. In order to provide a better flavor of how the Environment can be used interactively, actual keystrokes are shown, rather than the program interface bound to the keys.

The final chapter contains a cross-environment comparison of the Rational Environment and previously evaluated Environments in terms of performance figures.

2. Configuration Management/Version Control Experiments

The Configuration Management/Version Control (CM/VC) Experiments exercise the version control capabilities of the Environment. The first experiment simulates the system integration and testing phase of the life cycle by having three separate development lines of descent from a single baselined system. The second experiment investigates the Environment's support for activities such as system construction, reconstruction of previously generated baselined systems, and the construction of a mixture of new and old systems.

The third experiment in the area of configuration management investigates the activities involved in the management of a software product, including release control and release history. Although the Rational Environment provides the capability to track release control and release history through the work order management package, this package represents a primitive set of commands. Rational expects the user to construct a series of "skins" which invoke the work order commands, and to some extent the configuration management commands to enforce company policy. No "company policy" is presented in the third experiment, and for this reason the experiment is not instantiated. For a discussion of the work order management package, see *Evaluation of the Rational Environment* (see [4]).

The system described by the experiment is broken into three subsystems: VT, CLI, and SM. The subsystem boundary described by the experiment could have been ignored, and all units placed in one large subsystem. The functionality tested by the experiment would have been easily covered by the CM/VC facilities in the Environment. To more fully exercise the CM/VC package, the three experiment subsystems were instantiated as Rational Environment subsystems. A main program that depends on Ada units in each of the three systems is also provided, and acts as a subsystem driver through the course of the first experiment. To make use of more features provided by configuration management and version control packages in the Delta Release, the main program was placed in a fourth subsystem, MAIN.

Yet a fifth subsystem, AS, was required to instantiate the experiment due to a bug in the Cmvc.Release command. A package specification in SM depends upon two packages contained in the subsystem CLI. A separate subunit in CLI depends upon three packages in SM. According to the description of the CMVC.Initial command in Rational Environment Reference Manual 11, *Project Management*, if the parameter Subsystem_Type is set to Cmvc.Combined when subsystems are created, then circular import relations may hold between the subsystems. In an initial performance of the experiment, SM and CLI were created as combined views, SM was set to import CLI, and CLI was set to import SM. When the Cmvc.Make_Release command was issued to release the subsystems, the execution of the command never finished. The subsystem AS breaks the import circularity and consists of package Aim_Support and package specification and body for String_Utilities, which were originally assigned to subsystem CLI by the experiment description.

2.1. Introduction

The following two sections describe the instantiation of the two configuration management experiments. In each section the numbered experiment step is listed, followed by an overview of the Rational commands shown in braces ({ }), and finally a detailed list of the Rational commands. The experiment was conducted in a world created within the home world of a Rational user. The context for the experiment is presented as **!User.***experimenter*. If repeating the experiment is desired, a user may substitute a home world name where *experimenter* appears in the experiment transcripts.

The transcript indicates specific key commands by showing the key's label enclosed in angle brackets (e.g., <Promot>). The first time a command is used, all of its parameters and their values are detailed. Parameter values that must be supplied or that vary from the default are written in italics. Any subsequent uses of the command are listed with only required parameters or parameter settings that vary from the Rational Environment default settings written in italics.

The following paragraphs are examples of how to change current context, issue commands, and select an object. They are provided here once, so that the detailed list of Rational commands need not be an exact transcript of keystrokes.

The following sequence of key commands may be used to change to an example context directory *!users.experimenter.cm_experiment.cli*:

```
<Prompt For>
<Definition> -- A command window opens
             -- containing:
Common.Definition (Name => "<CURSOR>",
                   In_Place => False,
                   Visible => True);
```

The cursor is automatically placed at the parameter value for Name, enter the value for Name, by typing:

```
!users.experimenter.cm_experiment.cli
<Promot>
```

The cursor moves to the window least recently used, and a listing of the directory's contents *!users.experimenter.cm_experiment.cli* appears in the window.

Another method of changing contexts is to use the <Definition>, <Enclosing>, and cursor movement or arrow keys. For example, if the cursor is already in a window whose context is *!users.experimenter* to make *!users.experimenter.cm_experiment.cli* the current context:

```
<down arrow>     -- Press until the cursor is
                 -- on the line cm_experiment;
<Definition>     -- the cursor will move to a window
                 -- which will show the
                 -- !users.experimenter.cm_experiment
                 -- context.
<down arrow>     -- Press until the cursor is
                 -- on the line cli.
<Definition>
```

The cursor moves to a window listing the contents of the directory *!users.experimenter.cm_experiment.cli*. To move to the parent of the current context, press the key <Enclosing>. By combining the use of the <Enclosing>, cursor movement, and <Definition> keys, any context can be reached by traversing the directory tree structure.

Once the cursor is positioned in the correct context, the steps to issue and execute a command *Command_Name* are as follows:

```
<Create Command>  -- A command window opens.
Command_Name      -- Type in the command, or
                  -- a unique prefix of the
                  -- command.
<Complt>          -- The command will be
                  -- displayed with all
                  -- parameters and their
                  -- defaults.
```

The cursor is automatically placed at the value for the first parameter. To change the parameter, type the desired value. To change other parameters, move the cursor to the parameter value either with the arrow keys, or the <Next Item> key, and type the desired value. To start execution of the command, press the <Promot> key. A command has executed without error if no error message appears in the system message window at the top of the Rational screen, or if in the output window created by the command, no asterisks (*) appear in the left column.

To select an object, make the context containing the object the current context. Place the cursor on the line containing the object, and press the <Object> key, followed by the <Left Arrow> key; The object is then displayed in brighter characters to indicate that it is selected. Some commands have a default parameter setting of <SELECTION>; the environment resolves <SELECTION> to be an object that is selected in the window to which the command window is attached. It is often faster to select object, and allow <SELECTION> to resolve to that object.

2.2. Experiment #1

1. Experiment setup

   a. Create subdirectory in which the experiment will be performed.

      {The subsystems MAIN, CLI, SM, VT, and AS will all be subdirectories of Rational World "!Users.*experimenter*.Cm_Experiment".}

      ```
      {Make !Users.experimenter the current
      context}
      <Create_World>
      Library.Create_World (Name => "cm_experiment",
                            Kind => Library.World,
                            Vol => Library.Nil,
                            Model =>
                            "!Model.R1000_Portable",
                            Response => "<PROFILE>");
      ```

   b. Establish environment variables to be used in the experiment.

      {None are used; the logical names assumed by the experiment will be used as subsystem names.}

c. Develop a command named **record** to collect data file size measurement.

{See Appendix A.}

d. Develop a command named **timeit** to collect execution time measurements for any environment command.

{See Appendix A.}

2. Establish Ada program library structure for system integration.

{Use the Cmvc.Initial command to build an initial spec/load working view for the subsystems VT, AS, SM, CLI, and MAIN.}

```
{Make the Cm_Experiment world the current context.}
Cmvc.Initial(Subsystem => "VT",
             Working_View_Base_Name => "Rev1",
             Subsystem_Type => Cmvc.Spec_Load,
             View_To_Import => "",
             Model => "R1000_Portable",
             Comments => "",
             Work_Order => "<DEFAULT>",
             Volume => 0,
             Response => "<PROFILE>");
```

{Repeat the Cmvc.Initial command as above, while CM_Experiment is the current context for each of the other subsystems: SM, CLI, AS, and MAIN. Allow the default, "Rev1," to be the working view base name for all the subsystems.}

3. Copy existing subsystems into integration program library structure. Assume the existence of logical names: (AS), VT, CLI, SM, and MAIN, which are symbolic links to directories containing the source code of the respective subsystems. Using these logical names, copy all the files in each of the indicated source directories (AS, VT, CLI, SM, and MAIN) into the integration program library structure. *Record initial source file sizes.*

{Assume that !Users.*experimenter.*Cmvc_Source with subworlds VT, CLI, SM, MAIN, and AS exist. Each subworld contains its Ada source units. Use the Library Copy command and its recursive copy parameter to move the Ada source units into the subsystem structure set up above.}

```
{Make !Users.experimenter.Cmvc_Source the current
context.}
Library.Copy
 (From =>
  "!Users.experimenter.Cmvc_Source.As",
  To =>
  "!Users.experimenter.Cm_Experiment.
     As.Rev1_Working.Units",
 Recursive => True,
 Copy_Links => False,
 Response => <PROFILE>;
```

{Put the Ada units placed in the initial spec/load working view under access control by using the Cmvc.Make_Controlled command and the wildcard to indicate all units in the directory.}

```
{Make !Users.experimenter.Cm_Experiment.
 As.Rev1_Working.Units the current context.}
Cmvc.Make_Controlled(What_Object => "@",
                     Reservation_Token_Name =>
                          "<AUTO_GENERATE>",
                     Join_With_View => "<None>",
                     Comments => "",
                     Work_Order => "<DEFAULT>",
                     Response => "<PROFILE>");
```

{Repeat the above procedures (Library.Copy and Cmvc.Make_Controlled) for each of the subsystems (CLI, MAIN, SM, and VT), placing the subsystem's Ada units into the appropriate Rev1_Working.Units subdirectory, and placing the Ada units under access control.}

4. Define a new (integrated) system model from existing subsystems. This system model specifies the compilation dependencies in effect when integrating all of the individual subsystems.

   {Create for each subsystem a spec view composed of the default—a copy of all the package specifications within the subsystem.}

```
{Make !Users.experimenter.Cm_Experiment.As
the current context, and place the cursor on
Rev1_Working.}
Cmvc.Make_Spec_View(From_Path => "<CURSOR>",
                    Spec_View_Prefix => "As",
                    Level => 0,
                    View_To_Modify => "",
                    View_To_Import =>
                             "<INHERIT_IMPORTS>",
                    Only_Change_Imports => True,
                    Remake_Demoted_Units => True,
                    Goal => Compilation.Coded,
                    Comments => "",
                    Work_Order => "<DEFAULT>",
                    Volume => 0,
                    Response => "<PROFILE>");
```

   {Repeat the creation of spec views for subsystems SM, CLI, and VT, giving the Spec_View_Prefix as "Sm," "Cli," and "Vt," respectively.}

   {Due to a bug in the Make_Spec_View command, the specification units copied to the spec view are not automatically promoted to the state indicated by the Goal parameter. It is necessary to visit all spec views and promote any units they contain to coded state. Because of dependencies between the spec views, the order in which the spec views are promoted is important.}

```
{Make !Users.experimenter.Cm_Experiment.Vt.
the current context, and place cursor on
Vt_0_Spec.}
<Code (This World)>
```

   {Promote the Ada units in !Users.*experimenter*.Cm_Experiment.As.As_0_Spec to coded state also. Before the Ada units in !Users.*experimenter*.Sm.Sm_0_Spec and !Users.*experimenter*.Cli.Cli_0_Spec can be promoted, imports for these spec views must be created. The spec view of SM depends upon definitions in the spec view of AS and VT. The spec view of CLI depends upon definitions in the spec view of AS.

```
{Make !Users.experimenter.Cm_Experiment.Sm.Sm_0_Spec
the current context.}
Cmvc.Import(View_To_Import => !Users.experimenter.
              Cm_Experiment.
              [As.As_0_Spec,Vt.Vt_0_Spec],
            Into_View => "<CURSOR>",
            Only_Change_Imports => False,
            Import_Closure => False,
            Remake_Demoted_Units => True,
            Goal => Compilation.Coded,
            Comments => "",
            Work_Order => "<DEFAULT>",
            Response => "<PROFILE>");
{Place cursor on
!Users.experimenter.Cm_Experiment.Sm.Sm_0_Spec.}
<Code (This World)>
{Make !Users.experimenter.Cm_Experiment.Cli.Cli_0_Spec
the current context.}
Cmvc.Import(View_To_Import => !Users.experimenter.
              Cm_Experiment.As.As_0_Spec);
{Check that cursor is on
!Users.experimenter.Cm_Experiment.Cli.Cli_0_Spec.}
<Code (This World)>
```

{Create the import list for each subsystem's working load view; MAIN imports the spec views of VT, CLI, and SM. The order in which these imports are performed is unimportant; they must only occur before attempts to compile units in the working load views of the subsystems.}

```
{Make !Users.experimenter.Cm_Experiment.Main.
Rev1_Working.Units the current context.}
Cmvc.Import(View_To_Import =>
              !Users.experimenter.Cm_Experiment.
        [Vt.Vt_0_Spec, Cli.Cli_0_Spec,
         Sm.Sm_0_Spec]);
```

{CLI imports the spec views of AS, VT, and SM:}

```
{Make !Users.experimenter.Cm_Experiment.Cli.
Rev1_Working.Units the current context.}
Cmvc.Import(View_To_Import =>
              !Users.experimenter.Cm_Experiment.
        [As.As_0_Spec, Vt.Vt_0_Spec,
         Sm.Sm_0_Spec]);
```

{SM imports the spec view of VT and AS:}

```
{Make !Users.experimenter.Sm_Experiment.Sm.
Rev1_Working.Units the current context.}
Cmvc.Import(View_To_Import =>
              !Users.experimenter.Cm_Experiment.
        [As.As_0_Spec, Vt.Vt_0_Spec]);
```

{Create the Activity Table, which is a listing of subsystems with their corresponding spec views and load views.}

```
{Make !Users.experimenter.Cm_Experiment
the current context.}
Activity.Create(The_Activity => "AIM_B01",
                Source => Activity.Nil,
                Mode => Activity.Exact_Copy,
                Response => "<PROFILE>");
```

{Make AIM_B01 the current context; insert entries. The order in which entries are added is not important as the editor maintains them in alphabetical order by subsystem name.}

```
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "",
       Load_View => "Rev1_Working");
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "Sm_0_Spec",
       Load_View => "Rev1_Working");
<Object> I
Insert(Subsystem => "Cli",
       Spec_View => "Cli_0_Spec",
       Load_View => "Rev1_Working");
<Object> I
Insert(Subsystem => "Vt",
       Spec_View => "Vt_0_Spec",
       Load_View => "Rev1_Working");
<Object> I
Insert(Subsystem => "As",
       Spec_View => "As_0_Spec",
       Load_View => "Rev1_Working");


{The entries must be saved by typing:}
<Enter>

{Make AIM_B01 the default activity with the following
commands:}
Set_Default(The_Activity => "AIM_B01",
            Response => "<PROFILE>");
```

5. Build an executable load module named **AIM_B01_EXE** from all the Ada source code files; use the system model defined in Step 4 where appropriate. *Measure time taken to perform the build.*

   {Promote all the Ada units in the load views to coded state, using the Compilation.Promote command, which relies on the subsystem structure and imports to determine compilation order.}

   ```
   {Make !Users.experimenter.Cm_Experiment
   the current context.}
   Compilation.Promote(Unit => "$.[Main,As,Vt,Sm,Cli].
                  Rev1_Working",
                       Scope =>
                           Compilation.Subunits_Too,
                       Goal => Compilation.Coded,
                       Limit => "<ALL_WORLDS>",
                       Effort_Only => False,
                       Response => "<PROFILE>");
   ```

   {Select and promote Main in subsystem MAIN, to cause a link of the closure and the creation and execution of a load module.}

   ```
   {Make !Users.experimenter.Cm_Experiment.Main.
   Rev1_Working.Units the current context; select
   Main'body.}
   <Promot>
   {Observe the creation of an output window, and
   "It works!" appears in the window.}
   ```

6. Construct a configuration baseline named **B0.1** of the current system. *Measure time taken to create the CM files. Record initial sizes of CM files. Measure time taken to perform baseline operation.*

{Release all of the subsystems and create an Activity which specifies the spec views and released load views. Note that because the Cmvc.Release command is actually making a copy of each subsystem, it takes some time to complete.}

```
{Make !Users.experimenter.Cm_Experiment
the current context}
Cmvc.Release(From_Working_View =>
             "!Users.experimenter.Cm_Experiment.
            [Main, Sm, Cli, Vt, As].Rev1_Working"
             Release_Name => "<AUTO_GENERATE>",
             Level => 0,
             Views_To_Import => "<INHERIT_IMPORTS>",
             Create_Configuration_Only  => False,
             Compile_The_View => True,
             Goal => Compilation.Coded,
             Comments => "",
             Work_Order => "<DEFAULT>",
             Volume => 0,
             Response => "<PROFILE>");


Activity.Create(The_Activity => "B0_1",
                Source => "Aim_B01");
```

NOTE: *The value for the parameter Source must be in quotes.*

```
{Make !Users.experimenter.Cm_Experiment.B0_1
the current context and insert entries with
the following commands:}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "",
       Load_View => "Rev1_0_1");
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "",
       Load_View => "Rev1_0_1");
<Object> I
Insert(Subsystem => "Cli",
       Spec_View => "",
       Load_View => "Rev1_0_1");
<Object> I
Insert(Subsystem => "Vt",
       Spec_View => "",
       Load_View => "Rev1_0_1");
<Object> I
Insert(Subsystem => "As",
       Spec_View => "",
       Load_View => "Rev1_0_1");
```

{Save the new entries in the activity by typing:}

```
<Enter>
```

7. Parallel test the integrated system using three variants of main program (**MAIN.A**, **MAIN.B**, **MAIN.C**). *Measure time for single file fetch, reserve, and replace* operations.

   MAIN.A - test VT interfaces

   MAIN.B - test CLI interfaces

MAIN.C - test SM interfaces

{Create 3 parallel development paths from the released subsystem MAIN--vttester, clitester and smtester. Note that these names would be much more readable as "vt_tester," "cli_tester," and "sm_tester"; however, due to a limitation of the Cmvc.Build command used later, no underscores (_) can be used in a user-designated pathname.}

```
{Make !Users.experimenter.Cm_Experiment.Main
the current context}
Cmvc.Make_Path(From_Path =>
                "!Users.experimenter.
           Cm_Experiment.Main.Rev1_0_1",
               New_Path_Name => "Vttester",
               View_To_Modify => "",
               View_To_Import => "<INHERIT_IMPORTS>",
               Only_Change_Imports => True,
               Model => "<INHERIT_MODEL>",
               Join_Paths => False,
               Remake_Demoted_Units => True,
               Goal => Compilation.Coded,
               Comments => "",
               Work_Order => "<DEFAULT>",
               Volume => 0,
               Response => "<PROFILE>");
Cmvc.Make_Path(From_Path =>
                "!Users.experimenter.
           Cm_Experiment.Main.Rev1_0_1",
               New_Path_Name => "Clitester",
               Join_Paths => False);
Cmvc.Make_Path(From_Path =>
                "!Users.experimenter.
           Cm_Experiment.Main.Rev1_0_1",
               New_Path_Name => "Smtester",
               Join_Paths => False);
```

a. Test VT interfaces.

i. Build executable load module named **VT_MAIN** using MAIN.A as the main program. *Measure time taken to perform the build.*

```
{Make !Users.experimenter.Cm_Experiment.
Main.Vttester_Working.Units.Main'body the
current context}
<Check Out>
<Install Unit>
{Place cursor in body, after the
Text_IO.Put_Line statement.}
<Object> I
{An edit window opens and the
cursor is placed in it automatically;
insert the code line:
Text_IO.Put_line
    ("This is the VT Driver.");}
<Format>
{Place the new code line back
into the program.}
<Promot>
{Return the entire unit to
coded state.}
<Promot>
<Check In>
```

ii. Fix bugs in VT body (using variant line of descent). *Measure time taken for creating a variant line of descent. Measure CM file size increase caused by variant.*

{Since other programmers working on the system will be linked with the release version of subsystem VT, the programmer responsible for updates to Page_Terminal'body can continue to work in the Rev1_Working version of the subsystem.}

```
{Make !Users.experimenter.Cm_Experiment.Vt.
Rev1_Working.Units.Page_Terminal'body the
current context.}
<Check Out>
<Edit>
{Replace the null statement with:
Text_IO.Put_Line
("Elaborating Page_Terminal Body.");}
<Format>
<Code Unit>
<Check In>
```

iii. Construct a configuration baseline named **B0.2** of the current system using MAIN.A as the main program. *Record current sizes of CM files. Measure time taken to perform baseline operation.*

{Create a release of the VT subsystem; create an Activity B0_2 which uses the new release of the VT subsystem and Path VTtester for Main.}

```
{Make !Users.experimenter.Cm_Experiment.
Vt.Rev1_Working.Units the current context.
NOTE: Use all defaults in Cmvc.Release command.}
Cmvc.Release

{Make !Users.experimenter.Cm_Experiment
the current context.}
Activity.Create(The_Activity => "B0_2",
                Source => "B0_1");
{Make !Users.experimenter.Cm_Experiment.
B0_2 the current context.}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "",
       Load_View => "Vttester_Working");
<Object> I
Insert(Subsystem => "Vt",
       Spec_View => "",
       Load_View => "Rev1_0_2");
<Enter>
```

{Make Activity B0_2 the default activity and execute the system.}

```
Set_Default(The_Activity => "B0_2");
{Make !Users.experimenter.Cm_Experiment.
Main.Vttester_Working.Units the
current context and select Main'body.}
<Promot>
{Observe the creation of an output window;
"Elaborating Page_Terminal Body.",
"It works!" and "This is the VT Driver."
appear in the window.}
```

b. Test CLI interfaces

i. Build executable load module named **CLI_MAIN** using MAIN.B as the main program. *Measure time taken for creating a variant line of descent. Measure CM file size increase caused by variant. Measure time taken to perform the build.*

{Make Activity B0_1 the default activity, edit Main in Clitester subpath.}

```
Activity.Set_Default(The_Activity =>
       "!Users.experimenter.Cm_Experiment.
       B0_1"

{Make !Users.experimenter.Cm_Experiment.
Main.Clitester_Working.Units.Main'body
the current context.}
<Check Out>
<Install Unit>
{Place cursor in body after
the Text_IO.Put_Line statement.}
<Object> I
{An edit window opens and the cursor
is placed in it automatically.
Insert the code line:}
Text_IO.Putline
      ("This is the CLI Driver.");}
<Promot>
```

ii. Fix bugs in MAIN.B. {Note that Putline is underlined as an error.}

```
{Correct Putline to be Put_Line; place the
new code line back into the program.}
<Promot>
{Return the entire unit to coded state.}
<Promot>
<Check In>
```

iii. Construct a configuration baseline named **B0.3** of the current system using MAIN.B as the main program. *Record current size of CM files. Measure time taken to perform baseline operation.*

{Create Activity B0_3; make it the current activity; and link, load, and execute system.}

```
{Make !Users.experimenter.Cm_Experiment
the current context.}
Activity.Create(The_Activity => "B0_3",
                Source => "B0_1");
```

NOTE: *The value for the parameter Source must be in quotes.*

```
{Make !Users.experimenter.Cm_Experiment.
B0_3 the current context.}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "",
       Load_View => "Clitester_Working");

{Save the new entries.}
<Enter>

Set_Default(The_Activity => "B0_3");

{Make !Users.experimenter.Cm_Experiment
.Main.Clitester_Working.Units and select
Main'body.}
<Promot>
{Observe the creation of an output window.
"It works!" and "This is the CLI Driver."
 appear in the window.}
```

c. Test SM interfaces.

i. Build executable load module named **SM_MAIN** using MAIN.C as the main program. *Measure time taken to perform the build. Measure time taken for creating a variant line of descent. Measure CM file size increase caused by variant.*

{Make Activity B0_1 the default activity, edit Main in Smtester sub-path, link, load and execute.}

```
Activity.Set_Default(The_Activity =>
       "!Users.experimenter.Cm_Experiment.
       B0_1"

{Make !Users.experimenter.Cm_Experiment.
Main.Smtester_Working.Units.Main'body
the current context.}
<Install Unit>
{Place cursor in body after
the Text_IO.Put_Line statement.}
<Object> I
{An edit window opens and the
cursor is placed in it automatically;
insert the code line:
Text_IO.Put_Line
        ("This is the SM Driver.");}
{Place the new code line back into
the program.}
<Promot>
{Return the entire unit to coded
state.}
<Promot>
<Check In>

{Make !Users.experimenter.Cm_Experiment.
Main.Smtester_Working.Units the current
context and select Main'body.}
<Promot>
{Observe the creation of an output window.
"It works!" and "This is the SM Driver."
appear in the window.}
```

ii. Add new interface to Viewport_Manager package (using variant versions). *Measure time taken for creating a variant line of descent. Measure CM file size increase caused by variant.*

{Add function definition to Viewport_Manager specification, add function to Viewport_Manager'body, make a new spec view of subsystem SM, update subsystems which import subsystem SM.}

```
{Make !Users.experimenter.Cm_Experiment.Sm.
Rev1_Working.Units.Viewport_Manager'spec
the current context.}
<Check Out>
{Place cursor after subtype V_String
definition.}
<Object> I
{An edit window opens; type in the
specification:
"function I_Do_Nothing return Boolean;"}
<Format>
<Promot>
<Check In>

{Make !Users.experimenter.Cm_Experiment.Sm.
Rev1_Working.Units.Viewport_Manager'body
the current context.}
<Check Out>
<Edit>
{After "package body is" line, add:

  function I_Do_Nothing return Boolean is
  begin
    return True;
  end I_Do_Nothing;  }

{Replace null statement in package body
with:
  Text_Io.Put_Line("Elaborating
          Viewport_Manager Body.");  }
<Code Unit>
<Check In>

{Make !Users.experimenter.Cm_Experiment.Sm.
Rev1_Working the current context.}
Cmvc.Make_Spec_View(Spec_View_Prefix =>
                        "SM",
                    Level => 1);
{Due to bug in Cmvc.Make_Spec_View
command, units in the new spec view
must be manually promoted to coded state;
make !Users.experimenter.Cm_Experiment.
Sm.Sm_1_Spec the current context.}
<Code (This World)>

{Make !Users.experimenter.Cm_Experiment.
Main.Smtester_Working the current
context.}
Cmvc.Import(View_To_Import => "!Users.
   experimenter.Cm_Experiment.Sm.Sm_1_Spec);
{Make !Users.experimenter.Cm_Experiment.
Cli.Rev1_Working the current context.}
Cmvc.Import(View_To_Import => "!Users.
   experimenter.Cm_Experiment.Sm.Sm_1_Spec);
```

iii. Rebuild executable load module named **SM_MAIN** with new version of the Viewport_Manager. Test new interface along with previous interfaces of the SM. *Measure time taken to perform the build.*

{Construct an activity Aim_B04 which will test the new spec view and driver for Sm.}

```
{Make !Users.experimenter.Cm_Experiment
the current context.}
Activity.Create(The_Activity => "Aim_B04",
                Source => "B0_1");
```
NOTE: *The value for the parameter Source must be in quotes.*
```
{Make !Users.experimenter.Cm_Experiment.
Aim_B04 the current context}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "",
       Load_View => "Smtester_Working");
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "Sm_1_Spec",
       Load_View => "Rev1_Working");

{Save the new entries.}
<Enter>

Set_Default(The_Activity => "Aim_B04");
{Make !Users.experimenter.Cm_Experiment.
Main.Smtester_Working.Units the current
context, and select Main'body.}
<Promot>
{Observe the creation of an output window;
"Elaborating Viewport_Manager body.",
"It works!" and "This is the SM Driver."
appear in the window.}
```

iv. Construct a configuration baseline named **B0.4** of the current system using MAIN.C as the main program and the new versions of the Viewport Manager. *Record current sizes of the CM files. Measure time taken to perform baseline operation.*

{Create a release of the SM subsystem; create an Activity B0_4 which uses the new release of the SM subsystem and Path Smtester for Main.}

```
{Make !Users.experimenter.Cm_Experiment.
Sm.Rev1_Working.Units the current context.
NOTE: Use all defaults in Cmvc.Release command.}
Cmvc.Release

{Make !Users.experimenter.Cm_Experiment
the current context.}
Activity.Create(The_Activity => "B0_4",
                Source => "Aim_B04");
```
NOTE: *The value for the parameter Source must be in quotes.*
```
{Make !Users.experimenter.Cm_Experiment.
B0_4 the current context}
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "Sm_1_Spec",
       Load_View => "Rev1_1_1");

{Save the new entry.}
<Enter>
```

{Make Activity B0_4 the default activity and execute the system.}

```
Set_Default(The_Activity => "B0_4");
{Make !Users.experimenter.Cm_Experiment.
Main.Smtester_Working.Units the
current context and select Main'body.}
<Promot>
{Observe the creation of an output window;
"Elaborating Viewport_Manager Body."
"It works!" and "This is the SM Driver."
appear in the window.}
```

8. Merge bug fixes and enhancements back into main line of descent for:

    a. Main program

    b. VT package body

    c. VM package specification and body

    Measure time to perform merge operations. Record CM file size increases caused by merge operations.

    {Use the Cmvc.Merge_Changes command to merge Main'body from Vttester_Working, Clitester_Working, and Smtester_Working back into Rev1_Working.}

```
{Make !Users.experimenter.Cm_Experiment.
Main.Rev1_Working.Units the current context,
select Main’body.}
Cmvc.Merge_Changes(Destination_Object =>
                     "<SELECTION>",
                   Source_View => "!Users.
                  experimenter.Cm_Experiment.
                  Main.Vttester_Working,
                   Report_File => "",
                   Fail_If_Conflicts_Found => False,
                   Comments => "",
                   Work_Order => "<DEFAULT>",
                   Response => "<PROFILE>");
{Check Main_Merging_Report to see if merge occurred
successfully.}

{Select Main’body}
Cmvc.Merge_Changes(Destination => "<SELECTION>",
                   Source_View => "!Users.
                  experimenter.Cm_Experiment.
                  Main.Clitester_Working);
{Check report to see where conflicts occurred;
select Main’body}
<Edit>
{Remove "*;1" and "*;2" from the beginning of
"conflicting" lines.}
<Next Item>
<Control> D
<Control> D
<Control> D
<Format>
<Control> D
<Control> D
<Control> D
<Code Unit>
<Check In>

{Select Main’body.}
Cmvc.Merge_Changes(Destination => "<SELECTION>",
           Source_View => "!Users.
             experimenter.Cm_Experiment.Smtester_Working);
{Check report to see where conflicts occurred;
select Main’body.}
<Edit>
{Remove "*1;" and "*2;" from the beginning of
"conflicting" lines.}
<Code Unit>
<Check In>
```

{To "merge" the bug fixes and enhancements, the experimenter need only pick up the latest release of subsystems VT and SM in the subsequent experiment step.}

9. Add prologues to package specifications and bodies. *Measure time for single file reserve and replace operations.*

```
{Make !Users.experimenter.Cm_Experiment.
Vt.Rev1_Working.Units current context and
place cursor on Vt_Support'spec.}
<Mark> <Begin Of>
<Definition>
<Check Out>
{Place cursor at beginning of file.}
<Image> <Begin Of>
<Object> I
{An Edit window opens, enter the prologue:}

-------------
-- This is a sample Ada program prologue:
--
--            Author:
--         Unit Name:
--     Creation Date:
--    Update History:
--
-------------

<Promot>
<Check In>
<Mark> <End Of>

{To repeat the above prologue insertion
procedure for each of the package
specifications and bodies in the
Rev1_Working views of all the subsystems,
the <Mark> <Begin Of> and
<Mark> <End Of> commands above
have created a macro, which now may
be bound to a key.}

Editor.Macro.Bind(Key => "F1");

{Now place the cursor on any
package specification or body while
in its enclosing context, and the
commands to insert the
prologue template above will be
repeated with the press of key
<F1>.}
```

10. Construct a configuration baseline named **V1.0** of the current system. *Record current sizes of CM files. Measure time taken to perform baseline operation.*

    {Release all the subsystems, create Activity V1_0, and make Activity V1_0 the default activity.}

```
{Make !Users.experimenter.Cm_Experiment the current
context.}
Cmvc.Release(From_Working_View =>
                !Users.experimenter.
          Cm_Experiment.[Vt,Sm,Cli,As,Main].
          Rev1_Working);

Activity.Create(The_Activity => "V1_0",
                Source => "B0_1");
```

NOTE: *The value for the parameter Source must be in quotes.*

```
{Make !Users.experimenter.Cm_Experiment.V1_0
the current context and insert entries with
the following commands:}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "",
       Load_View => "Rev1_0_2");
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "Sm_1_Spec",
       Load_View => "Rev1_1_2");
<Object> I
Insert(Subsystem => "Cli",
       Spec_View => "",
       Load_View => "Rev1_0_2");
<Object> I
Insert(Subsystem => "Vt",
       Spec_View => "",
       Load_View => "Rev1_0_3");
<Object> I
Insert(Subsystem => "As",
       Spec_View => "",
       Load_View => "Rev1_0_2");

{Save the entries:}
<Enter>

Set_Default(The_Activity => "V1_0");
```

11. Build executable load module name **Product** using all current source code.

    {In order to create a code view of the subsystems which can be executed, a spec view for subsystem Main must be created.}

```
{Make !Users.experimenter.Cm_Experiment.
Main.Rev1_0_2 the current context.}
Cmvc.Make_Spec_View(Spec_View_Prefix =>
                    "Main",
                    Level => 1);
{Due to a bug in the Cmvc.Make_Spec_View
command, units in the new spec view
must be manually promoted to coded state,
make !Users.experimenter.Cm_Experiment.
Main.Main_0_Spec the current context.}
<Code (This World)>
```

    {Create a code view of each of the subsystems and an Activity which references them.}

```
{Make !Users.experimenter.Cm_Experiment.
the current context.}
Cmvc.Make_Code_View(
    From_View => "$.[Main.Rev1_0_2,
           Sm.Rev1_1_2,
           Cli.Rev1_0_2,
           Vt.Rev1_0_3,
           As.Rev1_0_2]",
    Code_View_Name => "Product",
    Comments => "",
    Work_Order => "<DEFAULT>",
    Volume => 0,
    Response => "<PROFILE>");

{Create an Activity named Product,
Make !Users.experimenter.Cm_Experiment
the current context.}
Activity.Create(The_Activity => "Product",
                Source => "V1_0");
```

NOTE: *The value for the parameter Source must be in quotes.*

```
{Make !Users.experimenter.Cm_Experiment.
Product the current context.}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "Main_0_Spec",
       Load_View => "Product");
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "",
       Load_View => "Product");
<Object> I
Insert(Subsystem => "Cli",
       Spec_View => "",
       Load_View => "Product");
<Object> I
Insert(Subsystem => "Vt",
       Spec_View => "",
       Load_View => "Product");
<Object> I
Insert(Subsystem => "As",
       Spec_View => "",
       Load_View => "Product");

{Save the entries:}
<Enter>

Set_Default(The_Activity => "Product");
```

```
{Make !Users.experimenter.Cm_Experiment.
Main.Main_0_Spec.Units the current context;
select Main'Spec}
<Promot>
{Observe the creation of an output window;
"Elaborating Viewport_Manager Body.
Elaborating Page_Terminal Body.
It Works!
This is the SM Driver.
This is the CLI Driver.
This is the VT Driver."
appears in the window.}
```

2.3. Experiment #1 Functionality Checklist

Primary Activities

Activity	Step #	Supported (Y/N)	Observations
Version Control			
Create element	6	Yes	Must create the Ada object in the Units directory of the subsystem and place under control using the Cmvc.Make_Controlled command.
Create new version			
Successive	9	Yes	Conventional checkin/checkout paradigm.
Parallel	7	Yes	Branch creation supported via the Make_Path and Make_SubPath commands with Join_Paths parameter set to False.
Retrieve specific version			
Explicit	7	Yes	Use the Cmvc.Revert command.
Dynamic	7,9	Yes	Defaults to most recent revision.
Configuration Control			
Define system model			
Specify source dependencies	5,7,11	Yes	Maintained by library and imports.
Specify translation rules	5,7,11	N/A	
Specify translation options	5,7,11	Yes	Set Model for subsystem.
Specify translation tools	5,7,11	N/A	
Build system			
Current default	5,7,11	Yes	Using Compilation. Promote and enumerate subsystems comprising the system.
Product Release			
Baseline system	6,7,10	Yes	Use Cmvc.Release on each subsystem and define an Activity for the Baseline.
Create system release class	11	Yes	Create coded views of all subsystems.

Secondary Activities

Activity	Step #	Supported	Observations
Version Control			
Merge variants	8	Yes	Merge Ada objects which have a common ancestor.

2.4. Experiment #2

1. Experiment setup

    a. Establish environment variables to be used in the experiment.

    {None are used; the logical names assumed by the experiment are used as subsystem names.}

    b. Change working directory to the **system_integration** directory created in Experiment #1.

    ```
    {Make !Users.experimenter.Cm_Experiment the
    current context.}
    ```

    c. Create a new program library named **build_lib** underneath the sys_integration directory.

    {New libraries, or working views of the subsystems, will be created as needed later.}

    d. Confirm that no files are currently reserved:

    ```
    Cmvc.Show_Checked_Out_In_View(In_View =>
        "$.[Main,Vt,Sm,Cli,As].Rev1_Working",
                                    Response =>
                                      "<PROFILE>");

    Cmvc.Show_Checked_Out_In_View(In_View =>
        "$.Main.[Vttester_working,Clitester_working,
        Smtester_working]");
    ```

    e. Remove any pre-existing copies of files used throughout the experiment.

    {Remove all working and release views, but allow configuration objects to remain.}

```
{Make !Users.experimenter.Cm_Experiment
the current context.}
Cmvc.Destroy_View(
 What_View =>
  "$.As.[Rev1_0_1,
      Rev1_0_2,
      Rev1_Working]",
 Demote_Clients => False,
 Destroy_Configuration_Also => False,
 Comments => "",
 Work_Order => "<DEFAULT>",
 Response => "<PROFILE>");
Cmvc.Destroy_View(
 What_View =>
  "$.Cli.[Rev1_0_1,
       Rev1_0_2,
       Rev1_Working]");
Cmvc.Destroy_View(
 What_View =>
  "$.Main.[Clitester_Working,
       Rev1_0_1,
       Rev1_0_2,
       Rev1_Working,
       Smtester_Working,
       Vttester_Working]");
Cmvc.Destroy_View(
 What_View =>
  "$.Sm.[Rev1_0_1,
      Rev1_1_1,
      Rev1_1_2,
      Rev1_Working]");
Cmvc.Destroy_View(
 What_View =>
  "$.Vt.[Rev1_0_1,
      Rev1_0_2,
      Rev1_0_3,
      Rev1_Working]");
```

2. Display configuration management historical information pertaining to the current state of the system. *Measure time taken to display history information.*

```
Cmvc.Show_History_By_Generation(
      For_Objects =>
          "$.[Main,Vt,Sm,Cli,As].Product",
      Display_Change_Regions => True,
      Starting_Generation => 1,
      Ending_Generation => Natural'Last,
      Response => "<PROFILE>");
```

3. Fetch all the Ada source code files belonging to the **B0.4** baseline and build an executable load module named **Version0.4**. *Measure time taken to fetch the source files in the* **B0.4** *baseline. Measure time taken to perform the build.*

   {Display Activity B0_4, note the version numbers of the spec views and load views in Activity B0_4. Invoke the **Cmvc.Build** command to reconstruct all the load views. Create working views. Create Aim_V1_2 to refer to the new working views. Set default Activity to Aim_V1_2. Execute main program.}

```
{Make !Users.experimenter.Cm_Experiment.B0_4 the
current context, note:

Subsystem     Spec View       Load View
---------     ---------       ---------
As            As_0_Spec       Rev1_0_1
Cli           Cli_0_Spec      Rev1_0_1
Main                          Smtester_Working
Sm            Sm_1_Spec       Rev1_1_1
Vt            Vt_0_Spec       Rev1_0_1   }

{Make !Users.experimenter.Cm_Experiment the
current context.}
Cmvc.Build(Configuration =>
              "$.[As.Configurations.Rev1_0_1,
          Cli.Configurations.Rev1_0_1,
          Main.Configurations.Smtester_Working,
          Sm.Configurations.Rev1_1_1,
          Vt.Configurations.Rev1_0_1]",
           View_To_Import => "",
           Model => "R1000_Portable",
           Goal => "Compilation.Coded",
           Limit => "<WORLDS>",
           Comments => "",
           Work_Order => "<DEFAULTS>",
           Volume => 0,
           Response => "<PROFILE>");
```

{Due to a bug in the creation of configuration-only mode and the Build command, the imports between the spec views and load views are lost; when the Build command attempts to recompile the system to coded state, the build fails. This leaves the frozen releases in an unfrozen state. Also, the State.Release_History text file is lost from each view, and must be replaced by a dummy blank file.}

```
{Replace Release_History files.
Make !Users.experimenter.Cm_Experiment.
As.Rev1_0_1.State the current context.}
<Create Text>
Text.Create(Image_Name => "Release_History",
            Kind => Text.File);
<Promot>
{A text editor window opens on text file
Release_History.}
<Enter>

{Repeat the above steps in contexts
!Users.experimenter.Cm_Experiment.Cli.
Rev1_0_1.State,
!Users.experimenter.Cm_Experiment.Main.
Smtester_Working.State,
!Users.experimenter.Cm_Experiment.Sm.
Rev1_1_1.State, and
!Users.experimenter.Cm_Experiment.Vt.
Rev1_0_1.State to create a blank
Release_History text file in each.}

{Create a working view for each subsystem; make
!Users.experimenter.Cm_Experiment the
current context.}
Cmvc.Make_Path(From_Path => "Users.experimenter.
       Cm_Experiment.As.Rev1_0_1",
               New_Path_Name => "Rev2",
               Join_Paths => "False");

Cmvc.Make_Path(From_Path => "Users.experimenter.
       Cm_Experiment.Cli.Rev1_0_1",
               New_Path_Name => "Rev2",
               Join_Paths => "False");

Cmvc.Make_Path(From_Path => "Users.experimenter.
       Cm_Experiment.Main.Smtester_Working",
               New_Path_Name => "Rev2",
               Join_Paths => "False");

Cmvc.Make_Path(From_Path => "Users.experimenter.
       Cm_Experiment.Sm.Rev1_1_1",
               New_Path_Name => "Rev2",
               Join_Paths => "False");

Cmvc.Make_Path(From_Path => "Users.experimenter.
       Cm_Experiment.Vt.Rev1_0_1",
               New_Path_Name => "Rev2",
               Join_Paths => "False");
```

```
{Due to problems with the Build command, run
Cmvc_Maintenance.Check_Consistency on the
working views.  Let !Users.experimenter.
Cm_Experiment remain the current context.}
Cmvc_Maintenance.Check_Consistency( Views =>
     "$.[As.Rev2_Working, Cli.Rev2_Working,
      Main.Rev2_Working, Sm.Rev2_Working,
      Vt.Rev2_Working]",
      Response => "<PROFILE>");

{Due to problems with the Build command, must
redefine the imports required to compile the
system.}
{Make !Users.experimenter.Cm_Experiment.Main.
Rev2_Working the current context.}
Cmvc.Import(View_To_Import =>
             !Users.experimenter.Cm_Experiment.
      [Sm.Sm_1_Spec, Vt.Vt_0_Spec,
      Cli_0_Spec]);

{Make !Users.experimenter.Cm_Experiment.Cli.
Rev2_Working the current context}
Cmvc.Import(View_To_Import =>
             !Users.experimenter.Cm_Experiment.
      [Sm.Sm_1_Spec, Vt.Vt_0_Spec,
      As.As_0_Spec]);

{Make !Users.experimenter.Cm_Experiment.Sm.
Rev2_Working the current context}
Cmvc.Import(View_To_Import =>
             !Users.experimenter.Cm_Experiment.
      [Vt.Vt_0_Spec, As.As_0_Spec]);

{Create an Activity to refer to the new
working views, make !Users.experimenter.
Cm_Experiment the current context}
Activity.Create(The_Activity => "AIM_V1_2",
                Source => "B0_4")
```

```
{Make !Users.experimenter.Cm_Experiment.Aim_V1_2
the current context.}
<Object> I
Insert(Subsystem => "Main",
       Load_View => "Rev2_Working");
<Object> I
Insert(Subsystem => "Sm",
       Load_View => "Rev2_Working");
<Object> I
Insert(Subsystem => "Cli",
       Load_View => "Rev2_Working");
<Object> I
Insert(Subsystem => "Vt",
       Load_View => "Rev2_Working");
<Object> I
Insert(Subsystem => "As",
       Load_View => "Rev2_Working");

{Save the entries for the new Activity:}
<Enter>

Set_Default(The_Activity => "Aim_V1_2")

{Rebuild the system, make
!Users.experimenter.Cm_Experiment
the current context.}
Compilation.Promote(
 Unit => "$.[As, Cli, Main, Sm, Vt].
    Rev2_Working",
 Goal => Compilation.Coded,
 Limit => <ALL_WORLDS>);

{Link and load the system by executing it; make
!Users.experimenter.Cm_Experiment.Main.
Rev2_Working.Units the current context, and
select Main'body.}
<Promot>

{Observe the creation of an output window,
and "Elaborating Viewport_Manager Body.",
"It works!", and "This is the SM Driver."
appear in the window.}
```

4. Move the Str_Utilities package specification and body of the current system (**V1.0**) into the local copies of the AIM_SUPPORT package specification and body. Recompile compilation units as necessary.

   {Checking Activity V1_0 shows that release Rev1_0_2 of subsystem AS would have the Str_Utilities package specification and body; rebuild subsystem As.Rev1_0_2.}

   ```
   {Make !Users.experimenter.Cm_Experiment.
   As the current context.}
   Cmvc.Build(Configuration => "Configurations.Rev1_0_2",
              Model => "R1000_Portable");
   ```

   {Edit the Aim_Support package specification to contain the definitions in the Str_Utilities package specification. Create Aim_Support package body to contain the function defined in Str_Utilities. Make Aim_Support package body controlled, create a new spec view for subsystem As. Change uses of package Str_Utilities to Aim_Support in subsystems Cli and Sm. Generate new spec view for Sm.}

```
{Make !Users.experimenter.Cm_Experiment.
As.Rev1_0_2.Units.Str_Utilities'Spec the current
context.}
<Window> <Promot>
{Make !Users.experimenter.Cm_Experiment.
As.Rev2_Working.Units.Aim_Support'Spec the current
context.}
<Check Out>
<Install Unit>
{Place cursor after subtype Aim_Name definition.}
<Object I>
{Place cursor back in Str_Utilities'Spec window,
with cursor before type Str_Access definition.}
<Region> [
{Place cursor after function Get_String definition.}
<Region> ]
{Note that the two lines become highlighted,
and move cursor back to Aim_Support'Spec window.}
<Region> C
<Format>
<Promot>
<Check In>

{Make !Users.experimenter.Cm_Experiment.
As.Rev1_0_2.Units.Str_Utilities'Spec the current
context.}
<Window> <Demote>

{Make !Users.experimenter.Cm_Experiment.
As.Rev1_0_2.Units.Str_Utilities'Body the current
context.}
<Window> <Promot>
{Place cursor before function Get_String definition.}
<Region> [
{Place cursor after "end Get_String;" statement.}
<Region> ]

{Make !Users.experimenter.Cm_Experiment.
As.Rev2_Working.Units the current context.}
<Create Ada>
{Enter:
"package body Aim_Support is".}
<Format>
{Place cursor after package body declaration in
!Users.experimenter.Cm_Experiment.As.
Rev2_Working.Units.Aim_Support'body.}
<Region> C
<Format>
<Code Unit>

{Make !Users.experimenter.Cm_Experiment.
As.Rev1_0_2.Units.Str_Utilities'Body the current
context.}
<Window> <Demote>
```

{Since Aim_Support'Body was added to subsystem As, it must be controlled. The configuration management system up to this point has assumed the existence of a null Aim_Support'Body. An attempt to use Cmvc.Make_Controlled to put the new actual Aim_Support'Body under version control results in error messages and command failure. The workaround for this bug is to use the Cmvc_Maintenance command Check_Consistency and then make Aim_Support'Body controlled.}

```
{Make !Users.experimenter.Cm_Experiment.
As, place cursor on Rev2_Working.}
Cmvc_Maintenance.Check_Consistency(
                    Views => "<CURSOR>",
                    Response => "<PROFILE>");
```
(*Note: use default values.*)

```
{Make !Users.experimenter.Cm_Experiment.
As.Rev2_Working.Units the current context.}
Cmvc.Make_Controlled(What_Object =>
                        "Aim_Support'body");

{Make !Users.experimenter.Cm_Experiment.
Cli.Rev2_Working.Units.
Command_Interpreter'Body the current
context.}
<Check Out>
{Select "with Str_Utilities" statement.}
<Edit>
{In edit window change "with Str_Utilities"
to "with Aim_Support".}
<Format>
<Promot>
<Check In>

{Make !Users.experimenter.Cm_Experiment.
Sm.Rev2_Working.Units.Image_Manager'Spec
the current context.}
<Check Out>
{Select "with" clause.}
<Edit>
{Delete "with Str_Utilities".}
<Format>
<Promot>
<Check In>

Cmvc.Make_Spec_View(
   Spec_View_Prefix => "Sm2",
   Level => 0);
```

5. Fetch the current version (from baseline **V1.0**) of the COMMAND_INTERPRETER.PERFORM_COMMAND subprogram. *Measure time taken to perform fetch operation.*

   {Checking Activity V1_0 shows that release Rev1_0_2 of subsystem CLI would have the subprogram Command_Interpreter.Perform_Command; rebuild subsystem Cli.Rev1_0_2.}

   ```
   {Make !Users.experimenter.Cm_Experiment.
   Cli the current context.}
   Cmvc.Build(Configuration => "Configurations.Rev1_0_2",
              Model => "R1000_Portable");
   ```

   {Copy Command_Interpreter.Perform_Command from the V1_0 release version of subsystem CLI to the Rev2_Working.Units version.}

```
{Make !Users.experimenter.Cm_Experiment.
Cli.Rev2_Working.Units.Command_Interpreter.
Perform_Command the current context.}
<Check Out>
Library.Copy(From => "!Users.experimenter.
      Cm_Experiment.Cli.Rev1_0_2.Units.
      Command_Interpreter.Perform_Command",
             To => "!Users. experimenter.
      Cm_Experiment.Cli.Rev2_Working.Units.
      Command_Interpreter.Perform_Command")
<Code Unit>
<Check In>
```

6. Generate an executable load module named **Product** using the Ada source files presently in the experiment's source code directory; perform this system build using the pragma SUPPRESS to disable the following checks during the translation phase:

   - access_check
   - discriminant_check
   - index_check
   - length_check
   - division_check
   - overflow_check
   - elaboration_check

   Measure time taken to perform the build.

   {Pragma SUPPRESS is not supported.}

7. Remove the configuration management file elements associated with the specification and body of the STR_UTILITIES package. *Measure time taken to perform remove operation.*

   {Make uncontrolled and remove Str_Utilities specification and body. Create new spec views for the subsystem As and Sm.}

```
{Make !Users.experimenter.Cm_Experiment.As.
Rev2_Working.Units the current context.}
Cmvc.Make_Uncontrolled(What_Object => "Str_Utilities'Spec",
       Comments => "", Work_Order => "<DEFAULT>",
       Response => "<PROFILE>");
       {Select Str_Utilities'Body.}
Cmvc.Make_Uncontrolled(What_Object => "<CURSOR>");
{Now delete the body and specification;
select Str_Utilities'Body.}
<Object> d
{Select Str_Utilities'Spec}
<Object> d

{Allow cursor to remain in context !Users.
experimenter.Cm_Experiment.As.Rev2_Working.Units.}
Cmvc.Make_Spec_View(Spec_View_Prefix => "As",
                    Level => 1);

{Make !Users.experimenter.Cm_Experiment.Sm.Rev2_Working
the current context.}
Cmvc.Make_Spec_View(Spec_View_Prefix => "Sm2");
```

{Due to a bug in the Make_Spec_View command, the specification units copied to the spec view are not automatically promoted to the state indicated by the Goal parameter. Visit the spec view and promote the units to coded state.}

```
{Make !Users.experimenter.Cm_Experiment.As.As_1_Spec.
Units the current context.}
<Code (This World)>
{Make !Users.experimenter.Cm_Experiment.Sm.Sm2_0_Spec.
Units the current context.}
<Code (This World)>
```

{Update all the referencers of the spec view of subsystem As and Sm.}

```
Cmvc.Import(View_To_Import => "!Users.experimenter.
       Cm_Experiment.As.As_1_Spec",
            Into_View => "!Users.experimenter.
       Cm_Experiment.[
       Cli.[Cli_0_Spec, Rev2_Working],
       Sm.[Sm2_0_Spec, Rev2_Working],
       Main.[Main_0_Spec, Rev2_Working]]";

Cmvc.Import(View_To_Import => "!Users.experimenter.
       Cm_Experiment.Sm.Sm2_0_Spec",
            Into_View => "!Users.experimenter.
       Cm_Experiment.[Main, Cli].Rev2_Working");
```

8. Add prologues to all Ada source code contained in the experiment's code directory.

```
{Make !Users.experimenter.Cm_Experiment.Vt.
Rev2_Working.Units the current context and
place cursor on Vt_Support.}
<Mark> <Begin Of>
<Definition>
<Check Out>
<Begin Of>
<Object> I
{Enter the following in the edit window:

------------
-- This is a sample Ada program prologue:
--
--            Author:
--         Unit Name:
--     Creation Date:
--    Update History:
--
------------
}
<Promot>
<Check In>
<Mark> <End Of>
```

{Repeat the above for all Ada Units in the Rev2_Working.Units areas of subsystems Vt, As, Cli, Sm and Main except for Command_Interpreter.Perform_Command, which will already have a prologue.}

```
{To repeat the above prologue insertion procedure,
the <Mark> <Begin Of> and <Mark> <End Of> commands
above have created a macro, which now may be bound
to a key.}

Editor.Macro.Bind(Key => "F1");

{Now place the cursor on any package specification
or body while in its enclosing context, and the
commands to insert the prologue template above
will be repeated when the key is pressed.
<F1>.}
```

9. Construct a configuration baseline named **V1.2** of the current system. In making this baseline, each source code file in the experiment's code directory should be compared against the latest version already baselined in version **V1.0**; only if the local copy is different (i.e., more up to date) than the already existing CM element shall it be placed into this new system baseline. *Measure time taken to perform the compare operations. Measure time taken to perform baseline operation.*

   {Release all of the subsystems, create Activity V1_2, and create a linked module Product_V1_2.}

```
Cmvc.Release(From_Working_View =>
    "$.[Main, Sm, Cli, As, Vt].rev2_working");

Activity.Create(The_Activity => "V1_2",
                Source => "Aim_V1_2");

{Make !Users.experimenter.Cm_Experiment.V1_2
the current context.}
<Object> I
Insert(Subsystem => "Main",
       Spec_View => "Main_0_Spec",
       Load_View => "Rev2_0_1");
<Object> I
Insert(Subsystem => "Sm",
       Spec_View => "Sm2_0_Spec",
       Load_View => "Rev2_0_1");
<Object> I
Insert(Subsystem => "Cli",
       Spec_View => "Cli_0_Spec",
       Load_View => "Rev2_0_1");
<Object> I
Insert(Subsystem => "Vt",
       Spec_View => "Vt_0_Spec",
       Load_View => "Rev2_0_1");
<Object> I
Insert(Subsystem => "As",
       Spec_View => "As_1_Spec",
       Load_View => "Rev2_1_1");

{Save the changes:}
<Enter>

Set_Default(The_Activity => "V1_2");
```

```
{Make !Users.experimenter.Cm_Experiment
the current context.}
Cmvc.Make_Code_View(
 From_View =>
      "$.[Main.Rev2_0_1, Sm.Rev2_0_1,
      Cli.Rev2_0_1, Vt.Rev2_0_1,
      As.Rev2_1_1]",
 Code_View_Name => "Product_V1_2");

Activity.Create(The_Activity => "Product_V1_2",
                Source => "Aim_V1_2");

{Make !Users.experimenter.Cm_Experiment.
Product_V1_2 the current context.}
<Object> I
Insert(Subsystem => "Main",
       Load_View => "Product_V1_2");
<Object> I
Insert(Subsystem => "Sm",
       Load_View => "Product_V1_2");
<Object> I
Insert(Subsystem => "Cli",
       Load_View => "Product_V1_2");
<Object> I
Insert(Subsystem => "Vt",
       Load_View => "Product_V1_2");
<Object> I
Insert(Subsystem => "As",
       Load_View => "Product_V1_2");

{Save the entries:}
<Enter>

{Set the default Activity to the new
product.}
Set_Default(The_Activity =>
               "Product_V1_2");

{Make !Users.experimenter.Cm_Experiment.
Main.Main_0_Spec the current context, and
select Main'Spec.}
<Promot>
{Observe the creation of an output window;
"Elaborating Viewport_Manager Body",
"It Works!" and "This is the SM Driver."
appear in the window.}
```

2.5. Experiment #2 Functionality Checklist

Primary Activities

Activity	Step #	Supported (Y/N)	Observations
Version Control			
Delete element	7	Yes	Use the Cmvc.Make_Uncontrolled command.
Create new version			
Derived	6	Yes	Create Path, with a different Model.
Retrieve specific version			
Referential	5	Yes	Use the Cmvc.Revert command either with the version number or a negative number to indicate the number of previous versions.
Compare different file versions	3	Yes	Cmvc.Show_History_By_Generations displays the change regions between generations.
Configuration Control			
Build system			
Earlier release	3	Yes	Either make the appropriate views the current context, or if they were destroyed, reconstruct using the Cmvc.Build command.
Hybrid	6	Yes	Use the Cmvc.Accept_Changes command to collect the latest objects among joined working views.

Secondary Activities

Activity	Step #	Supported (Y/N)	Observations
Version Control			
Display history attributes	1	Yes	
Product Release			
Display members of a released system	1	Yes	Display the Activity associated with the release.
Display system release history	1	Yes	Cmvc.Show_History_By_Generations for each of the subsystems in the system.

2.6. Experiment #1 Answers

CM1.1 **Describe the mechanics of constructing a software system. Are automated construction techniques supported (built-in, Makefile, command procedures)?**

First, the system should be partitioned into subsystems. Rational recommends that modules be placed in subsystems such that there is a minimum of dependencies crossing subsystem boundaries. The Cmvc.Initial command sets up the structure for a subsystem. Then Ada units may be copied into or created in the Units subdirectory of the first working view. The Cmvc.Make_Spec_View command is then used to create a view which represents the contents of a subsystem which may be used by other subsystems. Once spec views have been created for all the subsystems, Cmvc.Import must be used to allow an Ada unit in a working view to use an Ada unit in another spec view. Once all dependencies between subsystems have been established using the Cmvc.Import command, an Activity should be created. The Activity is a list of subsystems with their corresponding spec view version and load (or working) view version.

Rational recommends that Ada units, whether spec views or load views, be compiled as soon as possible, in a compile-as-you-go fashion. However, complete system rebuilds may be accomplished in two different ways. Either a Compilation.Promote command listing all the subsystems involved may be issued, or a Compilation.Promote command specifying a main driver module which uses all the modules in all the subsystems may be issued.

The first method was found to be more successful for the instantiation of Configuration Management Experiments #1 and #2. The second method would recompile only the minimum necessary for closure. When parameters were set to compile load views, Ada subunits did not get compiled. When parameters were set to compile "subunits too," only the specifications in the spec views were compiled (and not the contents of the load views).

CM1.2 **Elapsed time for performing a system build operation?**

Forcing a complete system build, by specifying all subsystems to be compiled in the Compilation.Promote command, required 43.50 seconds to compile Rational source code (already formatted) to Rational code in coded state.

Creating coded views (no Diana tree, just machine code) of the five subsystems from the subsystems all in coded state required 85.64 seconds.

CM1.3 **Describe the mechanics of creating a CM element.**

A subsystem must be created with the Cmvc.Initial command. An Ada unit must be placed or created in the Units subdirectory. Cmvc.Make_Controlled places the Ada unit under configuration management.

CM1.4 **Describe the mechanics of constructing a configuration baseline.**

For this experiment, a configuration baseline was created by releasing each of the subsystems with the Cmvc.Release command. This creates a frozen copy of the subsystem, which in turn can be used to generate working views if modifications must be done to the baseline.

The Rational model leaves considerable latitude, and other methods for baselining could be implemented.

CM1.5 **What kind of baselining mechanism is employed?**

For this experiment, baselines were created by creating releases of subsystems. The command to release a subsystem from working view copies portions of the working view's directory structure and its Units subdirectory, and places them under a "frozen" status. A release may be created in configuration-only mode, which causes a copy of only the configuration database. The Ada units can then be reconstructed using the Cmvc.Build command. Configuration-only mode can be used in order to save some space.

Subsequently, working views can be created from the released subsystems, if work must progress from an "old" baseline.

CM1.6 **How are baselines/releases tagged (numeric, alpha, alpha-numeric)?**

A release of a working view of a subsystem can either be named by the user or "auto-generated." The auto-generated name consists of the portion of the view name up to "_Working" followed by "_n_m", where n and m represent automatically incremented level numbers. The user can control how n and m are incremented via the level parameter, or through the model world specified when the subsystem was created.

If a code view of a subsystem is made, its name is entirely determined by the user. Policy can be established to logically name code views of subsystems.

In this experiment, Activities were used to specify which versions of the subsystems formed a system baseline or release. Activities can be given a name of any form. Policy can be established to logically name the Activities.

CM1.7 **Elapsed time for creating a CM file element?**

Elapsed time for placing a file under configuration control (using <Cmvc.Make_Controlled>: average time: 2.13 seconds. The average file size for the files in Rational Source Code state: 3072 bytes (characters).

CM1.8 **Elapsed time for performing baseline operation?**

- Initial baseline for all subsystems - 175.85 sec.
- B02, only VT had to be released - 41.15 sec.
- B03, no release needed - N/A.
- B04, only SM had to be released - 38.13 sec.
- V1.0, all subsystems released - 201.51 sec.
- V1.0, all subsystems configuration-only mode: - 42.86 sec.

CM1.9 **File size increase caused by baseline inclusion?**

Releasing one subsystem, SM, caused an increased disk usage of 100,460 bytes. Releasing all 5 subsystems caused an increased disk usage of 483,609 bytes. Releasing all 5 subsystems in configuration-only mode caused an increased disk usage of only 51,853 bytes.

CM1.10 **How easy/difficult is it to create a CM element?**

Very easy: use Cmvc.Initial to create the subsystem and Cmvc.Make_Controlled to put the Ada unit under version control in the Rev1_Working view Units subdirectory.

CM1.11 **How easy/difficult is it to create a baseline configuration?** In this experiment, using the Delta 0 release of the Rational Environment, creating a

baseline by releasing all the subsystems was easy using the Cmvc.Release command. However, editing an Activity to indicate the versions of the subsystems which form a system release was tedious.

CM1.12 **Are original files removed when a CM file is created?**

No, the "file" or Ada unit actually becomes the controlled element.

CM1.13 **Where are the CM files stored? (separate directory, maintained locally)**

A copy of the Ada unit under configuration control may be kept locally. A copy of the original unit is maintained in the configuration database in the subsystem's State subdirectory in Cmvc_Database. When a user edits the Ada unit, it must be checked out before it is changed. When the user checks the unit back in, the deltas are stored in the Cmvc_Database.

CM1.14 **How are the CM files stored? (text, binary)**

An Ada unit stored in a working view or Released View is stored as a decorated binary tree. The Cmvc_Database maintains a copy plus the deltas between checked in versions in an unreadable binary format.

CM1.15 **Are the CM files delete protected?**

The Ada units stored in a working view under configuration control are protected from deletion. An attempt to delete an Ada unit under configuration control raises a "Policy Error." An Ada unit must be removed from configuration control in a working view (Cmvc.Make_Uncontrolled) before it can be deleted.

The Cmvc_Database stored in the State subdirectory of a subsystem is NOT protected from deletion.

CM1.16 **Describe the mechanics of fetching a CM element.**

If a view of a subsystem has been stored in configuration-only mode, the view may be reconstructed by invoking the Cmvc.Build command with the Configuration parameter set to the "subsystem_name.configurations.view_name." Due to bugs, a dummy blank text file named Release_History will have to be created in the subsystem's State subdirectory. Cmvc_Maintenance.Check_Consistency should be run on the rebuild subsystem also. CM elements in the view's Units directory may then be "checked out" as described below.

If a view of a subsystem has not been reduced to configuration-only mode, then to edit a CM element, make the CM element or Ada unit the current context, and press one key—the <Check Out> key. Also, the Cmvc.Check_Out command can be invoked from a command window:

```
<Create Command>
Cmvc.Check_Out
{Supply name of Ada Unit
as value for parameter
What_Object.}
<Promot>
```

CM1.17 **Describe the mechanics of creating a variant of a CM element.**

A variant version of a CM element is created by making a path or subpath of its subsystem. The Cmvc.Make_Path command, with the Join_Paths parameter set to false, will allow the creation of a separate version of the CM element, which may be accessed simultaneously with the version in the original view. If the Join_Paths parameter remains as the default value of true, changes on the variant version must occur either before or after changes made to the original version.

CM1.18 **Describe the mechanics of fetching a variant of a CM element.**

Either connect to the view or path of the variant version and access the unit as it resides in the Units directory or use the Cmcv.Revert command with the desired generation indicated by the To_Generation parameter.

CM1.19 **Describe the mechanics of reserving a variant of a CM element.**

Make the CM element or Ada unit the current context and press one key—the <Check Out> key. Also, the Cmvc.Check_Out command can be invoked from a command window:

```
<Create Command>
Cmvc.Check_Out
{Supply name of Ada unit
as value for parameter
What_Object.}
<Promot>
```

CM1.20 **Describe the mechanics of replacing a variant of a CM element.**

With the CM element or Ada unit that is to be replaced as the current context, press the <Check In> key. Also, the Cmvc.Check_In command can be invoked from a command window:

```
<Create Command>
Cmvc.Check_In
{Supply name of Ada unit
as value for parameter
What_Object.}
<Promot>
```

CM1.21 **How are CM file versions maintained? (copy, deltas, data compression)**

When the Cmvc.Make_Controlled command is first executed, a copy of the Ada unit is placed in the Cmvc_Database in the State subdirectory of the subsystem. With each "check out" and "check in," deltas are stored in the Cmvc_Database.

CM1.22 **Elapsed time for fetching a CM element.**

Elapsed time for fetch operation (Check Out command): 1.37 seconds.

CM1.23 **Elapsed time for creating a variant of a CM element.**

Elapsed time to create a subpath with Join_Paths parameter set to false:

- Vttester - 22.92 seconds
- Clitester - 23.60 seconds
- Smtester - 22.22 seconds

CM1.24 **Elapsed time for fetching a variant of a CM element.**

Elapsed time for checking out various CM elements within the subsystems:

- Main - 1.00 seconds
- Page_Terminal'Body - 1.12 seconds
- Viewport_Manager'Spec - 1.10 seconds
- Viewport_Manager'Body - 1.16 seconds

CM1.25 **Elapsed time for reserving a variant of a CM element.**

No corresponding operation in the Rational Environment.

CM1.26 **Elapsed time for replacing a variant of a CM element.**

Elapsed time for checking in various CM elements within a subsystem:

- Main - 0.96 seconds
- Page_Terminal'Body - 1.04 seconds
- Viewport_Manager'Spec - 1.03 seconds
- Viewport_Manager'Body - 1.05 seconds

CM1.27 **File size increase caused by successive version?**

Size of change to the logs and the size of the change to the Ada unit are stored in 1024-byte chunks in the configuration database.

CM1.28 **File size increase caused by variant version?**

Should be same as CM1.27, above.

CM1.29 **How easy/difficult is it to fetch/reserve a CM element?**

Very easy: press one key, <Check Out>.

CM1.30 **How easy/difficult is it to replace a CM element?**

Very easy: press <Check In>.

CM1.31 **How easy/difficult is it to create a variant version of a CM element?**

Relatively easy, but must be done on the subsystem level with the Make_Path or Make_Subpath command.

CM1.32 **How are the reasons for version changes recorded? Is this mandatory or optional data collection?**

Comments parameter or through work orders. The data is optional but may be set to mandatory through the use of work orders.

CM1.33 **Can variant be placed into baselines easily?**

Yes, copy the variant to the desired subsystems, and then release the subsystem.

CM1.34 **Are fetching/reserving/replacing variants harder than for non-variant elements?**

No.

CM1.35 **What is the default protection of a fetched CM file element? Is the default reasonable?**

"Fetching" a CM file element is accomplished through reserving tokens in a table. Read and Write access are not used on the Rational to accomplish version control.

CM1.36 **What is the default protection of a reserved CM file element? Is the default reasonable?**

See CM1.35.

CM1.37 **Merging is defined to be the operation of incorporating the changes made in a variant branch of a CM element back into the (current) main trunk version of the original root CM element. Describe the mechanics of merging variant versions of a CM element.**

Cmvc.Merge_Changes allows the merging of Ada units in severed paths. The Cmvc.Merge_Changes command takes the name of the unit to have changes merged into it, and the view name of the object whose changes are to be applied.

The Cmvc.Accept_Changes allows the merging of Ada units in joined paths. The object specified in the Destination parameter is changed to reflect any modifications that have been made to corresponding source objects.

CM1.38 **How well are merge inconsistencies identified?**

Inconsistencies are identified by a text file generated which is the unit name concatenated with "_Merging_Report." Conflicting lines are marked with an asterisk and semi-colon (*;) followed by a number indicating which Ada unit the line came from. The report is very clear.

CM1.39 **How well are merge inconsistencies handled?**

Can edit the inconsistencies, they show up in a resulting Ada unit as lines beginning with "*;" and a number.

CM1.40 **Elapsed time of merging variant versions of a CM element.**

Elapsed time for merging:

- main.a into main: 7.58 seconds
- main.b into main: 6.17 seconds
- main.c into main: 5.40 seconds

Average elapsed time for merging variant versions of a CM element: 6.38 seconds.

CM1.41 **File size increase caused by merge operation.**

- main.a into main: 2572 byte increase
- main.b into main: 5078 byte increase
- main.c into main: 6358 byte increase

Total increase for merging 3 variants back into main: 14008 bytes.

CM1.42 **How easy/difficult is it to merge existing variant versions of a CM file element?**

The Cmvc.Merge_Changes command is easy to use, and the documentation is helpful.

CM1.43 **Describe mechanics of reserving a CM element.**

Make the unit to be checked out the current context by traversing the subsystem directory structure with the <Definition> and cursor movement keys, and press the <Check Out> key.

CM1.44 **Describe mechanics of replacing a CM element.**

See CM1.24.

CM1.45 **Elapsed time of reserving a CM element.**

See CM1.26.

CM1.46 **Elapsed time of replacing a CM element.**

Timings of <Check In>.

CM1.47 **How easily do the generic experiments map onto environment** operations?

Easily mapped, except that the experiment's concept of a "subsystem" differs from the Rational model's concept of a "subsystem." Rational advocates that subsystems be defined so that there are few dependencies across subsystem boundaries.

CM1.48 **How are CM files referenced? (local name, CM file name)**

"File Names" as well as "CM files" are maintained by the system and default to the Ada unit name.

CM1.49 **Describe error handling capabilities and error diagnostics.**

The error messages indicate that the CM/VC commands are actually implemented through other system-level commands, and command execution usually involves a long stream of messages to an execution window. Embedded in these long streams are error messages indicated by leading asterisks, and sometimes they are not indicative of the problem.

CM1.50 **Describe the command syntax. Awkward? Easy to learn and use?**

Very consistent with the entire user interface except in one area: the Activity Editor is not a "full screen" editor as are the Ada Editor and Text Editor.

The single keystrokes for Check In and Check Out, and the commands that would be used in the day-to-day development and maintenance of a system are easy to learn and use.

CM1.51 **Multiple views of a software system in this context is defined to be the capability of representing the software system from differing perspectives (e.g., functional subsystems versus baselined product class). Are multiple views of a product supported? Is concurrent use supported?**

Yes, many different working views of subsystems can exist at the same time. The model allows different activities to pick up different sets of released and working views. Multiple views and concurrent use are both supported. The model allows for great flexibility, or policies can be established and enforced.

CM1.52 **Is the CM capability integrated into the compilation system?**

Yes, a given Ada unit exists in source, installed, or coded state and can be checked in or checked out in any of these states, although it is recommended that a successful compilation (or coded state) be reached before the Ada unit is checked in.

CM1.53 **Describe Ada filename syntax.**

Filenames are maintained by the system and are the Ada unit name. Any valid Ada unit name is a valid "Ada filename."

CM1.54 **Does all source code have to be in the same directory or is a hierarchical project structure supported?**

No, source code can be maintained in several subsystems, or a hierarchical project structure can be supported by subdirectories residing under the units directory of a subsystem. In the Delta 0 release, a hierarchy of subsystems is not supported.

CM1.55 **What mechanism is used for sharing program libraries?**

Links and imports (which are actually built on top of and used to maintain links among subsystems).

CM1.56 **Can a package spec and body be separated in different program libraries?**

No. However, a copy of the spec goes into the spec view to allow for minimal recompilation and provide for "firewalling" between subsystems.

CM1.57 **What intermediate files are generated by the compilation system?**

None, an Ada unit moves between states, as the Diana Tree becomes more or less decorated.

2.7. Experiment #2 Answers

Question	Response
CM2.1	**Describe the mechanics of displaying history information for a CM** element. There are a number of Show commands: • Show • Show_All_Checked_Out • Show_All_Controlled • Show_All_Uncontrolled • Show_Checked_Out_By_User • Show_Checked_Out_In_View • Show_History, Show_History_By_Generations • Show_Image_Of_Generation • Show_Out_Of_Date_Objects All of these are invoked from a command window.
CM2.2	**Elapsed time for displaying history information for a CM element.** The elapsed time for Show_History_By_Generations for all the Ada units was 33.03 seconds. This is approximately 2 seconds for each unit.
CM2.3	**Describe the mechanics of rebuilding an earlier baselined system.** Invoke the Cmvc.Build command with the parameter configuration set to the subsystem name concatenated with ".Configurations." and the view name. Due to bugs in the configuration-only mode and the Build command, a dummy text file named Release_History must be created in the State directory. It is also advised to invoke Cmvc_Maintenance.Check_Consistency on the view. The bug also affects imports that are lost between subsystems and must be respecified. The units in the subsystems must also be recompiled.
CM2.4	**Elapsed time for rebuilding an earlier baselined system.** From a configuration-only mode, the Cmvc.Build command required 133.81 seconds; and 24.10 seconds elapsed for a recompilation of the system across the five subsystems. Rebuilding the earlier baselined system required a total elapsed time of 157.91 seconds.
CM2.5	**Describe mechanics of fetching a CM element.** Refer to answer CM1.16.
CM2.6	**Elapsed time for fetching a CM element.** Refer to answer CM1.22.
CM2.7	**Describe mechanics of deleting a CM element.** Invoke the Cmvc.Make_Uncontrolled procedure with the parameter What_Object set to the name of the Ada unit to be deleted. Then invoke the Library.Delete procedure with the parameter Existing set to the name of the Ada unit to be deleted.
CM2.8	**Elapsed time for deleting a CM element.**

Elapsed time to make an Ada unit uncontrolled: 1.71 seconds.

CM2.9 **Describe mechanics of comparing different versions of a CM element.**

The Show_History_By_Generations procedure provides a list of change regions between versions.

CM2.10 **Elapsed time for comparing different versions of a CM element.**

Not applicable.

2.8. Configuration Management/Version Control (CM/VC) Analysis

Although the problems are many in number, they do not represent a serious performance problem for the Delta 0 Release of the configuration management and version control system. Rational Customer Support was very prompt in acknowledging the problems and suggesting workarounds.

2.8.1. Functionality

The instantiation of Experiments #1 and #2 reflects one workaround required by a bug in the system: A combined load view could not be released. When releasing combined load views, which contain elements that depend on each other, the Release command goes into an infinite loop. The transcripts for Experiments #1 and #2 also reflect some workarounds for other bugs present in the Delta 0 Release of the CM/VC software. First, when a spec view is created from a load view, a parameter Goal defaults to compilation state Coded. The compilation of specifications to coded state does not occur when the spec view is created. As such, the user must go to the newly created spec view working units subdirectory and compile these units as a separate step. Second, due to a bug in the creation of a subsystem release in configuration-only mode and the Build command, the imports defined between spec views and load views were lost. When the Build command attempted to recompile the system to coded state, it failed. This left the released subsystems that were supposed to be frozen in an unfrozen state. Several steps must be taken to reestablish the lost import information. Also, each view loses its State.Release_History text file. As such, copies of the views cannot be made by the Make_Path or Make_SubPath commands. The work around consists of visiting the State subdirectory and editing in a blank Release_History text file. The loss of this file represents a potential loss of information if comments are provided to the Release command.

The functionality presented by the configuration management and version control system provides all of the "Primary Activities" and "Secondary Activities" outlined in the functionality checklists for Experiments #1 and #2. Specifying translation rules as in a Unix Makefile does not apply to the Rational Environment compilation strategy. Translation rules and order are maintained automatically as interdependency information by the directory structure of the Rational Environment. As such, translation rules are not needed. Translation tools do not need to be specified, since one command procedure, Compilation.Promote, with the proper parameters handles translation. The Rational Environment provides a very complete set of CM/VC functionality as detailed by the *SEI Evaluation Method* for the development, maintenance, and release of software.

2.8.2. Performance

From the perspective of a programmer using the CM/VC system, the most commonly used commands, such as "Check In" and "Check Out," are fast and easy to use interactively. System response for these common commands generally only required about one second of elapsed time. Some of the less common commands, which would probably be invoked only by project leaders or managers, required more time. The elapsed time for performing a baseline operation across all five subsystems required over two minutes. When the baseline was released in configuration-only mode, which would require that it be rebuilt later if needed, the elapsed time required was under a minute. The time required for releasing a system would increase with the size (number of lines of Ada code) of the system. For the system presented by the experiments, an initial system build required a total elapsed time of slightly over two minutes.

The initial creation of an empty subsystem consumed 105,814 bytes. Constructing releases, paths, and subpaths of a subsystem is by its nature also an expensive operation, as almost the entire subsystem directory structure is copied. A disk utilization increase of about 100,000 bytes was noted for each subsystem when a release was made. A subsystem may exist in configuration-only mode, which does return some disk space to the file system. A return of almost 80,000 bytes of available storage was noted when one view of a subsystem was destroyed and left in configuration-only mode. (Configuration-only mode would allow the subsystem to be rebuilt if needed.)

2.8.3. User Interface

The user interface to the configuration management and version control commands is the same interface as that used by all system commands. The commands are actually calls to Ada procedures. The procedures are invoked from a command window, which provides context and a begin-end block in which to insert the command. Command completion and the presentation of default parameters are very useful in these procedures. Although the configuration management and version control commands have the bugs previously noted and the problems described in the following, the most commonly invoked commands are easy to use and bound to the program function keys.

Version control is only available in the context of the subsystem structure. Objects placed in a subsystem must be controlled explicitly when they are first placed in the subsystem. Subsequent copies of the view created through the Make_Path or Make_SubPath will continue the object as a controlled object. However, if access to the objects is not controlled, they are lost if the subsystem is ever reduced to configuration-only mode. In this case, a simple oversight on the part of the user can have disastrous results over the life cycle of a subsystem.

The editor provided for creating and changing Activities differs from the whole screen editor provided for Ada Objects or command windows. The Activity Editor actually involves providing parameters to procedures such as Activity.Add and Activity.Change to add or change the contents of an Activity. It also requires pressing <Enter> to save changes. If this is not done, changes are not saved, which often leads to unexpected performance. The Activity.Create command requires the value of parameter The_Activity to be supplied as a text string. The prompt for this value supplies quotes that persist once the user has typed in a string. The optional

parameter Source requires the same type of value as The_Activity, yet the user is not provided with quotes that persist once the user has typed in a string. Also, if quotes are not typed, then the resulting error message is misleading:

```
Parameter list (THE_ACTIVITY => "B0_2", SOURCE => B0_1,
MODE => ACTIVITY.EXACT_COPY, ...) is invalid.
```

This is accompanied by an underscore appearing in the command window after "Activity.Create" and underneath the parameter value B0_1.

View names cannot have underscores. This seems inconsistent since the view name is concatenated with "_Working" and used as a directory name. Directories that are created using the regular directory creation commands do not have this restriction and can use any characters permitted in an Ada name. The restriction stems from the inability of the Build command to reconstruct a view with extra underscores in the name.

The configuration management and version control packages were released for the first time in the Rational Environment Delta 0 Release. The bugs encountered and the problems with the user interface reflect the immaturity of these packages.

2.8.4. System Interface

The configuration management and version control facility is completely integrated with the editor, browser, and compiler of the Rational Environment. It does not make use of access control facilities. The user is expected to construct "skins" which tie together the work order management, configuration management and version control, and access control facilities to reflect user policy for the system being created, tested, or maintained.

3. System Management Experiments

3.1. Introduction

System Management Experiment #1 investigates the procedures supporting the installation of an Ada software environment. The experiment requires the collection of information about loading the software from release media, integrating the software with the underlying operating environment, and exercising the installed environment.

The R1000 computer, the Rational Environment for Ada development, and a Rational service contract are a package. (Although the service contract is a separately priced item, Rational strongly recommends it.) As part of the service contract, Rational technicians perform all system software installation and updating. Also, the software does not need to be integrated with an underlying operating environment, as in a more traditional APSE. For these reasons, and because the other experiments exercise the installed environment, Experiment #1 was not conducted on the Rational Environment.

System Management Experiment #2 investigates the support of user account management activities. The operations of creating, deleting, modifying, copying, displaying, and verifying user account information are outlined in the experiment and instantiation in Section 3.2. Because the only system management attributes associated with an account are account name, password, and user group, some of the steps in the experiment are not applicable to the Rational Environment. However, many attributes and operations for work space management can be associated with an individual account by the Rational Environment, including key bindings, macro definitions, session switches, and so forth. However, these attributes and operations roughly correspond to the workspace customization that can be performed by a VAX/VMS login file, not the system management attributes required by this experiment.

System Management Experiment #3 does not contain individual steps and data collection. It is an assimilation of questions that address the issues of maintaining current releases of the Ada environment software, customer support and service, and archiving (and subsequently retrieving) the Ada environment software and/or database elements. These questions are presented and answered in Section 3.6.

System Management Experiment #4 investigates the procedure for the collection of accounting statistics. The experiment addresses the issues of monitoring the system's performance and collecting specific accounting information: CPU usage, disk space usage, connect time, and number of pages printed. The experiment requires the development of procedures, one to automate the collecting of system accounting statistics and one to facilitate dynamic, continuous monitoring of system's performance. Neither of these procedures had to be written for the experiment, since the Rational Environment supplies this functionality, as detailed in Section 3.4.

In the following instantiation of the experiments, specific keys to be pressed are denoted by the lettering on the key or key map designation enclosed in angle brackets (< >.) The first time a command is presented in text, all its parameters are detailed. Parameters that must be supplied

or changed are printed in italics. Subsequent uses of the command include only required parameters and those that differ from the default. Familiarity with creating and executing a command, selecting an object, traversing objects in a window, and moving between windows is assumed. Any variation in reporting the experiment step is noted.

3.2. Experiment #2

1. Experiment setup

    a. Log in to environment as the system administrator.

    {The R1000 does not have system administrator accounts. Members of group Operator (user name Operator) and users with write access to file *!Machine.Operator_Capability* can perform operations within the Environment that require *operator capability*. Log in to an account with operator capability, in this case **Operator**. System prompts are in bold type and the user response is in italics.}

    Name: *Operator <return>*
    Password: {enter the password} *<return>*
    Session: *<return>*

    b. Create subdirectory in which experimental results will be stored.

    {Results gathered by using the *timeit* and *record_size* procedures are reported to a standard output window. These programs are listed in Appendix A with an explanation of their use.}

    c. Establish environmental variables to be used in the experiment.

    {None are used.}

2. Create environment user account group name **ENV_USER**. *Measure time taken to create new user account group. Record file size increase caused by creating a new user account group.*

```
Create_Group(Group => "Env_User",
             Response => "<PROFILE>");
```

{Note: Access control groups are maintained in a directory *!Machine.Groups*, this is what should be measured for the increase in size caused by creating a new user account group.}

3. Create environment user account for John T. Smith; assume the last name is to be used for the user name, password, and pathname of the account's home directory. *Measure time taken to create new user account. Record increase in file size caused by creating a new user account.*

{The R1000 environment automatically assigns the path or context "!Users.*Name*" to the home directory of a new user *Name*.}

```
Create_User(User => "Smith",
            Password => "Smith",
            Volume => 0,
            Response => "<PROFILE>");
```

4. Add user **Smith** to user group **ENV_USER**. *Measure time taken to add new user to an account group. Record increase in file size caused by adding new user to an account group.*

```
Add_To_Group(User => "Smith",
             Group => "Env_User",
             Response => "<PROFILE>");
```

5. Copy **Smith** account characteristics into a new account for Thomas R. Jones; assume the last name is to be used for the user name, password, and pathname of the account's home directory. *Measure time taken to copy characteristics into a new user account. Record increase in file size caused by creating a new user account.*

   {The Rational environment has no provision for copying accounts. Create user Jones as in step SM2.3 and SM2.4.}

```
Create_User( User => "Jones",
             Password => "Jones");

Add_To_Group(User => "Smith",
             Group => "Env_User",
             Response => "<PROFILE>");
```

6. Copy *Smith* account characteristics into a default account named **DEFAULT** to be used in the future for creating new environment accounts. *Measure time taken to copy characteristics into a new user account. Record file size increase caused by creating a new user account.*

   {The only default account characteristic in the Rational Environment is the password. Access to certain groups is added later. Below are the details showing how to create an Ada program *Create_Default_Account*. When this program is run from an account with operator capability, with no user name provided, it will create a user **DEFAULT** with password "*default*" and place the user in group **ENV_USER**.}

```
<Create Ada>
Create_Default_Account
<Promot>
{Enter the following in the edit window,
which opens:}
with Operator;
procedure Create_Default_Account
                    (Name :String =: "Default") is
begin
Operator.Create_User(User => Name,
            Password => Name,
            Volume => 0,
            Response => "<PROFILE>");
Operator.Add_To_Group(User => Name,
             Group => "Env_User,
             Response => "<PROFILE>");
end Create_Default_Account;

{Type the following to compile and store the
program:}
<Promot>
<Promot>
<Enclosing>
```

   {Create default account **DEFAULT** by executing the procedure just created.}

```
{Check that Create_Default_Account'body is selected.}
<Promot>
```

7. Disable logins for the **DEFAULT** account. *Measure time taken to disable logins for an account.*

```
Delete_User(User => "DEFAULT",
            Response => "<PROFILE>");
```

8. Display characteristics of the **DEFAULT** account. *Measure time taken to display account characteristics.*

   {Cannot display account characteristics of a disabled account.}

9. Change account name of the **DEFAULT** account to be **ENV_USER**. *Measure time taken to modify one characteristic of a user account. Record increase in file size caused by modifying a characteristic of a user account.*

   {No procedure provided to change an account name.}

10. Display characteristics of the **DEFAULT** account. *Measure time taken to display account characteristics.*

    {See step 8.}

11. Modify account names as above (step 9) for the **Smith** and **Jones** accounts. *Measure time taken to modify one characteristic of a user account. Record file size increase caused by modifying a characteristic of a user account.*

    {One account characteristic which can be modified is the password.}

```
Change_Password(User => "Smith",
                Old_Password => "Smith",
                New_Password => "newsmith",
                Response => "<PROFILE>");

Change_Password(User => "Jones",
                Old_Password => "Jones",
                New_Password => "newjones");
```

12. Display characteristics of the **Smith** and **Jones** accounts. *Measure time taken to display account characteristics.*

```
Display_Group(Group => "Smith",
              Response => "<PROFILE>");
Display_Group(Group => "Jones");
```

13. Create an account for **Jane Doe** using characteristics from the **DEFAULT** account; assume the last name is to used for the user name, password, and pathname of the account's home directory. *Measure time taken to copy characteristics into a new user account. Record file size increase caused by creating a new user account.*

```
Create_Default_Account(Name => "Doe");
```

14. Create working directories containing login/logout command procedures for the **Smith**, **Doe**, and **Jones** accounts. *Measure time taken to create initial account directories.*

    {The Rational Environment command that creates accounts automatically creates a user home directory with the pathname *!User.user name*. This was performed in steps 3, 5 and 13.}

15. Update any environment-specific databases to grant **Smith**, **Doe**, and **Jones** access to the environment software.

    {Since the Rational Environment is not a system layered on top of a traditional environment, commands to create a user account automatically provide access to the environment. Steps 3, 5 and 13 have already accomplished this.}

16. Verify the creation and correctness of the **Smith**, **Doe**, and **Jones** accounts (e.g., login and edit a text file from these accounts).

```
{Log off the R1000 account Operator}
<Home Library>
<Create Command>
Quit
<Promot>
{Log in as Smith. System prompts are
in bold type, and user response is
in italics.}
User:  Smith <Return>
Password:  Smith <Return>
Session:  <Return>

{Create a text file.}
<Create Text>
My_file
{In edit window, which opens, type:}
This is a text file.
{Store the text file.}
<Enter>

{After My_file appears in the Smith home
directory, log out.}
<Create Command>
Quit
<Promot>

{Repeat the above steps for the accounts
Doe (password: Doe) and Jones
(password:  newjones).}
```

17. Revoke environment access from the **Jones** account. *Measure time taken to revoke environment access from a user's account.*

```
Delete_User(User => "Jones");
```

18. Remove **Jones** account from the **ENV_USER** account group. *Measure time taken to remove user from an account group. Record decrease in file size caused by removing user from an account group.*

```
Remove_From_Group(User => "Jones",
                  Group => "Env_User",
                  Response => "<PROFILE>");
```

19. Remove **Jones** account. *Measure time taken to remove user account. Record decrease in file size caused by removing user account.*

```
Library.Destroy(Existing => "!Users.Jones??",
                Threshold => 1,
                Limit => "<DIRECTORIES>",
                Response => "<PROFILE>");
```

3.3. Experiment #2 Functionality Checklist

Activity	Step #	Supported (Y/N)	Observations
User Account Management			
Create user account	3,13	Yes	
Delete user account	19	Yes	

Copy user accounts	5	No	
Add user account to group	4	Yes	
Delete user account from group	18	Yes	
Establish user account characteristics	3,5,13	Yes	Only system management characteristic is the account password. Membership in a user group is added later.
Modify user account chars	11	Yes	Password and group membership.
Establish default account chars	6	No	Can create a program to build default account.
Modify default account chars	9	No	
Display user account chars	12	Yes	Can display group membership.
Display default account chars	8,10	No	
Create initial working directories	14	Yes	Default working directories are created automatically by the create user account procedure.
Establish default login/logout macros	14	No	Default system wide login procedures (which can be changed) are automatically used for all users who do not override them with procedures in their home directory.
Verify creation of user accounts	16	Yes	

3.4. Experiment #4

1. Experiment setup

    a. Log in to underlying operating system as the system administrator.

    {The Rational Environment has no system administrator account. Therefore, log in as an ordinary user and acquire operator capability.}

    b. Create subdirectory in which experimental results will be stored.

    {As will be seen, this experiment generates no results to be logged.}

    c. Establish environment variables to be used in the experiment.

    {None are used.}

2. Establish default access control to restrict non-privileged users' access to all command files and log files to be used for System Management activities.

    {The Delta Release of the Rational Environment already restricts access of non-privileged users to system management commands by requiring operator capability for certain commands.}

3. Create a subdirectory named BILLINGS under the system root directory to house environment accounting statistics.

    {The directory name for accounting information is hardwired into the Rational Environment as **!Machine.Accounting**.}

4. Initially, enable the logging of environment accounting information. Measure time taken to enable logging of accounting information.

- CPU usage
- Connect time
- Disk usage
- Number of logins
- I/O activity
- Pages printed

{Usage logging in the Rational Environment is always enabled. Usage accounting is monolithic; there is no separate enabling of accounting for different resources.}

5. Write a command procedure to automate the monthly collection of accounting information to be used by a billing program, assuming logging is already enabled. Measure time taken to disable system logging. Record size of system accounting log file.

   {The current accounting file cannot be renamed, moved, or copied while the system has a lock on it. To remove the lock, the system must be shut down and rebooted (which creates a new file in !Machine.Accounting with the date of reboot included in the file name). Thus, the only step performed is b.}

   a. Disable the logging of environment accounting information.

      {Have operator reboot the system at midnight at the end of the month to close the old accounting file.}

   b. Rename current accounting log file to a file of the form: mmmddyy.LOG where

      mmm - previous month (e.g., Jan)

      dd - last day of previous month (e.g., 31)

      yy - year of the previous month (e.g., 88)

```
<Create Command>
"Library.Rename"
<Complt>
Library.Rename(From => "<SELECTION>",
               To => ">>NEW SIMPLE NAME<<",
               Response => "<PROFILE>");

{Supply as value to parameter From:}
"Activity_88_01_09_At_11_16_14"
<Next Item>
{Supply as value to parameter To:}
"Jan3188_Log"
<Promot>
```

      {If the system has been rebooted more than once in a time period, then multiple files for the time period will be generated and must be concatenated to generate one file for the period.}

   c. Re-enable the logging of environment information.

      {Logging was enabled for the new time period when the system was rebooted in step 5.a.}

6. Write a command procedure to continuously monitor the system's performance (i.e., number of processes currently active, CPU usage per process, physical memory user per process, program image running under process, page faults, etc.).

{This procedure already exists in the Rational Environment.}

```
<Create Command>
{Command window opens, and enter:}
"What.Jobs"
<Complt>
What.Jobs(Interval => 10,
          User_Jobs_Only => False,
          My_Jobs_Only => False,
          Running_Jobs_Only => True);
{Use all the default values.}
<Promot>
{System monitoring begins.}
```

3.5. Experiment #2 Answers

This section has been truncated to eliminate those evaluative questions for which the R1000 lacks required capabilities.

Question	Response
SM2.1	**Describe the mechanics of creating a user account group.**
	Create a command window and enter the command:

```
Create_Group(Group => ">>GROUP NAME<<";
             Response => "<PROFILE>");
```

Supply the desired group name as the value for the parameter Group.

Execution of this procedure requires that the executing job have operator capability.

SM2.2 **Elapsed time for creating a user account group?**

```
Wall Clock Time:  2.49 seconds
CPU  Time:        0.46 seconds
```

SM2.3 **File size increase caused by creating a user account group?**

!Machine.Groups increased by 1 byte after adding a user account group.

SM2.4 **Describe the mechanics of creating a user account.**

Create a command window, and enter the command:

```
Create_User(User => ">>USER NAME<<";
            Password => "";
            Volume => 0;
            Response => "<PROFILE>");
```

Supply the desired user name as the value for the parameter User, and if an initial password is desired, enter it as the value for the parameter Password.

Execution of this procedure requires that the executing job have operator capability.

SM2.5 **Elapsed time for creating a new user account?**

```
Wall Clock Time: 6.48 seconds
CPU Time:        4.24 seconds
```

SM2.6 **File size increase caused by creating a new user account?**

File increase results from creation of an entry in the world !Machine.Users and creation of a home world for the user. File size increased by 7473 bytes.

There was no direct correlation with the length of the user name or password.

SM2.7 **How easy/difficult is it to create a new user account?**

Creating a new user account is straightforward since only one procedure, !Commands.Operator.Create_User is called from a Rational command window.

SM2.8 **Describe (in detail) the user account information maintained.**

User account name, user password, and user account group membership.

SM2.9 **What are the resource requirements of an environment user?**

Since the Rational Environment is not layered on top of a traditional operating system, this question is not applicable.

SM2.10 **What privileges are necessary for an environment user?**

There are no additional privileges required by the user, other than access to his home world (i.e., the account login is not disabled).

SM2.11 **Describe mechanics of adding a user account to a user group.**

Create a command window and enter the command:

```
Add_To_Group(User => ">>USER NAME<<";
             Group => ">>GROUP NAME<<";
             Response => "<PROFILE>");
```

Supply the desired user name as the value for the parameter User, and supply the desired group name as the value for parameter Group.

Identities are established at login. Adding a user to a group will not be effective until the user's next login. The user must log out and then log in again for the new group membership to be added to the user's identity.

Execution of this procedure requires that the executing job have operator capability.

SM2.12 **Elapsed time for adding a user account to a user group?**

```
Wall Clock Time: 0.07 seconds
CPU Time       : 0.05 seconds
```

SM2.13 **File size increase caused by adding a user account to a user group?**

!Machine.Groups.Env_User increases by 1 byte. !Machine.Groups.*User_Name* increases by 1 byte. Any other storage costs could not be measured.

SM2.14 **How easy/difficult is it to add a user account to a user group?**

It is very easy to add a user account to a user group. It only requires one command, and standard environment techniques (such as window creation and command completion) facilitate entering the command.

SM2.15 **Describe the mechanics copying old account characteristics into a new account.**

The Rational Environment does not support copying of old account characteristics into a new account. However, a program can be developed using the Operator commands to create an account, give it a password, and add it to certain user groups. See System Management Experiment #2, Steps 6 and 13, for an example of the creation and use of a default account creation program.

SM2.16 - SM2.18 Accounts cannot be copied.

SM2.19 **Describe the mechanics of disabling logins for a user account.**

Make sure that the user is logged out before disabling the user's account. Create a command window and enter the command:

```
Delete_User(User => ">>USER NAME<<";
            Response => "<PROFILE>");
```

Supply the desired user name as the value for the parameter User.

Execution of this procedure requires that the executing job have operator capability.

SM2.20 **Elapsed time for disabling logins for a user account?**

```
Wall Clock Time: 1.77 seconds
CPU Time       : 0.89 seconds
```

SM2.21 **File size increase caused by disabling logins for a user account.**

There is no change in file size when logins to a user account are disabled.

SM2.22 **How easy/difficult is it to disable logins for a user account?**

It is very easy to disable logins to a user account. It only requires one command, and standard environment techniques (such as window creation, command completion) facilitate entering the command.

SM2.23 **Describe mechanics of displaying user account characteristics.**

The Rational Environment supports a limited number of "account characteristics"; see Question SM2.8.)

It is not possible to display a user's password. To display the user groups to which a user belongs, create a command window and enter the command:

```
Display_Group(Group => ">>GROUP NAME<<";
              Response => "<PROFILE>");
```

Supply the desired user name as the value for the parameter User.

SM2.24 **Elapsed time for displaying user account characteristics?**

To display the user groups a user belongs to:

```
Wall Clock Time: 0.03 seconds
CPU Time       : 0.02 seconds
```

SM2.25 **How easy/difficult is it to display user account characteristics?**

The only characteristic that can be displayed is the user groups to which a user belongs. This is very easy to display, requiring only one command. Standard environment techniques (such as window creation and command completion) facilitate entering the command.

SM2.26 **Describe the mechanics of modifying a user account's characteristics.**

The password of a user account can be changed by creating a command window and entering the command:

```
Change_Password(User => ">>USER NAME<<";
                Old_Password => "";
                New_Password => "";
                Response => "<PROFILE>";
```

See SM2.33 for removing a user from a user group and SM2.11 for adding a user to a user group.

SM2.27 **Elapsed time for modifying a user account's characteristics.**

To change a user password:

```
Wall Clock Time: 0.03 seconds
CPU Time       : 0.02 seconds
```

See SM2.34 for elapsed time of removing a user from a user group, and see SM2.12 for elapsed time of adding a user to a user group.

SM2.28 **File size increase caused by modifying a user account's** characteristics?

There is no change in file size for changing a user's password.

See SM2.35 for file size change caused by removing a user from a user group. See SM2.13 above for file size change caused by adding a user too a user group.

SM2.29 **How easy/difficult is it to modify an existing user account's** characteristics?

It is very easy to change a user password. It only requires one command and standard environment techniques (such as window creation and command completion) facilitate entering the command.

See Questions SM2.14 and SM2.36 for ease/difficulty of adding or deleting a user from a user group.

SM2.30 **Describe the mechanism for creating a home directory for a new user account.**

The Rational Environment command that creates accounts automatically creates a user home directory with the pathname *!User.user name*. See Question SM2.4 for mechanics of creating a user account.

SM2.31 **What is the default protection for a user's home directory for a new user account. Is this default modifiable? If so, by whom? (user alone, user and system administrator) Is this default protection reasonable?**

The Rational Environment does not set up protections for a user's home directory. When a user name is created, a group name corresponding to that user name is created by default. The user name is, by default, a member of that group, as well as a member of group **Public** and **Network_Public**. By default, other users have access to read, modify, and change the state of Ada objects in another user's home directory or subdirectories; however, other users do not have the ability to delete objects in another user's home directory. The read and edit capabilities are by virtue of the users' belonging to the **Public** group. This default can be modified by removing a user from the **Public** and **Network_Public** groups. The user can remove his user name from groups. Addition to a group requires operator capability. The default protection, which allows a user to modify objects in another user's directory, does not seem reasonable.

SM2.32 **What mechanism, if any, is employed to restrict access to the environment software? How can a user gain access to environment software.**

The Rational Environment software is provided as executable code on the system only, so access to modify or view is not available.

SM2.33 **Describe the mechanics of removing a user account to a user group.**

Create a command window and enter the command:

```
Remove_From_Group(User => ">>USER NAME<<";
                  Group => ">>GROUP NAME<<";
                  Response => "<PROFILE>");
```

Supply the desired user name as the value for the parameter User. Supply the desired group name for the value of the parameter Group.

Execution of this procedure requires that the executing job have operator capability.

SM2.34 **Elapsed time for removing a user account to a user group?**

```
Wall Clock Time: 0.11 seconds
CPU Time       : 0.04 seconds
```

SM2.35 **File size decrease caused by removing a user account to a user group?**

There is a 1-byte decrease in file size.

SM2.36 **How easy/difficult is it to remove a user account from an account group?**

It is very easy to remove a user account from an account group. It only requires one command, and standard environment techniques (such as window creation and command completion) facilitate entering the command.

SM2.37 **Describe the mechanics of deleting a user account?**

The procedure !Commands.Operator.Delete_User is called from a command window and disables login to a user account, but preserves the user's home world. The procedure Library.Destroy removes the user's home world.

SM2.38 **Elapsed time for deleting a user account?**

See Question SM2.20 for elapsed time to disable logins for a user account.

The elapsed time to execute the Library.Destroy command:

```
Wall Clock Time: 1.10 seconds
CPU Time:        0.49 seconds
```

SM2.39 **File size decrease caused by deleting a user account?**

No file size decrease was seen when disabling logins, and a 7556-byte decrease was seen when the user's home world was removed.

SM2.40 **How easy/difficult is it to delete a user account?**

Deleting a user account is straightforward. It involves calling the Operator.Delete_User procedure and Library.Destroy commands from a command window.

SM2.41 **How easy/difficult is it to learn the command syntax of the user account manager utility?**

The Rational Environment has no "user account manager" utility. Either a user account (called "Operator") with operator capabiliy can be created, or any user can be granted operator capability. For some commands to execute, the executing job must have operator capability. Because the syntax of these commands is consistent with other Rational Environment commands, the system management commands are easy to use.

SM2.42 **Is the user interface of the user account manager utility consistent to those of similar tools?**

The system management commands are very consistent with the rest of the Rational Environment commands.

SM2.43 **How useful are the error diagnostics of the user account manager utility?**

The error diagnostics are good, and they are consistent with error diagnostics present for other commands.

SM2.44 **How well is the user account manager utility documented? Is there an online help facility?**

Those commands requiring operator capability are documented mostly in the System Management Utilities volume of the Rational Environment reference manuals. The commands are well documented.

Online help is available for commands in the !Commands.Operator package.

SM2.45 **How well is the user account manager utility integrated into the underlying operating environment?**

The Rational Environment is not layered on top of any operating environment; as such, operator capability is a part of the Rational Environment.

SM2.46 **Describe the operating system's protection scheme. Protection masks? Access control lists?**

The Rational Environment is the operating system, and its protection scheme is access control lists.

SM2.47 **Describe the environment's protection scheme. Does the environment offer any more or less protection than the underlying operating system? If so, describe the differences.**

See Question SM2.46.

SM2.48 **How and where is user account information maintained? (text file, binary file)**

User account information is maintained as an entry in the !Machine.Users directory.

SM2.49 **What kind of scheme is used to protect account information?**

A person changing the user account information must have the operator capability.

SM2.50 **Can passwords be displayed in humanly readable format?**

No.

SM2.51 **What, if any, unique account attributes are available? Disallow changing of an account's password. Restricted system access based on current date and time? System-generated passwords? Automatic password expiration after a setable time period?**

Accounts have passwords, and a user may belong to certain user groups; these are the only account attributes available. There is no way to disallow the changing of an account's password. There is no way to restrict system access based on current date and time. There are no system-generated passwords. There is no capability for automatic password expiration.

SM2.52 **Is a command procedure provided for creating new user accounts?**

Yes, the Ada procedure Operator.Create_User creates new user accounts.

SM2.53 **Is a command procedure provided for removing user accounts?**

Yes, the Ada procedure Operator.Delete_User deletes user accounts.

3.6. Experiment #3 Answers

Question	Response

SM3.1 **What is the overall process for updating the environment software?**

Rational technical personnel perform all updates of system software. The form of software update provided up to this time has been a complete new release of the system software.

SM3.2 **How frequent are new software releases?**

Rational has marketed its system for about three years at the time of this evaluation. During this time there have been four releases, the original and three updates. This may imply that releases will occur a little more frequently than annually, but it is too early to assume that in the future they will continue to be as frequent.

SM3.3 **Are new releases accompanied by release notes? Updating procedures?**

The Rational Environment Delta Release was accompanied by a complete new set of manuals for the Environment. There are online release notes and a message describing how to read and print them upon login.

SM3.4 **Are new releases downward compatible? Are new releases upwards compatible, or do they supersede all previous releases?**

The Delta Release required some software that used some Rational-specific options to be rewritten. All Ada programs had to be recompiled, once the Delta Release was installed. The degree of compatibility between releases depends upon the features of the new release and is documented by the release notes. All new releases supercede previous releases.

SM3.5 **Can a new release be installed within a multi-user environment or must the machine be in the single user mode?**

See SM3.1.

SM3.6 **Can multiple versions of the environment be running simultaneously?**

No.

SM3.7 **What is the procedure for fixing bugs that are uncovered between releases? (object code patches, new object code, entirely new software release)**

Bugs in the kernel of the operating system are fixed with new system software releases. Changes can be made to the Environment procedures that appear in the Rational directory structure by recompiling the Ada units involved. If bugs are found in this part of the Environment, Rational technicians will make required changes and recompile that part of the system.

SM3.8 **Is patching of executable images supported? If so, is it facilitated via a command procedure?**

No. As noted above, some parts of the OS can be recompiled.

SM3.9 **Can patches be applied within a multi-user environment or must the machine be in single-user mode?**

See SM3.1.

SM3.10 **How easy/difficult is it to update the environment software?**

See SM3.1.

SM3.11 **How much human intervention is required during the updating** procedure?

See SM3.1.

SM3.12 **How easy is it to recover from errors during the update procedure?**

See SM3.1.

SM3.13 **How well is the update procedure documented?**

The update procedure is documented only in documentation used by Rational field service technicians.

SM3.14 **Newsletter? What is the frequency of publication?**

The Rational User's Group publishes a quarterly newsletter.

SM3.15 **Interest and/or user groups?**

There is a Rational User's Group which meets quarterly. It also holds an annual international meeting.

SM3.16 **Is there a dial-up computer number to access a database of previously encountered bugs?**

A list of problem reports made to Rational is available through their Support Information Management System (SIMS). However, the primary purpose of SIMS is to facilitate customer contact with Rational headquarters. The list of problems is not particularly useful as a set of bug reports because it is unindexed, and because hardware problems that are not of general interest are included in the problem database. See SM3.20 for further information about SIMS.

SM3.17 **Level of Software support? Levels are defined as follows:**

Level 1

7 day, 12-24 hour phone service; maintenance; revised versions of software and documentation; on-site consultation regarding problems.

Level 2

5 day, 8-12 hour phone service; maintenance; revised versions of software and documentation; remote consultation regarding problems.

Level 3

no phone service; no maintenance; revised versions of software and documentation; no consulting support; submit software trouble reports formally in writing.

Rational provides level 1 phone service and 1 consultation. The hardware/software maintenance contract includes phone access to the Rational 24 hour/7 day Response Center. See SM3.18. Revised versions of software are installed as part of regular maintenance. Revised documentation is provided with revised software.

SM3.18 **What is the cost for software maintenance?**

Software maintenance is bundled with Rational's overall customer support fee, which is $4500 per month in January 1988 for the Rational R1000 Model 200-20. The customer support fee has remained at that cost since the introduction of the Model 200-20 in November 1986. This fee includes comprehensive hardware and software onsite support. The hardware support includes parts and on site labor. The software support includes all updates, upgrades, and documentation.

SM3.19 **Is remote maintenance offered (i.e., vendor dials into system under maintenance contract to service remotely)?**

A diagnostic modem is part of the standard Rational hardware configuration. If certain system failures occur the Rational computer itself will call Rational and request that a customer representative be paged. The diagnostic modem will also be used as needed to diagnose customer reported problems as part of the standard maintenance contract.

SM3.20 **What is the method of reporting software bugs? Are there any automated tools available to report errors (e.g. a program that makes it easy to fill in the form that must be delivered to report the error or an electronic address to mail the problem report?).**

The standard Rational configuration includes a terminal dedicated to contacting Rational's Support Information Management System (SIMS). This terminal is used for electronic mail between the user and Rational and for logging problem reports. SIMS provides screen forms that the user fills in for sending mail messages and entering problem reports. A bulletin board containing news is available through SIMS.

SM3.21 **Average turnaround time from bug report to bug fix to distribution of patch?**

Turnaround time for fixes to the kernel is the interval between system releases. This was stated in the answer to SM3.2. Part of the job of Rational service representatives is to help customers develop workarounds to kernel bugs discovered in the interval between system releases. Company policy is to provide problem solutions rather than kernel patches so that the software in the field will be a known entity to service representatives developing the solutions.

Fixes to the functional part of the OS will be implemented as soon as the Rational technicians find the bug. Patches are not distributed; rather the section of code containing the bug is edited and recompiled.

SM3.22 **Is the software covered under a warranty? If so for how long?**

No.

SM3.23 **What is the policy and procedure for acquiring 3rd party software that will execute within the Ada environment? Is there an integration kit available to aid in integrating 3rd party software into the environment?**

Since the Environment runs Ada exclusively, only those third party tools written in Ada can be ported to the Environment. The Rational Environment supports ANSI standard tape format (such as is generated by VAX/VMS) and Ethernet as a means of importing programs. Although the Environment does not store Ada source in text format, Text_IO can read Rational Ada objects as though they were text. Thus, imported tools requiring read access to Ada source in text format will work provided they use Text_IO to access text files. Tools requiring access to system services (such as the directory system) will find that sections of the visible operating system interface (such as the procedures that access the directory structure) are undocumented. Rational technicians will provide assistance to users who are attempting to solve problems that require use of undocumented portions of the Environment system.

SM3.24 **Are full disk backups supported for both the software and the database?**

Yes, the full backup procedure takes a snapshot of the entire Environment. Rational recommends a weekly full backup.

SM3.25 **Are incremental disk backups supported for both the software and the database?**

Yes; primary backup procedure records changes to the Environment since the last full backup procedure. Rational recommends a daily primary backup.

SM3.26 **Is there automated support for restoring the software and/or database element from the backup?**

The entire process of both generating an Environment backup and restoring the Environment from primary or full backup is prompted from the operator's console. The restoration operation restores the entire Environment. There is no capability for restoring selected files from a full or primary backup. Individual users can request source archiving for selected files or directories. Rational does not recommend backing up the entire Environment in source archive form because performing a source archive of the entire Environment would be a very resource-intensive and long-running process.

3.7. Experiment #4 Answers

Question	Response
SM4.1	**Describe the mechanics of enabling the logging of system accounting information.**
	Logging of accounting information on the Rational R1000 is enabled by creating a directory called **!Machine.Accounting** when the system is booted. When the system is booted, a file is created that contains the date and time of the reboot. Disabling logging requires the intervention of a Rational technician and involves changing the system boot procedures. The files in this directory contain records of machine use between system reboots, with each file being given a name containing the data of reboot. To generate files that contain accounting information for a given time period, the system must be rebooted at the end of each period. Within each file a record is generated that logs usage data for each user session, system-initiated job, and user-initiated background job that terminates after the user session that initiated it.
	A session is a user interaction with the Rational Environment from logon to logoff. Accounting information for a session includes the resources used by all user-initiated background jobs that terminate during the session. Sessions have names that appear in the accounting files. The names represent sets of session switches that control Rational Environment attributes, such as maximum number of windows on the Rational display. Users edit a set of session switches to define a desired working environment. The session names are therefore not arbitrarily chosen, but selected by the user at logon from a set of already defined session names. If the user logs on with a session name that does not exist, the Rational Environment offers the user the choice of aborting the login or creating the session with its concomitant set of session switches.
SM4.2	**Elapsed time for enabling the logging of system accounting** information?
	Enabling logging requires only the time for directory creation, which is given in the Design and Development Experiment.
SM4.3	**Describe the mechanics of disabling the logging of system accounting information.**
	The Rational Environment is designed to have accounting always enabled. Disabling logging requires intervention of a Rational technician.
SM4.4	**Elapsed time for disabling the logging of system accounting information?**
	See SM4.3.
SM4.5	**What are the disk space requirements of the accounting log file?**

Each entry in an accounting log file is a line of text consisting of a fixed length and a variable length part. The first sixty-one characters record all information except the user name and session name. The variable length part contains the user name and session name.

SM4.6 **What is the execution overhead associated with continuous collection of accounting statistics?**

According to Rational designers at company headquarters, the overhead is so low that an attempt to measure it would be masked by such noise factors as disk latency. The statistics collection is part of the activity of the system daemon and cannot be separated at the programmer interface level from the measurement of the system daemon activity.

SM4.7 **What kind of system accounting information can be collected? CPU usage? Connect time? Disk usage? Number of logins? I/O Activity? Pages printed?**

For each user session, the following information is logged: time and date session starts and ends, elapsed time, CPU time, number of disk requests and number of jobs executed during the session. The user session information about CPU time and disk requests is a cumulative summary of the usage information about all the jobs initiated by the user that terminated during the session. The same information is logged for user-initiated jobs that extend beyond the end of the session, as well as for background jobs initiated by the Rational Environment itself.

SM4.8 **Are callable program interfaces provided for collecting accounting statistics? If so, do these interfaces support all appropriate services provided by the underlying operating environment?**

The callable interfaces for accounting statistics provided by the Rational Environment return elapsed CPU time and current working set size for a job.

SM4.9 **What format is employed for the accounting log file (ASCII text, compressed binary)?**

Accounting information is stored in ASCII text files.

SM4.10 **Describe the mechanics of dynamically monitoring the system's workload.**

The Rational Environment provides three procedures for monitoring the system workload. One of these, What.Jobs, continuously monitors the load and updates a screen display at a user-specified interval. The other two, What.Load and What.Users, provide snapshots and terminate after generating one screen display. What.Jobs is invoked by opening a command window, entering What.Jobs, and promoting the command window. What.Users and What.Load are bound to the keyboard and are invoked with a single keystroke.

SM4.11 **What is the execution overhead associated with dynamically monitoring the system's workload?**

What.Jobs does not show up in the What.Jobs window as a separate job. Therefore, the overhead it creates must be reported as part of either general system overhead or as part of the overhead of an idle user session (which is relatively stable at 1.75 percent of CPU time). Where the What.Jobs overhead is reported cannot be determined. From the user's point of view, invocation of the What.Jobs command simply causes the cursor to update a What.Jobs window at a user-specified interval (which has a default value of ten seconds). The user can work in other windows while monitoring the system.

SM4.12 **What type of system workload monitoring is supported? Which of the following can be monitored: Page faults, Swapping, I/O activity, Memory Usage, Process w workloads.**

The information provided by the three system monitoring procedures described in SM4.10 is listed separately.

What.Jobs lists for each job the following:

- user name (of user who started job)
- session name
- job number
- status (run, idle, wait, disabled, queued)
- elapsed time for job
- total CPU time for job
- percentage of CPU time consumed by job
- working set size
- disk waits since job inception
- job name

What.Users lists the following for each job initiated by a currently active user:

- user name
- port number
- job number
- status (run, idle, wait, disabled, queued)
- elapsed time for job, and
- job name

What.Load(False) lists the average number of tasks eligible for CPU time over the last 100 milliseconds, the last minute, the last 5 minutes, and the last 15 minutes. When invoked with the verbose parameter set to true (the default), it displays the number of tasks able to be run, averaged over the last 100 milliseconds, 1 minute, 5 minutes and 15 minutes. It also displays the number of tasks waiting for pages from disk and the number of withheld jobs averaged over the four time periods.

SM4.13 **Does a report generator exist for summarizing the accounting logs?**

Yes, a report generation tool is available in !Tools.System_Availability.

SM4.14 **Describe the mechanics of reconfiguring and rebooting the system.**

Reconfiguration of the operating system for performance involves two separate activities, scheduling the clients of the system daemon and setting the parameters for the system scheduler. Either activity can be performed by calling Environment procedures interactively or by incorporating calls to Environment procedures into the initialization procedure that is run as part of system booting. Interactive invocation follows the standard Rational method of executing a procedure from a command window. Incorporating procedure calls into the system initialization procedure follows the standard Rational methods for program editing and compilation.

The daemon's clients are procedures that perform housekeeping functions, such as disk garbage collection, in the Rational Environment. Scheduling these clients is important as several of them consume so many machine resources that the system is unusable when they are running, and others noticeably slow the system. Rational estimates that the period that the Rational system will be rendered unusable by daemon's clients will range from two to six hours per day, depending on the workload. The higher the workload, the more housekeeping the daemon's clients must perform.

The system scheduler allows tailoring of system-wide parameters for the algorithms that allocate CPU time, memory, and disk resources to the system jobs.

Additional system configuration capabilities are offered by the package Terminal, which allows setting communication port parameters, and by the package Queue, which provides for the management of print queues (creating print device lists, assigning queues to devices, etc.).

The Rational R1000 has a hardware switch to determine whether system reboot will be manual or automatic. The manual reboot process is used only by Rational technicians. When the hardware switch is set to automatic, rebooting the system proceeds automatically after calling the system shutdown procedure or when powering up the system. The system can be configured so that reboot is completely automatic or so that the operator must press <return> at the system prompt "Enter configuration to boot [Standard]."

SM4.15 **Elapsed time of reconfiguring and rebooting the system?**

System rebooting usually requires fifteen to twenty minutes. When system configuration procedures are called interactively, they execute within a few seconds. The initialization procedure executed at system reboot can be altered as fast as the user can edit it.

SM4.16 **How easy/difficult is it to reconfigure the operating system?**

Executing the reconfiguration procedures interactively or editing the initialization procedure is very easy.

SM4.17 **How much human intervention is required during the reconfigure** procedure?

The reconfiguration process is entirely manual.

SM4.18 **How easy is it to recover from error during the reconfigure procedure?**

Out of range values for system reconfiguration procedure parameters will generate constraint errors. If the procedure was invoked interactively, the user would simply return to the command window, edit the procedure parameters, and run the procedure again. If the error is generated during execution of the system initialization procedure, the system will complete booting, but only one Rational terminal that is connected to a port whose communications parameters are hardwired will be enabled. The initialization procedure must then be corrected from the hardwired terminal and run to enable the remaining terminals. This suggests that all changes to the system initialization procedure should be tested by running the procedure when the system is up. Editing the initialization procedure is as easy as any other program development work on the Rational.

SM4.19 **How well is the reconfigure procedure documented?**

Complete documentation is provided, including guidelines for scheduling daemon clients, an overview of the system's resource-scheduling algorithms,

and detailed descriptions of all daemon and scheduler procedures and parameters.

SM4.20 **Do system resource (CPU time, disk space, etc.) quotas exist? If so, at what level can they be set? (individual user, user account group, only all accounts)**

CPU time and working set size quotas exist. Disk usage is allocated by withholding jobs from running when the disk load becomes too high. The parameters for the disk load algorithm can be set by the user. There is no allocation of disk space. The CPU time and disk usage allocations are system-wide only. Working set size can be set system-wide and for individual jobs.

3.8. System Management Analysis

3.8.1. Functionality

Users of the R1000 computer buy a package that typically includes the R1000 computer, the Rational Environment, software updates, and a Rational service contract. The service contract is a separately priced item, which includes software updates. The R1000 and Rational Environment are installed by Rational technicians. Therefore, users are not involved in system installation. System management can supply only three account characteristics: the account name, initial password, and enrollment in certain user groups. The functionality to modify and display these few characteristics is provided to users with operator capability.

User work space management attributes and operations can be associated with an individual account via the Rational commands, including key bindings, macro definitions, and session switches. These are not considered system management attributes. However, a procedure could be created that would provide a baseline of work space management attributes, as well as an initial password and user group membership upon user account creation.

Although the R1000 does not allow tuning the system by adjusting system resource quotas for individual users, system management utilities do allow tuning by setting parameters for the system as a whole. These utilities are not covered by this experiment.

The system can log sufficient resource usage accounting information to serve as a basis for a billing system. There is no easy way to disable system accounting logging once it is enabled. Comprehensive accounting information is available interactively. There are no documented program interfaces for obtaining system accounting information except for cumulative CPU time for a job. Procedures are provided to control system-wide parameters affecting system performance both for CPU time, memory, and disk usage allocation and for scheduling of system daemon's clients.

Rational Environment installations and upgrades are done entirely by Rational technicians as part of the sales and maintenance contracts. Also, the Rational Response Center is available by phone to answer any questions. Throughout the performance of the experiments, the Response Center was prompt in providing guidance, admitting to problems, and providing usable workarounds.

3.8.2. Performance

Rational technicians reserve a machine for over a half day to a day to perform installation of a new release of the Rational Environment. Differences in installation time depend on whether the user has optional environment components such as networking software or cross compilers.

The commands to create a user account group, create a new user account, add a user account to a user group, disable logins for a user account, display account characteristics, modify account characteristics, remove a user account from a user group, and delete a user account are highly interactive. Most executed in an elapsed wall clock time of about one second.

The space required for creating a new user was 7473 bytes. If user access to the account is revoked, no space is reclaimed by the system, as the user's home world still exists. However, if the user's home world is deleted, then the space is reclaimed. Creating a user account group caused a space consumption increase of 1 byte in the !Machine.Groups directory. Adding a user to the user account group caused a space consumption increase of 2 bytes. Any other storage costs due to these two operations could not be measured. The space used by !Machine.Groups is reclaimed when a user account is removed from a user group.

The execution overhead of accounting logging and interactive resource usage display could not be determined. System reboot takes approximately fifteen to twenty minutes (and is required to close accounting log files). Interactive response to the system reconfiguration commands was quick.

3.8.3. User Interface

The System Management Experiments use the Rational Environment interface. This interface supports command recall, wildcards, command editing, command abbreviations, and parameter prompting. In the Rational Environment, the user is always in the Rational editor, whose commands for cursor movement, object selection, and deletion operate on directories, command windows, text, and Ada objects. All commands in the Rational Environment are Ada subprograms. They may be invoked from within the editor by selecting the command in its home directory and pressing the <Promot> key, by entering the subprogram name in a command window (which provides a block within which subprograms be run), or by binding the subprogram to a key on the Rational Terminal keyboard at logon. Procedures may be bound to the keyboard so that they execute immediately or so that they prompt for parameters. A Prompt_For key overrides a key binding for immediate execution and causes the command bound to a key to appear in a command window. A <Complt> key will generate a parameter list for a command whose name has been entered in a command window and will also complete the spelling of a procedure name if enough of the name is provided so that it is unambiguous. Many parameterless commands act on objects in the Environment, such as directory entries or sections of text or Ada objects that have been selected with the editor object selection commands.

The simple user account management activities provided by the Rational Environment can easily be performed directly from a command window in any directory that has established a link with the operator package. The only requirement of a user of most of the operator package commands is that the user's account have operator capability.

3.8.4. System Interface

The commands for user account management are in complete accordance with the standard Rational interface. The user account management commands are Ada procedures which are invoked from a command window.

The Rational Environment lacks the ability to disable machine usage accounting without the intervention of a Rational technician. According to Rational designers, the system overhead for logging accounting information is so low that an attempt to measure it would disappear into the noise generated by such factors as disk latency. Closing the current logging file requires rebooting the system, which is a clumsy arrangement. Its contents may be copied to another file using the Common.Write_File procedure. Operating system reconfiguration is performed completely in accordance with the standard Rational interface, which makes the reconfiguration process easy.

One problem concerning system interface was encountered across several of the experiments. Since this section deals with the duties of a system administrator, it will be noted here. A random, sporadic generation of input from an unused I/O port wasted 50 to 80 percent usage of the CPU. Until this hardware problem was corrected, it caused a degradation in system response to many of the commonly used window commands. Due to window manipulation commands often being only one or two keystrokes, system interface degradation was immediately noticeable and very frustrating. It would be the duty of the system administrator to recognize the problem and correct it. With the help of a Rational Service Representative the bad I/O port was found, and once corrected the system response improved dramatically.

4. Design and Development Experiment

4.1. Introduction

The Design and Development Experiment exercises the Environment support of detailed design, code development, and translation. The Experiment consists of creating a program library and using the Environment's editor to enter code seeded with errors. The Environment's capability to detect the errors is exercised. A second program library, which will contain dependencies on the first, is created; changes to the first are outlined, and required retranslation effort is observed.

The Rational Environment keeps the bodies of procedures declared as separate in the Ada object in which their specification was declared, rather than in separate files. Aside from this, they are treated independently of their parent unit. Once the bodies of subunits have been compiled, they appear in directories and can be visited independently of their parent unit and can also be promoted and demoted independently of their parent unit, provided that the state of the subunit body is not higher than the state of its parent unit.

The incorporation of subunit bodies in the Ada unit on which they depend requires that the Get_Row and Get_Col procedure bodies be created within the body of Matrix Management. To conform as closely as possible to the existing script, the bodies of Get_Row and Get_Col are entered into Matrix_Management in step 7 and coded in step 8.

The installed Ada objects in a library contain much of the semantic information in a library. The division of library information between Ada objects and their enclosing context cannot be determined. Consequently, when the Rational Environment measures disk usage, it combines the changes in size of the Ada object and the changes in size of the enclosing context into one figure rather than separating the figures into library disk utilization and procedure disk utilization.

The Rational Environment requires that any compilation unit that depends upon another compilation unit which is being edited also be demoted to source state. Compilation units in the source state are not visible to other compilation units. Obsolete units are never visible to other units in libraries. For this instantiation of the Design and Development Experiment, determining the recompilation status of a library will be interpreted as meaning generating a listing of the state of the Ada objects (source, installed, or coded) in a directory or world rather than determining which units in a library are obsolete.

In the following experiment instantiation, experiment steps are numbered and substeps are lettered. Sections of substeps are numbered with lowercase Roman numerals. Any comments concerning the experiment step in the context of the Rational Environment are in regular type with braces ({ }). The instantiation of the experiment is provided as a transcript of actual keystrokes. Comments as to the correct context in which to type the indicated keys, and comments as to what the keystroke will accomplish are indented, enclosed in braces ({ }) and printed in a different typeface `typeface`. The required keystrokes are indented and printed in typeface `typeface`, and are indicated either by the letter(s) on the key or by its keymap designation in angle brackets (< >).

4.2. Experiment

1. Set up experiment

   a. Create directory named EXP_LIB, in which the experiment will be performed.

```
<Create Directory>
{A command window opens,
supply Exp_Lib as value
of parameter Name.}
(Name => "")
"EXP_LIB"
<Promot>
```

   b. Create a subdirectory under the experimental directory, named ADA_LIB, to house Ada source code fragments that will be required throughout the experiment.

```
{Move cursor to Exp_Lib.}
<Definition>
<Create World>
{A command window opens;
supply Ada_Lib as value
of parameter Name.}
(Name => "")
"Ada_LIB"
<Promot>
<Create World>
{Supply Ada_Lib_Spec_Errors
as value of parameter Name.}
(Name => "")
"Ada_LIB_SPEC_ERRORS"
<Promot>
<Create World>
{Supply Ada_Lib_Body_Errors
as value of parameter Name.}
(Name => "")
"Ada_LIB_BODY_ERRORS"
<Promot>
```

   c. Create, as text, the source code fragments and data files in ADA _LIB. Appendix 5.A exhibits these files by filename.

   {Objects to be entered are shown below sorted by library with the step in which the object is used indicated. Also shown is the appendix section and exhibit number from *Evaluation of Ada Environments*, Chapter 5. Multiple libraries are used because the Rational directories function as program libraries and allow only one compilation unit of a given name and type in a directory.}

	Step	Appendix	Exhibit
In Ada_Lib_Spec_Errors:			
Matrix_Management spec	**3c**	**5.1**	**1.2a**
In Ada_Lib_Body_Errors:			
Vec_Main	**6b**	**5.A.7**	**1.4a**
Matrix_Management body	**7a**	**5.B.3**	**-**
In Ada_Lib:			

Vector_Management_Body	**7b**	**5.B.18**	**-**
Matrix_Main	**8a**	**5.B.5**	**-**

{Note: Vector_Management Spec (Appendix 5.1, Exhibit 1.1a) will be entered by hand, rather than copied from a library in step 3bi. Errors shown in Vector_Management spec cannot be committed to disk, because committing the file causes syntactic completion.}

d. Develop a command named recordit to collect general experimental data.

{For this script the library creation, the file copy command Library.Copy and Ada object promotion commands have been incorporated into procedures that instrument them by measuring time and disk space utilization. These instrumented procedures are then bound to the Rational keyboard. The code for these procedures is listed in Appendix B.}

e. Develop a command named time to collect experimental timing data.

{See 1.d.}

2. Identify objects and operations.

The R1000 provides no support for graphical design methods.

3. Create package specification(s).

a. Create program library named PROJECT_LIB. Measure the time it takes to create program library. Measure disk utilization for newly created program library.

```
<Create Directory>
{A command window opens;
supply Project_Lib as the
value for Directory_Name.}
(Directory_Name => "")
"Project_Lib"
<Promot>
```

b. Create package specification for a package named VECTOR_MANAGEMENT.

i. Enter the package specification, which is seeded with errors exactly as it is shown in Exhibit 1.1a.

```
{Move cursor over PROJECT_LIB in
the EXP_LIB directory.}
<Definition>
{Write a message to the system message
window (which shows the compiler error
messages) to label the error messages
that will be generated by attempting to
compile the specification of
Vector_Management.}
<Create Command>
{A command window opens; enter:}
"message.send"
<Complt>
{Enter user id as value for parameter
Who.}
<Next Item>
{Enter specification name as value
for parameter Message.}
"vector_management_spec"
<Promot>
{Message appears in system message window,
open window for anonymous Ada object
creation.}
<Object> I
{Enter source into the Ada object
as shown, in Evaluation of Ada
Environments, Chapter 5, Section 5.A.1
Exhibit 1.1a.}
```

ii. Display and correct translation errors.

```
<Format>
{Correct errors shown by format
key.}
```

{Note that the format key adds the missing "is" and "of" and adds the prompt "[expression]" for the missing "FLOAT."}

iii. Translate into program library PROJECT_LIB. Measure elapsed and CPU times for translation.

```
{Compile to installed state.}
<Promot>
{Compile to coded state.}
<Promot>
```

iv. Compare corrected package specification to Exhibit 1.1b. (Note that the file resides in Ada_LIB. Correct any differences and retranslate if necessary. Measure program library disk utilization attributable to the package specification.

{Package specification is as shown in *Evaluation of Ada Environments*, Chapter 5, Section 5.A.2 Exhibit 1.1b.}

c. Create package specification for a package named MATRIX_MANAGEMENT.

i. Enter in the package specification, which is seeded with errors, exactly as it is shown in Exhibit 1.2a.

```
{Close Vector_Management spec window
and move cursor back into PROJECT_LIB
world.}
<Object> G
{Move cursor to Project_Lib command
window and make notation in system
message window that any following
compiler errors refer to
Matrix_Management_Spec.}
<Window> <Down Arrow>
{Make message command editable.}
<Item Off>
{Change Vector_Management_Spec to
Matrix_Management_Spec and execute
message command.}
<Promot>

{Copy Matrix Management spec with errors
to Project_Lib in command window of
Project_Lib.}
Library.Copy
<Complt>
{Supply pathname to Matrix Management
spec and Matrix Management as value
for From parameter.}
"^Ada_Lib_Spec_Errors.Matrix_Management"
<Promot>
{Move cursor to directory entry
for Matrix_Management spec and select it.}
<Object> <Left Arrow>
{Open a window to edit Matrix_Management
spec.}
<Edit>
```

ii. Display and correct translation errors.

{Use format key to find syntax and local semantics errors, semanticize key to find semantic errors, next item key to move between errors, and explain item key to obtain explanation of syntax and semantics errors. Note that missing second <> for the array definition is inserted by pressing the format key.}

iii. Translate into program library PROJECT_LIB. Measure elapsed and CPU times for translation.

```
{Compile to installed state.}
<Promot>
{Compile to coded state.}
<Promot>
```

iv. Compare corrected package specification to Exhibit 1.2b. Correct any differences and retranslate if necessary. Measure program library disk utilization. Measure disk utilization attributable to the package specification.

{Package specification is as shown in *Evaluation of Ada Environments*, Chapter 5, Section 5.A.4 Exhibit 1.2b.}

4. Design subprogram control flows, identify subprogram interdependencies, and define subprogram specifications local to each package body.

The Rational Environment provides no graphical design aids.

5. Create package body for VECTOR_MANAGEMENT.

   a. Generate package body of VECTOR_MANAGEMENT using a null body generator if available. Otherwise, use vector_body_null in Ada_LIB.

```
{Cursor will be in spec of
Matrix_Management at end of
step 3.  Close Matrix_Management
spec window and move cursor back
to Project_Lib.}
<Object> G
<Window <Down Arrow>
{Retrieve command window with
message.send, in order to
note in system message window
that any following compilation
errors apply to the body of
Vector_Management.}
<Object> U
{Change Matrix_Management_Spec to
Vector_Management_Body and execute
message command.}
<Promot>
{Move cursor to Vector_Management'Spec
in the Project_Lib directory window, and
select it.}
<Object> <Left Arrow>
<Create Body>
{Note:  A window opens on skeleton of
Vector_Management'Body.}
```

   {From this point on, assume that a message indicating what unit is being compiled will be sent to the message window before every compilation of a unit containing errors.}

   b. Modify the pairwise vector multiplication function.

      i. Enter the function body, which is seeded with errors, exactly as it is shown in Exhibit 1.3a.

         {The missing "is" in line 2 of the code in *Evaluation of Ada Environments*, Chapter 5, Section 5.A.5 Exhibit 1.3a is prevented by use of the null body generator. Enter remaining code as shown.}

      ii. Display and correct translation errors.

         {Use format key to find syntax and local semantics errors, semanticize key to find semantic errors, next item key to move between errors, and explain item key to obtain explanation of syntax and semantics errors.}

      iii. Translate into program library PROJECT_LIB.

```
{Compile to installed state.}
<Promot>
{Compile to coded state.}
<Promot>
```

      iv. Compare corrected package body to Exhibit 1.3b. Correct any differences and retranslate if necessary. Measure program library disk utilization. Measure disk utilization attributable to the package body.

         {Package specification is as shown in *Evaluation of Ada Environments*, Chapter 5, Section 5.A.6 Exhibit 1.3b.}

6. Create a main program named VEC_MAIN in a separate program library to drive pairwise vector multiplication.

    a. Create a program library named TEST_LIB from within the directory PROJECT_LIB that will contain compilation units that have dependencies upon units in PROJECT_LIB.

```
{Cursor is in body of Vector_Management
at end of step 5; close Vector_Management'Body
window and move cursor to Project_Lib window.}
<Object> G
<Create Directory>
{Note that a unit compiled in a
directory within a world has as a
context all other units that have been
compiled in the world or in a directory
within the world.  Supply Test_Lib as
the value for parameter Name.}
(Name => "")
"Test_Lib"
<Promot>
```

    b. Create a test main program named VEC_MAIN that will be translated into TEST_LIB.

        i. Create the procedure VEC_MAIN, which is seeded with errors, by copying it from Ada_LIB. Refer to *Evaluation of Ada Environments*, Chapter 5, Section 5.A.7, Exhibit 1.4a.

```
<Open Test_Lib window by moving the
cursor over Test_Lib in Project_Lib
directory.}
<Definition>
{Copy Vec Main with errors into
Test_Lib.}
<Create Command>
Library.Copy
<Complt>
{Supply pathname to Vec_Main and
Vec_Main as value for parameter
From.}
(From => ""...)
"^^Ada_Lib_Body_Errors.Vec_Main"
<Promot>
{Move cursor over Vec_Main
directory entry and select it.}
<Object> <Left Arrow>
{Open a window containing Vec_Main
to edit.}
<Edit>
```

        ii. Display and correct translation errors. Display a cross-reference map.

            {Use format key to find syntax and local semantics errors, semanticize key to find semantic errors, next item key to move between errors, and explain item key to obtain explanation of syntax and semantics errors.}

            {The Rational cross reference utility requires that objects be in the installed state or, if in the source state, that they have been successfully semanticized immediately prior to use of the cross reference

utility. Therefore, Vec_Main must be successfully semanticized before the Xref calls are made.}

```
{Move cursor to Test_Lib window.}
<Enclosing>
<Create Command>
{Command window opens showing
previous command; enter:}
"Xref.Uses"
<Complt>
(List_Of_Names => "<IMAGE>",
Visible_Declarations_Only
               => True, ...)
{Supply Vec_Main as the value for
parameter List_Of_Names.}
"Vec_Main"
{Go through a list of Boolean switches
turning on the switches for information
that is not needed in the Xref
listing.}
Numeric 7 <Next_Item>
"true" {report use of constants}
Numeric 3 <Next Item>
"true" {report use of labels}
<Next Item>
"true" {report use of packages}
Numeric 5 <Next Item>
"true" {report use of variables}
Numeric 3 <Next Item>
{Save the Xref results.}
"Vec_Main_Xref"
<Promot>
```

iii. Translate into program library TEST_LIB.

```
{Move cursor to window containing Vec_Main
and compile it to installed state.}
<Promot>
{Compile to coded state.}
<Promot>
```

iv. Compare corrected package specification to Exhibit 1.4b. Correct any differences and retranslate if necessary. Measure program library disk utilization. Measure disk utilization attributable to the procedure.

{Package specification is as shown in *Evaluation of Ada Environments*, Chapter 5, Section 5.A.8 Exhibit 1.4b.}

c. Create executable module. Execute. Halt execution. Resume execution. Time module creation. Observe execution error message(s).

{Rational links object code contained in Ada objects into an executable module either at runtime or, if the pragma Main is used in the main program, at compilation time. Creation of an executable module is therefore not an observable, measurable step.}

```
{Move cursor to Test_Lib window.}
<Enclosing>
{Move cursor over Vec_Main and
select it.}
<Object> <Left Arrow>
{Execute Vec_Main.}
<Promot>
{Move job to background mode.}
<Control> G
{Halt execution of job.}
<Job Disable>
{Resume execution of job.}
<Job Enable>
```

{Note the job enable and job disable keys stop all jobs. These keys can prompt for a job number so that a specific job is disabled, but Vec_Main does not run long enough to go through the mechanics of disabling a single job. If Vec_Main were run under the debugger, stopping and starting it would be easy. Running under the debugger does not require recompilation, just use of the meta key in combination with the promote key (<Meta> <Promot>).}

d. Determine the cause of the execution error by first browsing VEC_MAIN and noticing that the variable v3 is of TYPE VECTOR(1..4). Examine the statement invoking pairwise vector multiplication: product3 := v3*u3. Then browse the pairwise vector multiplication function and notice that there is no check for compatible dimensions.

```
{Move cursor to Vec_Main'Body
in Test_Lib directory window.}
<Definition>
{Move to Test_Lib window.}
<Enclosing Object>
{Move to Project_Lib window.}
<Enclosing Object>
{Move cursor over Vector_Management
body.}
<Definition>
```

7. Create package body for MATRIX_MANAGEMENT.

a. Create package body for MATRIX_MANAGEMENT by copying existing version from matrix_body_errors in Ada_LIB. Correct all errors except for the exception declaration, which will be corrected in the next step.

```
{Cursor is in window containing
Vector_Management body after
step 6; move to Project_Lib window.}
<Enclosing Object>
{Turn off selection of Vector_Management.}
<Item Off>
<Create Command>
{Cursor moves to command window containing
last command issued with this window; change
command to:}
"Library.Copy"
<Complt>
(From => "<REGION>" ...);
{Supply parameter value for parameter
From.}
"^Ada_Lib_Body_Errors.Matrix_Management"
<Promot>
{Move cursor to Matrix_Management body
in Project_Lib directory and select it.}
<Object> <Left Arrow>
{Open window to edit Matrix_Management body.}
<Edit>
{Use format key to find syntax errors
and local semantics errors, semanticize
key to find semantic errors, next item
key to move between errors, and
explain item key to obtain explanation
of syntax and semantics errors.  Commit
incorrect Matrix_Management body to disk.}
<Enter>
```

b. Substitute for the VECTOR_MANAGEMENT package body a revised version copied from vector_body_excptn in Ada_LIB. This version contains a non-null INNER_PROD function and a test for incompatible dimensions in the pairwise vector multiplication function. Add "Dimension_Error : exception;" to the package specification and retranslate.

```
{At end of step a, the cursor is
in Matrix_Management'body; move cursor
to Project_Lib directory.}
<Enclosing Object>
{Turn off selection of Matrix_Management body.}
<Item Off>
<Create Command>
{Cursor moves to command window; replace
previous command:}
"Library.Copy"
<Complt>
(From => "<CURSOR>"...);
{Supply value for parameter From:}
"^Ada_lib.vector_management'body"
<Promot>

{Alter the spec of Vector_Management.}
{Move cursor to Vector_Management'Spec
in Project_Lib directory and select it.}
<Object> <Left Arrow>
{Demote Vector_Management'Spec to
installed state.}
<Install Unit>
{Open window to incrementally edit
Vector_Management'Spec.}
<Definition>
{Move cursor into declarative region
of Spec.  Open edit window with
cursor at the declaration prompt.}
<Object> I
{Enter the declaration of the
Dimension_Error exception and format
to fit in Vector_Management'Spec.}
<Format>
{Add declaration to rest of Spec.}
<Promot>
{Return Spec to coded state.}
<Promot>
{Move cursor to Project_Lib window.}
<Enclosing Object>

{Code the new Vector_Management body;}
move cursor to Vector_Management body
and select it.}
<Object> <Left Arrow>
{Promote Vector Management body to coded
state.}
<Code Unit>
```

c. Create function body for GET_ROW and null body for GET_COL by copying from GET_ROW in Ada_LIB, but do not translate until so directed in a subsequent step. Retranslate MATRIX_MANAGEMENT package body into PROJECT_LIB.

{The Rational Environment requires that separate unit bodies be part of the Ada object in which their specifications are declared. This can be achieved in at least two other ways besides the method shown in the script; the procedures can be compiled independently and copied into the Matrix_Management Ada Object by using the Library.Copy procedure, or they can be copied from their own window to the Matrix Management window with the "<Object> C" key combination. In either case, the separate

declaration in Matrix_Management is automatically generated as is the separate body syntax.}

```
{Move cursor to Matrix_Management
body in Project_Lib directory.}
<Definition>
{Move cursor to Get_Row function
and select it.}
<Object> <Left Arrow>
<Edit>
{Enter source as shown and commit
Get_Row body to disk.}
<Enter>
{Return to Matrix_Management body.}
<Object> G
{Select Get_Col.}
<Object> <Down Arrow>
<Edit>
{Enter source as shown and commit
Get_Col body to disk.}
<Enter>
{Return to Matrix Management body.}
<Object> G
{Compile Matrix Management body.}
<Code Unit>
{Return to Project_Lib directory.}
<Object> G
```

8. Create a main procedure named MAT_MAIN to drive matrix-vector multiplication.

    a. Create main procedure by copying Matrix_Main from ADA_LIB. Translate main procedure into program library TEST_LIB. List the compilation unit names and types in program library TEST_LIB and PROJECT_LIB. List package and subprogram interdependencies. Determine the completeness and recompilation status of both program libraries.

```
{At end of step 7, the cursor is in the
Project_Lib directory.  Move the
cursor over Test_Lib and copy
Mat_Main from Ada_Lib.}
<Definition>
<Create_Command>
"Library.Copy"
<Complt>
(From => "<REGION>" ...);
{Supply value for parameter From:}
"^^Ada_lib.mat_main'body"
<Promot>
{Move cursor to Mat_Main'Body, and
select it}
<Object> <Left Arrow>
{Promote Mat_Main to the coded state.}
<Code Unit>

{List compilation units' type and status
in Test_Lib.}
<Create Command>
"Ada_List"
<Complt>
"@'c(Ada)"
<Promot>

{Generate Xref listing for Test_Lib.}
<Create Command>
{Replace previous command in command window:}
"Xref.Uses"
<Complt>
(List_Of_Names => "<IMAGE>" ...)
"@"
Numeric 11
<Next Item>
"true"
<Promot>

{Move cursor to Project_Lib directory.}
<Enclosing Object>
{List compilation units' type and status
in Project_Lib.}
<Create Command>
"Ada_List"
<Complt>
"c(Ada)"
<Promot>
{Generate Xref listing for Project_Lib.}
<Create Command>
"Xref.Uses"
<Complt>
([List_Of_Names => "<IMAGE>" ...)
"@"
Numeric 11
<Next Item>
"true"
<Promot>
```

b. Create executable module. Execute. Time how long it takes to create module.

{Since the Rational Environment links at runtime, the only applicable part of this step is Execute.}

```
{Move cursor to Test_Lib in Project_Lib
directory.}
<Definition>
{Move cursor to Mat_Main in the
Test_Lib directory and select it.}
<Object> <Left Arrow>
<Promot>
{Note error message that Get_Row
and Get_Col have not been installed.
Move cursor to Project_Lib directory.}
<Enclosing Object>
{Move cursor to Get_Row and
select it.}
<Object> <Left Arrow>
<Code Unit>
{Move cursor to Get_Col and
select it.}
<Object> <Left Arrow>
<Code Unit>
{Move cursor to Test_Lib
in Project_Lib directory.}
<Definition>
{Move cursor to Mat_Main in the
Test_Lib directory and select it.}
<Object> <Left Arrow>
<Promot>
```

9. Modify package specifications and bodies and examine system retranslation behavior using MAT_MAIN as a main procedure (Figure 5-6).

    a. Change a package specification by removing a function specification that no other package depends upon: Delete pairwise vector multiplication specification and store temporarily in a separate location for subsequent reuse. Translate. Create an executable module. Observe system retranslation behavior.

```
{Cursor is in Test_Lib after step 8;
move cursor to Project_Lib window.}
<Enclosing Object>
{Create Temp Storage for function
to be deleted.}
<Object> I
{Return cursor to Project_Lib window.}
<Enclosing Object>
{Move cursor to Vector Management'Spec.}
<Definition>
{Demote Spec to installed state.}
<Install Unit>
{Move cursor to pairwise vector
multiplication function spec and
select it.}
<Object> <Left Arrow>
{Move cursor to window for
anonymous Ada object and copy
function declaration.}
<Region> C
{Return cursor to Vector_Management'Spec
window.}
{Select function again.}
<Object> D
{Note a window opens showing objects made
obsolete by removal of pairwise multiplication
(Vec_Main body and Vector_Management body).
Move cursor over Vec_Main'Body and select it.}
<Object> <Left Arrow>
{Demote Vec_Main'Body to source.}
<Source Unit>
{Move cursor over Vector_Management'Body and
select it.}
<Object> <Left Arrow>
{Demote Vector_Management'body to source.}
<Source Unit>
{Move cursor back to Vector_Management'Spec
window and again select the function.}
<Object> D
{Compile Vector Management'Spec to coded state.}
<Promot>
{Move cursor to Test_Lib Window and
select Mat_Main'Body.}
<Object, Left Arrow>
{Note:  A simple <Promot> keystroke could be
used below, but <Code (All Worlds)> generates
a log and <Promot> does not.}
<Code (All Worlds)>
```

b. Change package body by changing an algorithm in a subprogram body: Change INNER_PROD body so that it no longer uses pairwise vector multiplication. Translate into PROJECT_LIB. Create executable module. Observe system retranslation behavior.

```
{Move cursor back to Project_Lib window.}
<Enclosing Object>
{Move cursor to Vector_Management body.}
<Definition>
{Make Vector_Management'Body editable.}
<Install Unit>
{Move cursor to inner_product function
and select it.}
<Object> <Left Arrow>
<Edit>
{Note: inner_product body moves to an edit
window and demoted to source.  Change inner
product and reinstall it into
Vector_Management'Body.}
<Promot>
{Compile Vector_Management body to coded.}
<Promot>
{Return to Project_Lib window.}
<Enclosing Object>
{Move cursor to Test_Lib in Project_lib
window.}
<Definition>
{Move cursor to Mat_Main body.}
<Object> <Left Arrow>
<Code (All Worlds)>
```

c. Change package body by deleting an unused subprogram body: Delete pairwise vector multiplication function body and store temporarily in a separate location. Translate into PROJECT_LIB. Create executable module. Observe system retranslation behavior.

```
{At the end of step 9b the cursor
is in the Test_Lib window; return
to Project_Lib window.}
<Enclosing Object>
{Create Ada object to store function body.}
<Object, I>
{Return to Project_Lib directory.}
<Enclosing Object>
{Move cursor to Vector_Management body.}
<Definition>
{Make Vector_Management body
incrementally editable.}
<Install Unit>
{Move cursor over pairwise vector
multiplication function and select it.}
<Object> <Left Arrow>
{Move cursor to anonymous object window
and copy function to it.}
<Region> C
{Move cursor back to
Vector_Management'Body window.  Move
cursor over pairwise vector multiplication
function and select it.}
<Object> <Left Arrow>
{Delete function body.}
<Object> D
{Return Vector_Management body to coded.}
<Promot>
{Move cursor to Project_Lib window.}
<Enclosing Object>
{Move cursor over Test_Lib window.}
<Definition>
{Move cursor over Mat_Main body.}
<Object, Left Arrow> {Mat_Main is selected.}
<Compilation Make>
```

d. Change package body by adding a subprogram body: Add back pairwise vector multiplication function body. Translate into PROJECT_LIB. Create executable module. Observe system retranslation behavior.

```
{At the end of step 9c the cursor
is in the Test_Lib window; return
cursor to Project_Lib window.}
<Enclosing Object>
{Move cursor to object
containing function body.}
<Definition>
{Return to Project_Lib directory.}
<Enclosing Object>
{Move cursor to Vector_Management'Body.}
<Definition>
{Make Vector_Management body incrementally
editable.}
<Install Unit>
{Open edit window for function body.}
<Object> I
{Move cursor to window holding function and
select function.}
<Object> <Left Arrow>
{Move cursor to Vector_Management insertion
window and copy function
to it.}
<Region> C
{Install function body.}
<Promot>
{Code Vector_Management body.}
<Promot>
{Move cursor to Project_Lib window.}
<Enclosing Object>
{Move cursor to Test_Lib window.}
<Definition>
{Move cursor to Mat_Main body and
select it.}
<Object> <Left Arrow>
<Code (All Worlds)>
```

e. Change a package specification by adding a subprogram specification: Add back pairwise vector multiplication function specification. Translate into PROJECT_LIB. Create executable module. Observe system retranslation behavior.

```
{At the end of step 9d the cursor is
in the Test_Lib window; return cursor
to Project_Lib window.}
<Enclosing Object>
{Move cursor to object containing
function spec.}
<Definition>
{Return to Project_Lib directory.}
<Enclosing Object>
{Move cursor to Vector_Management'Spec.}
<Definition>
{Make Vector_Management'Spec
incrementally editable.}
<Install Unit>
{Open window to edit function spec.}
<Object> I
{Move cursor to window holding
function spec and select function.}
<Object> <Left Arrow>
{Move cursor to Vector_Management'Spec
declaration insertion window and copy
function to it.}
<Region> C
<Promot>
{Note:  window showing units that would
be made obsolete by promotion opens
(Mat_Main'Body).  Cursor moves to this
window.  Move cursor over Mat_Main'Body
and select it.}
<Object> <Left Arrow>
{Demote Mat_Main to source state.}
<Source Unit>
{Move cursor back to Vector_Management
insertion window.}
{Install function spec.}
<Promot>
{Code Vector_Management'Spec.}
<Promot>
{Move cursor to Project_Lib window.}
<Enclosing Object>
{Move cursor to Test_Lib window.}
<Definition>
{Move cursor to Mat_Main'Body and
select it.}
<Object> <Left Arrow>
<Code (All Worlds)>
```

f. Change package body by adding comments: Add comments to package body of VECTOR_MANAGEMENT. Translate into PROJECT_LIB. Create executable module. Observe system retranslation behavior.

   {Since comments can be added to a coded body, no retranslation is required.}

```
{The cursor is in the Test_Lib window
at the end of step 9e; move cursor to
Vector_Management'Body.}
<Definition>
{Move cursor to where comment is to
be inserted and open an insertion
window.}
<Object> I
{Enter comment.}
<Promot>
{Note:  comment is placed into body
and insertion window closes.  More comments
may be inserted by following the same steps.}
```

g. Add comments to package specification of VECTOR_MANAGEMENT. Translate into PROJECT_LIB. Create executable module. Observe system retranslation behavior.

{Since comments can be added to a coded specification, no retranslation is required.}

```
{The cursor is in the
Vector_Management'Body window
at the end of step 9f; move cursor to
Test_Lib window.}
<Enclosing>
{Move cursor to Vector_Management'Spec.}
<Definition>
{Move cursor to where comment is to
be inserted and open an insertion
window.}
<Object> I
{Enter comment.}
<Promot>
{Note:  comment is placed into body
and insertion window closes.  More comments
may be inserted by following the same steps.}
```

4.3. Functionality Checklist

Activity	Step #	Supported (Y/N)	Observations
Detailed Design			
Create system skeleton	3	No	
Code Development and Translation			
Create program library	3	Yes	
Create prog. lib. interdep.	6	Yes	
Develop package specs			
create package spec.	3	Yes	A private part generator is provided. Editor is syntax directed.
modify package spec.	3	Yes	
delete package spec	gen	Yes	
Develop package bodies			
create package bodies	5,7	Yes	

A null body generator is provided

modify package bodies	5,7	Yes	
delete package bodies	gen	Yes	
Query and manip. prog. lib.			
list unit names	8	Yes	
list unit types	8	Yes	
list prog. lib. interdep.	gen	Yes	Links.Display command
list package interdep	8	Yes	
list subprog. interdep.	8	Yes	Via Xref command
determine completeness	8	Yes	Via Compilation.Make with wildcard or <Compile (All Worlds)> key.
determine recomp	8	Yes	Compilation.Make, effort only switch on.
remove unit	gen	Yes	Bodies must be removed before specs. Specs cannot be removed until all dependent units have been demoted to source.
clear prog. lib.	gen	Yes	Compilation.Delete with wildcard, and Library.Expunge or Compilation.Destroy.
Translate code			
trans. into a prog. lib.	3,5,7	Yes	
create x-ref. map	6	Yes	
display error messages	3,5,6	Yes	
list subprog. interdep	8	Yes	
pretty print source code	gen	Yes	
Create executable image	6,8	N/A	R1000 links at run time. Pragma Main forces prelinkage.
Execute code			
halt/resume/terminate execution	6	Yes	See Unit Testing and
trace execution path	6	Yes	Debugging Experiment, Chapter 5.
clock CPU time by subprog.	gen	No	

4.4. Experiment Answers

Question	Response
DD1	**Describe the mechanics of importing data from the OS.**
	The Rational Environment is the host operating system for the R1000 computer. Data is made available to programs and command procedures through "operating system" routines. All of these routines are callable Ada functions or subprograms.
DD2	**What tools are provided by the environment to monitor/query CPU and elapsed time for a tool, memory utilization for a tool, storage requirements for files, directories, and program libraries?**
	The package !Tools.System_Utilities provides a function **Cpu** that returns the elapsed CPU time for a specified or current job. The execution of a tool can be the specified job. The memory utilization of a tool can be monitored by the procedures What.Users and What.Jobs which return information about processes in the system. The storage requirements for a tool are displayed using the library listing. Storage requirements for objects (files, directories,

and program libraries) are found either through the library list command or the package !Tools.Directory_Tools.Statistics function **Object_Size**. The library listing command also provides date modified, last user to modify, size and class of objects, and status of Ada objects (archived, source, installed, or coded.)

DD3 **Describe the extent to which user interface customization is possible, including support for user-defined command procedures, command aliases, and key bindings.**

All operations in the R1000 Environment are performed by Ada subprograms, which can be bound to a key or executed from a command window. All the procedures that constitute the R1000 Environment can be incorporated in user-written procedures, which can be executed from command windows or bound to keys. Since Ada procedures can be renamed, aliases can be provided for any Environment command or user-defined procedure.

The R1000 terminal provides twenty function keys and three modifier keys (Shift, Control, and Meta) for a total of 160 key combinations available for binding. Of these, 96 are bound to common operations in the Environment and 64 are available to be bound to user-created procedures or to other procedures in the Environment. Since Rational provides a keyboard template that indicates which procedure is bound to which key combination, the multiplicity of key combinations is easily mastered in practice. The keyboard template also provides space for recording any user-defined key bindings. Default key bindings include debugger operation; directory traversal and listing; help; promotion and demotion of Ada objects; creation of directories, worlds, and text files; browsing commands; job management; and system queries. The program that binds procedures to a key can specify whether the procedure will execute immediately or appear in a command window with prompts for the procedural parameters provided. In this experiment user-written procedures that were used to collect size and timing statistics were bound to the keyboard.

Command windows provide a block within which any procedure visible in the current default context can execute. Command windows use a standard context clause that makes all common procedures for operating the Environment visible. Users can edit the context clause of a command window to make any subprogram they want visible. Procedures can be created (and bound to keys) that will create a customized command window. When a procedure name has been typed into a command window, a named parameter association for the procedure can be generated with the key Complt. Associated with each parameter is a prompt. If a parameter has defaults then the defaults appear in the prompt. If there is no default for the parameter, the type or subtype indication appears in the prompt. The user can move from prompt to prompt using the next item key.

A user indicates at login the name of the current session. Associated with a session name is an extensive set of switches for modifying the operating characteristics of the Environment. Among other things, switches control the layout of the screen and the information recorded in system log files. Users can maintain as many sessions and associated sets of switches as they desire.

When a user logs in, the system looks for an Ada program called *login* in the user's home directory and executes it. Login can perform any additional customization of the Environment not provided by session switches such as selecting a directory in which to start work. Another procedure,

Rational_Commands in the user's home directory will perform any key bindings the user requires that are global to all sessions. The procedure login will always be invoked upon a user login, while Rational_Commands will only be invoked if the user is logging in from a Rational Terminal. Separating key binding commands from the login procedure prevents the misapplication of bindings to an inappropriate terminal. Templates to construct Rational_Commands and other Environment-supported terminals can be found in !Machine.Editor_Data.

DD4 **Describe the mechanics of using the environment to define objects and operations during the detailed design process.**

The Rational Environment provides no support for graphical design.

DD5 **Describe the mechanics of creating a program library.**

In the Rational Environment any directory or world can serve as a program library. Directories or worlds are created by the procedures Create_Directory and Create_World in package !Commands.Library. The procedures are bound to function keys <Create Directory> and <Create World>. Directories or worlds are easily created by pressing these keys and then typing the name of the directory or world in response to the prompt appearing in the command window.

DD6 **What are the CPU and clock times for creating a program library?**

CPU Time: 0.70 seconds. Wall Clock Time: 1.73 seconds

DD7 **What are the space utilization ramifications of creating a program library?**

Directory creation consumed 7426 bytes.

DD8 **How easy/difficult is it to create a program library?**

Very easy; the creation requires a single keystroke followed by typing the name of the library in response to a prompt, followed by the <Promot> key.

DD9 **Describe the mechanics of entering Ada source code: package specifications, package bodies, subunits, and subprograms.**

The R1000 does not maintain separate files for source code, object code, and executable images; it maintains one Ada object, which may be in one of four states, archived, source, installed or coded. Initial entry of a unit using the Ada Object Editor creates an image in source state. The Ada unit can be demoted to archived state, which is much more compact than the source state. In archived state, the unit need not be syntactically or semantically correct, and cannot be edited. The unit may be promoted back to source state. Units in the source state can be freely edited. Installed units are syntactically and semantically correct and are visible to other units. Only those portions of installed units on which there are no dependencies can be edited. Declarations and statements may be freely added to installed units. Coded units have had machine code generated. An Ada unit in the coded state cannot be edited except for the addition, deletion, and modification of comments; it must first be demoted to the installed or source state.

The R1000 editor can create either Ada objects or text. What the editor creates is determined by the method of invoking it. To create text, the editor is invoked by pressing the create text key and typing the name of the text file desired in response to the prompt generated by the keystroke. The name of the text file appears in the directory listing and a window into which text may be entered is opened. To create an Ada object, the user presses the object

key followed by "I" or "i" for insert. This causes a window into which the object may be typed to be opened. When the object is promoted to the installed state for the first time, the name of the compilation unit appears in the directory listing. Prior to the first promotion the object is anonymous (does not appear in directory listing) or appears with a system-generated name if certain operations such as committing the object to disk are performed. Subunits cannot be created as separate Ada objects. They are generated either by entering a procedure body in line, selecting the procedure with the object select keys and entering the Make_Separate command in a command window attached to the object being edited, or by selecting a subunit declaration and pressing the edit key, which creates a window containing a skeleton of the subunit. Subunits can be brought in line with the Make_Inline command.

A bug exists in the editor whereby complex editing operations make it fail to recognize a legal Ada statement. This bug was bypassed by deleting small sections of source and re-entering them. The observed frequency of the editor problem decreased from the Gamma0 and Gamma1 Releases of the Rational Environment. The problem was observed only once in conducting the experiment using the Delta Release. The power of the editor outweighs this problem.

Entering an Ada object is easy and efficient once the R1000 editor is mastered. Since the editor is highly sensitive to environment, its capabilities are more fully described in the answer to DD10.

DD10 **Is the editor sensitive to the environment?**

The Rational Environment is controlled either through keys to which environment procedures are bound or through command windows. Since the editor is used for editing Ada objects, text, and command windows, it is the interface to the Rational Environment; a user never leaves the editor. Even when a user is supplying input to a user-developed Ada program through a window generated by Text_IO, the editor functions are available. A user can, for example, use editor functions to copy part of a window generated by Text_IO into a request for input generated by Text_IO.

The Ada Object Editor offers syntactic and semantic completion of constructs and interactive syntax and semantics checking. The syntax-sensitive features and local semantics are accessed through the format key, and the semantics-sensitive features are accessed through the complete and semanticize keys.

When the format key is pressed to complete the syntax of an Ada construct, the editor prompts for missing elements. For example, if the programmer types "procedure Foo is" and then presses the format key, the editor responds by generating the following completion:

```
procedure Foo is
begin
  [statement]
end Foo;
```

The cursor is positioned on the prompt ‘‘[statement].’’ The programmer can move between prompts with the next item and previous item keys. Prompts disappear when the programmer types on them. The syntactic completion capability of the editor allows a programmer to prevent many syntax errors by having the editor format Ada constructs. The format key will also correct some syntax errors of omission by performing syntactic completion.

The editor knows all the semantic information available in any library. The

editor uses this information to offer semantic completion of subprogram calls. The user enters the name of a subprogram in an Ada object and selects it with the object select keys. When the complete key is pressed, the editor generates a named parameter association for the subprogram, with prompts for the actual values giving the type of the expression required. If a parameter has a default value then this is supplied in the named association instead of a prompt. When the complete key is used to generate named parameter associations, code containing a semantic error will be generated if all of the following conditions are true:

1. The procedure being completed is from a package in the with clause of the unit being edited.
2. There is no use clause for the package.
3. The parameter has as a default value of a type declared in the package in the with clause.
4. The default value is not written using the dot notation to include the package name.

In these circumstances, the default copied into the parameter association will be undefined.

In addition to syntactic and semantic completion, the editor provides interactive syntax and semantics error correction with the format and semanticize keys respectively. When syntax or semantics errors are found, they are underlined. The user can move between errors discovered using next item and previous item keys. Explanations of each error are available by pressing the explain item key. By performing syntax and semantics checks after every statement or small group of statements, a programmer can be assured that when the last line of a program has been entered and checked, the entire program is completely correct—both syntactically and semantically.

DD11 **Is the editor an OS editor or is it specific to the environment?**

The editor is specific to the Rational Environment and the Ada language.

DD12 **Describe the mechanics of translating a compilation unit into a specified program library.**

A program library is either a directory or a world. An Ada object will be compiled into the library in which it was inserted. The procedure for inserting an Ada object into a library is described in DD9. The Rational system divides compilation into two separate stages:

1. The creation of the DIANA tree that represents a syntactically and semantically correct Ada program.
2. Code generation.

The DIANA tree is created and incrementally checked for syntactic and semantic correctness in the editor. In the source state the DIANA tree can be freely edited. In the installed state elements can be added to the tree, and elements on which nothing is dependent can be changed or deleted. In the coded state the DIANA tree cannot be edited except for its comment nodes. While editing, each time the format key is pressed, the syntax of new text entered is checked. Each time the semanticize key is pressed the entire unit is checked for semantic correctness. At any point when the DIANA tree is syntactically and semantically correct, the compilation unit can be made visible in the program library by promoting it to either the installed or coded

state. A correct DIANA tree in the coded state may be incomplete in the sense that its image still contains prompts indicating incomplete Ada constructs. Programs containing incomplete Ada constructs will execute up to the time that a call is made to an incomplete construct, at which time the system raises Program_Error.

To perform promotion either the unit name is highlighted in a directory listing or the cursor is placed in an image of the Ada object. The user then has the following alternatives:

1. Press the promote key, which moves object up one state (from source to installed or installed to coded).
2. Press the install key, which moves the object to the installed state (from either the source or coded state).
3. Press the code key, which moves the object to the coded state (from the source or installed state).

The R1000 also provides procedures for converting text files into Ada objects. The primary use of these procedures is not program development but rather importing Ada programs from other computers, such as when running the Ada Compiler Validation Suite.

DD13 **How easy/difficult is it to translate a compilation unit?**

Very easy. One keystroke will translate a unit when the cursor is in an image of the object or when the object has been selected in a directory listing with the object select keys.

DD14 **How much translator error correction is automated?**

The R1000 translator is designed to compile incrementally on a statement-by-statement basis and is intended to be used frequently during interactive program editing. Syntax and semantics checking are separate operations that the user can invoke. They are integrated with the editor so that corrections made by the translator are inserted in the source. To this end the primary error correction performed by the translator is syntactic completion.

DD15 **How tolerant of simple syntax errors is the translator?**

Very tolerant. It will correct many syntax errors of omission.

DD16 **How informative are the translator error messages?**

For interactive incremental compilation, translator error messages are of two types. When an error is discovered it is underlined in the source. An explanation of any underlined error can be requested with the explain item key. Often, having the location of the error pointed out with underlining is sufficient for the programmer to immediately locate the problem. Sometimes the underlining provided by the syntax checker is misleading in that a correct construct preceding the error is underlined. In these cases the message produced by the explain item key will usually provide a clue that allows location of the error.

Semantic error messages are usually straightforward and informative. Examples are:

```
"Operator "*" contains no return statement"
"INDEX is undefined"
```

A syntax error message is expressed in terms of an unexpected token recognized by the parser and a list of expected tokens. When the list of what was expected is short this is helpful. For example, the function declarations in the

matrix management specification that separates parameters with commas rather then semicolons generates an underlining of the comma and the following message:

```
Saw ",", expected: ":=", ")", ";"
```

A long message from the parser is less helpful. For example, the missing "is" in the declaration of pairwise vector multiplication in Vector_Management body generates an underlining of U_Len and the following message:

```
Saw "U_LEN", expected: "eof", "/=", "<=", ">=",
".", ":=", "=>", "**", "<", ">", "=", "(",
")", "*", "/", "+", "&", "-", ",", "|", ".",
"'", "and", "in", "is", "loop", "mod", "not",
"or", "range", "rem", "renames", "then", "xor"
```

The Environment is designed with the intent that the semantics and syntax checking capabilities be used frequently. Proper use of these facilities will insure that the scope within which any part of the program can be incorrect is kept small.

The Environment also provides facilities for compiling text files. The compilation process turns them into Ada objects. The error reporting provided by the procedures used to compile text files is rudimentary compared to the error reporting available when performing interactive incremental compilation in the editor. An error report for a text file compilation consists of the line number and column in which the error occurred, the token(s) expected, and the token found. After one error is found, the text compilation procedures abandon the compilation effort.

DD17 **What are the CPU and elapsed times for translating a compilation unit into a specified library?**

The following times include both installation and coding of the given procedure.

```
Time for Vector_Management specification
Elapsed Time: 5.52 seconds
CPU     Time: 2.48 seconds

Time for Matrix Management specification
Elapsed Time: 3.00 seconds
CPU     Time: 1.37 seconds

Time for Vector_Management body
Elapsed Time: 6.13 seconds
CPU     Time: 3.09 seconds

Time for Vec_Main body
Elapsed Time:  3.83 seconds
CPU     Time:  2.76 seconds
```

DD18 **What are the space utilization ramifications of translating a compilation unit into a specified program library?**

The semantics of a Rational library are conveyed by both the Ada objects contained in a library and by information maintained in the enclosing context. The division between these two is not clear. Therefore, space utilization is reported as the sum of the change in library size generated by installing an object and the size of an object installed. (Note that the size of a directory does not include the size of the objects contained within it; it includes only the information in the directory itself. Some information separate from an Ada object is generated in the directory by installing an object). No space is consumed by coding: only installation consumes space.

```
Space for Vector_Management specification
Library space:   125 bytes
Object   size: 13281 bytes

Space for Matrix_Management specification
Library space:   125 bytes
Object   size: 11365 bytes


Space for Vector_Management body
Library space:   125 bytes
Object   size: 13083 bytes

Space for Vec_Main body
Library space:  1365 bytes
Object   size: 34386 bytes
```

DD19 **Describe the mechanics of using the environment to design data structures, program units, program units interfaces, and control flows.**

No graphic design tools are available.

DD20 **Is a graphical interface supported?**

No. The Rational Environment does support a character-oriented, windowing interface on a 66 line by 80 column screen. The number of windows is user definable, with from one to four windows being usable. The Rational windows do not overlap; they split the screen into horizontal segments.

DD21 **How much source code generation is automated (e.g. null body generation and completion of matching begin .. end statements)?**

Both a private part generator and a body part generator are available with single keystrokes in the editor. The body part generator can create bodies for single procedures and functions, as well as packages; however, it fails when attempting to create a body part for a task with no entries. Further automatic code generation capabilities are described in the answer to DD10.

DD22 **What are the CPU and elapsed times for translating a compilation unit into a specified program library?**

See DD17.

DD23 **Describe the mechanics of creating inter-program library dependencies.**

A program unit contained in library A is made visible in library B by executing the Links.Add command in library B passing the pathname of the unit in library A. Use of wildcards in the pathname allows making more than one unit in A visible to B with one Links.Add command. The Rational Environment checks dependencies across libraries so that a change to a unit in A will not be permitted if it would make obsolete a unit in B. In general, the Rational Environment will not allow changes in one unit that make another unit obsolete unless the unit that would be made obsolete is demoted to the source state. The Environment provides the Compilation.Demote procedure (which is bound to the keyboard) for performing the demotion of all units dependent on a given unit.

DD24 **Describe the mechanics of creating an executable module.**

Because the Rational Environment performs linking at runtime this question is inapplicable. Linking before runtime can be forced by including the pragma Main in a main procedure. Linking then occurs when the main unit is promoted to the coded state. This would be done only if speed of program execution had become a problem.

DD25 **How easy/difficult is it to create an executable module?**

See DD24.

DD26 **What are the CPU and elapsed times necessary for creating an executable module?**

See DD24.

DD27 **What are the space utilization ramifications of creating a executable module?**

See DD24.

DD28 **How informative are the execution time error messages?**

When an exception is raised, the Rational Environment reports only the name of the exception and the name of the package in which it was raised.

DD29 **Describe the mechanics of executing a module, including I/O redirection, execution interruption and resumption and termination.**

Modules are executed by promoting a command window containing the name of the module. The Rational Environment generates humanly readable output and obtains input from users through Standard_Input and Standard_Output of Text_IO. Both normally default to an editor window. For any particular job Standard_Input and Standard_Output can be reset to be any text file by using the procedure Program.Run_Job. The file names of the desired input and output files are passed to Run_Job as parameters, as is the procedure name. Output can also be redirected for any particular job using Log.Set_Output. Run_Job is also used for scheduling a job to be run at a given time. Programs running in the background can be stopped, started, and killed with the Job.Enable, Job.Disable, and Job.Kill commands. The user can connect to any job with a Job.Connect command and can disconnect from the current job, forcing it to run in the background, by typing Control G.

DD30 **Describe the environment's mechanism for enforcing source code, object module, and executable module consistency.**

By combining the functions of source code, object code, and executable image into the Ada object, the Rational Environment ensures consistency. Note that the Environment will not allow a compilation unit A that depends on another compilation unit B to remain in the coded or installed state if changes are made in B that make A obsolete.

DD31 **Does the environment permit browsing of Ada Source code at varying levels of abstraction?**

Once Ada code has been installed, browsing of its uses and definition can be done with a single keystroke by selecting a construct.

DD32 **What are the space utilization ramifications of browsing a compilation unit?**

Reading an object using the editor does not create another version of the object.

DD33 **How easy/difficult is it to find an Ada object, subprogram specification, or body when its file or package location is unknown?**

The Rational Environment provides a browsing facility for locating definitions of objects and subprograms. By highlighting a name with the object selection keys and then pressing the definition key, the place where an object, subprogram, or compilation unit is defined will automatically be displayed. If the

definition is in the current image, the window on the image will be repositioned to show the definition. Otherwise, another window will be opened to show the definition. The Environment provides markers for a series of definition requests so that the users path through a series of windows can be retraced to the starting point. The Environment provides an Ada Other Part key to automatically bring a specification into view from a body, or to bring a body into view from a specification.

DD34 **How well integrated are the following environment tools: browser, editor, translator, and program library utilities.**

The browser, editor, and translator are completely integrated. Editing an object automatically places the user in the library into which the object is compiled.

DD35 **Describe the mechanics of querying and manipulation of a program library.**

Commands accessed with a single key press will be indicated by the name of the key enclosed in angle brackets. For example, the Create Ada key will be shown as <Create Ada>.

Manipulation of Libraries

Two commands, <Create World> and <Create Directory> create a library within the current context. A full pathname for the directory or world can also be supplied.

The current library is defined by the cursor position; it will be either the world or directory listing containing a cursor or the world or directory containing the object in whose image the cursor is positioned.

Libraries can be cleared by entering the compilation.delete command with the all object wildcard "@" in a command window opened on the library. The library pathname could also be supplied as a prefix to the wildcard.

Cleared libraries are deleted by highlighting their name in the context that encloses them and then pressing the <Object> D key sequence.

Interlibrary Linkage

A compilation unit in library A may reference a compilation unit in library B by creating a link to the other compilation unit in library A. Links are manipulated with link commands and are associated only with worlds, not directories. Links to a world outside (or inside) a given world are called external links. Links are also required to make units in a single program library visible to each other. Such links are called internal links and are automatically generated by translating into a library. The automatic generation of internal links can be disabled with a session switch. The link commands are as follows:

Links.Add takes the pathname of the unit for which a link is to be created. The name of the link is by default the name of the unit. If a name other than the name of the unit is specified, then the referenced unit is effectively renamed by the link.

Links.Copy copies the links or a subset of the links in one world to another world.

Links.Display shows all links associated with a world.

Links.Delete deletes a link or multiple links through use of wildcards.

Links.Dependents shows all Ada units dependent on a link.

Links.Display shows all links in a world.

Links.Replace replaces a specific link with another link

Several other link commands exist that offer subtle variations on the ones above.

Manipulation of Ada Objects

Ada units are made visible to other units in a library by promoting them to the installed state and made invisible to other units by demoting them to the source state. The method of inserting an Ada object into a library was described in the answer to DD9. They are deleted from the library by selecting them with the object select keys and pressing the object key followed by "D" or "d" for delete. An object cannot be deleted until all the units that depend on it have been demoted to the source state. Units may be copied from one library to another with the Library.Copy and Library.Move commands.

Library Queries

Directory listings (which include all objects in a world, or directory,) can be set to indicate the type of an Ada object (procedure body, package spec or body, subunit). This can be set either while in a directory or by session switches. Session switches can be set to cause the directory listings to display the state of Ada object (source, installed, or coded), although this significantly slows listings of long directories.

Listings of only the Ada objects in a library together with their state (archived, source, installed, or coded) and the type of unit can be generated by the Ada_List command. Ada_List has one deficiency; it does not show subunits in the library that appear in directory listings. Further information about all objects in a library such as last modifier, date of last modification and size can be obtained with the Library.Verbose_List command. A listing of the files in a library (which holds text or data) can be obtained with Library.File_List command.

The command Compilation_Make, with the effort-only switch set to true, shows all compilation activity required to bring the transitive closure of a given unit to a given state (installed or coded). Compilation_Make will not show missing bodies. However, a program exception will be raised during execution if an empty body is required during execution.

The Xref.Uses command generates a listing of all constructs upon which a given unit depends.

The Xref.Used_By will generate a listing of all units dependent on a given construct.

A series of parameters controls the granularity with which the Xref commands display dependency information.

DD36 **How easy/difficult is it to manipulate and query the program library?**

Most library query and manipulation operations are performed with either one or two keystrokes or by executing one command from a command window.

The following note applies to questions DD37 through DD42. The Rational Environment does not allow alteration of coded Ada units except for alteration, addition, and deletion of comments. Thus, to make changes other than to comments, the unit being changed must be demoted to the installed state. If other units are dependent on some part of a compilation unit being changed, then all units involved must be brought to the source state before

changes can be made. However, if upwardly compatible changes are made (such as adding type declarations to a specification) then dependent units need not be demoted. If changes are made to a package body, it must be demoted to the installed state, and then changes can be made incrementally. Moving dependent units to the installed state is automatic when a given unit is demoted to the installed state. Moving dependent units to the source state is accomplished with the Compilation.Demote procedure, which is bound to the keyboard.

This behavior is unlike that of a conventional edit, compile, link system in that the demotions must take place before editing is permitted. One might say that decompilation behavior precedes recompilation behavior. The requirement that units dependent on a declaration in a given unit be demoted to the source state prior to making changes ensures that the system will never be in an inconsistent state.

DD37 **What is the system recompilation behavior resulting from modifying a referenced package specification?**

To modify the Vector_Management specification, it had to be demoted to the installed state. This automatically demoted all dependent units to the installed state. Before the specification of the pairwise vector multiplication function could be incrementally deleted, the Rational Environment required that the units depending on that specification, Vec_Main and Vector_Management body, be demoted to the source state. A system recompilation using Mat_Main as the main procedure produced the following results:

```
Vector_Management body was installed.
The following were moved from installed to coded:
Matrix_Management spec
Mat_Main body
Vector_Management body
Matrix_Management body
Get_Col body
Get_Row body
```

DD38 **What is the system recompilation behavior resulting from modifying a subprogram body in a package?**

Only package had to be recompiled, not system.

DD39 **What is the system recompilation behavior resulting from deleting a subprogram body in a package?**

Only package had to be recompiled, not system.

DD40 **What is the system recompilation behavior from adding a subprogram body to a package body?**

Only package body had to be recompiled, not system.

DD41 **What is the system recompilation behavior resulting from adding a subprogram specification to a package specification?**

Same as DD37 except that Mat_Main had to be moved to source before the function specification could be added to Vector_Management spec.

DD42 **What is the system recompilation behavior resulting from adding comments to a package body?**

No recompilation (or compilation). Comments can be added to a coded unit without moving it from the coded state.

DD43 **What is the system recompilation behavior resulting from adding comments to a package specification?**

See DD42.

General Questions

Functionality

DD44 **Describe the mechanics of using the online help facility.**

The help facility is accessed by using five keys: <Help>, <Help on Help>, <Help on Key>, and the <Prompt For> <Help> combination. <Help on Help> key opens a window describing how to use the other help keys.

<Help on Key> prompts for a keystroke or two keystroke sequence. In keystroke sequences, the first key always indicates the entity of interest (object, region, window, image, line, work, or mark) and the second key indicates an operation to be performed on the entity, such as "D" for delete. After the keystroke, the <Help on Key> presents a window that describes the procedure (if any) bound to the key or keystroke sequence that was entered.

<Prompt For> <Help> prompts for a string. If the string is the name of a command, then a window explaining the command is opened. Otherwise, a list of all commands containing the string is displayed. For example, typing "delete" at the prompt displays a list of all the delete commands, such as library.delete, links.delete, compilation.delete, object.delete, etc. To identify a command, and thereby present a procedure description rather than a list of procedures, <Prompt For> <Help> requires only enough of the command name to uniquely identify it.

DD45 **Does the program library use a DIANA tree or any intermediate representation?**

Ada objects are stored in the Rational system only as DIANA trees. When machine code is generated, it is attached to the DIANA tree representing an Ada object.

DD46 **If an intermediate representation is used, is it instead of or in addition to source code files?**

The Rational Environment DIANA trees are used instead of source code files.

User Interface

DD47 **Characterize the interactivity of the environment in terms of general responsiveness and information content of the environment's feedback.**

The Rational Environment presents a user-definable number of windows on a character-oriented, 66-line screen. Each window representing a library or object is labeled with a pathname of the library or object. Windows representing program output are labeled with the name of the program unit generating them. When a window is open on a directory, a scrollable directory listing appears in the window. The Rational Environment thus provides excellent feedback on where a user is within the hierarchical directory structure and what the contents of any window represent. The Environment maintains a listing of windows that have been opened during a session and provides commands to return to any window on the list. To allow the user to manage a work session, windows can be selectively removed from the list so that a long session need not generate an overwhelming list of objects visited.

Execution of many system commands generates a log indicating all actions that the system is taking and any error conditions that have occurred. Log messages generated by the system are maintained for an entire session and are available for review at any time. This logging capability is also provided for the debugger so that all commands issued and responses generated during a debugging session are automatically available for review.

DD48 **Qualitatively summarize the learning curve as it applies to using the environment for programming in the small activities.**

The methods of navigating, issuing commands, and generating programs in the Rational Environment are sufficiently different from the methods of command line oriented environments that use the traditional edit, compile, link cycle that intensive initial training is required to use the Environment at all. This training is typically provided in a three day, hands-on seminar by the staff of Rational and is sufficient to allow a programmer who knows Ada to perform all the kinds of work addressed by this experiment.

Since the user is always in the editor, and since Ada procedures are the only means of performing operations in the Environment, the Environment has a highly consistent interface. Thus, once the user is over the first major hurdle of learning how to manipulate the Environment, the primary additional learning required of users is to assimilate the range of Ada procedures that the Environment provides.

When users attempt to use a range of features broader than those covered in initial training, the Environment will provide occasional minor but frustrating puzzles about how to accomplish specific tasks for some time. However, this problem is by no means unique to the Rational Environment.

DD49 **Describe the user interface's error handling, including its tolerance for minor errors and clarity of error messages.**

Using an invalid combination of keys for invoking a command generates a beep. Since the translator parses the command window, errors in commands issued through the command window generate standard Rational translator error messages. The location of the error is underlined, and an explanation of the error is available with the explain item key.

DD50 **Assess the general helpfulness of the user interface (e.g., command completion and command history retrieval mechanisms).**

The Rational Environment allows the contents of any command window to be re-executed or edited and re-executed. Previous command window contents can be retrieved by placing the cursor in the command window and then using the object undo and object redo keys to traverse a stack of command window contents maintained by the system. When a retrieved command window is executed, the windows above it on the stack are discarded. The Environment will recognize the shortest command abbreviation that provides an unambiguous name for the command. The complete key will bring up a named parameter association for any command typed into the command window with prompts for the parameter indicating the type of input expected. Command parameter defaults automatically appear in the parameter association.

DD51 **Assess the quality of written documentation. Pay particular attention to support for the translator and other primary tools.**

Rational documentation (see [6]) is organized into a *User's Guide*, the *Basic Operations Manual*, the *Reference Summary*, and a series of 11 reference volumes describing all the procedures in the Environment.

The *User's Guide* contains material on becoming familiar with the terminal, editing text files, and developing simple Ada programs.

The *Basic Operations Manual* summarizes the material covered in Rational training and is an easy to use and concise guide to how to perform the basic Environment operations for those who have had Rational training.

The remainder of the documentation is reference material describing the procedures and parameter types comprising the Environment. Within each package, procedures and parameter type descriptions are listed in alphabetic order. Packages are grouped into logical sections such as work space management, editing images, and so forth. Each section contains a table of contents and an index. Many sections contain introductory material that covers information of a scope wider than a package (such as information about wildcards used across the Environment procedures).

Software engineers who had the initial Rational training, but no experience with the Rational Environment, had difficulty using the reference material to solve some problems because the material describes the Environment at the program interface level of the procedures and parameter types in the packages comprising the environment. The common reaction has been "If you already know how to do it, you can easily find the references."

DD52 **Describe the helpfulness of the online help facility, including support for common activities (i.e. translation, editing and creating executable modules).**

The help facility is described in the response to DD44.

DD53 **Describe any noted inconsistencies exhibited by the environment.**

Some commands which require string parameters provide the quotes in the command window prompt, while others require the user to remember to type the quote marks.

Missing subunit bodies are not detected by keys <Code (All Worlds)> and <Code (This World)> bound to the Compilation.Make command. However, missing package bodies are detected. During system development this serves as a stubbing facility. A system with missing subunits may be executed, and will execute unless a missing subunit is called. In this case execution will halt with a Program Error. For later phases of development (integration testing, product testing), missing subunits not discovered until runtime could be a problem. Missing subunits and missing package bodies should be treated identically by a compilation procedure which checks for closure.

DD54 **How well does the environment support the concept of representing multiple views of Ada code?**

See DD31.

System Interface

DD55 **Assess the communications bandwidth of the editor (e.g. screen oriented and/or line oriented; supports multiwindowing).**

The Rational editor is multiwindowing and allows text to be copied or moved from one window to another using exactly the same commands used to copy or move text within a window. Multiple objects can be open at once, and multiple windows can be opened on an object by copying an open window with management attributes and operations can be associated the <Window> C key combination. The editor offers syntax and semantics checking and completion. Rational stated in a training session that the maximum effective baud rate of the Environment for some operations is 4800 baud due to the speed of Rational terminal hardware. Although powerful, the multiple windows, command windows, and accompanying reverse video banners make a cluttered screen, and can be confusing to the novice user.

DD56 **Assess the communication bandwidth of the environment.**

The Rational Environment has the capability of presenting large amounts of information simultaneously. The system message window, the 3 default user windows, command windows, and reverse video window banners can be confusing for the novice system user. Learning to make use of the information presented and learning to tailor the windows contributes to the steepness of the novice user's learning curve.

DD57 **How is the underlying OS file system utilized for the implementation of the environment database?**

There is no host OS underlying the Rational Environment.

DD58 **How difficult is it to use OS tools from the environment?**

Since there is no distinction between the OS and the Rational Environment, this question does not apply.

4.5. Design and Development Analysis

4.5.1. Functionality

Features of the Rational Environment allow the use of Ada as a compilable design language. The Environment does not require that Ada constructs in a compilation unit be complete before they can be compiled; prompts for declarations, expressions, and statements can be left in an Ada object that can be compiled and run. Typical strategies for allowing compilation of incomplete designs, such as use of a TBD package, are not required in the Rational Environment. A program unit containing prompts can even be executed, and will only result in a runtime Program Error when incomplete code is encountered.

The Rational Environment does not provide any graphical design aids for code development.

Ada source is entered through the Rational editor. The editor is unusually powerful, providing template generation, syntactic completion of Ada constructs, semantic templates for subprogram calls (by generating a named parameter association), and interactive, incremental syntax and semantics checking. By using the editor's error prevention and detection capabilities after entering each statement or small group of statements, entry of syntactically and semantically correct programs is greatly facilitated.

The procedures for installing source code in a library and subsequently generating machine code are all bound to the keyboard, so installation and coding are one-keystroke operations. These keystrokes can be used either when the cursor is in an image of an Ada object or when an Ada object is selected in a directory listing. Since the Rational Environment links at runtime, users need not perform linking before running a program. Linking before runtime can be forced by the use of pragma Main in a main procedure.

The Rational Environment will also generate a stubbed out Ada body when provided with a specification when the <Create Body> key is used. Also, if no specification in provided for a body, the Rational Environment provides one.

Program library creation is fast and easy, requiring only the press of a key to invoke the

Create_World or Create_Directory procedures. Specification of a current library is performed automatically by placing the cursor in a library or in the image of an Ada object contained in the library. A program compiled into one library can reference a program compiled into another library through the use of link commands that create pointers from one library to another.

The Rational Environment library system does not allow units in the library to become obsolete. If a change is made to a unit A that would make unit B obsolete, unit B must be made invisible to other units by demoting it to source before unit A can be changed. The Environment provides tools for automatically performing demotion that are similar to the tools provided for automatically performing promotion. The Environment also tracks dependencies between libraries so that a unit in library A referenced by a unit in library B cannot be changed without first demoting the unit in library B.

Environment switches can be set to show the state (source, installed, or coded) of all Ada units in a directory. Program libraries can be queried using the Ada_List and Verbose_List commands that list units and their state (source, installed, or coded) and the Xref command that generates cross reference listings. The tool for checking library completeness does not find missing subunit bodies.

All the basic functionality necessary to develop Ada code is provided in a very easy to use environment. Most common operations are initiated with single keystrokes. The editor greatly facilitates the generation of syntactically and semantically correct code.

4.5.2. Performance

A dedicated R1000 Model 200-20, running the Delta 0 release Environment was used when timing was performed. Program library creation and small compilation unit translation (consisting of package specifications of less than twenty lines of Ada code) were timed. Executable module creation takes place in the Rational Environment links at runtime. Linking can be forced during compilation by use of the pragma Main in the main procedure.

```
- Program library creation:

Elapsed time: 1.73 seconds.
CPU     time: 0.70 seconds.

- Small Compilation Unit (average of installation and coding
  of Vector_Management specification and Matrix_Management
  specification):

Elapsed time: 4.26 seconds.
CPU     time: 1.93 seconds.
```

Program library creation consumed 7426 bytes. Translating a small unit resulted in the utilization of 12448 bytes (average of Vector_Management specification and Matrix_Management specification object size and library size increase).

The recompilation characteristics of the Rational Environment are discussed in *Evaluation of the Rational Environment*.

4.5.3. User Interface

The user of the Rational Environment is always in the system editor and therefore has an extremely consistent system interface. The editor capabilities available to the user depend on the type of window being edited. The most restrictive is the directory window in which only the editor cursor movement and object selection and deletion operations are available. The editor is at its most powerful in editing Ada objects and command windows.

The command language of the Environment is Ada. Ada procedures can be invoked either by binding them to the keyboard of the Rational Terminal (which offers 160 function key combinations, 96 of which are prebound to commonly used Environment commands) or by entering them in a command window (created with the Create_Command key) that provides a declarative block in which Ada procedures may execute. The full editor capacities including syntactic and semantic completion, and interactive syntactic and semantic error checking are available for editing command windows. Since the Environment commands are Ada procedures, syntactic or semantic errors in entering a command generate Ada translator error messages.

The Rational Environment is a character-based windowing system. Windows can contain directories, Ada objects, text files, program output, etc. Windows are always labeled with a pathname of a directory or file or with the name of the program generating output in the window. Procedures for navigating the directory structure are bound to the keyboard, so many common movements (up a level, down a level, home, from spec to body, and from body to spec) are available with one keystroke. The system also has a browsing capability that provides a direct jump from the place where any type, subprogram, or variable is used to the place where it is declared. Markers can be set in the directory system to retrace a navigational path generated by browsing.

The Ada procedures comprising the system commands have been written to use a large set of wildcards, which makes navigation and complex directory operations easy. Command history is supported with the ability to re-execute command windows and to retrieve the previous contents of command windows. Many commands that change the state of the system (such as Compilation.Promote or Library.Copy) generate logs, which are retained for an entire user session. Since the user interface consists entirely of Ada procedures and key bindings, it is highly customizable. A problem in customizing the Environment is that many system-level packages, such as those that manipulate directories, are currently undocumented.

The help system is comprehensive and convenient to use. Help on any key or Environment session switch is available at a keystroke, and a facility for searching the help files by keyword is also available with a command procedure. Keywords are components of command names, not command parameters.

The user interface of the Rational Environment is very sophisticated and user friendly. It incorporates many features that enhance human computer interaction, including a windowing system, easy navigation through the directory system, a true system browsing capability, a powerful command language (Ada) and good online help.

The learning curve for the user interface is fairly steep. Users with three days training required about two weeks of using the Environment before they were completely comfortable with it, but productive work was performed during the acclimatization period. After using the Environment for an extended period users were reluctant to return to a conventional, command line oriented environment.

4.5.4. System Interface

The Rational Environment does not run on top of a host operating system. It was designed from the ground up as an Ada development environment, which has led to the complete integration of the Ada development tool set with the Rational operating system.

5. Unit Testing and Debugging Experiment

5.1. Introduction

The Unit Testing and Debugging Experiment exercises many aspects of unit testing and debugging. Provided are Ada units to test, support routines for a test harness, and test data files. Experiment instantiation calls for the construction of a test harness, using the capabilities of the APSE as much as possible. It also exercises browsing facilities, debugger capabilities (breakpoints, displaying variable values, tracepoints, etc.), static and dynamic analysis capabilities, and regression testing capabilities.

In the following experiment instantiation, experiment steps are numbered, and substeps are lettered. Any comments concerning the experiment step in the context of the Rational Environment are in regular type in braces ({ }). The instantiation of the experiment is provided as a transcript of actual keypresses. Comments as to the correct context in which to type the indicated keys, and comments as to what the keystroke will accomplish are indented, enclosed in braces ({ }), printed in a different `typeface`. The required keystrokes are indented and printed in `typeface`, and are indicated by the letter(s) appearing on the key or its keymap designation appearing in angle brackets (< >). When a command results in information appearing in the debugger window, the information appears following the command, indented and printed in a different `typeface`.

5.2. Experiment

1. Build and browse a test harness and the supporting subprograms to test the functions which implement matrix-vector multiplication, vector multiplication, and vector inner-product.

    a. Build a test harness by copying a subset of the required files from **Ada_LIB**: the main procedures, **test_harness** and a supporting I/O package, **testio_spec** and **testio_body_exe-errors**. (Note that the **_exe-errors** suffix indicates the presence of execution errors).

    {See 1.b.}

    b. Translate the previously mentioned procedure and package into the program library named **TEST_LIB** and then attempt to create an executable module. *Take note of the quality and content of the error messages.*

```
{Place cursor in the Test_Lib library and create
an Ada object.}
<Object> I
{Enter code for procedure Test_Harness and check
semantic correctness.}
<Semanticize>
{No errors found; promote to
coded state.}
<Code Unit>
{Return cursor to the enclosing library.}
<Enclosing>

{Enter in Testio'spec and Testio'body
in the same manner.}

{Run Test_Harness.}
<Create Command>
{Enter "Test_Harness" in the
command window.}
<Promot>
{Note Errors:  procedure bodies
Test_Matrix_Vector_Mult,
Test_Vector_Mult, and Test_Inner_Prod have
no coded bodies.}
```

c. Retrieve the missing subprograms from the following files in **Ada_LIB**: **test_matmult**, **test_vecmult**, and **test_innerprod**. Translate the subprograms into the program library **TEST_LIB**. Create an executable module.

{Same as 1.b.}

d. In order to become familiar with the test harness structure, browse its constituent subprograms. *Take note of the various browsing methods* available.

i. Browse the main procedure **TEST_HARNESS**.

```
{Move cursor to Test_Harness'body.}
<Definition>
{Now use the object keys and arrow keys to
browse through the procedure.}
```

ii. Browse the dependent (called) procedure **TEST_MATRIX_VECTOR_MULT**.

```
{Cursor is still in Test_Harness.  Place
cursor on declaration of procedure
Test_Matrix_Vector_Mult and select it.}
<Object> <Left Arrow>
<Definition>
{Now browse as in previous substep.}
```

iii. Browse the dependent package (WITHed) package specification **TEST_IO**.

```
{Cursor is still in
Test_Matrix_Vector_Mult, return cursor
to Test_Harness'Body.}
<Enclosing>
{Return cursor to TEST_LIB library.}
<Enclosing>
{Move cursor down to Testio'Spec.}
<Definition>
{Now browse as in previous substep.}
```

iv. Browse the function body of **GET_MATRIX**.

```
{Cursor is still in Testio'spec.  Move
cursor to declaration of Get_Matrix and
select it.}
<Object> <Left Arrow>
<Definition>
{Now browse as in previous substep.}
```

v. Browse the function body of **TEST_VECTOR_MULT**.

```
{Cursor is still in Get_Matrix'Body;
return cursor to Test_Lib.}
<Enclosing>
{Move cursor to .Test_Vector_Mult.}
<Definition>
{Now browse as in previous substep.}
```

2. Debug the test harness and associated supporting subprograms.

a. Create a small test data file using the format described in Exhibit 2.1 and the data (which abides by that format) shown in Exhibit 2.2.

{See 2.b.}

b. Execute the module using the test data file created above. This results in a CONSTRAINT_ERROR. (Note: the module reads from standard input and writes to standard output).

```
{Return to Test_Lib library.}
<Enclosing>
<Enclosing>
{Run Test_Harness using the test data from
exhibit 2.2.}
<Create Command>
"Test_Harness"
<Promot>
{Vector-Multiplication data works correctly.}
{Inner-Product data works correctly.}
{When running the Matrix-Vector multiplication
data a CONSTRAINT_ERROR is raised.  Terminate
the execution of Test_Harness.
<Job Kill>
```

c. Determine the cause of the CONSTRAINT_ERROR, modify the subprogram in error, and continue to execute the module. *Note the level of integration between the debugger, the editor, and the translator.*

i. Set breakpoints upon the raising of an exception. Execute the program and determine the problem statement and subprogram.

{The Rational debugger automatically breaks on exceptions unless the debugger behavior is modified to propagate exceptions with the propagate procedure.}

```
{Return to command window containing
"Test_Harness"}
<Window> <Up Arrow>
{Run Test_Harness again but under the
debugger.}
<Meta> <Promot>
<Execute>
```

{"Exception CONSTRAINT_ERROR (Array Index) caught at .TESTIO.GET_MATRIX.3s". Get_Matrix procedure is displayed with the line where the exception occurred highlighted.}

ii. Query the values of the variables **num_rows** and **num_cols** and the loop counters **i** and **j** in the procedure **TEST_IO.GET_MATRIX**. Determine the dimensionality of the variable **M**. Conclude that the upper bounds for the loop counters for the inner and outer loops must be reversed.

{There are two ways to display a value. The following transcript uses both.}

```
<Prompt For>
<Put>
{Command window opens; enter i
as value of parameter "Value"}
[Value => ""]
"i"
<Promot>
{Output to debugger execution
window:}
4
{To display value of j,
place cursor on "j," and select it.}
<Object> <Left Arrow>
<Put>
{Output to debugger execution
window:}
1
```

{Routine is trying to read in an 4 x 3 matrix instead of a 3 x 4 matrix. The loop counters need to be switched.}

iii. Modify **GET_MATRIX** so that the upper bound for the outer loop is **num_cols** and the upper bound for the inner loop is **num_rows**.

```
{Cursor is already in Get_Matrix'body.
Turn off the selection of the selected
line.}
<Item Off>
{Demote the state of Testio'Body to
installed.}
<Install Unit>
{Place cursor at the beginning of the loop
to be changed and select the loop.}
<Object> <Left Arrow>
<Edit>
{An edit window opens displaying
loop code to be edited; make change
and return the corrected version to
the Testio'Body.}
<Promot>
{Return Testio'Body to coded state.}
<Promot>
```

iv. Restart execution at the beginning of the procedure GET_MATRIX, verifying that the error has been corrected.

{Job cannot be restarted. It must be terminated and a new one must be executed.}

```
{Return to Test_Lib library.}
<Enclosing>
{"Test_Harness" is still in command
window.}
<Create Command>
{Terminate the old debugger job and begin
a new one.}
<Meta><Promot>
<Execute>
{Type input to the executing program:
3 4 <Enter>
1.0 2.0 3.0 4.0 <Enter>
2.0 4.0 6.0 8.0 <Enter>
3.0 6.0 9.0 12.0 <Enter>
4 <Enter>
1.0 2.0 3.0 4.0 <Enter>
{Correct results appear in the
debugger execution window.}
```

3. Perform a static analysis of the module created in above (Step 1c). *Measure the CPU and elapsed times for performing the static analysis.*

   a. Examine the overall quality of the program's structure by:

      i. Identifying violations against a prescribed set of programming guidelines.

      ii. Producing a measure of each subprogram's complexity (e.g., McCabe's Cyclomatic).

      iii. Identifying unreachable statements.

   {No static analysis tools are available.}

   b. Collect statistics including (but not limited to):

      i. number of executable lines

      ii. percent comment lines

      iii. frequencies of statement types

   {No supported statistics tools are available.}

4. Create a library of test data using the specified test data files residing in **Ada_LIB** as necessary.

   a. Create a test data file to test the "normal" functionality of matrix-vector multiplication including the following cases (use **test_input_normal** in **Ada_LIB**):

```
{Return to Test_Lib library.}
<Window> <Up Arrow>
<Window> <Up Arrow>
<Create Text>
{A command window opens; supply
"Test_Input_Normal as the value of
parameter "File_Name."}
[File_Name => ""]
"Test_Input_Normal"
<Promot>
{Test_Input_Normal window opens,
enter data; when finished,
commit the data to disk.}
<Promot>
```

b. Create a test data file to test the following boundary cases (use **test_input_boundary** in **Ada_LIB**):

   {Same as above.}

c. Create a test data file to structurally test (i.e., test subprogram control flows) the subprograms implementing matrix-vector multiplication, vector multiplication, and inner product (use **test_input_structure** in **Ada_LIB**). (Note that the **MATRIX_MANAGEMENT** and **VECTOR_MANAGEMENT** packages are included in Appendix 6.B.)

   {Same as above.}

d. Create a test data file to stress test the three subprograms using data with combinations of large and small numbers (use **test_input_stress** in **Ada_LIB**).

   {Same as above.}

5. Perform the initial baseline test of the three mathematical functions: matrix-vector multiplication, vector multiplication and inner product.

   a. Create a file containing the expected output when using the files previously created in the preceding step. (Use **test_output_expected** from **Ada_LIB**.)

      {Create file in the same manner as in 4.a.}

   b. Execute the module using the test input data and create a file containing the actual test output.

      {In order to have the Test_Harness input be a given text file and the output be sent to a text file, Test_Harness must be run by using the command "!Commands.Program.Run_Job." The command Run_Job allows "Test_Input" specified as the input filename and "Test_Output_Actual" to be specified as the output filename.}

```
{Go to Test_Lib}
<Create Command>
"File_Utilities.Append"
<Complt>
(Source => "Test_Input_Normal",
 Target => "Test_Input");
<Promot>
{Repeat using Test_Input_Boundary,
Test_Input_Structure, and
Test_Input_Stress as the source file.}
<Create Command>
"Program.Run_Job"
<Complt>
(S => "Test_Harness",
Debug => False,
Context => "$",
After => 0.0,
Options => "Input := Test_Input;
Output := Test_Output_Actual",
Response => "<PROFILE>");
<Promot>
```

c. Compare the actual output to the expected output.

```
<Create Command>
"File_Utilities.Difference"
<Complt>
(File_1 => "Test_Output_Expected,
File_2 => "Test_Output_Actual",
Result => "",
Compressed_Output => False,
Subobjects => False);
<Promot>
```

{The only significant difference was in the result for the first case in Test_Input_Stress.}

```
{Expected}
Numeric_Error_Raised
{Actual}
-- input matrix
1.00E+00 2.00E+20 3.00E+40 4.00E+60 5.00E+80 6.00E+100
1.00E+00 2.00E+00 3.00E+00 4.00E+00 5.00E+00 6.00E+00
-- input vector
1.00E+01 2.00E+01 3.00E+01 4.00E+01 5.00E+01 6.00E+01
-- result vector
3.60E+102 9.10E+02
```

{The actual output does represent the correct answer.}

6. Perform a dynamic analysis of the module. *Measure the CPU and elapsed time for performing the analysis.*

   a. Collect performance statistics, including CPU time for the currently implemented subprograms which perform matrix, vector arithmetic.

      {No performance analysis tools are available.}

   b. Perform a test data coverage analysis, and identify sections of code that are not executed when using the test input data.

      {No coverage analysis tools are available.}

c. Collect general statistics, including the number of conditional and unconditional branches traversed and the number of times each subprogram was executed.

{No general statistics tools are available.}

7. Create a variation of matrix-vector multiplication using a parallel algorithm.

a. Substitute for the current implementation of matrix-vector multiplication the new implementation retrieved from **parallel_matmult** in **Ada_LIB**.

```
{Cursor is in Test_Lib library;
go to Project_Lib library.}
<Enclosing>
{Place cursor on Matrix_Management'Body
and select it.}
<Object> <Left Arrow>
{Try to edit Matrix_Management'Body.}
<Edit>
{Edit not allowed; note message in message
window that Get_Col'Body and Get_Row'Body
would be obsolete.}
{Go to Project_Lib.}
<Window> <Down Arrow>
{Place cursor on Matrix_Management'Body
and select it.}
<Object> <Left Arrow>
<Uncode (This World)>
{Open window on Matrix_Management'Body.}
<Definition>
<Edit>
{Enter code for Parallel_Matmult.}
```

b. Define a task type as follows in the declarative part of the package body of **MATRIX_MANAGEMENT**:

```
task type ROW_PRODUCT is
   entry SEND_VECTORS (Row, Col : in VECTOR);
   entry RETRIEVE_PRODUCT (P : out FLOAT);
end ROW_PRODUCT;

task body ROW_PRODUCT is separate;
```

c. Create the task body for **ROW_PRODUCT** by copying it from **row_product_exe-errors** in **Ada_LIB**.

```
{Enter code for Row_Product'body.  Place
cursor on first line of its declaration and
select the declaration.}
<Object> <Left Arrow>
<Create Command>
{Command window opens; enter:}
"Make_Separate"
<Promot>
{Commit data to disk.}
<Enter>
```

d. Create a new module employing the parallel implementation.

```
{Go back to Matrix_Management'Body.}
<Enclosing>
 {Go back to Project_Lib.}
<Enclosing>
{Place cursor on Matrix_Management'Body
and select it.}
<Object> <Left Arrow>
<Code (This World)>
{Get_Col, Get_Row, Row_Product, and
Matrix_Management go from Source (S)
to Coded (C) state.}
```

8. Regression test the new module and discover that the output differs from the initial baseline test output created when using the sequential algorithm. Discover that the new parallel algorithm yields different answers and also results in a deadlock situation.

   {Testing is done manually. The answers to the Matrix_Vector multiplication tests differ. The first case from Test_Input_Structure results in a deadlock situation.}

9. Debug the parallel implementation of matrix-vector multiplication.

   a. Set breakpoints in **GET_MATRIX** after every iteration of the outer loop.

```
{Go to Test_Lib.}
<Create Command>
"Test_Harness"
{Run with the debugger.}
<Meta><Promot>
<Create Command>
"Debug.Take_History"
<Promot>
{Activate history recording in order to
do a Show (histories) and History_Display
later on in the script.}
{Go to Test_Lib}
<Window> <Up Arrow>
<Window> <Up Arrow>
{Place cursor on Test_Io'body}
<Definition>
{Place cursor at beginning of the inner loop
in Get_Matrix'body and select it.}
<Object> <Left Arrow>
{Set breakpoint.}
<Break>
```

   b. Set a tracepoint after every iteration of the outer loop for the loop counter **i** and for the **i**th row of the matrix **M**.

      {The Rational Environment does not have tracepoints in the VAX/VMS sense. Therefore, the breakpoints which were activated in 9.a. will be used along with the proper Debug Put procedures.}

   c. Set a tracepoint for the **v** in **GET_VECTOR** for every time **v** changes value.

```
{Cursor is still in Test_Io.body;
move cursor to statement in loop in
Get_Vector'body and select it.}
<Object> <Left Arrow>
{Set breakpoint.}
<Break>
```

   d. Compare the values of **M** in **GET_MATRIX** and **v** in **GET_VECTOR** with the

respective values in the test input file and conclude that the data is being read properly.

```
<Execute>
"Matrix_Vector_Mult"
<Promot>
Testing Matrix-Vector Multiplication
"2 3"
<Promot>
{Break at Get_Matrix.2s}
<Run>
<Run>
"1.0 2.0 3.0"
<Promot>
{Break at Get.Matrix.3s}
<Prompt For>
<Put>
[Put => ""]
"i"
1
<Promot>
<Run>
<Run>
<Run>
{Read in first row of the matrix.}
{Display values read in for the matrix.}
<Prompt For>
<Put>
[Put => ""]
"M(1,1)"
<Promot>
1.00e+00
<Prompt For>
<Put>
[Put => ""]
"M(1,2)"
<Promot>
2.00e+00
<Prompt For>
<Put>
[Put => ""]
"M(1,3)"
<Promot>
3.00e+00
<Execute>
"4.0 5.0 6.0"
<Promot>
"3"
<Promot>
<Run>
"10.0 20.0 30.0"
<Promot>
```

```
{Break at Get_Vector.2s}
<Prompt For>
<Put>
[Put => ""]
"i"
<Promot>
2
<Run>
{Display values read in for the vector.}
<Prompt For>
<Put>
[Put => ""]
"v(1)"
<Promot>
1.00e+01
<Run>
{Break at Get_Vector.2s}
<Prompt For>
<Put>
[Put => ""]
"v(2)"
<Promot>
2.00e+01
```

e. Set a breakpoint upon rendezvous with **PARALLEL_ROW_PROD.SEND_VECTORS**.

{Due to naming conventions on the Rational Environment, "Parallel_Row_Prod.Send_Vectors" is listed as "Matrix_Management.Row_Product."}

```
{Go to Matrix_Management.Row_Product,
place cursor on rendezvous statement, and
select it.}
<Object> <Left Arrow>
{Set breakpoint.}
<Break>
```

f. Display task status, the current breakpoints and tracepoints, the program stack and history.

```
<Task Display>
Task_Display ("", All_Tasks);
Job:  232 Root task: #B54E8
ROOT_TASK, #B54E8: Step complete at
.TEST_HARNESS.TEST_MATRIX_VECTOR_MULT.9s [Pri = 1}

<Show Breaks>
Show (BREAKPOINTS>;
Active Permanent Break 1 at .TEST_IO.GET_MATRIX.2S
                                        [any task]
Active Permanent Break 2 at .TEST_IO.GET_VECTOR.2S
                                        [any task]
Active Permanent Break 3 at
                 .MATRIX_MANAGEMENT.ROW_PRODUCT.1S
                                        [any task]

{Go to debugger window.}
<Debugger Window>
<Create Command>
```

```
{Enter in command window:}
"Show (Traces)"
<Promot>
Show (TRACES);
No tasks are tracing calls.
No tasks are tracing statements.
No tasks are tracing exceptions.

<Stack>
Stack ("", 0, 0);
Stack of task ROOT_TASK, #B54E8:
_1:  TEST_MATRIX_VECTOR_MULT.9s
_2:  TEST_HARNESS.2s.3s
_3:  TEST_HARNESS.2s
_4:  command_procedure.1s
_5:  command_procedure  [library elaboration block]

<Create Command>
{Enter in command window:}
"Show (Histories)"
<Promot>
Show (Histories);
History of Calls is being recorded for:
        all tasks at all locations
History of Statements is being recorded for:
        all tasks at all locations
History of Exceptions is being recorded for:
        all tasks at all locations

<Create Command>
{Enter in command window:}
"History_Display"
<Promot>
History_Display (0, 0, "");
History of statements executed by all tasks :
                                          (oldest..newest)
Timestamp Depth Location and Task
251269226399  6 .TEST_IO.GET_VECTOR_DIM.1s
                                    [ROOT_TASK, #B54E8]
251276928611  5 .TEST_HARNESS.TEST_MATRIX_VECTOR_MULT.6s
 +    336     5     ....7s
 +    996     5     ....8s
 +   1265     5 .TEST_IO.GET_VECTOR.1s
 +   1383     6     ....1s
 +   1771     6     ....2s
251370692649  5 .TEST_HARNESS.TEST_MATRIX_VECTOR_MULT.8s
 +    487     5     ....9s
```

g. Display the values for **row_vector.all** and **col_vector.all** in **PARALLEL_ROW_PROD.SEND_VECTORS** and notice that they are equal at each rendezvous.

```
<Run>
<Execute>
<Run>
<Run>
<Prompt For>
<Put>
[Put => ""]
"Col_Vector.all"
[1..3]
[1 => 4.00E+00 ...
 2 => 5.00E+00 ...
 3 => 6.00E+00 ...]
<Prompt For>
<Put>
[Put => ""]
"Row_Vector.all"
[1..3]
[1 => 4.00E+00 ...
 2 => 5.00E+00 ...
 3 => 6.00E+00 ...]
```

h. Modify the line in **PARALLEL_ROW_PROD.SEND_VECTORS** which assigns **col_vector** to be

```
Col_Vector := New Vector'(Col);

{Go to Parallel_Row_Prod.Send_Vectors.}
<Item Off>
{Move cursor to and select statement
to be changed.}
<Object> <Left Arrow>
<Edit>
{Edit window opens with selected
statement in it; make change in edit
window and place change back in body.}
<Promot>
{Place body back in coded state.}
<Promot>
```

i. Set a breakpoint at the entry and exit point for the matrix-vector multiplication (MVM) function.

{Due to a problem in the debugger, was unable to set a breakpoint in the overloaded function "*". Overload resolution of infix operators was not working. To solve the problem, had to change all occurrences of "*" to the procedure call "Times(U, V)." The transcript of steps following still applies, even with the change.}

```
{Go to Matrix_Management'body, move
cursor to the first declaration statement,
and select it.}
<Object> <Left Arrow>
{Set breakpoint.}
<Break>
{Move cursor to the last statement in
the procedure and select it.}
<Object> <Left Arrow>
{Set breakpoint.}
<Break>
```

j. Jump to the entry point of the MVM function.

{In order to make use of changes made above, current job is terminated and a new one is started.}

```
{Put job into the background.}
<Control> <G>
<Job Kill>
{Go to Test_Lib.}
<Create Command>
"Test Harness"
<Meta><Promot>
{To re-use previous jobs' breakpoints,
must reactivate them.}
<Activate>
{Enter data.}
```

k. Set a tracepoint for the vector variable **product** upon exiting the MVM function.

```
{Continue through the breakpoints
to the second breakpoint, which is
just before the return statement
in the MVM function.}
<Execute>
<Prompt For>
<Put>
[Value => ""]
"Product"
1.4E+01
```

l. Using a test case that previously (before the parallel algorithm was implemented) resulted in the raising of an **ERROR** condition in **TEST_HARNESS**, determine where the deadlock occurs. *Assess the level of difficulty in determining the cause of the deadlock.* (Note that the **ERROR** condition is initially caused by the raising of **DIMENSION_ERROR** in the MVM function.)

{The deadlock occurs as a result of the Dimension_Error exception being raised. The function '*' attempts to end but a Debug Task Display shows that it has children tasks running and waiting at an accept for an entry call. Since a parent cannot end before its children end, the program becomes deadlocked. Upon examination of the situation the reason for the deadlock was fairly obvious.}

```
<Meta> <Promot>
<Execute>
5 3
1.0 2.0 3.0
1.0 2.0 3.0
1.0 2.0 3.0
1.0 2.0 3.0
1.0 2.0 3.0
5
10.0 20.0 30.0 40.0 50.0
Exception Vector_Management.Dimension_Error
caught at .Matrix_Management.times'N(2).2s
<Task Display>
#444FB(>Matrix_Management.Row_Product):Running,
waiting at accept for entry call [Pri = 1]
```

m. Use **row_product** from **Ada_LIB** to create a new module that no longer deadlocks. (Note that this new version of **ROW_PRODUCT** includes a set statement with a terminate alternative.)

```
{Define Matrix_Management'body.Row_Product.}
<Edit>
{Change to coded state.}
<Code Unit>
{Go to Matrix_Management'body.}
<Enclosing>
{Go to Project_Lib.}
<Enclosing>
```

10. Regression test the corrected module.

```
{Place cursor on Test_Lib.}
<Definition>
<Create Command>
"Test_Harness"
<Promot>
```

{Test is done manually. It works.}

5.3. Functionality Checklist

PRIMARY ACTIVITIES

Activity	Step #	Supported (Y/N)	Observations
Unit testing			
Create and debug test harness	1,2	No	
Create test input data for			
functional testing	4a	No	
boundary case testing	4b	No	
structural testing	4c	No	
stress testing	4d	No	
Perform initial test			
create expected output data	5a	No	Done manually.
produce actual output data	5b	Yes	
compare actual and expected data	5c	No	Done manually.
Perform dynamic analysis			
measure execution time by subprogram	6a	No	
perform test data coverage analysis	6b	No	
identify code not executed	6b	No	
measure statement execution frequency	6c	No	
Perform regression testing	8,10	No	
Debugging			
Set/reset breakpoints on program unit			
entry/exit	9i	Yes	
exception	2c	Yes	
statement	2c	Yes	
nth iteration of a loop	9a	Yes	
variable changing value	gen	No	
variable taking on a specified value	gen	No	
rendezvous	9e	Yes	
Control execution path			
jump n statements	gen	Yes	

enter a specified subprogram	2c,9i	Yes	
exit the current subprogram	gen	Yes	
Query program state			
display source code	2c	Yes	
display breakpoints	9f	Yes	
display tracepoints	9f	Yes	
display stack	9f	Yes	
display history	9f	Yes	
display task status	9f	Yes	
Modify program state			
modify variable values	2c	Yes	
add, modify and delete code	2c,9h	No	Changes can be made but they will not affect current debugging session.

SECONDARY ACTIVITIES

Unit testing

Perform static analysis			
check against prog. guidelines	3	No	
measure subprogram's complexity	3	No	
identify unreachable statements	3	No	

Debugging

Set/reset tracepoints on			
program unit entry/exit	gen	Yes	
exception	2c	Yes	
statement	9b	Yes	
nth iteration of a loop	9b	No	
variable changing value	gen	No	
variable taking on a specified value	gen	No	
rendezvous	gen	Yes	

5.4. Experiment Answers

Question	Response
TD1	**Describe the mechanics of using the environment for creating and debugging the test harness.**
	There are no tools for generating a test harness.
TD2	**Describe those aspects of test harness generation which are automated.**
	See TD1.
TD3	**How easy/difficult is it to use the environment for creating and debugging a test harness?**
	The same level of difficulty as developing any Ada program in the Rational Environment. Procedures to execute a program (Program.Run and Program.Run_Job), as well as routines to compare output with expected output (File_Utilities.Difference), are available to be used in a test harness.
TD4	**Does the debugger operate in a screen mode and/or command line mode?**

The debugger is always operating in a screen mode. Commands are entered either through use of keys to which debugger procedures are bound or through a command window that can be attached to any window on the screen. This treatment of the command interface is completely consistent with any other use of the Rational Environment.

TD5 **Is a multi-windowing capability available from within the debugger?**

The Rational debugger only operates in a multi-window mode. It uses a debugger window in which output of debugger commands is displayed and a source code window that displays source code when either single stepping or hitting break points. Commands and their parameters are echoed to the debugger window so that the window forms a complete log of a debugging session. During single stepping or when hitting breakpoints, the debugger highlights the line about to be executed in the source code window. If the program being debugged generates readable output, the output appears in a third window.

TD6 **Describe the mechanics of accessing test data from within the** debugger.

The mechanics of accessing test data from within the debugger are the same as they are outside the debugger since it is possible to move freely in and out of the debugger during a debugging session. The user opens a window containing the contents of the test data file.

TD7 **How easy/difficult is it to access test data from within the debugger?**

Very easy. See TD6.

TD8 **Describe the mechanics of setting and resetting breakpoints.**

There are two ways to set breakpoints. The simplest method is to select a statement in the source code window and then press the <Break> key. The second method is for the user to be prompted for the Debug.Break procedure in a command window. When executed from a command window, the statement on which the break is to be set is passed to the Debug.Break either by selecting it (as in the simple method) or by passing the name of the statement at which a breakpoint is to be set as a parameter. Statement names are the name of the procedure in which the statement occurs concatenated with "." and a line number. The line number may be determined with the Debug.Display procedure. Declarations and statements are numbered separately. When Debug.Break is executed from a command window, three parameters may be set: Stack_Frame, Count, and In_Task. Stack_Frame provides a means for specifying a frame in which to set a breakpoint. Count specifies the number of times the statement is to be executed before a break occurs. In_Task specifies that a break is to occur only if the code is executed by a particular task.

Once the debugger is invoked it remains active until the user logs off. This allows the debugger to remember breakpoints that have been set in a program. If a program is re-executed, the command "Activate (0)" reactivates all breakpoints previously set in the program. Specific breakpoints can be reactivated by passing in the break point identification number.

TD9 **How easy/difficult is it to set and reset breakpoints?**

Very easy using the select a statement and push a key method.

TD10 **Are graphical tools available from within the debugger to convey the program state?**

No.

TD11 **Describe the differences (if any) between invoking a module and invoking the module for debugging.** A module can be executed either by selecting it in a directory and pressing the promote key or by opening a command window, typing in the module name, and pressing the promote key. Similarly, a module can be invoked for debugging by selecting it in a directory and pressing the meta key, followed by the promote key, or by opening a command window, typing in the module name, and pressing the meta key followed by the promote key.

TD12 **Describe the mechanics of controlling the execution path.**

Two debugger procedures that control execution are bound to the keyboard. Debug.Execute causes the program to run until a breakpoint is hit. Debug.Run steps through a program until a specified event has taken place. A parameter to Debug.Run defines the event. Debug.Run is bound to the keyboard with two default events, Local_Statement and Statement. Local_Statement causes Debug.Run to step to the next statement in a subprogram without descending into the code of subprogram calls. Statement causes Debug.Run to step through a program descending into the code of subprogram calls. Additional parameters to Debug.Run allow stopping at the point of entry to procedures, just prior to returning from a subprogram, just after returning from a subprogram, and at the beginning and end of rendezvous.

For concurrent programs the debugger offers a rich set of commands to stop and start the execution of individual tasks or groups of tasks.

TD13 **Describe the mechanics of invoking the debugger. Does the program have to be retranslated?**

The mechanics of invoking the debugger are explained in TD11. Programs do not have to be retranslated before they can be run under the debugger.

TD14 **Describe the mechanics of modifying the program state.**

The value of a variable in a program can be changed with the procedure Debug.Modify. The variable to be modified can be indicated either by selecting it or by providing its pathname. The components of records and arrays must be modified individually; there is no provision for modifying them with aggregates. The debugger does not allow the following objects to be modified:

- access types (can be modified only to the null value)
- variables of task types
- constants
- *in* parameters
- discriminants of variant records
- *for* loop iteration variables

Code can be modified at any point from within the debugger in the same manner that it is modified from outside the debugger: Define the module to be modified, put it in the installed or source state and edit. Because the source code and the executable module are one object in the Rational Environment, this action causes the debugger to lose track of its location in the code. The statement to be executed ceases to be highlighted in the source

code window, the debug stack command displays the message "Program has been recompiled since debugger started," and variable values can no longer be displayed. Although an edited program will continue to execute, debugging must be restarted in order for the debugger to generate useful information.

TD15 **Describe the mechanics of querying the program state.**

Display source code:

Source code being single-stepped, source code around a breakpoint, and statements that raise an exception are automatically displayed in the source code window. The procedure Debug.Display will display source around a pathname. However, the pathname must include a declaration or statement number.

Display variable values:

Variable values are displayed by selecting the variable and pressing the debug put key or by invoking the Debug.Put procedure from a command window and passing a variable name. When Debug.Put is invoked from a command window, the variable name is evaluated in a default context which is the current scope. A separate procedure can reset the context in which the variable name is evaluated. Display of a selected variable is independent of the evaluative context.

Structured and designated objects can be displayed by user-written procedures to improve the readability of the output. This capacity is useful, for example, when displaying a linked list.

Display breakpoints and tracepoints:

Breakpoints are shown by pressing the Show Breaks key. Tracepoints are shown by invoking Debug.Show from a command window and passing the Debug.Trace parameter. Rational tracepoints do not show values of variables, they print a message when a particular event occurs in a task.

Display stack:

Press the Stack key.

Display history:

Invoke the Debug.Show procedure from a command window and pass the Debug.History parameter. History recording must be turned on by the Take_History procedure before a call of Debug.Show with the history parameter has an effect.

Display task status:

Press the Task Display key.

TD16 **How easy/difficult is it to control the execution path?**

Very easy.

TD17 **How easy/difficult is it to modify the program state?**

Very easy.

TD18 **How easy/difficult is it to query the program state?**

Very easy.

TD19 **How effective is the debugger in conveying the program state at any given point (e.g., source code straddling the current breakpoint, variable values in the current scope, the name of the calling subprogram)?**

The debugger can convey most aspects of the program state. Below is a sample session with the debugger.

```
 <Execute>
Execute ("");

Break ("", 1, "");
Break at selected object.
The breakpoint has been created and activated:
Active Permanent Break 3 at .MATRIX_MANAGEMENT.ROW_PRODUCT.1S [anyta
 <Task Display>
Task_Display ("", ALL_TASKS);

Job:  232, Root task:#B54E8
ROOT_TASK, #B54E8: Step complete at
.TEST_HARNESS.TEST_MATRIX_VECTOR_MULT.9s  [Pri = 1]
 <Show Breaks>
Show (BREAKPOINTS);
Active Permanent Break 1 at .TEST_IO.GET_MATRIX.2S [any task]
Active Permanent Break 2 at .TEST_IO.GET_VECTOR.2S [any task]
Active Permanent Break 3 at .MATRIX_MANAGEMENT.ROW_PRODUCT.1S [any t
 <Create Command>
 "Show (Traces)"
 <Promot>
Show (TRACES);
No tasks are tracing calls.
No tasks are tracing statements.
No tasks are tracing exceptions.
 <Stack>
Stack ("", 0, 0);
Stack of task ROOT_TASK, #B54E8:
_1:  TEST_MATRIX_VECTOR_MULT.9s
_2:  TEST_HARNESS.2s.3s
_3:  TEST_HARNESS.2s
_4:  command_procedure.1s
_5:  command_procedure  [library elaboration block]
 <Create Command>
 "Show (Histories)"
 <Promot>
Show (HISTORIES);
History of Calls is being recorded for:
     all tasks at all locations
History of Statements is being recorded for:
     all tasks at all locations
History of Exceptions is being recorded for:
     all tasks at all locations
 <Create Command>
 "History_Display"
 <Promot>
History_Display (0, 0, "");
   History of statements executed by all tasks : (oldest..newest)
    Timestamp Depth Location and Task
    251269226399   6 .TEST_IO.GET_VECTOR_DIM.1s [ROOT_TASK, #B54E8]
     +    127      6     ....1s
    251276928611   5 .TEST_HARNESS.TEST_MATRIX_VECTOR_MULT.6s
     +    336      5     ....7s
     +    996      5     ....8s
     +   1265      6 .TEST_IO.GET_VECTOR.1s
     +   1383      6     ....1s
     +   1771      6     ....2s
    251370692649   5 .TEST_HARNESS.TEST_MATRIX_VECTOR_MULT.8s
```

TD20 **How integrated is the translator and editor with the debugger?**

The three tools are basically standalone tools. The editor can be used with the debugger running in another window, but if program code is modified the

debugger will still run the old versions of the source code. The old program must be terminated and a new one invoked in order to run the updated version.

The debugger can no longer report any information about a program unit that has been edited while it is being debugged. This is due to the fact that source code and object code are contained in the same Ada object in the Rational Environment. Editing an object apparently destroys some linkages that allow the debugger to show the source corresponding to the object code. This seems to be a reasonable limitation.

The debugger is completely integrated with the Rational browsing capability. Selecting a variable and requesting a definition by pressing the <Definition> key causes a window containing the source code for the definition of the variable to open.

TD21 **What are the space utilization ramifications of instrumenting code for debugging?**

None.

TD22 **Do the analysis tools present a graphical representation of the program?**

No analysis tools are available.

TD23 **What are the space utilization ramifications of instrumenting code for analysis?**

See TD22.

TD24 **Describe the information available from static analysis.**

No supported static analysis tools are available.

TD25 **Describe the mechanics of performing static analysis.**

Not applicable, see TD24.

TD26 **How easy/difficult is it to perform static analysis?:**

Not applicable, see TD24.

TD27 **What are the CPU and clock times for performing a static analysis of the test_harness and modules to be tested?**

Not applicable, see TD24.

TD28 **Describe those aspects of performing tests which are automated.**

There are no tests available which are automated.

TD29 **Describe the mechanics of using the environment to create a test plan and test data.**

Test data must be created manually with the editor.

TD30 **Describe those aspects of test data generation which are automated.**

Test data generation is not automated.

TD31 **How easy/difficult is it to use the environment to create a test plan and test data?**

See TD30.

TD32 **Describe the differences (if any) between invoking a module and invoking the module for unit testing.**

There are no available tools for unit testing.

TD33 **Describe the mechanics of performing initial unit testing.**

See TD32.

TD34 **How easy/difficult is it to perform initial unit testing?**

See TD32.

TD35 **Describe the information available from dynamic analysis.**

There are no available tools for dynamic analysis.

TD36 **Describe the mechanics of performing dynamic analysis.**

See TD35.

TD37 **How easy/difficult is it to perform dynamic analysis?**

See TD35.

TD38 **What are the CPU and clock times for performing a dynamic analysis of the test_harness and modules to be tested?**

See TD35.

TD39 **Describe the mechanics of performing regression testing.**

There are no available tools for regression testing.

TD40 **How easy/difficult is it to perform regression testing?**

See TD39.

TD41 **Describe the mechanics of setting and resetting tracepoints.**

The tracing facility of the debugger causes a message to be displayed each time a certain kind of event occurs in a task for which tracing is enabled. Traces can generate a message for a statement, call, rendezvous, or exception. They do not provide values of variables. Tracing is enabled or disabled by executing Debug.Trace from a command window. The parameters are On (enable/disable toggle), Event (what is to be traced), In_Task (in which tasks is tracing enabled), At_Location (which specifies a scope within which tracing is enabled), and Stack_Frame (which specifies the frame or subprogram to perform tracing.)

TD42 **How easy/difficult is it to set and reset tracepoints?**

Very easy.

TD43 **How accessible are environment tools from within the debugger?**

All tools available outside the debugger are also available while using the debugger.

TD44 **Assess the quality of written documentation. Pay particular attention to support for the debugger and testing related tools.**

The documentation for the Rational debugger includes an introductory overview of the debugger functions and naming conventions, a section giving detailed information about each debugging procedure and the types of special Debug procedure parameters, and an index to debugging topics that covers technical terms and procedure and type names. This documentation should be adequate for any user to teach himself how to use the debugger. The *Basic Operations Manual* provides four pages of step-by-step instructions of common debugger operations designed to help the novice user get started.

TD45 **Describe any noted inconsistencies exhibited by the Environment.**

There are no noteworthy inconsistencies exhibited by the environment.

TD46 **Describe the helpfulness of the online assistance, especially as it relates to unit testing and debugging.**

The online assistance provides descriptions of all debugging procedures. It does not include the introductory material or description of procedure parameter types provided in the written documentation. In providing information about a debugging procedure, the online assistance is as helpful and easier to use than the written documentation since it is basically an online version of the written documentation with searches on name fragments of the debugging procedures added. In addition, the online help immediately provides a description of any procedure bound to a key whenever the user presses the <Help on key>, followed by the key of interest.

TD47 **Do the analysis tools use an underlying database to store and/or retrieve program related information?**

No supported analysis tools are available.

TD48 **How accessible are the underlying OS tools from within the debugger?**

There is no underlying operating system for the Rational Environment. For the availability of Rational Environment tools while debugging, see TD43.

TD49 **How accessible is online assistance from within the debugger?**

Very accessible. It is the same as from outside the debugger.

TD50 **How informative are the debugger error messages?**

In general, they are very informative.

TD51 **How informative are the test manager error messages?**

No test manager tools are available.

TD52 **How tolerant of simple errors is the test manager?**

No test manager tools are available.

TD53 **How tolerant of simple errors is the debugger?**

The debugger describes the problem in a message and highlights the line in which the error occurred. A more detailed explanation of the error is usually provided if the <Explain> key is pressed.

TD54 **Qualitatively describe the response times for interacting with the** debugger.

Responses are instantaneous for most operations. There does seem to be a startup time associated with the debugger of several seconds.

TD55 **Qualitatively summarize the learning curve as it applies to using the environment for unit testing and debugging.**

The debugger is a powerful tool. Some learning is required to become acquainted with all its capabilities. The basic operations of stepping through a program, setting breakpoints, and displaying object values are very easy to learn and to use.

5.5. Unit Testing and Debugging Analysis

5.5.1. Functionality

There are no tools for test harness generation, regression testing, or test management.

There are no tools for performance analysis. However, the R1000 is intended as a universal host development system. Since code generated for the R1000 will never run on target machines, a performance analyzer is more important for the target environments than for the R1000. Performance patterns on the R1000 should be expected to differ from performance patterns on targets since the R1000 has hardware optimizations for implementing certain Ada operations that will very likely not exist for target machines. The R1000 has target build tools that can be used to develop code that will be recompiled on target machines that support an APSE. The recompilation is driven by a script generated by the R1000. If the target machine APSE includes a performance coverage analyzer, then that tool will be available to examine programs developed on the R1000.

The browsing capability in combination with the debugger provides a very powerful tool. Stepping through code while the next line to be executed is highlighted in a window makes source-level debugging easy, as does the ability to display variable definitions and uses interactively.

The Rational Environment debugger also has a full complement of commands for setting, resetting, and displaying breakpoints and tracepoints. Rational breakpoints can be defined so that they will break only when the code containing the breakpoint is called by a specific task or after a certain number of executions. Rational tracepoints do not display values; they simply print a message indicating that some event has taken place in a task. Commands are provided to determine whether the debugger breaks on exceptions or propagates them. This behavior can be localized by task and by code location. For example, the debugger could be set to propagate all exceptions except ones raised in a specific procedure when called by a specific task. Commands are also provided to stop and start the execution of tasks and to display task states. There are commands for stepping the execution of programs by statements or by events (such as making or returning from procedure calls), to display program variable values, and to modify program variable values. The debugger also provides a facility for defining how the debugger will display user-created types such as linked lists. This functionality was not tested by the experiment.

5.5.2. Performance

No recompilation is required to allow a program to be run under control of the debugger. There does appear to be a startup time associated with the debugger, but it is minimal compared with the time that would be required by a recompilation. The debugging and browsing facilities are highly interactive.

The Delta 0 Release of the Rational Environment did have a problem in the debugger. The experiment requires the setting of a breakpoint in an overloaded function. The function happens to define an infix operator. The debugger was unable to resolve the reference. The Rational Customer Response Center indicated that they were aware of the bug, that it would fixed in the

next release, and suggested as a workaround that the function be redefined as a normal function call. The workaround was easy to implement using the browsing and incremental editing facilities. It also corrected the problem of overload resolution.

5.5.3. User Interface

Although no test management tools are provided in the Rational Environment, they could be easily constructed given the nature of the user interface. Procedures can be written in Ada, making use of the programmatic Ada interface to such packages as **Program.Run_Job** and **File_Utilities.Difference** to manage different kinds of testing.

The debugger always operates in a screen-oriented mode with one window devoted to showing the results of debugger commands and another window devoted to showing the source code being debugged. The source code window highlights the line about to be executed when stepping or when a breakpoint is hit. Most common debugger commands are bound to function keys and will, in many cases, operate on objects selected in the source code window. Thus, displaying an object's value can be as simple as selecting the object in the source code window and pressing the Debug Put key. The ability to browse code is completely integrated with the debugger, and the browsing interface is consistent with the debugger interface. In general, the debugger interface is unusually easy to learn and use.

The Rational debugger interface is consistent with the overall Rational Environment interface. Thus, a user familiar with the rest of the Rational Environment will already know the debugger and browsing interface and needs only to learn the debugger commands.

5.5.4. System Interface

Because the Rational Environment is the operating system for the Rational R1000 series computers, there is no interface to tools of an underlying operating system. The debugger and browsing facility are well integrated into the Environment and actually sold as part of the Rational Environment. The debugger and the browsing facility are also completely integrated.

6. Prototype ACEC

6.1. Introduction

The final experiment of the Environment Evaluation Methodology is the compilation and execution of the Ada Compiler Evaluation Capability (ACEC) test suite that was assembled by the Institute for Defense Analysis (IDA). The version of the ACEC that was publicly available and used for this evaluation is described in the *User's Manual for the Prototype Ada Compiler Evaluation Capability*, Version 1, by Audrey A. Hook, Gregory A. Riccardi, Michael Vilot and Stephen Welke, Institute for Defense Analysis, October 1985.

The SEI experience with the ACEC is described in Chapter Eight of *Evaluation of Ada Environments*. Since the prototype ACEC did not provide any capability for analyzing the ACEC results, the SEI developed an analysis program that generates statistics from the ACEC results files. The SEI discovered that the present design of the ACEC does not generate measurements of sufficient reliability for the ACEC suite to be used for its primary purpose, the evaluation of individual language features. Lack of reliability was indicated by two problems: lack of repeatability and negative deltas. Repeatability was directly tested using DEC's VAX Ada compiler by running a subset of the ACEC three times. Variations in CPU usage of up to 4% were found for compilation and of up to 50% for run time. The repeatability problems led the SEI to the following conclusion:

> This jitter, if representative of other environments, does not invalidate the observed ratios of overall compiler performance, but it does invalidate any fine-grained measurements, particularly any differential statistics.

Negative deltas occurred when a test using a language feature took less time than a control test that did not use the language feature. Clearly, negative deltas are a byproduct of the non-repeatability of a measurement that was measured directly with VAX Ada. Negative deltas appeared in the results of VAX Ada running under VMS, Verdix's VADS compiler running under ULTRIX (DEC Unix), and with the Rational R1000. The SEI conclusion about using the ACEC to measure individual language feature performance is that "... the differential statistics produced by the analysis program must be viewed with extreme skepticism" and that " ... no conclusions can be drawn."

In accordance with the SEI finding, the report on the results of implementing the ACEC on the Rational R1000 does not discuss individual language features. The aggregate measures for all tests and the aggregate measures for each of the major test categories (normative performance, normative capacity, optional performance, and optional special algorithms) are reported for the compilation of the ACEC from source state to coded state. According to the *IDA User's Manual for the Prototype ACEC*, Version 1, the optional special algorithms tests "are combinations of language constructs that are characteristic of synthetic benchmark programs." As such, the aggregate measures for just this category are reported for compilation from ASCII text file format to a loaded main program.

6.2. Implementing the Prototype ACEC

The ACEC was implemented on the Rational R1000 using the Delta 0 Release of the Rational system software. Since the command language of the Rational R1000 is Ada, and since Ada procedure calls were available to obtain the required statistics (such as CPU time) used by a job, generation of the support software required to drive the ACEC suite was easy. (See Appendix C for listings of the support software.) No problems were encountered using the Rational Environment to compile the components of the support software supplied with the ACEC. The Environment was dedicated to running the ACEC suite for each run.

6.2.1. Implementation Choices

The ACEC test suite collects compilation and execution timings based on a traditional compile, link, load, and execute cycle. The Rational Environment departs from the traditional cycle, and it is difficult to arrive at a timing method that provides a fair comparison to other APSEs.

There are three areas where implementation choices were made so that the collected data would be more comparable to previously studied APSEs. First, the ACEC tests could be compiled from a text file to a coded state or from an Ada object in source state to a coded state. Second, linking could occur when the tests were run or when they were compiled. Finally, compilation time could reflect the cost of loading a main program or execution time could reflect the cost of loading.

Two sets of results are presented. The first, Section 6.3.1 shows timings collected from the compilation and execution of all four categories of the ACEC test suite. Compilation time in this table was measured as the time to promote an Ada object in source state to coded state and to link the object. Compiling from Ada objects in source state eliminates the parsing which traditional compilers perform when starting with text files. Because compilation on the Rational is typically performed on Ada objects in source state, these results are significant when compared with the other APSE results.

The second set of results (see Section 6.3.3) shows timings collected from the compilation and execution of only the architecture category "optional special algorithms." Compilation time was collected as the time to compile from text to coded state plus the time to produce a loaded main program. The optional special algorithms were chosen because they represent more standard benchmark-type code, rather than code that is written to test language features. Note that this compilation time also incurs the cost of "pretty printing" the source code. More traditional environments provide formatting as a function separate from compilation.

For both sets of results, the pragma Main was added to the source in order to force compile time linkage. This is not standard or recommended practice when developing or maintaining Ada units in the Rational Environment, but was done here to make results comparable to the results from other APSEs. Link and load usually occur when execution of a program is requested.

One incompatibility between the architecture of the ACEC suite and the Rational treatment of program libraries had to be resolved to enable the use of Ada objects. Ten of the text files in the suite contained non-nested compilation units of the following form:

```
package Small_Unit is
      .
      .
      .
end Small_Unit;

with Small_Unit;
procedure An_ACEC_Main_Procedure is
      .
      .
      .
end An_ACEC_Main_Procedure;
```

These were the tests BSRCA2, BSRCA3, CENTB2, NULLA1, NULLA2, PKGEA1, PKGEA2, PKGSA1, PKGSA2 and SHARA2. The Rational library system stores non-nested compilation units in separate Ada objects. Thus, the packages contained in the ACEC text files are split into multiple Ada objects. The objects split out of the text files must be compiled into the program library before the test harness can compile the ACEC main procedures. To include the compilation times of the packages split out of the ACEC text files, a separate record was kept of their compilation times. The times in the compilation log generated by the main ACEC test harness were then adjusted by hand for the ten test programs that contained non-nested compilation units.

6.2.2. Problems Found in the ACEC Suite

When the ACEC suite was first executed, the Rational compiler detected four erroneous ACEC programs (LOSCA1, LOSCA2, SRTEA1 and SRTEA2) at runtime. These erroneous programs have not been detected by any other Ada runtime system. Erroneous programs use the language incorrectly but need not be caught by either an Ada compiler or by Ada runtime. Those incorrect uses that are treated as erroneous are specified in the *Ada Language Reference Manual*, (ANSI/MIL-STD-1815A-1983). In each case where the Rational Environment detected an erroneous program, an uninitiated variable was passed to a procedure. Passing a variable as an *in* or *in out* parameter causes the variable to be evaluated, and evaluation of an uninitialized variable is defined as erroneous. Once the erroneous programs were detected, they were corrected by initializing variables in their declaration. Rational indicates this class of erroneous program by raising numeric errors, regardless of the type of the uninitialized variable.

Another problem was found with the programs BSRCA2 and BSRCA3. Both assume that an implementation has no predefined integer types with a range greater than the predefined type Integer. This is not the case with the Rational, where the largest integer type is Long_Integer. The problem arose when assigning System.Min_Int to a variable of type Integer. The *Ada Language Reference Manual* (see [2]) defines System.Min_Int to be the smallest value of all predefined integer types in an implementation. Assigning System.Min_Int, which on the Rational had the value Long_Integer'First, to a variable of type Integer caused a constraint error. The problem was corrected by changing the type of the offending variable.

Two capacity tests, BLEMA2 and RCDSA2, generated no instrumentation measurements. BLEMA2 contains sixty-five nested blocks. Compilation determined that the program was semantically and syntactically correct, but object code was not generated since this exceeded the com-

piler capacity of fifteen static nestings. The problem was reported by the compiler with the following message:

```
BLEMA2 could not be promoted to coded;
it was promoted to installed.  Static
nesting level exceeds 15 (note:
inserting a package into your sequence
of nested blocks/subprograms will
fix this).
```

RCDSA2 was a capacity test that used a record with 400 fields. This exceeded the documented Rational limit of 256 fields, and generated the following runtime error message:

```
Instruction_Error(type mismatch).
```

In both these cases, the ACEC performed its job, which was to detect capacity limits in the compiler being tested.

A problem previously detected in the Design and Development Experiment also affects the ACEC results; a procedure that compiles a program cannot also measure the size of the resulting Ada object. An attempt to do so generates a random number. Thus, the object code size field is not available in the test results. Since object code generated by the Rational is stored in Ada objects that contain far more information than an object code file, the size of Ada objects is not an indication of the size of Rational's object code. There is no way to measure object code size directly. Section 6.3.4 is provided as a means to compare disk utilization for executable images with other APSEs.

Once the changes to individual tests described in Section 6.2.1 were made, the Rational Environment had no trouble compiling and executing the prototype ACEC suite.

6.3. Numeric Results

6.3.1. Aggregate Measurements for All Tests

MEAN VALUE	Rational	VMS/VAXSet	UNIX/VADS
Compilation Quantity			
Elapsed Time	30.0 (1.0)	52.6 (1.8)	61.8 (2.1)
CPU Time	24.9 (1.0)	15.2 (0.6)	44.8 (1.8)
Instrumentation Quantity			
Elapsed Time	4.4 (1.0)	16.2 (3.7)	23.6 (5.4)
CPU Time	4.2 (1.0)	16.2 (3.9)	0.2 (0.0)
Run Time Quantity			
Elapsed Time	15.8 (1.0)	28.8 (1.8)	36.7 (2.3)
CPU Time	5.1 (1.0)	17.0 (3.3)	23.3 (4.6)

MINIMUM VALUE	Rational	VMS/VAXSet	UNIX/VADS
Compilation Quantity			
Elapsed Time	6.7 (1.0)	39.8 (5.9)	39.2 (5.9)
CPU Time	4.8 (1.0)	6.3 (1.3)	26.3 (5.5)
Instrumentation Quantity			
Elapsed Time	0.0 (-)	0.0 (-)	0.2 (-)
CPU Time	0.0 (-)	0.0 (-)	0.0 (-)
Run Time Quantity			
Elapsed Time	11.2 (1.0)	12.2 (1.1)	12.8 (1.1)
CPU Time	1.0 (1.0)	0.6 (0.6)	0.5 (0.5)

MAXIMUM VALUE	Rational	VMS/VAXSet	UNIX/VADS
Compilation Quantity			
Elapsed Time	1760.1 (1.0)	297.2 (0.2)	606.1 (0.3)
CPU Time	1684.6 (1.0)	121.3 (0.1)	590.2 (0.4)
Instrumentation Quantity			
Elapsed Time	204.7 (1.0)	402.4 (2.0)	460.5 (2.2)
CPU Time	201.6 (1.0)	402.0 (2.0)	4.6 (0.0)
Run Time Quantity			
Elapsed Time	216.1 (1.0)	415.2 (1.9)	473.8 (2.0)
CPU Time	202.6 (1.0)	402.8 (2.0)	459.9 (2.3)

Table 6-1: Aggregated Measurements for All Tests

6.3.2. Aggregated Measurements for Each Architecture Category

COMPILATION-TIME: ELAPSED TIME (SECONDS) ---- TOTALS

ARCH. CATEGORY	# TESTS	RTNL[1]	DEC[2]	VADS[3]
MEAN_VALUE				
ALL_CATEGORIES	173	30.0 (1.0)	52.6 (1.8)	61.8 (2.1)
NORMATIVE_PERFORMANCE	131	15.6 (1.0)	49.8 (3.2)	58.1 (3.7)
NORMATIVE_CAPACITY	11	213.9 (1.0)	69.2 (0.3)	120.6 (0.6)
OPTIONAL_FEATURES	3	67.9 (1.0)	139.0 (2.0)	91.0 (1.3)
OPTIONAL_ALGORITHMS	28	21.0 (1.0)	50.3 (2.4)	53.0 (2.5)
MINIMUM_VALUE				
ALL_CATEGORIES	173	6.7 (1.0)	39.8 (5.9)	39.2 (5.9)
NORMATIVE_PERF.	131	6.7 (1.0)	39.8 (5.9)	39.3 (5.8)
NORMATIVE_CAPACITY	11	7.7 (1.0)	40.4 (5.2)	42.2 (5.5)
OPTIONAL_FEATURES	3	15.2 (1.0)	45.8 (3.0)	47.3 (3.1)
OPTIONAL_ALGS.	28	9.2 (1.0)	39.9 (4.3)	39.2 (4.3)
MAXIMUM_VALUE				
ALL_CATEGORIES	173	1760.1 (1.0)	297.2 (0.2)	606.1 (0.3)
NORMATIVE_PERF.	131	95.1 (1.0)	88.5 (0.9)	135.8 (1.4)
NORMATIVE_CAPACITY	11	1760.1 (1.0)	158.2 (0.1)	606.1 (0.3)
OPTIONAL_FEATURES	3	132.1 (1.0)	297.2 (2.2)	151.0 (1.1)
OPTIONAL_ALGORITHMS	28	65.0 (1.0)	99.7 (1.5)	81.1 (1.2)

COMPILATION-TIME: TOTAL CPU TIME (SECONDS) ---- TOTALS

ARCH. CATEGORY	# TESTS	RTNL	DEC	VADS
MEAN_VALUE				
ALL_CATEGORIES	173	15.1 (1.0)	15.2 (1.0)	44.8 (3.0)
NORMATIVE_PERFORMANCE	131	11.7 (1.0)	13.0 (1.1)	40.6 (3.5)
NORMATIVE_CAPACITY	11	201.0 (1.0)	33.7 (0.2)	106.2 (0.5)
OPTIONAL_FEATURES	3	39.3 (1.0)	46.6 (1.2)	63.2 (1.6)
OPTIONAL_ALGORITHMS	28	15.8 (1.0)	14.6 (0.9)	38.4 (2.4)
MINIMUM_VALUE				
ALL_CATEGORIES	173	4.8 (1.0)	6.3 (1.3)	26.3 (5.5)
NORMATIVE_PERFORMANCE	131	4.8 (1.0)	7.3 (1.5)	27.4 (5.7)
NORMATIVE_CAPACITY	11	4.9 (1.0)	8.3 (1.7)	30.3 (6.2)
OPTIONAL_FEATURES	3	9.3 (1.0)	8.4 (0.9)	28.0 (3.0)
OPTIONAL_ALGORITHMS	28	6.2 (1.0)	6.3 (1.0)	26.3 (4.2)
MAXIMUM_VALUE				
ALL_CATEGORIES	173	1684.6 (1.0)	121.3 (0.1)	590.2 (0.4)
NORMATIVE_PERFORMANCE	131	86.4 (1.0)	55.6 (0.6)	99.9 (1.2)
NORMATIVE_CAPACITY	11	1684.6 (1.0)	121.3 (0.1)	590.2 (0.4)
OPTIONAL_FEATURES	3	77.1 (1.0)	106.4 (1.4)	107.4 (1.4)
OPTIONAL_ALGORITHMS	28	45.2 (1.0)	39.3 (0.9)	61.8 (1.4)

Table 6-2: Compilation Results

[1]Rational 1000 Model 200-20, Rational Environment Release, Delta0.

[2]Digital Equipment Corporation Ada Compilation System Version 1.2, VMS, 6 megabytes main memory, 102 megabytes disk space.

[3]Verdix Ada Development System, Version 5.1, ULTRIX Version 1.2, 6 megabytes main memory, 202 megabytes disk space.

INSTRUMENTATION: ELAPSED TIME (SECONDS) ---- TOTALS

ARCH. CATEGORY	# TESTS	RTNL	DEC	VADS
MEAN_VALUE				
ALL_CATEGORIES	171	4.4 (1.0)	16.2 (3.7)	23.6 (5.4)
NORMATIVE_PERFORMANCE	131	3.4 (1.0)	17.9 (5.3)	26.1 (7.7)
NORMATIVE_CAPACITY	9	1.3 (1.0)	1.3 (1.0)	2.4 (1.8)
OPTIONAL_FEATURES	3	1.1 (1.0)	8.4 (7.6)	4.9 (4.5)
OPTIONAL_ALGORITHMS	28	10.3 (1.0)	15.0 (1.5)	21.7 (2.1)
MINIMUM_VALUE				
ALL_CATEGORIES	171	0.0 (-)	0.0 (-)	0.2 (-)
NORMATIVE_PERFORMANCE	131	0.1 (1.0)	0.0 (0.0)	0.3 (3.0)
NORMATIVE_CAPACITY	9	0.0 (-)	0.0 (-)	0.2 (-)
OPTIONAL_FEATURES	3	0.7 (1.0)	3.5 (5.0)	4.0 (5.7)
OPTIONAL_ALGORITHMS	28	0.0 (-)	0.0 (-)	0.2 (-)
MAXIMUM_VALUE				
ALL_CATEGORIES	171	204.7 (1.0)	402.4 (2.0)	460.5 (2.2)
NORMATIVE_PERFORMANCE	131	96.2 (1.0)	402.4 (4.2)	460.5 (4.8)
NORMATIVE_CAPACITY	9	7.2 (1.0)	4.9 (0.7)	11.8 (1.6)
OPTIONAL_FEATURES	3	1.4 (1.0)	11.3 (8.1)	5.4 (3.9)
OPTIONAL_ALGORITHMS	28	204.7 (1.0)	289.9 (1.4)	358.0 (1.7)

INSTRUMENTATION: CPU TIME (SECONDS) ---- TOTALS

ARCH. CATEGORY	# TESTS	RTNL	DEC	VADS
MEAN_VALUE				
ALL_CATEGORIES	171	4.2 (1.0)	16.2 (3.9)	0.2 (0.0)
NORMATIVE_PERFORMANCE	131	3.3 (1.0)	17.8 (5.4)	0.2 (0.1)
NORMATIVE_CAPACITY	9	1.0 (1.0)	1.3 (1.3)	0.0 (0.0)
OPTIONAL_FEATURES	3	1.1 (1.0)	8.4 (7.7)	0.4 (0.3)
OPTIONAL_ALGORITHMS	28	9.7 (1.0)	14.7 (1.5)	0.4 (0.0)
MINIMUM_VALUE				
ALL_CATEGORIES	171	0.0 (-)	0.0 (-)	0.0 (-)
NORMATIVE_PERFORMANCE	131	0.1 (1.0)	0.0 (0.0)	0.0 (0.0)
NORMATIVE_CAPACITY	9	0.0 (-)	0.0 (-)	0.0 (-)
OPTIONAL_FEATURES	3	0.7 (1.0)	3.5 (5.1)	0.0 (0.0)
OPTIONAL_ALGORITHMS	28	0.0 (-)	0.0 (-)	0.0 (-)
MAXIMUM_VALUE				
ALL_CATEGORIES	171	201.6 (1.0)	402.0 (2.0)	4.6 (0.0)
NORMATIVE_PERFORMANCE	131	94.7 (1.0)	402.0 (4.2)	4.6 (0.0)
NORMATIVE_CAPACITY	9	4.5 (1.0	4.9) (1.0)	0.0 (0.0)
OPTIONAL_FEATURES	3	1.3 (1.0)	11.3 (8.7)	1.0 (0.8)
OPTIONAL_ALGORITHMS	28	201.6 (1.0)	289.5 (1.4)	4.5 (0.0)

Table 6-3: Instrumentation Results

RUN-TIME: ELAPSED TIME (SECONDS) ---- TOTALS

ARCH. CATEGORY	# TESTS	RTNL	DEC	VADS
MEAN_VALUE				
ALL_CATEGORIES	171	15.8 (1.0)	28.8 (1.8)	36.7 (2.3)
NORMATIVE_PERFORMANCE	131	14.8 (1.0)	30.4 (2.0)	39.1 (2.6)
NORMATIVE_CAPACITY	9	17.7 (1.0)	13.7 (0.8)	15.5 (0.9)
OPTIONAL_FEATURES	3	12.8 (1.0)	20.8 (1.6)	18.2 (1.4)
OPTIONAL_ALGORITHMS	28	21.7 (1.0)	27.6 (1.3)	34.8 (1.6)
MINIMUM_VALUE				
ALL_CATEGORIES	171	11.2 (1.0)	12.2 (1.1)	12.8 (1.1)
NORMATIVE_PERFORMANCE	131	11.3 (1.0)	12.4 (1.1)	12.8 (1.1)
NORMATIVE_CAPACITY	9	11.3 (1.0)	12.2 (1.1)	13.2 (1.2)
OPTIONAL_FEATURES	3	11.9 (1.0)	15.7 (1.3)	17.3 (1.5)
OPTIONAL_ALGORITHMS	28	11.2 (1.0)	12.3 (1.1)	13.2 (1.2)
MAXIMUM_VALUE				
ALL_CATEGORIES	171	216.1 (1.0)	415.2 (1.9)	473.8 (2.2)
NORMATIVE_PERFORMANCE	131	107.4 (1.0)	415.2 (3.9)	473.8 (4.4)
NORMATIVE_CAPACITY	9	19.3 (1.0)	17.3 (0.9)	25.0 (1.3)
OPTIONAL_FEATURES	3	13.7 (1.0)	23.9 (1.7)	18.8 (1.4)
OPTIONAL_ALGORITHMS	28	216.1 (1.0)	302.3 (1.4)	370.8 (1.7)

RUN-TIME: CPU TIME (SECONDS) ---- TOTALS

ARCH. CATEGORY	# TESTS	RTNL	DEC	VADS
MEAN_VALUE				
ALL_CATEGORIES	171	5.1 (1.0)	17.0 (3.3)	23.3 (4.6)
NORMATIVE_PERFORMANCE	131	4.2 (1.0)	18.6 (4.4)	25.8 (6.1)
NORMATIVE_CAPACITY	9	2.0 (1.0)	1.9 (1.0)	2.8 (1.4)
OPTIONAL_FEATURES	3	2.1 (1.0)	9.5 (4.5)	4.6 (2.2)
OPTIONAL_ALGORITHMS	28	10.7 (1.0)	15.4 (1.4)	21.5 (2.0)
MINIMUM_VALUE				
ALL_CATEGORIES	171	1.0 (1.0)0.6 (0.6)		0.5 (0.5)
NORMATIVE_PERFORMANCE	131	1.0 (1.0)	0.7 (0.7)	0.8 (0.8)
NORMATIVE_CAPACITY	9	1.0 (1.0)	0.6 (0.6)	0.5 (0.5)
OPTIONAL_FEATURES	3	1.7 (1.0)	4.1 (2.4)	2.2 (1.3)
OPTIONAL_ALGORITHMS	28	1.0 (1.0)	0.6 (0.6)	0.6 (0.6)
MAXIMUM_VALUE				
ALL_CATEGORIES	171	202.6 (1.0)	402.8 (2.0)	459.9 (2.3)
NORMATIVE_PERFORMANCE	131	95.7 (1.0)	402.8 (4.2)	459.9 (4.8)
NORMATIVE_CAPACITY	9	5.5 (1.0)	5.5 (1.0)	12.1 (2.2)
OPTIONAL_FEATURES	3	2.4 (1.0)	12.8 (5.3)	5.8 (2.4)
OPTIONAL_ALGORITHMS	28	202.6 (1.0)	290.2 (1.4)	357.6 (1.8)

Table 6-4: Run-Time Results

6.3.3. Measurement on 28 Optional Algorithms from ACEC Suite

	Rat. S->C*	Rat. T->L**	VMS/VAXSet***	UNIX/VADS***	ALS***
MEAN VALUE					
Elapsed Time (sec)	21.0	62.6	50.3	53.0	777.6
CPU Time (sec)	15.8	26.1	14.6	38.4	492.6
MINIMUM VALUE					
Elapsed Time	9.2	28.5	39.9	39.2	677.3
CPU Time	6.2	12.1	6.3	26.3	426.9
MAXIMUM VALUE					
Elapsed Time	65.0	145.0	99.7	81.1	1285.8
CPU TIME	45.2	62.1	39.3	61.8	771.7

Table 6-5: Compilation Time

	Rat. S->C	Rat. T->L	VMS/VAXSet	UNIX/VADS	ALS
MEAN VALUE					
Elapsed Time (sec)	10.3	10.3	15.0	21.7	21.4
CPU Time (sec)	9.7	9.7	14.7	0.4	20.8
MINIMUM VALUE					
Elapsed Time	0.0	0.0	0.0	0.2	0.0
CPU Time	0.0	0.0	0.0	0.0	0.0
MAXIMUM VALUE					
Elapsed Time	204.7	206.8	289.9	358.0	386.6
CPU TIME	201.6	201.5	289.5	4.5	386.0

Table 6-6: Instrumentation Quantity

* Rational Source (parsed) to Coded State.

** ASCII Text to Linked and Loaded via use of pragma Main.

*** Compilation time includes compile and link time.

	Rat. S->C	Rat. T->L	VMS/VAXSet	UNIX/VADS	ALS
MEAN VALUE					
Elapsed Time (sec)	21.7	22.4	27.6	34.8	45.6
CPU Time (sec)	10.7	10.8	15.4	21.5	26.8
MINIMUM VALUE					
Elapsed Time	11.2	11.5	12.3	13.2	23.9
CPU Time	1.0	1.0	0.6	0.6	5.3
MAXIMUM VALUE					
Elapsed Time	216.1	218.3	302.3	370.8	410.3
CPU TIME	202.6	202.5	290.2	357.6	391.6

Table 6-7: Run Time Quantity

akera2	Ackermann function (test)
akera3	Ackermann function (pragma suppress)
bsrca2	BINARY SEARCH PKG AT EXTREME LIMITS OF ITS INDEX TYPE: LOWER
bsrca3	BINARY SEARCH PKG AT EXTREME LIMITS OF ITS INDEX TYPE: UPPER
chssa2	Char. String Search (test)
chssa3	Char. String Search (pragma suppress)
facta2	RECURSIVE FACTORIAL FUNCTION
hsdra2	HEAPSORT BENCHMARK DRIVE USES X0BMHSPK
intqa2	A FULL INTEGER QUEUE USING X0QUE PACKAGE
iseqa2	GENERIC SEQUENCE MANIPULATION PACKAGE, 50 INTEGERS
minia2	MINIMAL PROGRAM WITH 2 STMT , 1 DECLARATION
mtcqa2	EMPTY CHARACTER QUEUE USING X0QUE PACKAGE
mtesa2	EMPTY SET OF ENUMERATION TYPE USING X0SET PACKAGE
mtisa2	EMPTY SET OF INTEGERS USING X0SET PACKAGE
pgqua2	PUT_END AND GOT_END WITH AN ENUMERATED TYPE USING X0QUE PKG
piala2	PI Algorithm (test)
prcoa2	PRODUCER/CONSUMER PROBLEM
puzza2	PUZZLE
puzza3	PUZZLE (PRAGMA SUPPRESS)
randa2	RANDOM NUMBER GENERATOR
shara2	READERS/WRITERS PROBLEM
sieva2	Sieve of Eratosthenes (test)
sorta2	INSERTION SORT USING X0SORT PACKAGE
sq10a2	PUT 10 INTEGERS IN SEQUENCE AND IF EMPTY USING X0SEQ PACKAGE
sqpga2	PUT AND GET 10 INTEGERS IN SEQUENCE USING X0SEQ PACKAGE
vpgsa2	VARIOUS PUTS AND GETS IN SEQUENCE USING X0SEQ PACKAGE
wheta2	WHETSTONE INSTRUCTIONS WITH FLOATS
wheta3	WHETSTONE INSTRUCTIONS WITH FLOATS (PRAGMA SUPPRESS)

Table 6-8: Optional Algorithms Programs

6.3.4. Comparison of Executable Image Size

	Bytes	Words	Lines
LilAda	1134	81	40
MedAda	9260	1101	289
BigAda	47411	4995	1436

Table 6-9: Size of Ada Source Comparison Programs

	Verdix Ada[4] (kbytes)	Alsys 3.2[5] (kbytes)	Rational[6] (kbytes)
LilAda executable	64	120	53
LilAda program library	76	125	N/A
MedAda executable	12	120	94
MedAda program library	129	195	N/A
BigAda executable	81	136	249
BigAda program library	245	191	N/A

Table 6-10: Comparison of Executable and Program Library Sizes

[4]Verdix Ada, VADS 5, MicroVAX II/ULTRIX 1.2

[5]Alsys 3.2, Sun 3/140, OS 3.2

[6]Rational, Model 200-20, Rational Environment Delta0

6.4. Prototype ACEC Analysis

When compiling from Rational source code state, the Rational compiler was 1.8 times faster than VMS/VAXSet and 2.1 times faster than UNIX/VADS in terms of elapsed time on an average across 173 ACEC Suite Ada programs. Compiling from Rational source code state does provide an advantage to the Rational compiler, as parsing has already occurred. However, it is not an unreasonable comparison as source state is easily achieved using the Ada Object Editor. Also, incremental compilation techniques prevent the need to re-parse an Ada program once it has been parsed successfully.

An anomalous figure for compile time is presented for the maximum value for the Normative Capacity Tests. The elapsed time to compile is 1760.1 seconds. The particular ACEC test which required such a great amount of time for compilation was **Centb2**, described in the *User's Manual for the Prototype ACEC* as:

```
CENTB2 CHECKS ENUMERATION TYPES UP TO 2000 ELEMENTS
```

Procedure **Centb2** consists of 35 lines of Ada code which rely on package **compp** for the definition of several enumeration types. The compilation of **Centb2** requires only 49.9 seconds of wall clock time. However, this must be adjusted for the compilation time needed for package **compp**—1710.2 seconds. Package **compp** consists of the definition of four types. The four types each enumerate 500, 1000, 1500, and 2000 elements. **Compp** is clearly designed to stress a compiler's ability to deal with the individual language feature of enumerated types. If the time to compile **Centb2** and **compp** is discarded, the Maximum for elapsed time for the Normative Capacity Tests is 258.0 seconds, which falls between the observed maximum for the Normative Capacity Tests for DEC and VADS. It was due to non-benchmark style programs such as **Centb2** that compilation time from ASCII text files was considered only on the 28 Optional Algorithms.

When compiling from ASCII text files, the Rational compiler was 1.2 times slower than VMS/VAXSet and UNIX/VADS in terms of elapsed time, across 28 benchmark style programs from the ACEC suite. This compilation time includes parsing and "pretty-printing," as well as machine code production. This result is significant to the porting of large Ada systems to be maintained on the Rational Environment. The initial compilation of such a system may take slightly longer than would be expected for the other evaluated APSEs.

The instrumentation results represent the time to execute the body of the Ada programs in the ACEC suite, excluding time required for elaboration. The execution of the programs on the Rational Environment was on the average faster than the execution times seen on DEC and VADS. This speed advantage may be attributed to the architecture of the R1000, which was designed to accommodate Ada.

The runtime results represent the time to elaborate and execute the body of the Ada programs in the ACEC suite. Here again, the Rational Environment was faster than DEC and VADS in terms of elapsed time. The speed advantage was not as great as that shown by the instrumentation results.

When **LilAda**, **MedAda**, and **BigAda** are in coded state on the Rational Environment, they require more space than the same executable images produced by Verdix Ada running on a MicroVAX II with ULTRIX 1.2. However, when program library support is taken into account, the Rational Environment requires less space than Verdix Ada and VADS 5. The Rational Environment coded state programs require slightly less storage space than the Alsys 3.2 compiler-generated executable images. The space required for the coded state programs and Rational Environment directories is much less than the space required by the Alsys 3.2 compiler program library. The Rational Environment maintains program library information in its directory structure. It is not attempting to layer Ada program libraries on top of an external operating system and capitalizes on this advantage.

7. Cross Environment Performance Comparison

The following presents questions from each experiment category, with a summary of answers from previous Ada environment evaluations and the Rational Environment evaluation. Each question is labeled by the question's number and the experiment section in which it appeared.

- Configuration Management and Versions Control - CM
- System Management - SM
- Design and Development - DD

The numeric results of the cross-environment performance comparison show that the Rational Environment is a highly interactive Ada development environment. Most commonly used commands have a quick system response, often an elapsed time of under two seconds. Disk utilization for Ada units and directories is on par with the other environments evaluated.

The first value presented (VMS/ALS) is from an evaluation of the SofTech Ada Language System Version 3.0 developed by the Army and designed to be retargetable and rehostable. The ALS is hosted on DEC's VMS operating system (Version 4.2 of MicroVMS) and was run on a hardware configuration consisting of a MicroVAX II with 9 megabytes of main memory and disk space consisting of three RD53 disk drives (213 megabytes).

The second value presented (VMS/VAXSet) is from an evaluation of the DEC product VAX Ada (Version 1.2) and five additional tools collectively referred to as VAXSet. VAX Ada and VAXSet were run on Version 4.2 of MicroVMS. The hardware consisted of a MicroVAX II with 6 megabytes of main memory and 102 megabytes of disk space (one 31 megabyte RD52 disk drive and one 71 megabyte RD53 drive).

The third value presented (UNIX/VADS) is from an evaluation of the Verdix Ada Development System (VADS, Version 5.1) running on ULTRIX, DEC's version of UNIX (Version 1.2). The tests were run on a MicroVAX II configured with 6 megabytes of memory and 202 megabytes of disk space (one 31 megabyte RD52 drive and one 171 megabyte Fujitsu drive).

The fourth value presented is from the Rational Environment Evaluation and labeled "Rational." The Rational equipment used is described in the introduction to this report.

VMS/ALS	VMS/VAXSet	UNIX/VADS	Rational	
CM1.2 Elapsed time for performing a system build operation.				
1937.65 sec	205.04	169.35	43.50	(source -> coded)
			85.64	(coded -> loaded)
CM1.7 Elapsed time for creating a CM file element.				
13.64 sec	5.61	1.42	2.13	
size: 512 bytes	(ave.)512	(ave.)225	3072 bytes	
CM1.8 Elapsed time for performing baseline operation.				
initial 212.93 sec	90.23	42.95	175.85	(all subsystems)

```
B02      225.41          92.41          45.45          41.15  (only VT)
B03      226.3           95.10          45.10          no release needed
B04      223.55          95.33          46.55          38.13  (only SM)
V1.0     207.15         101.45          47.05         201.51  (all subsystems)
                                                      NOTE: in config-only mode:
                                                       42.86 sec to release all
                                                             subsystems
```

CM1.9 Files since increase caused by baseline inclusion.

```
      no increase       negligible       90 bytes      100,640 to release SM
                                                       483,609 to release all subsystems
                                                       NOTE: in config only mode:
                                                        51,853 to release all subsystems
```

CM1.22 Elapsed time for fetching a CM element.

```
(non-variant)
         14.46 sec         4.46           0.82           1.37 (check out command)
```

CM1.23 Elapsed time for creating a variant of a CM element.

```
aim_Ada  26.45 sec        12.06           4.23           1.00
vt_body  48.28            12.38           4.10           1.12
vm_spec_Ada
         48.81            12.25           4.40           1.10
vm_body_Ada
         48.13            12.23           3.55           1.16
```

CM1.24 Elapsed time for fetching a variant of a CM element.

```
aim_Ada  28.27 sec         4.95           0.98           not applicable
vt_body_Ada                                              (see CM1.23)
         28.47             5.26           0.90            "   "
vm_spec_Ada
         27.79             4.91           0.90            "   "
vm_body_Ada
         28.52             5.05           0.95            "   "
```

CM1.25 Elapsed time for reserving a variant of a CM element.

```
aim_Ada  14.19 sec         5.60           1.56           not applicable
vt_body_Ada                                              (see CM1.26)
         15.75             5.56           2.00            "   "
vm_spec_Ada
         15.66             5.54           1.95            "   "
vm_body_Ada
         14.93             5.53           1.25            "   "
```

CM1.26 Elapsed time for replacing a variant of a CM elment.

```
aim_Ada  19.61 sec         6.54           3.85            0.96 (check in command)
vt_body_Ada
         19.23             6.75           2.65            1.04
vm_spec_Ada
         19.03             6.63           3.85            1.03
vm_body_Ada
         19.38             6.69           2.70            1.05
```

CM1.27 File size increase caused by successive version.

```
        100%           majority:                          size of change
      + or - change    no change        107 bytes         logs and deltas
                       some: 1 block                      stored in 1k bytes
```

```
                                                    chunks in configuration
                                                    database
```

CM1.28 File size increase caused by variant version.

```
         100%           majority:     (ave)                Make_Path, with tokens
       + or - change    no change        111 bytes         severed, for variant
                        some: 1 block                      versions of main:
                                                           sm_tester  77518 bytes
                                                           cli_tester 77519 bytes
                                                           vt_tester  77516 bytes
```

CM1.40 Elapsed time of merging variant versions of a CM element.

```
main.Ada  N/A              12.54 sec       4.25             6.38 sec (average)
vt_body.Ada
          N/A              12.93           3.75             no merge needed

vm_spec.Ada
          N/A              13.35           3.75             no merge needed
Vm_body.Ada
          N/A           3.69 sec         3.03           Not Supported

              12.91           3.70             no merge needed
```

CM1.41 File size increase caused by merge operation.

```
         N/A            majority:      (ave)
                        no change        126 bytes          some merges not needed
                        some: 1 block                       for main.Ada:
                                                            1st merge  2572 byte increase
                                                            2nd merge  5078 byte increase
                                                            3rd merge  6358 byte increase
                                                            ----------------------------
                                                            total     14008 byte for
                                                            merging variant back into
                                                            main
```

CM1.45 Elapsed time of reserving a CM element.

```
(non-variant)        (non-variant)    (non-variant)
        14.21 sec           5.78            0.82            not applicable
```

CM1.46 Elapsed time of replacing a CM element.

```
(non-variant)        (non-variant)    (non-variant)
        19.36 sec           6.94            2.70            not applicable
```

CM2.2 Elapsed time for displaying history information for a CM element.

```
 not directly supported     0.33 sec       3.55             33.03 for all units

                                                        (approx. 2 sec per unit)
```

CM2.4 Elapsed time for rebuilding an earlier baseline system.

```
      3656.89 sec         325.22         188.15            (from config. only status)
                                                           for build  133.81 sec
                                                           for recompile
                                                                      24.10 sec
                                                           --------------------
                                                           TOTAL:     157.91 sec
```

CM2.8 Elapsed time for deleting a CM element.

not performed	5.44 sec	N/A	(making uncontrolled) 1.71 sec

SM2.2 Elapsed time for creating a user account group.

3.52 sec	3.44	5.75	2.49

SM2.3 File size increase caused by creating user account group.

majority: no change some: predefined chunks (eg. 3 blocks)	same as VMS/ALS	10 bytes (size of group name)	1 byte (other costs could not be measured)

SM2.5 Elapsed time for creating a new user account.

3.20 sec	3.05	20.20	6.48

SM2.6 File size increase caused by creating a new user account.

majority: no change some: predefined chunks (eg. 3 blocks)	same as VMS/ALS	56 bytes (size of new user name and full name)	7473 bytes (size of empty home world)

SM2.12 Elapsed time for adding a user account to a user group.

3.22 sec	3.18	included in SM2.5	0.07 sec

SM2.13 File size increase caused by adding a user account to a user group.

majority: no change some: predefined chunks (eg. 3 blocks)	same as VMS/ALS	5 bytes (one byte more than size of user name being added)	2 bytes (other costs could not be measured)

SM2.16 Elapsed time for copying old account characteristics into a new account.

3.79 sec	3.74	Not Supported	time to execute command that creates a user with some default characteristics: 1.77 sec

SM2.17 File size increase caused by copying old account information into new account.

majority: no change some: predefined chunks (eg. 3 blocks)	same as VMS/ALS	N/A	7473 bytes (empty home world)

SM2.20 Elapsed time for disabling logins for a user account.

N/A	N/A	N/A	1.77 sec

SM2.24 Elapsed time for displaying user account characteristics.

2.64 sec	3.11	0.33	(display group) 0.03 sec

SM2.27 Elapsed time for modifying a user account's characteristics.

```
2.84 sec        2.75            Not Supported      0.03 (change password)
```

SM2.34 Elapsed time for removing a user account from a user group.

```
3.69 sec        3.03            Not Supported      0.11 sec
```

SM2.35 File size decrease caused by removing a user account to a user group.

```
no decrease     no decrease     decrease by length  1 byte decrease
                                of user name + 1
                                byte
```

SM2.38 Elapsed time for deleting a user account.

```
3.19 sec         3.03          15.80               delete access: 1.81 sec
                                                   delete home world:
                                                                 1.10 sec
```

SM2.39 File size decrease caused by deleting a new user account.

```
no decrease     no decrease     55 bytes               7556 bytes
                                (size of user name
                                and full name)
```

DD6 What are the CPU and clock times for creating a program library?

```
Elapsed:
      17 min 17 sec        13.00 sec      3.2 sec             1.73 sec
CPU:  12 min 26 sec         2.85 sec      0.1 sec             0.70 sec
```

DD7 What are the space utilization ramifications of creating a program library?

```
      1 block             115 blocks      1.3 blocks          14.5 blocks
   (ie 512 bytes)                       (ie 690 bytes)     (ie 7425 bytes)
```

DD17 What are the CPU and elapsed times for translating a compilation unit into a specified program library?

```
Vector_Management Spec:
Elapsed:                                                  (source to coded)
       1 min 19 sec        11.38 sec      9.0 sec             5.52 sec
CPU:         41 sec         4.11 sec      1.0 sec             2.48 sec
Matrix_Management Spec:
Elapsed:
       1 min 27 sec        11.91 sec      6.9 sec             3.00 sec
CPU:         48 sec         5.94 sec      0.7 sec             1.37 sec
```

DD18 What are the space utilization ramifications of translating a compilation unit into a specified program library?

```
Vector_Management Source:
      1024 bytes           1024 bytes                     library space:  125 bytes
                           (optional)                     object size:  13281 bytes
Vector_Management Object Code:
    81,920 bytes           4608 bytes  6751 bytes

Matrix_Management Source:
      1024 bytes           1024 bytes                     library space:  125 bytes
                           (optional)                     object size:  11365 bytes
Matrix_Management Object Code:
    65,536 bytes           5120 bytes  (not available)
```

DD22 What are the CPU and elapsed times for translating a compilation unit into a specified program library?

```
Vector_Management body:
Elapsed:                                                    (source to coded)
       1 min 49 sec        15.11 sec      12.2 sec         6.13 sec
CPU:   1 min  6 sec         8.63 sec       4.9 sec         3.09 sec
```

DD26 What are the CPU and elapsed times necessary for creating an executable module?

```
Elapsed:
       7 min 45 sec        23.86 sec      25.1 sec           N/A
CPU:   4 min 50 sec         1.53 sec      10.5 sec
```

DD27 What are the space utilization ramifications of creating an executable module?

```
   an additional
      151,040 bytes     30,208 bytes     68,127 bytes          N/A
```

DD32 What are the space utilization ramifications of browsing a compilation unit?

```
      no increase       no increase      browsing not         no increase
                                          supported
```

Appendix A: Size and Time Reporting Procedures

The following procedures were used to obtain the instrumentation data for the Configuration Management/Version Control Experiments and the System Management Experiments. Each routine is followed by a short explanation of its use.

A.1. Specification Record_Size'Spec

Procedure **Record_Size** has three parameters: **Message**, **Object**, and **Recursive**. **Message** can be any string which will be reprinted with the size figures and should be used to annotate the resulting log. **Object** is the object whose disk utilization in being measured and reported. **Recursive** indicates whether subobjects of the Unit should be included in the size measurement. If **Recursive** is True, then the object's space utilization plus sub-objects' space utilization will be reported. If **Recursive** is false, then only the space utilization of that object itself will be reported.

```
procedure Record_Size (Message : String := "";
                       Objects : String := "";
                       Recursive : Boolean := True);
```

A.2. Procedure Record_Size'Body

```
with Directory_Tools, Io;
procedure Record_Size (Message : String := "";
                       Objects : String := "";
                       Recursive : Boolean := True) is

    package Dt renames Directory_Tools;
    package Stat renames Directory_Tools.Statistics;
    Orig_Obj, Obj : Dt.Object.Handle;
    Iter : Dt.Object.Iterator;
    Sum_Object_Size, Sum_Total_Size : Long_Integer := 0;
    Answer2 : String (1 .. 10);
begin
    if Recursive then
        Iter := Dt.Naming.Resolution (Objects & "??");
    else
        Iter := Dt.Naming.Resolution (Objects);

    end if;
    Orig_Obj := Dt.Object.Value (Iter);
    while not Dt.Object.Done (Iter) loop
        Obj := Dt.Object.Value (Iter);
        Sum_Object_Size := Sum_Total_Size + Stat.Object_Size (Obj) / 8;
        Sum_Total_Size := Sum_Total_Size + Stat.Total_Size (Obj) / 8;
        Dt.Object.Next (Iter);
    end loop;
    if Recursive then
        Answer2 := "everything";
    else
        Answer2 := "itself    ";

    end if;

    declare
        Answer1 : constant String := Dt.Naming.Simple_Name (Orig_Obj) & " ";
        Answer3 : constant String :=
           " object_size => " & Long_Integer'Image (Sum_Object_Size) &
              ", total_size => " & Long_Integer'Image (Sum_Total_Size);
    begin

        Io.Put_Line (Message);
        Io.Put_Line (Answer1 & Answer2 & Answer3);
    end;
end Record_Size;
```

A.3. Using Record_Size

Record_Size'Spec and **Record_Size'Body** can be placed in the *experimenter*'s home directory and compiled and executed from the home directory. To execute the procedure:

```
Open a command window
<Create Command>
Enter
Record_Size
<Complt>
Record_Size(Message => "",
            Objects => "";
            Recursive => True);
```

Supply a message, if desired, to annotate the log and provide the full pathname to the object and the object named for the value of **Objects**. If a measurement of only the object and not its sub-objects is desired, change the value of **Recursive** to False.

```
Execute the command
<Promot>
```

The message, followed by the object name and its space utilization requirements in bytes, is reported to the standard output window.

A.4. Specification Timeit'Spec

Procedure **Timeit** requires no parameters.

```
procedure Timeit;
```

A.5. Procedure Timeit'Body

```
with Compilation, Cmvc, Text_Io, Time_Utilities, System_Utilities;
procedure Timeit is

-- variables for timing

    Begin_Time : Duration := 0.0;
    End_Time : Duration := 0.0;

    Begin_Cpu_Time : Duration := 0.0;
    End_Cpu_Time : Duration := 0.0;

-- output for timings
    package Duration_Io is new Text_Io.Fixed_Io (Duration);

--
begin
    Text_Io.Put_Line ("    CLOCK      CPU");

    -- record the clock time since system boot & cpu time since job start
    Begin_Time := System_Utilities.Elapsed;
    Begin_Cpu_Time := System_Utilities.Cpu;
-- PLACE COMMAND TO TIME HERE:
    Cmvc.Release (From_Working_View =>
                  "!users.experimenter.cm_experiment.vt.rev2_working",
                  Release_Name => "<AUTO_GENERATE>",
                  Level => 0,
                  Views_To_Import => "<INHERIT_IMPORTS>",
                  Create_Configuration_Only => False,
                  Compile_The_View => True,
                  Goal => Compilation.Coded,
                  Comments => "",
                  Work_Order => "<DEFAULT>",
                  Volume => 0,
                  Response => "<PROFILE>");
    End_Cpu_Time := System_Utilities.Cpu;
    End_Time := System_Utilities.Elapsed;


    Duration_Io.Put (End_Time - Begin_Time, 4, 2, 0);
    Text_Io.Put ("    ");
    Duration_Io.Put (End_Cpu_Time - Begin_Cpu_Time, 4, 2, 0);
end Timeit;
```

A.6. Using Timeit

Procedure **Timeit** is shown here set up to time the **Cmvc.Release** command. The procedure **Timeit** can be placed in the *experimenter*'s home directory and compiled. In order to time any command or series of commands, the **Timeit** procedure can be modified using the incremental editing possible with the Rational Environment. Following is a transcript to change the **Timeit** procedure after it has been promoted to coded state in the *experimenter*'s home directory.

```
Make Timeit'body the current context.
<Install Unit>
Select the statement appearing below
the comment.
PLACE COMMAND TO TIME HERE:
<Edit>
An edit window will open; replace
the Cmvc.Release command with
the new command to be timed; no
parameters need be typed at this
point.
<Format>
Select the new command.
<Complt>
```

The parameters and their default values will be supplied. Change the default values as needed. If there is an error message "No completion for X," where X is the name of the command inserted, its package must be added to the **with** clause at the beginning of the program. Do this by moving the cursor to the line after the existing **with** clause. Type <Object> I, then in the edit window, type "With PACKAGE," where PACKAGE is the name of the package in which the command is defined. Type <Promot> to return to the edit window containing the command to be completed, and retype <Complt>, supplying any needed parameters.

```
Place the contents of the edit window
back into the body.
<Promot>
Return the entire body to coded
state.
<Promot>
```

The procedure **Timeit** can then be executed from a command window. The results of the timeit command, and any messages from the timed procedure, will appear in the standard output window.

Appendix B: Design and Development Instrumentation Procedures

The library creation, file copy, and Ada object promotion commands have been incorporated into the following procedures that measure them by recording time and disk space utilization. Section B.12 describes how they can be used when executing the experiment instantiation in Chapter 4. In the following program segments, *experimenter* is used in directory names to indicate where a user may insert his own user name.

B.1. Package Kluge_Stuff

The package **Kluge_Stuff** defines some common types and constants. The type **Selection_Methods** allows a user to perform an operation on an object that is either selected by having the cursor in its image or by selecting the object (<Object> Left Arrow) in its enclosing directory. It also sets up a file to serve as a place to store timing information taken at the beginning of an operation to be used with timing information taken at the end of an operation.

```
package Kluge_Stuff is

    type Selection_Methods is (Cursor_In_Image,
                               Highlight_In_Directory);

    The_Name : constant String :=
       "!Users.experimenter.exp_lib.info_file";

    type Info is
        record
            Selection_Method : Kluge_Stuff.Selection_Methods;
            Library_Size_Before_Promotion : Integer;
        end record;

end Kluge_Stuff;
```

B.2. Specification Timed_Code'Spec

The procedure to initiate and time the compilation of an Ada unit requires no parameters as indicated by its one line specification.

```
procedure Timed_Code;
```

B.3. Procedure Timed_Code'Body

Package **Timed_Code** collects the size of the enclosing library and the time required to compile an Ada Object.

```
-- Ada, Ada_Object_Editor, Common and Object_Editor are Rational
-- Environment packages.

with Common, Direct_Io, Kluge_Stuff, Size_Of, Text_Io, Timing_Log,
     Timing, Ada, Ada_Object_Editor, Object_Editor;

procedure Timed_Code is

    Library_Size_Before_Promotion : Integer := 0;
    -- initialized by unit name function

    Selection_Method :
       Kluge_Stuff.Selection_Methods;


    -- The following declarations are required to pass information
    -- generated by this procedure to
    -- Finish_Coding_Instrumentation.  This kluge is necessitated
    -- by a bug in the Rational Environment that prevents timed
    -- code from reading library size or unit size after promoting
    -- a unit.

    package Info_Io is new Direct_Io (Kluge_Stuff.Info);
    The_File : Info_Io.File_Type;
    The_Info : Kluge_Stuff.Info;


    -- The Unit_Name function has the side effect of setting
    -- Selection_Method.
    -- Selection_Method in turn is used to determine how the
    -- library context of the unit being promoted is referenced.
    -- If cursor is in image then the enclosing context special
    -- character '^' is used.  If the unit it highlighted then the
    -- current context string "[]" is used.

    -- The library size function depends on the side effect of
    -- unit_name which must be called before library_size.


    function Unit_Name return String is
        package Aoe renames Ada_Object_Editor;
        package Oe renames Object_Editor;
    begin

        if Aoe.Image_Name = "###Unknown###" then
            Selection_Method :=
               Kluge_Stuff.Highlight_In_Directory;
            return Oe.Get_Name (Oe.Selection);
        else
            Selection_Method := Kluge_Stuff.Cursor_In_Image;
            return Aoe.Image_Name;
        end if;

    end Unit_Name;


    function Library_Size return Natural is
    begin

        case Selection_Method is
            when Kluge_Stuff.Cursor_In_Image =>
                return Size_Of ("^");
            when Kluge_Stuff.Highlight_In_Directory =>
                return Size_Of ("[]");
        end case;

    end Library_Size;


begin

    Timing_Log.Append_Line;
    Timing_Log.Append_Line ("Coding unit:  " & Unit_Name);
```

```
      Library_Size_Before_Promotion := Library_Size;
      Timing_Log.Append_Line
         ("Library size: " & Integer'Image
                               (Library_Size_Before_Promotion) &
          " before promotion.");

      Timing.Reset;
      Ada.Code_Unit;
      Timing_Log.Append_Line (
                              " Clock time:  " & Timing.Wall_Time &
                              "           CPU time:" & Timing.Cpu);
      Timing_Log.Close_Log;



      -- Store data that will be picked up by
      -- finish_coding_instrumentation.

      The_Info.Selection_Method := Selection_Method;
      The_Info.Library_Size_Before_Promotion :=
         Library_Size_Before_Promotion;
      Info_Io.Create
         (File => The_File,
          Mode => Info_Io.Out_File,
          Name => Kluge_Stuff.The_Name,
          Form => "");

      Info_Io.Write
         (The_File, The_Info);
      Info_Io.Close (The_File);

   end Timed_Code;
```

B.4. Specification Finish_Coding_Instrumentation'Spec

The package specification for **Finish_Coding_Instrumentation** shows that the body can take one parameter. When **Both** is set to true, the size of both the package specification and body will be recorded. The default is false, which causes the recording of the size of just the indicated Ada Unit (either the Ada Unit is selected, or the cursor currently resides in the Ada Unit's image.)

```
procedure Finish_Coding_Instrumentation (Both : in Boolean := False);
```

B.5. Procedure Finish_Coding_Instrumentation'Body

An explanation of **Finish_Coding_Instrumentation** can be found in the comments in the code.

```
with Ada, Ada_Object_Editor, Common, Direct_Io, Kluge_Stuff,
     Object_Editor, Size_Of, Text_Io, Timing, Timing_Log;

-- This procedure logs the post promotion size of either a spec, a
-- body, or both.  Both should be set to true only when compiling
-- a main procedure body for which a spec does not exist.  In this
-- case the system automatically generates a spec, which consumes
-- object and library space.  The program spec or body is indicated by
-- either the cursor's being present in an image of the spec or body, or by
-- the program spec or body selected with the cursor beside, in its parent
-- directory listing.  The parent directory listing must be in standard format,
-- that is, with (proc_body) or (proc_spec) indicated.

procedure Finish_Coding_Instrumentation (Both : in Boolean := False) is

    -- The following declarations are required to read information
    -- generated by Timed_Code.  This kluge is necessitated by a
    -- bug in the Rational Environment that prevents library and
    -- object sizes from being read after promotion by a procedure
    -- that promotes an object.

    package Info_Io is new Direct_Io (Kluge_Stuff.Info);

    The_File : Info_Io.File_Type;       --Name of file containing information
    The_Info : Kluge_Stuff.Info;        --Record that holds information

    -- The following are measured by this procedure
    Library_Size_After_Promotion, Unit_Size_After_Promotion :
       Integer := 0;

    -- Functions Unit_Name and Library_Size are cloned from
    -- Timed_Code.
    function Unit_Name return String is
        --Returns the name of the unit that was promoted and is to be measured.

    begin
        -- Procedure Timed_Code determined how the user was indicating the
        -- code to be promoted, and recorded it.
        case The_Info.Selection_Method is

            -- When the code is indicated by the cursor's being in a window
            -- containing the file's image, then the file's name can be
            -- determined by a call to Image_Name.
            when Kluge_Stuff.Cursor_In_Image =>
                return Ada_Object_Editor.Image_Name;

                -- When the code is indicated by being highlighted in the
                -- directory listing, then the file's name can be determined
                -- by a call to Get_Name.
            when Kluge_Stuff.Highlight_In_Directory =>
                return Object_Editor.Get_Name (Object_Editor.Selection);

        end case;
    end Unit_Name;

    function Library_Size return Natural is
        -- Determine the size of the object's parent library.
    begin
        -- Depending on how the user has indicated the object, determine
        -- the size of the parent library.

        case The_Info.Selection_Method is

            when Kluge_Stuff.Cursor_In_Image =>
                return Size_Of ("^");

            when Kluge_Stuff.Highlight_In_Directory =>
                return Size_Of ("[]");

        end case;
    end Library_Size;

    -- Unit size is measured only after promotion because size of
```

```
-- object returned by the Size_Of function appears to be
-- random if the object has never been installed.
function Unit_Size return Natural is
    -- Measures the size in bytes of the selected object or objects if both
    -- a spec and a body were compiled.
begin

    case The_Info.Selection_Method is

        when Kluge_Stuff.Cursor_In_Image =>
            if Both then
                -- When in a body the current context symbol
                -- "[]" is interpreted as referencing the
                -- spec.  The image name returned by
                -- Ada_Object_Editor always has a spec or body.
                return Size_Of ("[]") +
                          Size_Of (Ada_Object_Editor.Image_Name);
            else
                return Size_Of (Ada_Object_Editor.Image_Name);
            end if;

            -- When the object is selected by highlighting it in the
            -- parent directory, the Object Editor only returns a name,
            -- it does not return 'spec or 'body types, which the Size_Of
            -- function needs in order to locate the proper object to
            -- measure.  The text manipulation below relies on the directory
            -- listing to be in standard format, and actually drags all of
            -- the characters off the highlighted lines and searches for
            -- (proc_body) or (proc_spec) in order to get the proper
            -- measurement.
        when Kluge_Stuff.Highlight_In_Directory =>

            Tricky_Text_Manipulation:
                declare
                    Big_String :
                       String (1 .. 120) := (others => ' ');
                    -- Will hold text grabbed off of highlighted line.
                    Start_Of_Type : Natural := 0;   --holds location of
                    -- Open paren. of the paren's. around the object
                    -- type.
                    Body_Selected : Boolean;  --true if object is a
                    -- Package body, false if object is a package spec.

                begin

                    -- Grab highlighted text and stuff in Big_String.
                    Big_String
                       (1 .. Object_Editor.Get_Text
                                 (Object_Editor.Selection)'Last) :=
                       Object_Editor.Get_Text (Object_Editor.Selection);
                    --Locate open paren. in selected text.
                    for I in Big_String'Range loop
                        if Big_String (I) = '(' then
                            Start_Of_Type := I;
                            exit;
                        end if;
                    end loop;


                    -- Find out what's selected.
                    Body_Selected :=
                       Big_String
                          (Start_Of_Type + 1 .. Start_Of_Type + 9)
                        = "Proc_Body";

                    if Both then
                        -- Print out notice that both a body and spec
                        -- are being measured.
                        Timing_Log.Append_Line
                           ("BOTH 'body and 'spec of " & Unit_Name &
                            " sized and compilation timed");
                        return
                           Size_Of (Unit_Name
```

```
                                        & "'body") +
                                  Size_Of (Unit_Name
                                        & "'spec");
                            else
                                -- Only size of selected text is
                                -- desired, be it a body or a spec.
                                if Body_Selected then
                                    -- Print out notice that just a body is
                                    -- being measured.
                                    Timing_Log.Append_Line
                                       ("ONLY 'body of " & Unit_Name &
                                        " sized and compilation timed");
                                    return
                                       Size_Of (Unit_Name &
                                                "'body");
                                else
                                    -- Print out notice that just a spec is
                                    -- being measured.
                                    Timing_Log.Append_Line
                                       ("ONLY 'spec of " & Unit_Name &
                                        " sized and compilation timed");
                                    return
                                       Size_Of (Unit_Name & "'spec");
                                end if;
                            end if;
                        end Tricky_Text_Manipulation;
            end case;
        end Unit_Size;

    begin

        -- Retrieve data stored by Timed_Code (selection method and library
        -- size before promotion of the object.
        Info_Io.Open (The_File, Info_Io.In_File,
                      Kluge_Stuff.The_Name);
        Info_Io.Read (The_File, The_Info);
        Info_Io.Delete (The_File);

        -- Obtain after promotion data on size of the library.
        Library_Size_After_Promotion := Library_Size;

        -- Get size of unit after promotion.
        Unit_Size_After_Promotion := Unit_Size;

        -- Record the information in log file.
        Timing_Log.Append_Line ("Library Size After Promotion: " &
                                Integer'Image (Library_Size_After_Promotion));

        Timing_Log.Append_Line ("Unit Size After Promotion: " &
                                Integer'Image (Unit_Size_After_Promotion));

        Timing_Log.Append_Line
           ("Library space used by coding " &
            Unit_Name &
            " is " & Integer'Image
                        (Library_Size_After_Promotion -
                         The_Info.Library_Size_Before_Promotion));

        -- all done, the log
        Timing_Log.Close_Log;

    end Finish_Coding_Instrumentation;
```

B.6. Specification Timed_Directory'Spec

The package specification for **Timed_Directory** indicates that it takes one parameter, the name of the directory to be created.

```
-- Times the creation of a directory and calculates the space
-- required for the newly created directory and the space used by
-- its parent directory to store information about it.
-- Records the information in file designated by file read by
-- Timing_log.retrieve_current_log_name.
procedure Timed_Directory (Directory_Name : String := "");
```

B.7. Procedure Timed_Directory'Body

The package body **Timed_Directory** times the creation of a directory. The directory is created as an object in the closest enclosing context that is either a world or directory. It is given the name passed in the **Directory_Name** parameter.

```
-- Ada is a Rational Environment package that used to contain procedure
-- Ada.Create_Directory.  Ada.Create_Directory is now found in
-- !Commands.Library.Create_Directory (LM-222) for Operating System Version
-- Delta.

--with Size_Of, Timing_Log, Timing, Ada;
with Size_Of, Timing_Log, Timing, Library;

procedure Timed_Directory (Directory_Name : String := "") is

    -- Times the creation of directory passed as quoted string, also
    -- calculates the space used by the newly created directory; and the
    -- space required by its parent to store information about it; it
    -- records the information in file designated by file read by
    -- Timing_log.retrieve_current_log_name.

    Size_Before_Creation,          -- Size of parent directory before
                                   -- creation of requested directory in bytes.
    Size_After_Creation : Integer; -- Size pf parent directory after creation
                                   -- of requested directory in bytes.

begin

    -- The string "[]" indicates the current context, which is
    -- the library in which the directory is to be created.
    Size_Before_Creation := Size_Of ("[]");

    -- Send blank line to log file.
    Timing_Log.Append_Line;

    -- Record name of directory to be created.
    Timing_Log.Append_Line ("Creating directory:  "
                            & Directory_Name);

    -- Record cpu time and wall clock time before creation of directory
    -- invoked.
    Timing.Reset;

    -- Create the requested directory.
    Library.Create_Directory (Directory_Name);

    -- Calculate and record the CPU and Wall Clock time elapsed in the
    -- creation of the directory in seconds.
    Timing_Log.Append_Line ("Clock time (sec):  " & Timing.Wall_Time &
                            "           CPU time (sec):" & Timing.Cpu);

    -- Determine size of parent directory with its new subdirectory
    -- information.
    Size_After_Creation := Size_Of ("[]");

    -- Determine size of new directory itself and bytes added to parent
    -- directory and record.
    Timing_Log.Append_Line
       ("Directory creation consumed "
         & Integer'Image (
                         (Size_After_Creation - Size_Before_Creation) +
                         Size_Of (Directory_Name)) & " bytes.");

    -- Directory creation and recording finished, close the log file.
    Timing_Log.Close_Log;

end Timed_Directory;
```

B.8. Specification Timed_World'Spec

The package specification for **Timed_World** indicates that it takes one parameter, the name of the world to be created.

```
procedure Timed_World (World_Name : String := "");
```

B.9. Procedure Timed_World'Body

The package body **Timed_World** times the creation of a world. The world is created as an object in the closest enclosing context that is either a world or directory. It is given the name passed in the **World_Name** parameter.

```
-- Ada is a Rational Environment package that used to contain procedure
-- Ada.Create_World.  Ada.Create_World is now found in !Commands.Library.
-- Create_World (LM-227) for Operating System Version Delta.

--with Size_Of, Timing_Log, Timing, Ada;
with Size_Of, Timing_Log, Timing, Library;
with Editor;

procedure Timed_World (World_Name : String := "") is
-- Times the creation of world passed in as quoted string; also
-- calculates the space used by the newly created world and the space
-- required by its parent to store information about it; it
-- records the information in file designated by file read by
-- Timing_Log.Retrieve_Current_Log_Name.

    Size_Before_Creation,       --size of parent directory before creation of
                                --requested world in bytes

    Size_After_Creation : Integer;  --size of parent directory after creation
                                    --of requested world in bytes
begin
    -- The string "[]" indicates the current context, which is
    -- the library in which the world is to be created.
    Size_Before_Creation := Size_Of ("[]");

    -- Send blank line to log file.
    Timing_Log.Append_Line;

    -- Record name of world to be created.
    Timing_Log.Append_Line ("Creating world:  " & World_Name);

    -- Record CPU time and wall clock time before creation of world invoked.
    Timing.Reset;

    -- Create the requested world.
    Library.Create_World (World_Name);

    -- Calculate and record the CPU and wall clock time elapsed in the
    -- creation of the world in seconds.
    Timing_Log.Append_Line (" Clock time:  " & Timing.Wall_Time &
                            "           CPU time:" & Timing.Cpu);

    -- Determine size of parent directory with its new "sub-world" information.
    Size_After_Creation := Size_Of ("[]");

    -- Determine size of new world itself and bytes added to parent
    -- directory and record.
    Timing_Log.Append_Line ("World creation consumed "
                             & Integer'Image ((Size_After_Creation -
                                               Size_Before_Creation) +

                                             Size_Of (World_Name)) & " bytes.");

    -- World creation and recording finished; close the log file.
    Timing_Log.Close_Log;

end Timed_World;
```

B.10. Specification Sized_Copy'Spec

The package specification **Sized_Copy** indicates that the procedure requires one parameter a string value for **Unit_To_Copy** which indicates the Ada unit that is to be copied in order to measure its size.

```
procedure Sized_Copy (Unit_To_Copy : String := "");
```

B.11. Procedure Sized_Copy'Body

Procedure **Sized_Copy** is described in its comments.

```
-- Procedure to copy a specified file, checking the size of the directory before
-- the file is copied and after the file is copied, in order to determine the
-- amount of space the file and its directory-level information consumes.

with Profile, Timing_Log, Size_Of, Library;

procedure Sized_Copy (Unit_To_Copy : String := "") is

    Library_Size_Before_Copy : Natural := 0;    -- Size of directory in bytes
                                                -- before file added.

    Library_Size_After_Copy : Natural := 0;     -- Size of directory in bytes
                                                -- after file added.

begin
    -- Get size of directory.
    Library_Size_Before_Copy := Size_Of ("[]");

    -- Send blank line to log file indicated in file read by
    -- Timing_Log.retrieve_current_log_name.
    Timing_Log.Append_Line;

    -- Record name of file being copied.
    Timing_Log.Append_Line ("Before copying " & Unit_To_Copy);

    -- Record size of directory before file copied to it.
    Timing_Log.Append_Line ("library size is " & Integer'Image
                                                (Library_Size_Before_Copy));
    -- Copy the specified file.
    Library.Copy (From => Unit_To_Copy,
                  To => "[]",
                  Recursive => True,
                  Response => Profile.Get,
                  Copy_Links => True,
                  Options => "");

    -- Below is the copy_into command as it was implemented for Operating
    -- System Version Gamma; the above was implemented for Operating System
    -- Version Delta.
    -- Library.Copy_Into
    --  (Existing => Unit_To_Copy,
    --   New_Context => Library.Current_Image, Before => "",
    --   Recursive => True, Response => Profile.Get,
    --   Copy_Links => True);

    -- Measure directory size with the new file.
    Library_Size_After_Copy := Size_Of ("[]");

    -- Record the sizes and their difference.
    Timing_Log.Append_Line ("After copy library size is "
                            & Integer'Image (Library_Size_After_Copy));

    Timing_Log.Append_Line
       ("A change of " & Integer'Image
                         (Library_Size_After_Copy - Library_Size_Before_Copy)
        & " bytes.");

    -- All done, close log file.
    Timing_Log.Close_Log;

end Sized_Copy;
```

B.12. Binding and Using Instrumentation Code

The code for the instrumented procedures can be placed in the *Experimenter's* home directory or placed in a subdirectory of the experiment library called **recordit**. If the latter is done, then a link from the *Experimenter's* home directory should be set up to the procedures in **recordit** by using the **Links.Add** command.

The instrumented procedures may be bound to the Rational keyboard by inserting the following code in the *Experimenter's* home directory in a file named **Rational_commands**.

```
with Visible_Key_Names;
with Finish_Coding_Instrumentation;
with Timed_Directory;
with Timed_World;
with Sized_Copy;
with Timed_Code;

procedure Rational_Commands is
    use Visible_Key_Names;

    type Intent is (Prompt, Execute, Interrupt);

    Action : Intent;

    Key_1 : Rational_Key_Names;
    Key_2 : Rational_Key_Names;

begin
    case Action is

        when Prompt =>
            case Key_1 is
                when S_F5 =>
                    Finish_Coding_Instrumentation (Both => False);
                when M_F5 =>
                    Timed_Directory (Directory_Name => "");
                when Cs_F5 =>
                    Timed_World (World_Name => "");
                when F5 =>
                    Sized_Copy (Unit_To_Copy => "");
                when others =>
                    null;
            end case;

        when Execute =>
            case Key_1 is
                when C_F5 =>
                    Timed_Code;
                when others =>
                    null;
            end case;

        when Interrupt =>
            case Key_2 is
                when others =>
                    null;
            end case;

    end case;

end Rational_Commands;
```

The procedure **Rational_Commands** should be promoted to coded state in the *Experimenter's* home directory. Logging out and logging in will bind the instrumentation commands to the following keys:

Procedure	**Key Binding**

```
Timed_Code                          <Control> <F5>
Finish_Coding_Instrumentation       <Shift> <F5>
Timed_Directory                     <Meta> <F5>
Timed_World                         <Control> <Shift> <F5>
Sized_Copy                          <F5>
```

As long as Rational_Commands is available at login time, the instrumentation procedures will be bound to the listed key.

In order to time the promotion of an Ada Unit, either place the cursor in a window containing the image of the unit, or select the unit in a directory listing and type:

```
<Control> <F5>
```

followed by

```
<Shift> <F5>
```

This will cause the time required for compilation and the size of the coded object to be recorded.

To time the creation of a directory, make the world or directory that is to contain the new directory the current context. Type:

```
<Meta> <F5>
```

A command window will open, prompting for Directory_Name; supply the desired name as the value for the parameter and type:

```
<Promot>
```

This causes the creation of a directory with that name and records the time required for the operation.

To time the creation of a world, make the world or directory that is to contain the new world the current context. Type:

```
<Control> <Shift> <F5>
```

A command window will open, prompting for World_Name; supply the desired name as the value for the parameter and type:

```
<Promot>
```

This causes the creation of a world with that name and records the time required for the operation.

To copy an object and determine its size, type:

```
<F5>
```

A command window will open, prompting for Unit_To_Copy; supply the name of the unit to be copied, and type:

```
<Promot>
```

This will copy the named object to the current context and log the amount of disk space used by the object in the current context.

Appendix C: ACEC Suite Timing Harnesses

The following routines must be compiled successfully to run the ACEC Test Suite.

C.1. Package Specification Cpu_Time'Spec

```
package Cpu_Time is

    function Cpu_Clock return Duration;

end Cpu_Time;
```

C.2. Package Cpu_Time'Body

```
with Calendar, System_Utilities;

-- System Utilities is a standard Rational utility package.

package body Cpu_Time is

    function Cpu_Clock return Duration is
    begin
        return System_Utilities.Cpu;
    end Cpu_Clock;

end Cpu_Time;
```

C.3. Specification Harness_Many'Spec

```
procedure Harness_Many;
```

C.4. Procedure Harness_Many'Body

```
with Text_Io, Calendar, System_Utilities, Time_Utilities, Compilation,
     Profile, Log, Library, File_Utilities, Program, Io_Package;

procedure Harness_Many is

    -- Files for results generated by ACEC run
    Comp_Data_File : Text_Io.File_Type;
    Instr_Data_File : Text_Io.File_Type;
    Run_Data_File : Text_Io.File_Type;
    Total_Time_File : Text_Io.File_Type;

    -- The following are used for reading names of ACEC program
    Source_File : Text_Io.File_Type;
    Test_Name : String (1 .. 7);
    Last : Natural;

    -- Various directory-dependent hardwired names

    -- Source World contains the Ada objects comprising
    -- the ACEC benchmark tests.
    Source_World : constant String := "!users.experimenter.acec2";

    -- List_of_ACEC_Programs is a text file containing the names of
    -- the Ada objects that comprise the ACEC benchmarks.
    List_Of_Acec_Programs : constant String := Source_World & ".acec_list";

    procedure Harness (Test_File : in Io_Package.Name_Type) is separate;

begin
    -- Create the logging files
    Text_Io.Create (File => Comp_Data_File,
                    Mode => Text_Io.Out_File,
                    Name => "!users.experimenter.acec2.C_data",
                    Form => "");

    Text_Io.Create (File => Instr_Data_File,
                    Mode => Text_Io.Out_File,
                    Name => "!users.experimenteracec2.I_data",
                    Form => "");

    Text_Io.Close (File => Instr_Data_File);

    Text_Io.Create (File => Run_Data_File,
                    Mode => Text_Io.Out_File,
                    Name => "!users.experimenter.acec2.R_data",
                    Form => "");

    Text_Io.Create (File => Total_Time_File,
                    Mode => Text_Io.Out_File,
                    Name => "!users.experimenter.acec2.Total_Time_Data",
                    Form => "");

    -- Open file containing test procedure names.
    Text_Io.Open (File => Source_File,
                  Mode => Text_Io.In_File,
                  Name => List_Of_Acec_Programs,
                  Form => "");

    Text_Io.Put_Line (Total_Time_File,
                      "ACEC run begins at " & Time_Utilities.Image
                                                (Time_Utilities.Get_Time));

    while not Text_Io.End_Of_File (Source_File) loop

        Text_Io.Get_Line (Source_File, Test_Name, Last);
        Harness (Test_Name (1 .. Last));

    end loop;

    Text_Io.Put_Line (Total_Time_File,
                      "ACEC run ends at " & Time_Utilities.Image
                                              (Time_Utilities.Get_Time));
```

```
    Text_Io.Close (File => Source_File);
    Text_Io.Close (File => Comp_Data_File);
    Text_Io.Close (File => Run_Data_File);
    Text_Io.Close (File => Total_Time_File);

    Text_Io.Put ("All tests have been submitted for testing");

exception
    when others =>

        -- Save results.
        Text_Io.Close (File => Source_File);
        Text_Io.Close (File => Comp_Data_File);
        Text_Io.Close (File => Run_Data_File);

        -- Log death point.
        Text_Io.Put_Line (Total_Time_File,
                          "Run died on " & Test_Name (1 .. Last));

        Text_Io.Put_Line (Total_Time_File,
                          "ACEC run ends at " & Time_Utilities.Image
                                                (Time_Utilities.Get_Time));

        Text_Io.Close (File => Total_Time_File);

        -- Notify Initiator of ACEC run.
        Text_Io.Put_Line
           ("++++++++++++++++++++++++++++++++++++++++++++++++++++++++");
        Text_Io.Put_Line
           ("++++++++++++++          KABOOM!!!          ++++++++++++++");
        Text_Io.Put_Line
           ("++++++++++++++++++++++++++++++++++++++++++++++++++++++++");

end Harness_Many;
```

C.5. Harness (Source State to Coded State)

The following is the Ada separate needed to compile the ACEC tests, already in Rational source state to Rational coded state. This version of **Harness** also records link and load time as a part of program execution time.

```
with Io_Exceptions, Directory_Tools;
use Directory_Tools;
separate (Harness_Many)
procedure Harness (Test_File : in Io_Package.Name_Type) is

    subtype Constrained_Duration is Duration delta 0.01;

    Year : Calendar.Year_Number;
    Month : Calendar.Month_Number;
    Day : Calendar.Day_Number;
    Begin_Time : Calendar.Day_Duration;
    End_Time : Calendar.Day_Duration;
    Total_Elapsed_Time : Calendar.Day_Duration;

    Beg_Cpu_Time : Constrained_Duration;
    End_Cpu_Time : Constrained_Duration;
    Total_Cpu_Time : Constrained_Duration;

    -- The software adds no comments to Compilation, Run_Time or
    -- Instrumentation data records.
    Comment_Width : constant Natural := 0;
    Comments : constant String (1 .. Comment_Width) :=
       (others => ' ');

    Compilation_Data :
       Io_Package.Compilation_Record_Type (Comment_Width);

    Run_Time_Data :
       Io_Package.Run_Time_Record_Type (Comment_Width);

    -- A bug in the Rational Environment prevents a procedure that compiles
    -- an Ada object from measuring its size.  We therefore return an
    -- arbitrary value until the bug in the Rational Environment is fixed.
    function The_Object_Code_Size_Of
                (Test_File : in Io_Package.Name_Type) return Natural is
        Obj : Object.Handle := Naming.Resolution (Test_File & "'body");
    begin
        return Natural (Statistics.Object_Size (Obj) / 8);
    end The_Object_Code_Size_Of;


begin
    -- stage 1: Recording Compilation and Link Time
    -- record the current elapsed and CPU times before compilation --
    Calendar.Split (Calendar.Clock, Year, Month, Day, Begin_Time);

    Beg_Cpu_Time := System_Utilities.Cpu;

    -- compile the test + record end times --
    Compilation.Promote (Unit => "!users.experimenter.acec2." & Test_File,
                         Scope => Compilation.Subunits_Too,
                         Goal => Compilation.Coded,
                         Limit => Compilation.Same_World,
                         Effort_Only => False,
                         Response => Profile.Get);

    End_Cpu_Time := System_Utilities.Cpu;

    Calendar.Split (Calendar.Clock, Year, Month, Day, End_Time);

    -- calculate the elapsed times (in hundredths of seconds) --

    Total_Cpu_Time := End_Cpu_Time - Beg_Cpu_Time;
    Total_Elapsed_Time := End_Time - Begin_Time;

    -- put the compilation statistics in the output record --
    Compilation_Data :=
       (Comment_Width, Test_File, Total_Elapsed_Time, Total_Cpu_Time,
        The_Object_Code_Size_Of (Test_File),
        Comments);

    Io_Package.Put (File => Comp_Data_File,
                    Value => Compilation_Data);
```

```
        -- stage 2:  Recording Execution Time --
        -- record the current elapsed and CPU times before execution --
        Calendar.Split (Calendar.Clock, Year, Month, Day, Begin_Time);
        Beg_Cpu_Time := System_Utilities.Cpu;

        -- execute test + record end times--
        Program.Run (S => Test_File, Context => "!users.experimenter.acec2");
        End_Cpu_Time := System_Utilities.Cpu;

        Calendar.Split (Calendar.Clock, Year, Month, Day, End_Time);


        -- calculate the elapsed times (in hundredths of seconds) --
        Total_Cpu_Time := End_Cpu_Time - Beg_Cpu_Time;
        Total_Elapsed_Time := End_Time - Begin_Time;

        -- put the runtime statistics in one output record --
        Run_Time_Data :=
           (Comment_Width, Test_File, Total_Elapsed_Time,
            Total_Cpu_Time, 512, 512, Comments);

        Io_Package.Put (Run_Data_File, Run_Time_Data);

        -- stage 3: Append Instrumentation Statistics to
        --              Instrumentation_Statistics_File;
        File_Utilities.Append (Source => "!users.experimenter.acec2.Instr",
                               Target => "!users.experimenter.acec2.I_Data");

    end Harness;
```

C.6. Harness (Text File to Loaded Main Programs)

This version of the Ada separate compiles the ACEC test suite programs from ASCII file state to Rational loaded main programs. It counts compilation time as time to promote from text file to coded state and time to load. Runtime represents the time to execute the already loaded main program. (Each ACEC test program is a main program.)

```
with Io_Exceptions, Directory_Tools;
use Directory_Tools;
separate (Harness_Many)
procedure Harness (Test_File : in Io_Package.Name_Type) is

    subtype Constrained_Duration is Duration delta 0.01;

    Year : Calendar.Year_Number;
    Month : Calendar.Month_Number;
    Day : Calendar.Day_Number;
    Begin_Time : Calendar.Day_Duration;
    End_Time : Calendar.Day_Duration;
    Total_Elapsed_Time : Calendar.Day_Duration;

    Beg_Cpu_Time : Constrained_Duration;
    End_Cpu_Time : Constrained_Duration;
    Total_Cpu_Time : Constrained_Duration;

    -- The software adds no comments to Compilation, Run_Time, or
    -- Instrumentation data records.
    Comment_Width : constant Natural := 0;
    Comments : constant String (1 .. Comment_Width) :=
       (others => ' ');

    Compilation_Data :
       Io_Package.Compilation_Record_Type (Comment_Width);

    Run_Time_Data :
       Io_Package.Run_Time_Record_Type (Comment_Width);

    -- A bug in the Rational Environment prevents a procedure that compiles
    -- an Ada object from measuring its size.  We therefore return an
    -- arbitrary value until the bug in the Rational Environment is fixed.
    function The_Object_Code_Size_Of
                (Test_File : in Io_Package.Name_Type) return Natural is
        Obj : Object.Handle := Naming.Resolution (Test_File & "'body");
    begin
        return Natural (Statistics.Object_Size (Obj) / 8);
    end The_Object_Code_Size_Of;


begin
    -- stage 1: Recording Compilation and Link Time
    -- record the current elapsed and CPU times before compilation --
    Calendar.Split (Calendar.Clock, Year, Month, Day, Begin_Time);

    Beg_Cpu_Time := System_Utilities.Cpu;

    -- compile the test + record end times --
    Compilation.Compile (File_Name  =>
                          "!users.experimenter.acec." & Test_File,
                         Library => "!users.experimenter.acec3",
                         Goal => Compilation.Coded,
                         List => False,
                         Source_Options => "",
                         Limit => Compilation.Same_World
                         Response => Profile.Get);

    Compilation.Load (File_Name =>
                        "!users.experimenter.acec3." &
                        Test_File(1..6),
                      To => "!users.experimenter.acec4",
                      Response => <PROFILE>);

    End_Cpu_Time := System_Utilities.Cpu;

    Calendar.Split (Calendar.Clock, Year, Month, Day, End_Time);

    -- calculate the elapsed times (in hundredths of seconds) --

    Total_Cpu_Time := End_Cpu_Time - Beg_Cpu_Time;
    Total_Elapsed_Time := End_Time - Begin_Time;
```

```
      -- put the compilation statistics in the output record --
      Compilation_Data :=
         (Comment_Width, Test_File, Total_Elapsed_Time, Total_Cpu_Time,
          The_Object_Code_Size_Of (Test_File),
          Comments);

      Io_Package.Put (File => Comp_Data_File,
                      Value => Compilation_Data);

      -- stage 2:  Recording Execution Time --
      -- record the current elapsed and CPU times before execution --
      Calendar.Split (Calendar.Clock, Year, Month, Day, Begin_Time);
      Beg_Cpu_Time := System_Utilities.Cpu;

      -- execute test + record end times--
      Program.Run (S => Test_File, Context => "!users.experimenter.acec4");
      End_Cpu_Time := System_Utilities.Cpu;

      Calendar.Split (Calendar.Clock, Year, Month, Day, End_Time);


      -- calculate the elapsed times (in hundredths of seconds) --
      Total_Cpu_Time := End_Cpu_Time - Beg_Cpu_Time;
      Total_Elapsed_Time := End_Time - Begin_Time;

      -- put the runtime statistics in one output record --
      Run_Time_Data :=
         (Comment_Width, Test_File, Total_Elapsed_Time,
          Total_Cpu_Time, 512, 512, Comments);

      Io_Package.Put (Run_Data_File, Run_Time_Data);

      -- stage 3: Append Instrumentation Statistics to
      --              Instrumentation_Statistics_File;
      File_Utilities.Append (Source => "!users.experimenter.acec2.Instr",
                             Target => "!users.experimenter.acec2.I_Data");

   end Harness;
```

C.7. Commands to Run the ACEC Test Suite

The above procedures, specifications, and either of the separates must be compiled in addition to the support programs provided by the ACEC Test Suite.

The ACEC Test Suite can then be run in batch mode by providing the following command in a command window and promoting the command to execute.

```
Program.Run_Job(S => "Harness",
                Debug => False,
                Context => "!Users.experimenter.Acec2",
                After => 0.0,
                Options => "Output =: Run_Job_Output; Error =: Run_Job_Error",
                Response => "<PROFILE>");
```

Note that a value of positive seconds may be provided for the **After** parameter. The **Harness** would then not start executing until after that number of seconds had elapsed since the command window was promoted.

References

[1] Hook, Audrey A., Riccardi, Gregory A., Vilot, Michael, Welke, Stephen.
User's Manual for the Prototype Ada Compiler Evaluation Capability (ACEC) Version 1.
Institute for Defense Analyses, 1801 N. Beauregard Street, Alexandria, VA 22311, October 1985.

[2] *VAX Ada Language Reference Manual*
Digital Equipment Corporation Maynard, Massachusetts., .

[3] Bassman, Mitchell J., Dahlke, Carl.
An Evaluation of the Rational R100 Development System Using the DoD Software Engineering Institute Methodology.
Computer Sciences Corporation, January 1987.

[4] Feiler, Peter H., Dart, Susan A., Downey, Grace.
Evaluation of the Rational Environment.
Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie Mellon University, July 1988.

[5] Feiler, P. H., Smeaton, R.
The Project Management Experiment: Evaluation of Ada Environments.
Technical Report CMU/SEI-88-TR-7, Software Engineering Institute, Carnegie Mellon University, July, 1988.

[6] Rational Environment documentation.
User's Guide (8001A-05), *Basic Operations Manual* (8001A-03), and *Reference Manuals 1-11* (8001A-03).
Delta Release, Rev 5.0, 1987.

[7] Weiderman, N.H., et al.
Evaluation of Ada Environments.
Technical Report CMU/SEI-87-TR-1, ADA180905, Software Engineering Institute, Carnegie Mellon University, January 1987.

Table of Contents

List of Tables