Technical Report
CMU/SEI-88-TR-015
ESD-TR-88-016

Evaluation of the Rational Environment

Peter H. Feiler
Susan A. Dart
Grace Downey

July 1988

Technical Report
CMU/SEI-88-TR-015
ESD-TR-88-016
July 1988

Evaluation of the Rational Environment

Peter H. Feiler
Susan A. Dart
Grace Downey

Evaluation of Environments Project

Unlimited distribution subject to the copyright.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the SEI Joint Program Office HQ ESC/AXS

5 Eglin Street

Hanscom AFB, MA 01731-2116

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.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF, SEI Joint Program Office

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

Copyright 1988 by Carnegie Mellon University.

Permission to reproduce this document and to prepare derivative works from this document for internal use is granted, provided the copyright and \'No Warranty\' statements are included with all reproductions and derivative works. Requests for permission to reproduce this document or to prepare derivative works of this document for external and commercial use should be addressed to the SEI Licensing Agent.

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN \'AS-IS\' BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTIBILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc. / 800 Vinial Street / Pittsburgh, PA 15212. Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page at http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service / U.S. Department of Commerce / Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides acess to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential con tractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: 1-800-225-3842 or 703-767-8222.

ALS is a trademark of SofTech. APOLLO and DSEE are registered trademarks of Apollo Computer, Inc. Apple is a trademark of Apple Computers, Inc. DEC, DEC/CMS, MicroVAX, VAX, VAX/VMS, VMS, and VT100 are trademarks of Digital Equipment Corporation. IBM and MVS are registered trademarks of International Business Machines Corporation. ISTAR is a trademark of Imperial Software Technology, Ltd., London. Macintosh is a trademark of MacIntosh Laboratories Inc. and is licensed to Apple Computer, Inc. Rational, R1000, and Rational Environment are

trademarks of Rational. Smalltalk-80 is a trademark of Xerox. Sun is a trademark of Sun Microsystems, Inc. UNIX is a registered trademark of AT&T Bell Laboratories. Verdix is a trademark of Verdix Corporation. XEROX is a registered trademark of Xerox.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.

Table of Contents

1. Introduction **3**
1.1. Background 3
1.2. The Rational Environment as Evaluated 4
1.3. Scope of Evaluation 5
1.4. Road Map for the Reader 6

2. Distinguishing Architectural Characteristics **9**
2.1. A Language-Centered Environment for Code Development and Maintenance 9
2.1.1. Consequences of Diana as Primary System Structure 10
2.1.1.1. Ada Objects 11
2.1.1.2. Changing Ada Objects 11
2.1.1.3. Browsing Syntactic and Semantic Information 12
2.1.1.4. Uniformity Through Objects 12
2.1.1.5. Uniform Use of Ada 12
2.1.2. Smart Processing 13
2.1.2.1. Basis for Effective Smart Processing 14
2.1.2.2. Processing in Small Pieces 14
2.1.2.3. Minimizing Reprocessing: A Cooperative Effort 15
2.1.2.4. Subsystems for Partitioning Large Systems 15
2.1.2.5. Support for Incomplete Programs 17
2.2. A Specialized System 17
2.2.1. Consequences of Specialized Hardware 18
2.2.1.1. Benefits of an Ada Machine 18
2.2.1.2. Space Utilization 18
2.2.1.3. Integration into a Computing Environment 19
2.2.2. Consequences of Specialized Software 20
2.2.2.1. Learnability 20
2.2.2.2. Maturity 21
2.2.2.3. Porting of Software 21
2.2.2.4. Integration of Tools 22
2.2.2.5. Environment Extensibility 23
2.2.2.6. Host/Target Support 23
2.3. Multiple User Support 24
2.3.1. Version and Configuration Control 24
2.3.1.1. CMVC Model 24
2.3.1.2. CMVC Implementation 28
2.3.2. Workorder Management 30

3. Capabilities of the Rational Environment **33**
3.1. Functionality 33
3.1.1. Objects and Common Operations 33
3.1.1.1. Objects and Naming 34
3.1.1.2. Common Operations 35

3.1.2. Editing 35
3.1.2.1. General Editor Support 36
3.1.2.2. Text Editing 38
3.1.2.3. Structure Editing 39
3.1.3. Browsing 40
3.1.4. Ada Code Development 41
3.1.4.1. Code Creation 41
3.1.4.2. Compilation 42
3.1.4.3. Error Handling 44
3.1.4.4. Execution 45
3.1.4.5. Library Management 46
3.1.4.6. Debugging 47
3.1.4.7. Testing 50
3.1.5. Configuration Management 50
3.1.5.1. Partitioning Concepts 51
3.1.5.2. Partitioning Management 52
3.1.5.3. Controlled Configurations 54
3.1.5.4. Cooperative Code Development 56
3.1.5.5. Independent Code Development 57
3.1.5.6. System Composition 58
3.1.5.7. Database Maintenance 59
3.1.6. Operating System and System Administration Features 60
3.1.6.1. Tailoring of Display and Logs 60
3.1.6.2. File Handling and Logging 61
3.1.6.3. Job and Program Control 62
3.1.6.4. System Administration 62
3.1.6.5. Access Control 63
3.2. Documentation 64
3.2.1. Printed Documentation 65
3.2.2. Online Help 66
3.3. User Interface 67
3.4. Performance 69
3.4.1. Timing Issues 70
3.4.2. Space Issues 71

4. Conclusions 73
4.1. Technological Advances in the Rational Environment 73
4.1.1. Semantics-Based Interaction 73
4.1.2. Large-Scale Code Development Support 74
4.2. The Rational Environment as a Product 76
4.2.1. Functionality Coverage 76
4.2.2. Learnability and Maturity 77
4.2.3. Effectiveness of a Specialized System 78
4.2.4. Integration into a Project Organization 79

References 81

List of Figures

Figure 2-1: Rational Subsystems 16
Figure 2-2: Multiple System Configurations 26
Figure 2-3: Team Development on One Subsystem Version 27
Figure 2-4: Parallel Development and Merging 27
Figure 3-1: Common Object Functionality 35
Figure 3-2: General Editor Functionality 37
Figure 3-3: Textual Editing Functionality 38
Figure 3-4: Structural Editing Functionality 39
Figure 3-5: Browsing Functionality 40
Figure 3-6: Code Creation Functionality 41
Figure 3-7: Compilation Functionality 42
Figure 3-8: Error Handling Functionality 45
Figure 3-9: Execution Functionality 45
Figure 3-10: Library Management Functionality 46
Figure 3-11: Debugging Functionality 48
Figure 3-12: Partitioning Management Functionality 52
Figure 3-13: Controlled Configurations Functionality 54
Figure 3-14: Versioning and Configuration in a Subsystem 55
Figure 3-15: Cooperative Code Development Functionality 56
Figure 3-16: Parallel Code Development Functionality 57
Figure 3-17: System-Level Composition Functionality 58
Figure 3-18: Database Maintenance Functionality 59
Figure 3-19: Display and Log Tailoring Functionality 61
Figure 3-20: File Handling and Logging Functionality 62
Figure 3-21: Job and Program Control Functionality 63
Figure 3-22: System Administration Functionality 63
Figure 3-23: Rational's Printed Documentation 65

Evaluation of the Rational Environment

Abstract. This report presents an analysis of the Rational R1000 Development System for Ada, also called the *Rational Environment*. The evaluation combined the use of the Software Engineering Institute (SEI) methodology for evaluation of Ada environments, an analysis of functionality not covered by that methodology, and an assessment of the novel environment architecture of the Rational Environment. In addition to this report, *Experiment Transcripts for the Evaluation of the Rational Environment*, by Grace Downey, Mitchell Bassman, and Carl Dahlke (CMU/SEI-88-TR-21, Software Engineering Institute, Carnegie Mellon University, 1988) contains support material for the experimental results. The support material is the result of performing experiments based on the SEI's environment evaluation methodology. It consists of transcripts of the experiments, the detailed answers to the evaluative questions, and the detailed performance results.

1. Introduction

This report is organized as follows. The introductory chapter presents some background information for the evaluation, describes the configuration of the environment being evaluated, defines the scope of the evaluation, and gives the reader a road map to assist in the reading of this document. Chapter 2 discusses the distinguishing architectural characteristics of the Rational Environment, highlighting in general its novel aspects compared to more conventional Ada language systems. Chapter 3 summarizes the capabilities of the Rational Environment in terms of the provided functionality, the user interface, the documentation, and some performance aspects. The report concludes with a summary of major findings.

1.1. Background

One of the goals of the SEI is to assess advanced software development technologies and to accelerate the transition of those that appear promising. Ada Programming Support Environments (APSEs), Software Development Environments (SDEs), and Integrated Project Support Environments (IPSEs) have been recognized as a targets of opportunity for the SEI. In 1985 the Evaluation of Ada Environments (EAE) Project was started. The two major results have been the definition and development of a systematic methodology for evaluating environments and the study of three of commercial APSEs (ALS, Verdix Ada, and DEC Ada). A description of the methodology can be found in [9], and the results of the study are published in [10].

Under the name of Evaluation of Environments Project, a continuation of the EAE Project has made advances in two areas. First, the project has started to investigate unconventional environment architectures by evaluating the Rational Environment. The SEI environment evaluation methodology assesses the functional capabilities of an environment by performing a predefined set of experiments. During the course of this evaluation, it became evident that architectural issues, especially those relating to unconventional architectures, are not adequately covered by this methodology, nor are all functionality experiments included in it. This evaluation report addresses some of these deficiencies.

Second, the Environments Project has extended the coverage of the methodology with an extensive project management experiment for assessment of SDEs and IPSEs [3]. The emphasis of this experiment is on computer-based support for management of the development process, i.e., for project management and its integration with development support. The project has evaluated the ISTAR IPSE from Imperial Software Technology. The results of the ISTAR evaluation are documented in [5].

1.2. The Rational Environment as 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 DEC server.

The software configuration is Release *D_9_25_1* or *delta0* of the Rational Environment. The evaluated environment is 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 form of the subsystem concept.
- Configuration management and version control support.
- Workorder management support.

The base environment can be extended with additional software packages that are sold as separate products by Rational or that are unsupported tools contributed by users. Product packages include two products that enhance the code development capability of the Rational Environment, a facility for document production, and a facility for electronic communication. The products are:

- The Cross-Development Facility for the Motorola 68000 family, the DEC/VAX architecture, and the MIL-STD-1750A architecture.
- The Target Build Facility for downloading of Ada source and compilation on other hosts.
- The Design Facility, which supports the use of Ada as a program design language (PDL) with structuring of program unit comments and generation of DoD-STD-2167 documentation from information provided therein, as well as a document formatter language processor.
- An electronic mail facility (Rational Network Mail).

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.

1.3. Scope of Evaluation

The evaluation of the Rational Environment involves the use of the SEI methodology for evaluation of environments, the analysis of functionality not covered by the experiments of the SEI methodology, and an assessment of the Rational Environment architecture.

The SEI methodology requires the evaluator to perform a set of experiments on the environment and answer questions based on the experience. The experiments cover different functionality areas. The current set of experiments consists of system management (system installation and administration), detailed design and coding, testing and debugging, compiler quality (Ada compiler evaluation capability [ACEC]), configuration management, and project management. The questions cover four evaluation areas: functionality, performance, user interface, and system interface. The methodology is described in detail in [9]. We have performed experiments on the Rational Environment in a manner similar to previous evaluations.

The Rational Environment, however, provides additional functionality that is not evaluated by the experiments in the SEI method. Hence, we have analyzed those facilities based upon our own hands-on experience. Also, we have assessed the architecture of the Rational Environment for its properties. No prescribed method existed for assessing architectural aspects of environments. Since the Rational Environment's architecture is quite different from that of conventional environments, we chose to present the distinguishing aspects that implement the end-user capabilities.

The term *conventional environment* refers to a toolset on top of a common operating system such as UNIX, Dec's VMS, or IBM's MVS—the toolset consisting of a text editor or language sensitive editor, a compiler, a linker, a build tool, a debugging tool, and a version control tool.

The set of basic mechanisms provided by the Rational Environment is intended to be used as is, following conventions outlined in the Rational documentation. However, as the functionality is not expected to satisfy all needs, the Rational Environment has been made extensible, permitting customers to add a layer of Ada functions on top of the delivered environment. This layer would encode some of the conventions and policies. We chose to perform the evaluation experiments on the environment as it was delivered.

The Rational Environment differs from conventional architectures in the following way. Using knowledge of the programming language, the program structure, and its semantics, the Rational Environment optimizes user interaction by providing immediate feedback (even at the semantic level) upon request, by reducing the number of entities the user has to manage, by limiting them to logical entities, and by reducing the wait time on processing by processing in small pieces and only what is minimally necessary. The result is a more responsive interaction between the user and the environment, the essence of which cannot be captured by performing measurements of compilation of lines per minute, or source and object code sizes of program units.

The execution of the SEI methodology experiments on the Rational Environment resulted in the collection of some performance measures. The experiments were designed to collect measures for space, time, and responsiveness of various commands, operations, files, and objects. Stress testing of the environment through these experiments was limited to stress testing of the compiler by running the ACEC test suite. The collected measurements do not include performance figures under system load or for development of very large Ada systems.

Because the Rational Environment is unconventional and readers may not have available a description of it, this report describes as well as analyzes new and distinguishing concepts of the Rational Environment.

1.4. Road Map for the Reader

This report does not require the reader to be familiar with the Rational Environment. Nor does the reader need to be familiar with the syntax and semantics of the Ada language, although familiarity with the concepts of module interface descriptions, separation of specification and implementation units, and separate compilation is useful.

Readers who want a quick summary of this report should read the introductory chapter (Chapter 1) and the conclusions (Chapter 4).

The distinguishing characteristics of the Rational Environment architecture and the conceptual models of its novel features are discussed in Chapter 2, while the practicality of the provided capabilities is summarized and analyzed in Chapter 3. The following outline of the report gives a synopsis of the various sections of these two central chapters:

1. **Distinguishing Architectural Characteristics:** This chapter highlights three areas in which the Rational Environment distinguishes itself from conventional environments: being an interactive, language-centered code development and maintenance environment, being an environment with specialized hardware and software, and being an environment that provides innovative version and configuration control.

    - **A language-centered environment for code development and** maintenance: The Rational Environment as evaluated can be called a language-centered code development and maintenance environment. This section defines language-centered and describes the benefits of such. The two factors that are the major contributors to this are discussed in detail.

        - **Consequences of Diana as primary system structure:** This section describes Diana and the way in which the Rational Environment exploits its use. It discusses the benefits of Ada objects over the use of multiple files, the way in which the user modifies Ada objects and compiles them, the benefits of treating all environment structures uniformly as objects, and the benefits of uniform use of Ada as a consequence of using Diana.
        - **Smart processing:** This section discusses the basis for effective smart processing, the benefits of processing in pieces, a cooperative approach to minimizing reprocessing, the concept of subsystem as a partitioning and composition mechanism for large systems, and the

ability to support execution of incomplete programs—making the Rational Environment a candidate for exploratory programming and prototyping.

- **A specialized system:** The Rational Environment is a specialized system in several respects. Its functionality is concentrated on Ada code development, its style of interaction with users differs from that of conventional Ada environments, its user interface and terminal are unconventional, and the processor was specifically designed to support the execution of Ada. The consequences of such specialized hardware and software are discussed.

  - **Consequences of specialized hardware:** This section first discusses the benefits of a processor that is an Ada machine in terms of execution speed and a hardware-supported, persistent object management system. Next, space utilization is discussed since the space consumption of Diana representations can be an issue. Being a special purpose machine, the Rational Environment will have to be integrated into the regular computing environment. In that context, issues such as use as a compilation engine, network support, and remote access are discussed.

  - **Consequences of specialized software:** This section discusses consequences of the specialized software in the Rational Environment. The consequences are organized into the following issues: the ease with which the environment is learned, the maturity of the environment, the acquisition and porting of software from other hosts and Ada environments, the extension of the environment through integration of tools and tailoring, and support for host/target development.

- **Multiple user support:** The Rational Environment emphasizes support for large-scale development of Ada systems. Key elements to such support are versioning, composition, and configuration control facilities and support for managing and tracking the tasks of developers.

  - **Version and configuration control:** The version and configuration control model (CMVC) supported by the Rational Environment is an improvement over common approaches of combining conventional Ada compilers and program library facilities with separate source code version control tools. This section discusses the benefits of the CMVC model, which is built around the subsystem concept and nicely integrates Ada program libraries and version and configuration support. The section also talks about the implementation of CMVC.

  - **Workorder management:** The Rational Environment provides some support for management and tracking of user tasks.

2. **Capabilities of the Rational Environment:** This chapter analyzes the capabilities provided by the Rational Environment in some detail. The major portion of the chapter concentrates on the functionality provided by the Rational Environment. The remaining three sections discuss the quality of documentation, the on-line help facility, and the user interface model supported by the Rational Environment, as well as some of the performance figures.

   - **Functionality:** This section covers the spectrum of available functionality in the environment, with the exception of workorder management since it is discussed in the previous chapter. Subsections present brief overviews of the available functionality in that area. The section starts out with highlighting the concept of common operations on objects, elaborates on general editing and

browsing capabilities, and then details the area of Ada code development—beginning with code creation and including debugging and testing. The section continues with configuration management support and closes with a discussion of operating system and system administration facilities such as file handling and logging, job and program control, archiving, access control, and tailoring of displays.

- **Documentation:** This section analyzes the organization and quality of the documentation and on-line help that is provided with the Rational Environment.
- **User interface:** This section describes the user interaction model of the Rational Environment and the ease with which the user interface is learned.
- **Performance:** This section discusses the major results from the performance data collected during the execution of the experiments. The performance results are in terms of space consumption, timing, and responsiveness. (The reader is reminded that given the time frame for the evaluation we have not been able to perform stress tests on the Rational Environment other than execution of the ACEC test suite.)

2. Distinguishing Architectural Characteristics

This chapter summarizes the Rational Environment architecture, including its derived characteristics. We consider the Rational Environment as it was evaluated to be a language-centered environment for code development and maintenance, and discuss the components of its architecture that characterize it as such. These components are the use of Diana as primary program representation, the introduction of the concept of subsystems, and the application of smart reprocessing techniques. This is followed by a discussion of the benefits and handicaps of such a specialized system, including the effects of specialized hardware as well as software in terms of performance and space utilization, integration of the Rational Environment into an organization's computing environment, and portability of Ada software. This section closes with a discussion of the support for large program development by multiple users, highlighting the configuration management and version control model and the workorder management model provided by the Rational Environment.

2.1. A Language-Centered Environment for Code Development and Maintenance

We have analyzed environments by categorizing them along several dimensions. Two of these dimensions are *functionality* and *environment architectures*.

Functionality can be viewed in terms of how much of the life cycle is supported or what development and management roles are supported. The Rational Environment as evaluated in this report (see Section 1.2) is best described as an *Ada code development and maintenance environment*. It provides support for interactive coding and debugging of Ada programs. Detailed design is supported in as far as Ada can be used for that purpose. Testing is supported in limited form with the capability of generating program unit stubs and executing incomplete programs. System integration is supported through the concept of subsystems. It permits partitioning of a system and efficient composition with different versions of subsystems. The Rational Environment provides support for version control and configuration management of Ada program units and text files. It specializes in supporting the management of Ada programs through subsystem interface checking and through selection and dynamic composition of subsystem versions. In addition, the Rational Environment provides mechanisms for workorder management—primitives for a task management facility.

Environment architectures can be characterized by categories defined in [1]. Environments can be *language-centered*, *structure-oriented*, *toolset-based*, or *method-based*. Language-centered environments are centered around a single programming language, are responsive (making use of, and available to the user, semantic information about the program), tend to be implemented in the language they support, and embed the application in the development environment, giving it access to many of the facilities used to implement the environment. Language-centered environments tend to provide little support for multiple programmers. Structure-oriented environments view programs as structures and allow users to manipulate them as such. Different from language-centered environments, language-specific information is not hard-coded in structure-

oriented environments, but passed to the environment in a generation process. Toolkit environments represent collections of tools that use a standard operating system as their common platform. Generic programming-in-the-large tools can be found. Efforts are underway to raise the level of functionality of the platform, especially in the areas of data management and networking. Method-based environments support a particular method of software development. They fall into two groups: interactive (and often graphical) tools supporting one or several methods in a particular phase of the life cycle (such as design) and environments with support for managing the development process. The latter are often referred to as Integrated Project Support Environments (IPSE).

The Rational Environment architecture can be characterized as a *language-centered environment*. As such, the Rational Environment is tailored to support the Ada programming language. It provides a responsive facility for code development and maintenance. It makes effective use of a representation that maintains the structural and semantic information of programs rather than storing programs with text as its primary program representation. Through partitioning, runtime type and interface checking, and incremental processing, the Rational Environment is able to provide a language-sensitive facility for exploratory programming and prototyping in Ada. The Rational Environment is implemented in Ada. Application programs have a rich base on which to build components of the Rational Environment and execute components embedded in it. By the same token, version control and configuration management facilities are provided to support the production of large Ada systems with teams of developers.

The following subsections discuss two key components of the Rational Environment as a language-centered environment: the representation and effective use of structural and semantic information of programs and incremental processing techniques to achieve a responsive environment for a language with a high degree of static semantic checking.

2.1.1. Consequences of Diana as Primary System Structure

This section gives a short background sketch of Diana, Rational's primary system structure, and discusses the basic unit of Diana structures—the program unit—its role as the single representation of a program unit, the states of its representation, and utilization of the Diana structure within program units. The section continues with a description of program libraries, the entity in which program units are maintained, and a discussion of the uniform object manipulation paradigm supported by the Rational Environment. The section concludes with a discussion of the facilities for manipulation of Diana structures by program, for dynamically binding to existing Ada system components, and the use of Ada as command language. These facilities give the Rational Environment some of the characteristics found in Lisp systems, which encourages experimental programming and prototyping.

Diana, a Descriptive Intermediate Attributed Notation for Ada [4], defines an abstract syntax representation of Ada programs that is attributed with semantic information. Its original intent was to provide a standard intermediate Ada program representation that is passed between phases of a compiler, and thus would permit the combination of different Ada compiler front-ends and back-ends.

2.1.1.1. Ada Objects

Rational has chosen Diana as the primary program representation for its Rational Environment. Ada program units are maintained as Ada objects. Ada objects are implemented as Diana structures, i.e., structures representing the syntactic structure and carrying semantic information and executable code. The textual representation of a program unit is derived from the Diana structure and is updated whenever the Diana structure changes. Users can also edit the text representation directly. Those changes are then parsed into the Diana structure and a formatted text representation derived from it. As such, Ada objects combine the notions of source code and object code into one entity. This differs from the traditional view that source code, object code, and executable images are stored in different files, resulting in replication of information and more objects to be managed directly by the user.

Ada objects are modified with an Ada object editor. The user can do so by modifying the text representation with text editing facilities, or by applying structure-editing operations to the Diana structure. The user can move the cursor textually or structurally. During structural cursor movement, the appropriate syntactic Ada construct represented by the Diana structure is highlighted. The user does not have to know Diana to use the structure-editing capabilities; knowledge of syntactic Ada constructs is sufficient.

2.1.1.2. Changing Ada Objects

Ada objects exist in one of four states: *archived*, *source*, *installed*, and *coded*. These states can be viewed as states of the ‘‘compiledness’’ of an Ada object. The state of an Ada object is changed by the user’s promoting or demoting it. Promotion moves the object to a higher state, while demotion moves the object to a lower state.

In archived state, program units are stored in compacted form, i.e., pure text, which will have to be reparsed and recompiled before use.

In source state, the Ada object can be freely edited as if it were a text file; the user can move the cursor on a character and line basis as well by syntactic structure. The modified text is incrementally parsed and Ada construct completion is provided.

An Ada object in its installed state is semantically correct and becomes available in the program library, i.e., other program units can be semantically analyzed against it. When modifications are restricted to additions of declarations and statements and modification or deletion of declarations and statements (if there are no dependencies on them), demotion to source state and reprocessing can be avoided.

Promotion of the object to coded state will result in the generation of code. In coded state, editor operations are restricted to addition, modification, and deletion of comments, as well as addition of declarations to package specifications. Use of these restricted operations does not require recompilation.

If the state of the Ada object is too restrictive for modification, the user can demote its state. A program unit can be demoted to *installed* or *source* state. A program fragment within a program unit can be demoted to a source state while the program unit is in an installed state. Depending

on the chosen demote command, demotion will report obsolescent program units and abort the demotion, or automatically cause the demotion of any dependent program units to the same state. By demoting a selected part of program representation to a particular state, the user indicates to the editor the scope of a planned change and the type of change. Taking advantage of the semantic information in the Diana representation, the editor can be smart about the consequences of the change and minimize reprocessing. For further discussion of smart processing see Section 2.1.2.

2.1.1.3. Browsing Syntactic and Semantic Information

The availability of semantic information is also beneficial to the user directly. The user can find the declaration of any selected identifier, can query and browse all *usage* sites of a declaration (not related to the Ada USE clause), and can ask for parameter templates when entering procedure calls. These capabilities are available for programming as well as debugging. They can be a powerful tool when Ada software has to be maintained. The maintainer can use the Rational Environment to get an impression of the "gestalt" (structure and dependencies) of unknown systems quickly and to better understand the effect of future maintenance changes.

2.1.1.4. Uniformity Through Objects

An Ada object is created in a program *library* and is stored under the same name as the program unit. Libraries can contain libraries and are manipulated through a library editor. Entries in a library are treated as objects to which operations can be applied. The basic editing commands are the same as those of the Ada object editor.

As a matter of fact, all Rational Environment components can be treated as objects. *Text files* are objects and are manipulated with a text *object editor*, whose functionality corresponds to the text editing capabilities of the Ada object editor. Other objects, such as *activities* and *workorders*, also have object editors with functionality similar to that of the library object editor. All object editors have functionality that is common to all object types and functionality that is type-specific. Consequently, the user interface has a large degree of uniformity.

2.1.1.5. Uniform Use of Ada

The Rational Environment is implemented in Ada. Ada packages exist for the different object types of the Rational Environment, as well as their operations. This includes the Ada object types, i.e., the Diana representation of programs. These Ada packages are visible to and can be invoked by the user of the Rational Environment in two ways: interactive invocation of system functions through the command processor, and extension of the Rational Environment through a layer of command procedures. The availability of Diana in program form permits programs to be written that manipulate Diana structures and take advantage of the parser available in the environment.

The Rational Environment uses Ada as its command language. The "commands" are the set of procedures available through a collection of Ada packages. For example, directory commands are the procedures available through the package(s) representing the two types of library objects. A user interacts with the command processor through the Ada object editor; all capabilities of this editor, such as call completion, are available. The command processor supports full Ada and executes commands by compiling the program unit in the command window. The "commands" of

the different system packages are available through a search list and are made visible, i.e., can be invoked without qualifying the package name, through a use clause in the command window program unit. The object-oriented user interface behavior is achieved by the command processor's supporting special default values for parameters, such as cursor or selected object. If such a procedure is invoked without a selection beforehand, the user will have to supply the parameter explicitly. Since the command language is Ada, command procedures simply become a matter of writing Ada procedures. In particular, this permits the user to extend the Rational Environment by adding a layer of functionality to the base environment, e.g., to implement certain development policies.

As a result of the availability of Ada interfaces to the Rational Environment, application developers have a large set of building blocks available. This permits certain classes of applications (i.e., interactive program code support tools) to be built quickly. However, because the application is now embedded in the Rational Environment, the application becomes dependent on the environment's being resident at runtime (similar to interpretive systems such as CommonLisp or Smalltalk80). If Ada code is to be developed for targets other than the R1000, reuse of the software available in the Rational Environment must be limited to those parts that are supported by the target runtime system as well (or can be taken out of the Rational Environment and added to the target runtime system).

2.1.2. Smart Processing

Smart processing refers to limiting the amount of processing or reprocessing necessary to reflect changes to a program. Smart processing holds that programs can be processed in small pieces more effectively than as a whole, especially if processing is done between user modifications and/or in background. This was the incentive for introducing separate compilation for many languages. Module interface descriptions and strong type checking (as can be found in Ada), however, can necessitate substantial (i.e., involving a large number of program units) reprocessing because of a change. Smart processing and reprocessing applies to the whole edit/compile/link/load cycle, i.e., to the use of incremental techniques in the link and load process as well as compilation. Furthermore, smart processing also includes the ability to handle incomplete programs.

The term *smart recompilation* has been coined by Tichy [8] to mean a technique for limiting the scope of propagation due to certain program changes. In this technique, a tool compares a representation (in Tichy's case parse trees) of a modified program with the original to determine the extent of changes made by the user. From the attained information, what has to be recompiled (if anything) is determined.

The Rational Environment supports smart processing. As a result of using incremental processing techniques throughout and handling incomplete programs, the Rational Environment is able to exhibit the behavior of an interpretive system allowing exploratory programming. In contrast to Tichy's approach, the Rational Environment approach to smart reprocessing requires cooperation from the user in that the user indicates the scope of the intended change by demoting the appropriate structure in the Diana representation.

2.1.2.1. Basis for Effective Smart Processing

The Rational Environment is able to provide smart processing because it maintains Ada programs as logical units, storing them in the Diana representation. This permits the Rational Environment to track user modifications to the program at the granularity of program units and smaller units (i.e., individual declarations and statements). The Diana representation not only reflects the abstract syntax of Ada programs (from which textual representations can be regenerated), but also contains semantic information. Semantic information includes type information and dependency information. Dependency information not only reflects export/import dependencies due to Ada *with* clauses, but actual usage sites (e.g., all procedure call sites). These actual usage sites are maintained within program units as well as across program units at the level of individual declarations.

On one hand, the Rational Environment makes this semantic information available to the user through queries. On the other hand, the Rational Environment takes advantage of this information to perform "smart reprocessing," i.e., keeping the reprocessing due to a change at a minimum. It maintains semantic information in the Diana representation consistently by demoting dependent program units when a declaring unit is demoted to allow modification. Note that demotion to source state results in removing the demoted unit from semantic information in the Diana representation, i.e., units in source state will not be found when queries are run on semantic information such as finding usage sites.

Conventional Ada environments require reprocessing of dependent program units any time any change is made to a program unit specification. The Rational Environment takes advantage of the Diana structure to permit certain modifications without invalidating dependent program units. These include modification of comments, addition of declaration to package specifications, etc. (see Section 2.1.1.2). As a result, fewer recompilations are required than in conventional Ada environments.

Over the life of a program, dependencies—such as "with" clause dependencies—will deteriorate in the sense that new dependencies are introduced and dependencies are rarely removed. This results in unnecessary recompilation after changes with side effects unless support is provided to remove unused imports. In the Rational Environment unused "with" clauses in a program unit can be queried and displayed through underlining. The underlined "with" clauses can then be removed with an edit command. Similarly, the Rational Environment provides a function to remove any unused imports of subsystems, a system partitioning facility similar to packages (see Section 2.1.2.4). Many conventional Ada environments do not provide such support.

2.1.2.2. Processing in Small Pieces

The Rational Environment supports the philosophy of processing in small pieces, both for consistently maintaining the Diana representation and for providing the user timely feedback about an operation. One form of it is "incremental" parsing, i.e., parsing of program text at any syntactic unit level. This means that users can edit Ada programs textually and ask for syntactic completion at any time. The system will complete the entered (partial) program fragment as far as possible, leaving placeholders for holes in the syntactic structure. For example, in case of procedure invocations the user will be provided with a template for the parameters to be supplied, displaying the formal parameter names and their default values.

Semantic analysis and code generation are performed at the granularity of program units by promoting them to installed and coded state. In each of the states certain modifications can be made without requiring demotion or propagation, e.g., insertion of comments or new declarations in coded state. When a program unit is demoted, dependent program units (based on actual dependencies) are demoted to the respective state automatically.

2.1.2.3. Minimizing Reprocessing: A Cooperative Effort

Users can limit semantic analysis and code generation to a unit smaller than a program unit. They can do so by demoting a substructure of a program unit specification into source state while the remainder of the program unit specification remains in coded state. For program unit implementations the program unit has to be brought to installed state before a substructure can be demoted to source state. Only those program units dependent on the demoted substructure, e.g., a single declaration, are demoted to the same state. The Rational Environment knows that the user only modifies the indicated program fragment and can minimize the necessary reprocessing. For example, if the change is to a comment no recompilation is necessary. Similarly, for program unit implementations only the substructure is semantically analyzed when promoted, and code is regenerated for the whole program unit when it gets promoted to coded state.

The user can further reduce the amount of reprocessing due to a modification by first querying all use sites of a declaration and demoting the respective statements. As a result, the dependencies are temporarily removed and not affected by the demotion and modification of the declaration. Once the declaration is promoted, the usage sites can be promoted after possible correction at the cost of only reprocessing the temporarily demoted statements. This latter technique, however, can be quite labor intensive for the user.

Notice, that the Rational Environment views smart reprocessing as a cooperative effort between the user and the environment. The user cooperates by indicating to the Rational Environment the scope of the intended changes by selecting and demoting the appropriate substructure in the Diana representation, e.g., only the comment of a procedure. If the user demotes a whole program unit, but does not make any change or only a small change such as a change to a comment, the Rational Environment will reprocess the whole program unit on promotion. It does not attempt to analyze the representation in source state to determine a scope of change smaller than the demoted Diana structure.

2.1.2.4. Subsystems for Partitioning Large Systems

The Rational Environment introduces the concept of *subsystem* as a facility for partitioning large Ada systems into manageable collections of Ada program units at a granularity larger than Ada packages. Subsystems act as *name scope boundaries*, *propagation boundaries*, and *composition boundaries*. The next three paragraphs elaborate in each of these scoping concepts.

Subsystems basically are program libraries with additional properties. Program units declared in a subsystem are not visible outside a subsystem unless they are exported, i.e., listed in the subsystem specification (*spec view*). By importing a subsystem specification, the facilities ex-

ported by that subsystem are made available in the importing subsystem. The Rational Environment checks against imported subsystem specifications during semantic analysis, i.e., during promotion to installed state. The facilities listed in a subsystem specification must be satisfied by its implementation (*load view*), i.e., the collection of program units contained in it. However, this condition is not checked until run time, or through explicit invocation of a checking function. As a result, subsystems can be developed independently, i.e., a subsystem specification can be provided for outside use while the subsystem implementation is still in progress. Figure 2-1 illustrates the scope of compile-time checking and run-time checking. The scope of compile-time checking for the implementation (load view) of subsystem B is shown with a bold line. The implementations of subsystems A and C have similar compile-time checking scopes, which are determined by their import dependencies. Subsystem B has two out of three program units exported, while the other two subsystems have all program units exported. All subsystems are shown with only one spec view and one load view each.

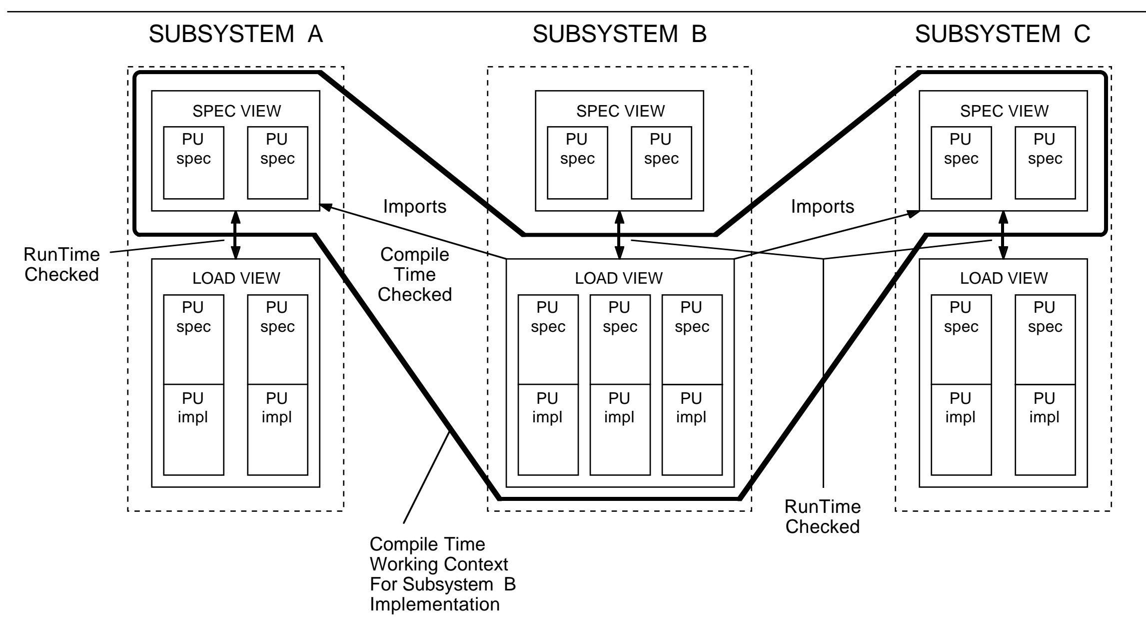

Figure 2-1: Rational Subsystems

Subsystems limit the propagation of changes, i.e., the invalidation of program units affected by the change, to the program units contained in a subsystem. This is accomplished by treating the subsystem specification and subsystem implementation as two separate program libraries. A developer making a change to a package specification only pays for the cost of recompiling the affected program units in the subsystem being worked on, rather than all program units of a system—an expensive proposition in large systems. If the changed specification is exported from the subsystem, the propagation beyond the subsystem boundary is not done at the cost of the original developer. Instead, the developer provides a new version of the subsystem specification, and owners of other subsystems are responsible for the recompilation cost of upgrading their subsystem import list to the newer version. The propagation is controlled as the owners of the affected subsystems can decide when to upgrade at a time appropriate for them.

In the Rational Environment Ada systems are composed of a collection of subsystems. This system composition is defined in an *activity file*, which is a table listing all subsystems comprising the system as well as the version of each subsystem specification and implementation. The compatibility of the subsystems specified in the activity file is checked dynamically. The Rational Environment checks that the facilities indicated in the selected version of each subsystem specification are being provided by the program units in the selected version of the subsystem implementation. It also checks that the version of the imported subsystem specifications that each subsystem implementation has been compiled against is the version of the subsystem specification as indicated in the activity file. When the user invokes the program for execution, subsystems listed in the activity file are dynamically linked and loaded. Incompatibilities are recognized at runtime and an exception is raised. Users can request the checking to occur before either by invoking a checking command or by prelinking and preloading. The interpretation of activity files allows users to produce different system configurations by building several activity files. The cost of building a new configuration is the construction of the activity file and the cost of interface checking at load or preload time (unless there are mismatches in the subsystems which require correction in the subsystem itself). The benefits are especially apparent at system integration and test time, when multiple system configurations are put together.

By comparison, the use of program libraries without subsystems in the Rational Environment or the use of program libraries in conventional Ada environments can require extensive recompilation. The user is allowed to split Ada systems into multiple program libraries. They represent separate name scopes. Search lists or links are used to make program units in libraries visible to command windows. Changes in program libraries that are depended upon are generally propagated to the using program libraries. A change in the composition of program libraries requires recompilation of the dependent program libraries. Thus, in conventional Ada environments, the cost of parallel development and processing side effects of changes is comparatively high, especially for large Ada systems.

2.1.2.5. Support for Incomplete Programs

The Rational Environment supports the execution of incomplete programs. The Rational Environment facilitates stub generation for program unit specifications. It also permits gaps to be left in a program unit at the statement level, i.e., placeholders for statements to be filled in or statements temporarily demoted to source state, while the whole program unit is in coded state, and executable. Similarly, *separate* subunits are not required to be in coded state for their parent unit to be executable. At runtime the Rational Environment raises an exception when execution reaches an incomplete program fragment. This is a convenient feature when exploratory programming or prototyping is exercised.

2.2. A Specialized System

A specialized environment architecture such as the Rational Environment has certain costs associated. In this section we examine the Rational Environment as a specialized system by first discussing the specialized hardware and its consequences, and then elaborating on the issue of specialized software.

2.2.1. Consequences of Specialized Hardware

The Rational Environment executes on special hardware, referred to as the R1000 processor. This hardware directly supports a virtual machine that can execute a high-level instruction set tailored to the Ada language. This is similar to hardware directly executing p-code for Pascal programs, Xerox's machines providing a Mesa/Cedar instruction set, or stack machines such as Burroughs' to better support block structured languages—in particular Algol.

2.2.1.1. Benefits of an Ada Machine

The instruction set reflects the Diana representation quite closely. Thus, the translation from Diana to object code is almost a one-to-one mapping. As a result source code debugging can be provided without excessive effort by the debugger to understand and compensate for compiler optimizations. At the same time the executable code is more compact because the instruction set is higher-level than that of conventional hardware architectures.

Rational's decision to support an Ada virtual machine through special hardware permits them to provide an efficiently executing Ada runtime system. The subsystem concept is supported by providing runtime support for dynamic linking/loading. Generics are implemented as shared code. Ada types, such as float, access, vector, matrix, array, and record, are directly supported. User-defined types are represented through hardware-interpreted type descriptors. Data objects are stored efficiently through the use of hardware supported bitpacking techniques. Constraint checks are performed by hardware instructions. Task context switching is implemented in hardware. (Hardware refers to physical hardware and microcode, i.e., firmware).

The R1000 hardware implements a virtual memory system, i.e., the memory system responds to virtual addresses and there is no software address translation. Both system software and application software executes in virtual memory. Virtual memory consists of a segmented virtual address space. This virtual address space manages all disk space. Physical memory can be viewed as a cache providing fast access to objects. Objects allocated from the virtual memory system can be persistent, i.e., they may outlive the creating task. There is no separate file system on the disk. Files are represented as persistent objects. In short, the R1000 hardware provides a persistent object management system.

2.2.1.2. Space Utilization

It is difficult to compare the space efficiency of the Rational Environment and conventional architectures due to the differences in data management in those architectures. The Rational Environment uses a variety of space reduction techniques in order to address the potentially space intensive approach of using Diana as primary representation. Space reduction techniques range from more compact executable code due to a high-level instruction set, efficient storage of Diana representation through hardware support, and delta techniques on program unit versions, to user controllable techniques such as bringing program units into compact form (archived state), generating compact executable views of subsystems (code views), maintaining descriptions that permit regeneration of full Diana representations for subsystem views from the delta storage of program units, and sharing subsystem versions across system variants as specified in multiple activity files. The different techniques and their effect on space are discussed in more detail in Section 3.4.

Overall, the Rational Environment's space requirements are similar to those of other Ada environments and generally more space intensive than what users are accustomed to from other languages. Notice, however, that the Rational Environment has an advantage in that the composition and dynamic linking capabilities of subsystem program libraries can result in space savings over conventional Ada environments, especially for large Ada systems, if used appropriately.

2.2.1.3. Integration into a Computing Environment

The Rational Environment is intended to be used as an interactive environment rather than a batch compilation server. Its compilation facility has not been optimized for batch-processing of Ada programs. Therefore, some of the effectiveness of the system is lost if it is configured in a workstation network as a pure compilation engine, while users edit Ada code on their personal workstation. Despite this fact, the compilation speed compares to or exceeds those of various Ada compiler systems on DEC/MicroVAX II, and Sun 3/140 workstations.

Notice, that the Rational Environment, due to its smart reprocessing techniques, will reduce (in some cases drastically) the number of lines of code to be recompiled after a change compared to conventional Ada environments. Thus, the time spent in compilation can be smaller even though a conventional Ada compiler may be compiling more lines per minute. However, smart reprocessing is most effective if users develop code on the Rational Environment rather than using it as a batch compilation engine.

One R1000 hardware system is intended to support a number of simultaneous interactive users. In general, the Rational Environment can support approximately ten persons developing new code interactively. However, one or two large-volume compilations or large- system integrations results in sluggishness of the Rational Environment—a property more annoying to the user in a highly interactive environment than in a batch compilation setting.

The Rational Environment does not support network transparent access to files, program libraries, etc. This means that copies of the same must be maintained by explicitly copying them between machines using the file transfer capability, which operates over an Ethernet connection. The result is replication of data—increasing as project size and number of installed R1000 systems increase. The Rational Environment requires subsystem specification views to be copied in full size while load views can be copied in their compact representation. The Rational Environment does not require all subsystem versions to exist as copies on all machines. Some primitives are provided in an attempt to aid the management of primary and secondary copies of program libraries (i.e., subsystem specifications and subsystem implementations). They are expected to be used following certain conventions (see Section 2.3.1.1).

The Rational Environment has been tailored to be an Ada code development and maintenance environment. Usually, other computing services such as document preparation, electronic mail, etc. already exist in a work environment. It is desirable to continue to use them rather than attempting to port them onto the Rational Environment or use the separately packaged products of the Rational Environment. This, however, requires an integration of the Rational Environment into the computing environment. Currently, such support consists of user-initiated file transfer, support for remote procedure call (RPC), and remote access through terminal emulation based on IP/TCP.

Since the R1000 supports remote access based on IP/TCP, the need for both a regular terminal or workstation screen and a special R1000 terminal can be eliminated. The R1000 uses special terminals with 60 lines and a keyboard with a large number of special function keys. Some of the function keys are augmented by pushing combinations of three other keys at the same time. The user interface of the Rational Environment has been tailored to such a terminal. The screen is divided into multiple windows. To compensate for the lack of a pointing device, the capabilities of the keyboard are used extensively. Some commands are bound to the special keypad, while others are invoked through the assignable function keys augmented by the *meta*, *control*, and *shift* keys, and some commands are invoked through modifiers on keys of the regular keyboard. Remote access is provided through an emulation package that supports VT100 type terminals and keyboards. Since VT100 is a industry standard for terminals, it is simple to set up terminals, PCs, and workstations to provide remote access. Such emulators require adjustments to the user interface by adapting the binding of commands to the keyboard, as the layout and availability of keys may differ from that of the Rational keyboard.

2.2.2. Consequences of Specialized Software

The Rational Environment software has a variety of special properties. Instead of being layered on top of an existing operating system, the Rational Environment itself implements many of the operating system functions. As a result, these components are going through a maturity process and have the potential of differing in provided functionality from that of conventional environments and operating systems. All information is stored in objects based on the Diana representation. An object management system manages all storage. Text files are a particular type of object. Objects can be named through a directory system. The characters chosen for directory and file extension separation are different from those of other file systems. A Rational-specific family of editors has a great commonality between its members, but differs from popular editors on other systems. The window system is proprietary and not compatible with other screen-oriented user interface packages (terminal-independent packages such as UNIX curses or window systems such as X windows). The Rational Environment provides its own job management system. It uses Diana as the primary program representation, introduces the concept of subsystem for partitioning large Ada systems, and applies smart processing techniques (see Section 2.1).

This specialized software raises a set of issues, which are discussed in the following sections. Some of the issues deal with the differences visible to the user of a Rational Environment, while others address the differences in the functional model available through the programmatic interface.

2.2.2.1. Learnability

The user interface of the Rational Environment is reasonably consistent through the use of an editor family for all interactions and Ada as the universal interaction language. The interaction style may be different from that to which users are accustomed—multiple window system, keyboard with many function keys, no pointing device, object-oriented user interface, availability of structural and semantic information for browsing. Once acquainted with the model, even infrequent users can use the Rational Environment without repeated learning. Since the Rational Environment will most commonly be accessed remotely from a developer's workstation or developers will use other systems to do work other than Ada programming, users will have to

face dealing with two window systems, two editors, two command languages, etc. For further discussion of the user interface the reader is referred to Section 3.3.

2.2.2.2. Maturity

All software components of the Rational Environment were developed new and, as all software does, this environment must go through a maturing process. We found that the Rational Environment and the R1000 hardware have been quite reliable. We have had no system crash. Tools could be used throughout the experiments without major failures. Some bugs were encountered, but we were able to complete the experiments through work-arounds after consultation with Rational. We have encountered some areas in which additional functionality is desirable (including operating system functionality such as incremental garbage collection of objects and network transparent object/file access).

In general, the Rational Environment is quite consistent in the provided functionality, but the consistency breaks down in some cases. This is an artifact of the Rational Environment architecture. In a conventional environment the command interpreter provides a mapping from a command language to the functions implementing a facility. This mapping encourages the command language designer to hide implementation concerns and concentrate on providing end-user functionality. In the Rational Environment, Ada is the command language and the Ada procedures implementing the facility are directly callable. In some of the more recently introduced facilities, the functionality and names of procedures reflect the particulars of the implementation. Examples are the choice of the name *Initial* for creation of a subsystem in contrast to other objects being created with *Make_<object>*, and the provision of several operations through parameterization of one implementation procedure. However, a new layer of procedures can be added by the user to provide a set of operations that disguise implementation details. This can be done easily due to the extensibility of the environment (see Section 2.1).

2.2.2.3. Porting of Software

Issues of porting of software fall into two categories: acquisition of existing Ada software into the Rational Environment and adapting the software in order to compile, link, and execute in the new environment.

The Rational Environment supports acquisition of existing Ada software in the following way. Source code can be transferred in the form of text files from other hosts via a file transfer facility. The text files can be moved into Ada program units residing in program libraries through a parsing facility. Notice that on the Rational Environment each program unit is listed under its name, while on conventional environments file names can be given independent of the program unit names, and files can be used to group multiple program units. In order to preserve this grouping information program units in different files can be placed into different directories within the same program library, or they can be placed into different subsystems. Placement into different subsystems requires additional work as export/import structures will have to be defined—information that is beneficial for partitioning large Ada systems. When Ada source code is moved to other hosts, this information would be lost as other Ada environments do not currently support the subsystem concept.

Getting Ada software into an executable state on the Rational Environment can create more problems; however, these problems are not unique to the Rational Environment. Applications often use functionality that is not defined as part of the Ada language standard in the Ada LRM. Examples are file system access, operating system access including job control and network communication, and screen-oriented display. Thus, porting involves adapting the application to its new execution environment (which is more than the Ada runtime system). This is an Ada porting problem between any two systems. The Rational Environment supports the development of code that is independent of the Rational Environment by making several subsets of the environment functionality available (through the Rational concept of *model*—see Section 3.1.5.2)—one of them being the subset specified as part of the Ada standard (i.e., calendar, text_io, sequential_io, etc.).

The Rational Environment adds to the problem by having made a choice that is in conflict with the general choice. One example is object/file naming. Most systems use "." as a file name and extension separator, while in the Rational Environment it is the directory name separator. Thus, an application opening a file named "test.dat" will run well on many systems, while on the Rational Environment it expects test to exist as a directory.

For two-dimensional screen display some systems provide a device-independent package, such as *curses*, which is available on UNIX systems as well as on DEC's VMS. Ada applications that make use of such display handling and window packages require some effort to be ported to the Rational Environment. Several choices are available:

- Change the application to use a different display and window package.
- Port or reimplement the missing packages; some display and window packages are implemented in languages other than Ada and provided with an Ada call interface.
- Use the cross development facility of the Rational Environment executing on the target only (see Host/Target support), when Ada programs have been originally developed on a host other than the Rational Environment.
- Leave the non-portable components of a system on the target and provide remote access with the remote procedure call (RPC) facility.

On the Rational Environment the package *window_io* allows applications to interface to its windowing facility. Porting software using this package to other systems requires adaptation to other window systems as the package has not been ported.

2.2.2.4. Integration of Tools

The Rational Environment currently concentrates its user support on Ada programming. Tools providing additional services must be built or ported and integrated. Some of the porting issues have already been mentioned. Once ported, a tool may have to be assimilated into the Rational Environment if uniformity with that environment is desired. This means adaptation of the user interaction model as well as the information storage and processing philosophy.

The Rational Environment user interface relies heavily on the use of function keys and on the use of Ada as command language—an approach not commonly found in tools. The Rational Environment stores information in structured form using the Diana representation and makes this struc-

ture available to the user for browsing and querying, while conventional tools tend to use a text representation, or devise their own structural representation (especially in the case of graphical tools). Users of the Rational Environment interact with objects that represent both source code and executable representation, while many other tools require the user to manage source representation and tool-processed representations through different files. The Rational Environment does processing (semantic analysis and code generation) in small pieces and is therefore able to provide feedback on semantic inconsistencies for user modifications while the user is still in context, while many tools provide the analysis capability in a form that processes the complete source entity rather than the modified portion. Thus, integration of tools to fit the Rational Environment philosophy is a non-trivial effort.

2.2.2.5. Environment Extensibility

The Rational Environment provides extensibility in two forms. The Rational Environment is extensible, because in certain functionality areas (e.g., workorder management) the user is given the ability to tailor the functionality through parameters. The Rational Environment is also extensible, because part of the system is made available through Ada interface specifications. New Ada code can be added quickly to provide additional operations, using the available functionality as building blocks.

The user of a Rational Environment is in fact expected to write such an envelope of procedures, if the provided functionality is insufficient. Such envelope procedures can reflect policies that the environment should enforce. For example, the Rational Environment provides a basic checkout mechanism, whose purpose is coordination of updates. While the basic mechanism guarantees that only one person can have a program unit checked out, it does not enforce that only one person has access rights to the checked-out unit. Such access control can be accomplished by manually setting access rights on the appropriate objects or by encoding the setting of those rights in an envelope procedure that users will call instead of the built-in procedures.

2.2.2.6. Host/Target Support

The Rational Environment provides host/target support for several machine architectures, where the Rational Environment on the R1000 hardware acts as a universal host development environment. That means code is developed and possibly tested on the Rational Environment before being cross-compiled and tested on a target system, still using the Rational Environment as the primary development and debugging vehicle. Such a development strategy has to be carefully planned by the user. One reason is the potential incompatibility of the execution environment (see Porting of Software). A second reason is the use of software components that are available to the application developer through the Rational Environment, i.e., applications can be embedded in the Rational Environment (see Section 2.1). Some of the components are specific to the Rational Environment and may not be available on the target system. The Rational Environment supports limiting the user to facilities defined in the Ada standard. A third reason is the nature of the Rational cross-compilation support. Rational provides a *Cross-Development Facility* for the 68020, MIL-STD-1750a and VAX/VMS, and a *Target Build Facility* for other architectures. The former provides a cross-compiler on the R1000, down-loading compiled code, and debugging through the debugger in the Rational Environment on the R1000. The latter provides a facility for downloading Ada source code and compilation on the target machine.

2.3. Multiple User Support

The Rational Environment supports the development of large Ada systems and to support the coordination of development by multiple teams. Its support consists of a configuration management and version control facility (CMVC) that uses the subsystem concept and a facility for managing user activities, i.e., the workorder concept. This section discusses each of these multiple-user support facilities in turn.

2.3.1. Version and Configuration Control

The CMVC facility of the Rational Environment supports composition of large systems, versioning and tracking of versions and configurations, coordination of multiple developers and parallel development activities, and integration of configuration and version control with the programming environment, especially Ada program libraries. The available support is an improvement over the commonly found approach of combining a conventional Ada compiler and program library system with a separate source code version control tool. CMVC has a two-level scheme for composing configurations of Ada systems, which is based on the concept of subsystems and was introduced in Section 2.1.2. To remind the reader, Ada systems are composed of subsystems, and subsystems are composed of Ada program units. Subsystems are beneficial for building large systems, because they provide a partitioning mechanism for systems as well as a grouping mechanism for entities larger than Ada packages, while still supporting full interface checking.

2.3.1.1. CMVC Model

Subsystems exhibit the following properties:

- Subsystems can be developed independently.
- Subsystems can be maintained in versions.
- Subsystem versions can be composed into systems.
- A subsystem version can be developed by a team in a coordinated manner.
- Subsystem versions can be developed independently and merged.
- Subsystems are the base unit for distributed development.

Each of these properties and the resulting CMVC facilities are discussed in the next paragraphs.

Different teams of programmers can work on different subsystems independently. The Rational Environment supports *independent development* of subsystems through to the separation of subsystem specification and implementation into spec views and load views respectively. Subsystem implementations are compiled against the imported spec views of other subsystems. The consistency between the specification and implementation of a subsystem is not checked until link/load time. This means that recompilation due to changes does not propagate beyond the boundary of a subsystem, i.e., the subsystem concept acts as a system-partitioning mechanism (see also Section 2.1.2 and Figure 2-1). In conventional Ada environments multiple program libraries can be used to permit independent development to occur. They either contain a complete copy of an Ada system or are structured into a program library hierarchy. In the first case, coordination has to happen at the source code level and compilation of the complete system is required. In the second case, concurrent development can occur safely only at the leaf nodes of

the program library hierarchy. The Rational Environment permits the use of libraries and links instead of subsystems; this corresponds to the use of multiple program libraries for independent development in conventional Ada environments.

Subsystems can exist in *versions*, i.e., as subsystem views. A subsystem view represents the collection of program units that make up a particular subsystem configuration. Subsystem views can be *released*, i.e., frozen, or can be *working views*. Working views are the work areas in which changes can be made to subsystems, i.e., work areas that are under the control of the Rational Environment. The implementors of one subsystem can develop, i.e., compile against a stable specification of other subsystems, while their implementation is progressing. The subsystem implementation can be tested against a released (i.e., stable) implementation of the imported subsystem if it is compatible with the specification view (determined dynamically at run-time). In conventional Ada environments multiple copies of program libraries can be made use of to maintain Ada program versions in compiled form. Usually, naming conventions are used to record version related information. Copying and moving program libraries can require recompilation on conventional Ada environments, while this is not the case on the Rational Environment when duplicating subsystem views on the same machine.

A system is composed by selecting versions of subsystem specifications and subsystem implementations. A system configuration is represented by an *activity file*, i.e., a list of particular versions of subsystems. An instance of the executable image of a system is constructed dynamically, or at user request through an explicit prelink/preload operation. System composition through activity files is illustrated in Figure 2-2. The figure shows how several system compositions can share subsystem versions. No special support is provided for the management of activity files, i.e., system configurations. They are managed in a manner that is similar to that used for text files. Activity files can be placed anywhere in the file system. Access control can be used to limit access to activity files. Multiple versions of system configurations are maintained using appropriate discipline and naming conventions on activity files. Even though subsystems are tailored to support grouping and versioning of Ada program units, activity files as well as text files can be placed into them. In this case, the versioning facility of subsystems is used. In conventional Ada environments system composition through program libraries requires some of them to be recompiled to establish the correct compilation context. System configurations are maintained through appropriate naming conventions.

A collection of sequentially released subsystem versions is referred to as a *path*. Within one path multiple people can work on producing the next released version. This is done through a collection of *working views*, also called *subpaths*. This is illustrated in Figure 2-3. Individual programmers do work in working views. In order to modify a program unit, programmers have to check the program unit out. This results in a new *generation* of the program unit to be created and made available for modification in the working view. While a program unit is checked out, it is not available to other developers for modification. Other working views refer to a previous generation of the checked-out program unit, thus, are not affected by the changes to it. Upon checkin, the new program unit generation becomes immutable and available for other team members working on the particular subsystem path. They can inquire whether their working view is up-to-date and can upgrade it to new generations with an explicit operation. Checkout of a program unit will

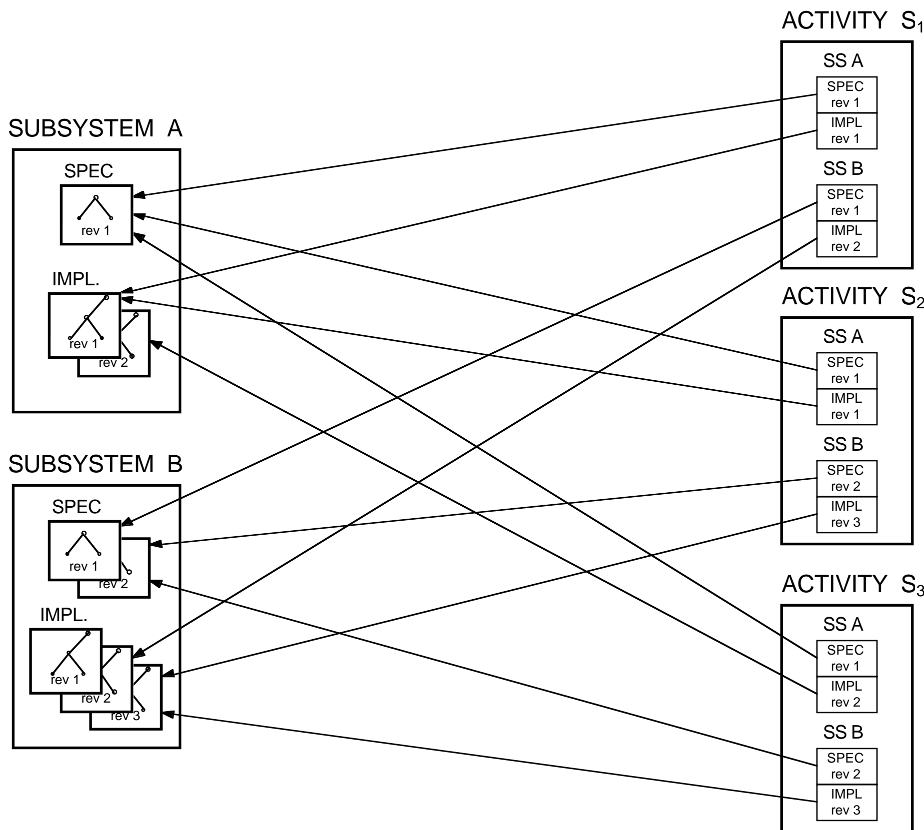

Figure 2-2: Multiple System Configurations

always result in the latest generation's being checked out regardless of what generation is referred to in the working view. One working view is used as the integrating view and its content becomes the new release of the subsystem path.

A subsystem can have multiple paths, each representing an independent progression of development for the given subsystem. This means that two teams of programmers can checkout and modify the same program units concurrently in two different paths. A subsystem starts out with one path. Additional paths are created with respect to a version in an existing path. A merge operation allows two paths to be merged, i.e., one path to be upgraded with changes made to a second path. This can be done selectively on individual program units. It is the user's responsibility to determine which subsets of modified program units are a consistent update. Both paths can continue to evolve after a merge operation. This is illustrated in Figure 2-4. A variety of development scenarios can be supported. One such scenario is that one path represents a field release and bug fixes to it, while a second path represents further development. Bugs fixes can be made to the field release without affecting development. At an appropriate time during development, these bug fixes can be merged into the development path. The Rational Environment supports this merge through an operation that takes the field release with bug fixes and the development release, determines the set of changed program units and performs the merge as much as it can automatically.

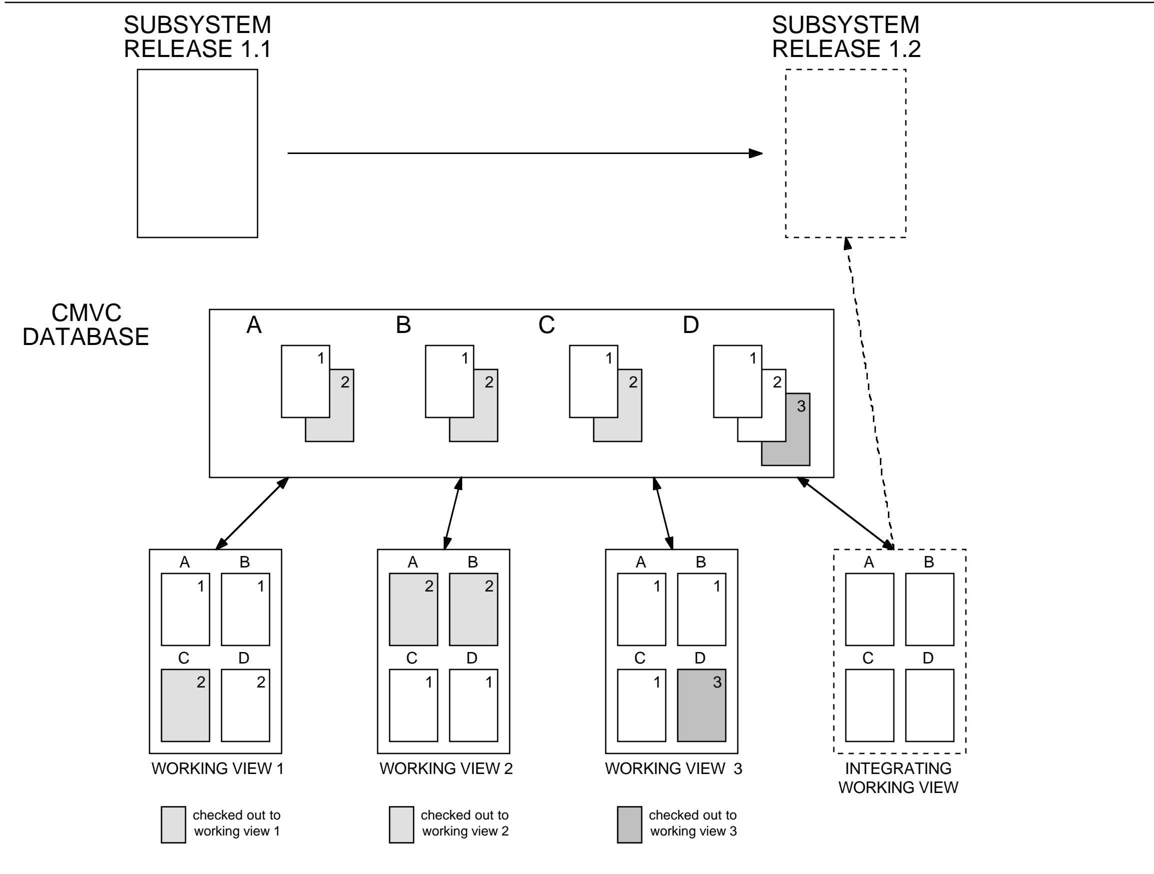

Figure 2-3: Team Development on One Subsystem Version

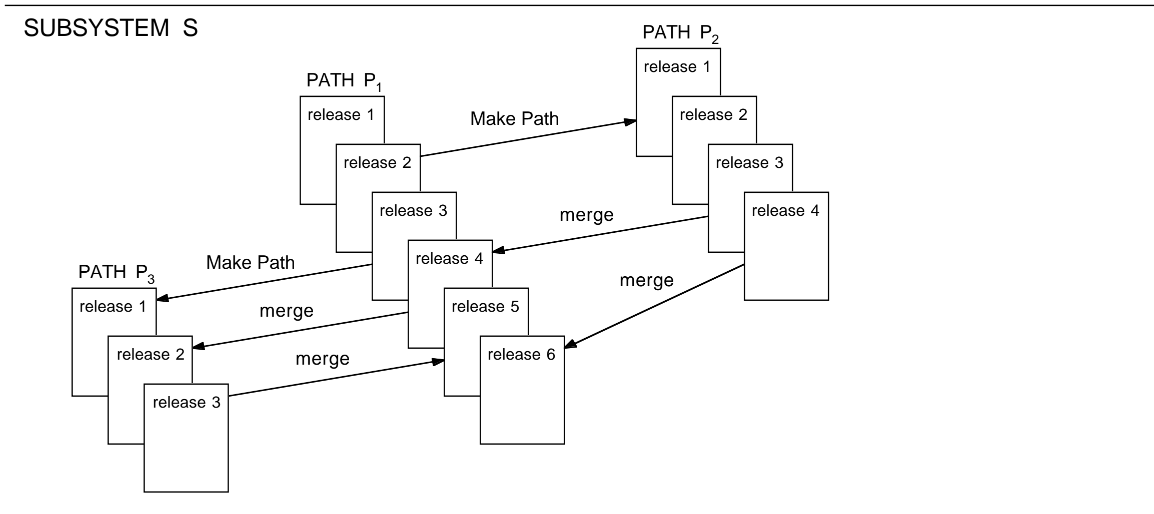

Figure 2-4: Parallel Development and Merging

While different paths of one subsystem support independent development, subpaths are intended for sequential development of releases in a path. A program unit can only be checked out into

one working view at a time. However, the path/subpath model has been augmented to allow independent development within one path at the program unit level. Individual program units in subpaths can be marked for independent development (*severed*), and program units in independent paths can be marked for coordinated updates (*joined*), i.e., sequential updates. Furthermore, program units in a subsystem view can be marked as *uncontrolled*, i.e., they are not subject to any version control operations. Conventions will have to be carefully adhered to in order to maintain consistent development.

Distributed development is supported at the subsystem level. Since the Rational Environment does not provide network transparent file and program library access, copies of a subsystem have to exist on several machines. Conventions and mechanisms for making copies primary and secondary help manage the problem of a coordinated update of a multiple copy subsystem.

To reduce space consumption due to replication and to eliminate recompilation after copying, developers can distribute copies of *code views* (i.e., a more compact representation with a stripped down Diana structure) to other machines. Debugging capabilities are limited for code views due to the stripped Diana representation. Similarly, conventions can be established for only copying new subsystem releases when the receiving site is ready to deal with them, i.e., potentially some subsystem releases may not get copied.

Conventional Ada environments are available on computer systems with network file support. However, some of the file locking mechanisms may have different semantics for accessing a remote file than that for accessing a local file. Thus, tools that coordinate multiple users may not work correctly in a distributed setting.

2.3.1.2. CMVC Implementation

Even though the CMVC support of the Rational Environment is quite successful in integrating version and configuration control with Ada program libraries and in providing large scale development and team support, the implementation of CMVC as found in the *delta0* release has some maturity problems. The concept of subsystem, its separation into specification and implementation view, and the selection of subsystem views to compose a system through activity files has already existed in the *gamma* release of the Rational Environment, and has been proven a successful concept both in-house and to Rational customers. The concepts of path, subpath, coordination of developers, configuration object, etc. are new and the implementation reflects that fact. The maturity problems fall into several categories:

- addition of complexity to the CMVC model
- potential damage through visibility of implementation layers
- provision of end-user functionality
- clarity of documentation

The following paragraphs address each of these problems.

In addition to providing parallel paths and separate subsystems for independent team development and subpaths for cooperative team development, the CMVC facility of the Rational Environment allows the user to fine-tune the environment’s control over concurrent development.

Developers can *sever* and *join* individual program units. Severing means that even within one development path (which supports coordinated cooperative development) severed program units can be worked on independently, i.e., can be checked out by more than one working view. Joining means that a joined program unit cannot be worked on independently even though parallel paths (which support independent development) may exist.

It may be desirable to have program units joined across paths. If software is developed for two targets, and target-specific and target-independent program units cannot be separated into different subsystems, joining target-independent program units results in coordination of their modification across the two paths, while severed target-specific program units can evolve independently as variants.

In addition, developers can selectively mark program units to be excluded from or included in CMVC by making them *uncontrolled* and *controlled* respectively. Being uncontrolled means that the program unit is not under the control of CMVC and users can change them freely without coordination of the changes or a record of the change history. One major use of the ability to uncontrol program units is to allow program units to be moved. Notice that uncontrolled program units may be lost when a view is rebuilt from its configuration object. The Rational Environment does not provide an easy way for users to be aware of which program units are uncontrolled and which program units are severed within a path or joined across paths. This added complexity for consistent management of configurations has to be weighed against the additional flexibility.

CMVC is implemented using lower-level mechanisms of the Rational Environment. Those mechanisms as well as intermediate functions of CMVC are available to the end user. However, they are expected not to be used or used only according to certain conventions. For example, the link mechanism is used by CMVC to implement import of subsystems. Users are not expected to, but also not prevented from, performing link operations in subsystems, with the danger of subverting the consistency of subsystem structures. Some of the functionality described in the previous paragraph (e.g., the ability to individually control and uncontrol program units) could be interpreted as being intermediate functions that should not be available to end users.

The set of functions provided by CMVC reflects its implementation in that some functions (such as *copy*) are primitives used to implement other functions. They generally do not have to be invoked by the user who wants to use the standard CMVC facilities. It is difficult for the end user to determine the relevant subset of functions and to use them because they seem to be inconsistent with the CMVC model as well as incomplete. For example, the function *Make_Subpath* is provided for the creation of subpaths (i.e., all program units are joined to enforce coordinated checkout). Execution of the function *Make_Path* using its default parameters also results in the creation of a view with all program units joined.

The reference manual for CMVC was provided to us in draft form two months after installation of the delta0 release. Other documentation on CMVC consisted of copies of tutorial slides on advanced topics. The documentation can be improved in several ways. The conceptual description of the supported CMVC models can be improved. The documented functions can be better sorted according to the usage of the end user and policy implementor. The description of func-

tions can be improved by giving a more precise description of the semantics of the function and its side effects on the environment. As conceptual descriptions have evolved in the existing documentation, terminology has changed from terms used in the implementation of the system. As a result, terminology is not always consistent and several terms describe the same concept. These comments apply only to the documentation of CMVC and workorder management (see next section). For general comments on documentation, see Section 3.2.

2.3.2. Workorder Management

CMVC supports the management of software products; in the case of the Rational Environment it is tailored to the management of Ada code. In support of the development process, the Rational Environment provides a facility of management of user activities. Users of the Rational Environment are provided with ventures, workorders, and workorder lists. These have some similarities with task templates, task description, and task list in Apollo/DSEE [6].

A venture is a template for the creation of workorders. It specifies the fields that exist in a workorder. It also enforces a predefined set of conventions (*policies*). These determine whether logging of commands and supply of comments to commands by users are enforced. The adequacy of these policies will have to be determined through practical use on a real project.

Workorders are always created with respect to a venture. Users create workorders manually by invoking the workorder editor and filling in the appropriate fields. Workorders fulfill several tasks. They serve as a description of a task that can be assigned to programmers. They can carry project management information. They maintain a log of commands being executed. Workorders are typically assigned by attaching them to workorder lists.

Workorders consist of three components: a state, user defined fields, and a log. Workorders have three states: pending (being created), in progress (active), and closed. User-defined fields are defined through a venture and are not interpreted by the Rational Environment. Logging of commands is determined by the policy set by the venture. Currently, execution of most modifying CMVC commands can get logged (commands that have workorder as a parameter). If enabled by policy a comment supplied by the user to the command is logged as well. Commands are logged in the current workorder. Similar to Apollo/DSEE, the Rational Environment does not check for any relationship between the task described in the workorder and the actions performed by the logged commands. This means that users have to switch the current workorder manually through discipline in order to get logs that reflect actions related to the task. Sessions can have a default workorder which is set to be the current workorder upon login.

Workorder lists are lists to which workorders can be attached. A workorder can be attached to more than one list. A workorder list and the attached workorders can represent tasks assigned to a person or a team. Workorder lists are created relative to a venture. Only workorders of that venture can be attached to a list. This restriction implies that a user may have several workorder lists, one for each type of workorder rather than one workorder list or workorder lists organized according to priorities instead of venture type.

Unlike Apollo/DSEE, the workorder management facility of the Rational Environment does not

provide a mechanism for monitoring changes to the development database and automatically triggering activities, e.g., the creation of new task descriptions and appending them to task lists. Furthermore, there is no mechanism for informing users of newly arrived workorders. The workorder management facility provides a simple set of capabilities. The appropriateness of the provided functionality will have to be demonstrated through practical use. As is the case with CMVC, this facility is provided for the first time in release delta0. It is intended to be a set of mechanisms. Development policies are expected to be embedded in a layer of Ada functions written by a systems manager. The documentation for workorder management is sparse.

3. Capabilities of the Rational Environment

This chapter represents an analysis of the functionality of the Rational Environment as described in Section 1.2. The functionality is determined from the commands available in the Rational Environment and from experimentation. Along with the functionality, it is necessary to examine the documentation, the user interface, and performance details. A section is devoted to each of these evaluation criteria, giving an analysis of that area. Some experimental results, based upon using the SEI evaluation method, have transcripts of the actual command sequences documented in a separate report [2].

Overall, Rational provides most of the necessary coding features for Ada program development and maintenance. It has many facilities that are in advance of those provided in a conventional Ada environment. The Rational Environment is impressive in many areas but has some deficiencies that we believe are due to its short lifetime to date as a product.

3.1. Functionality

This section discusses in detail the following areas: fundamental concepts; editing; browsing; Ada coding support pertaining to code development, compilation, error handling, execution, library management, debugging, and testing; configuration management; and operating system and system administration facilities.

The purpose of this section is to give the reader a thorough idea of the spectrum of the Rational Environment's functionality. This functionality basically includes what the user can do given the commands and the way in which the Rational Environment enforces policies or supports the user. The results are presented as lists summarizing the functionality along with accompanying text to explain features in more detail when necessary. The functionality lists only serve as a quick reference for summarizing the nature of the functionality rather than as a list of all the commands available in the Rational Environment (the Rational reference manuals provide the latter). For example, the operating system list indicates that file comparison functionality is available. It does not show that there are three commands (*compare, difference,* and *equal*) for performing comparisons since they only differ in the presentation of result. Only when there are extraordinary differences between commands, thereby providing additional capabilities, are such highlighted.

In general, the Rational Environment provides very good capabilities for code development, compilation, error handling, library management, debugging, and configuration management.

3.1.1. Objects and Common Operations

A user of the Rational Environment needs to become familiar with the notion of objects, how to select and name them, and their related operations in order to make best use of all the facilities.

3.1.1.1. Objects and Naming

In Rational terminology, objects are any code created via the Rational Environment's tools, such as an Ada unit or a selected item on the screen, such as a directory in a directory listing. Given an object, the user can perform textual or structural operations on it. Ada objects represent Ada program units. There are four states that an Ada object can attain, as described below. The state notion is easily comprehended since it relates to the state of ''compiledness'' which provides the capabilities available for that object (such as browsing or debugging). The user merely needs to press a key (<promot>) in order to move the object to a higher state (known as *promoting*) or to a lower state (known as *demoting* the object). The states for an Ada object are:

1. *archived state*: The object is not necessarily syntactically or semantically correct (since it was demoted from a source state); it is not known to other units in the system; it does not have any browsing capability since no Diana tree exists. This state minimizes storage requirements for an object, since it is stored in textual form rather than as a Diana tree. As in the other states, it can be viewed at any time without changing state; it has no dependent units; and it can be edited (it will automatically be promoted to source state at edit time).

2. *source state*: The object is not necessarily syntactically or semantically correct; it cannot be known to other units in the system. It can be changed independently of other units in the system except for its signature and its kind of unit specifics. It can be copied, deleted, removed, or renamed. Although not documented by Rational, there appear to be two substates of which the user should be aware:

    a. *pure text*—Only textual editing can be performed.

    b. *formatted text*—After the initial formatting of the text (by pressing the <format> key) the editor understands the structure, so structural as well as textual editing can take place.

3. *installed state*: The object is syntactically and semantically correct. It can be semantically referenced by other units, and it can be copied, deleted, moved, renamed or demoted if no semantic dependencies are affected. It can be changed using incremental addition and change operations.

4. *coded state*: The same as installed state plus the object has code generated for it.

A user can identify an object by selecting it or by naming it. An object is selected by moving the cursor to it and highlighting it via a combination of keys. To name objects which may be needed for parameter values, shortcut naming conventions are available. Aliases, wildcards, and substitution characters provide for conventional expression and pattern matching facilities. Special characters can be used to indicate relative context, such as "search up the hierarchy from this location" or absolute context such as "search throughout the search list and find all occurrences" or "search only through with clause" for resolving names. Object names can include attributes. Attributes are syntactically similar to Ada attributes and specify restrictions on the evaluation of names. These include class, version, and nicknames for overloaded names. Nicknames are very helpful to the user since they aid in disambiguating the reference by providing distinct names for each overloaded name.

Anonymous objects are those that represent substructures of an Ada unit. These anonymous objects are visible when a library is examined. For instance, doing incremental updates causes the Rational Environment to generate an insertion point in the code for, say, an Ada statement.

The Rational Environment creates a temporary object for this statement (which is made permanent when the change is committed). The user can select, but not name them, for editing and promotion.

3.1.1.2. Common Operations

Figure 3-1 presents the common operations available for objects. A *common operation* is that which can be applied to any kind of object in the Rational Environment. The semantics of the operation is based upon the object's type. The user merely selects an object and applies a common operation. An Ada program unit can be selected in one of two ways for application of a common operation. A program unit is selected by placing the cursor in the actual program text, or by highlighting the program unit name in the enclosing program library. Most of the common operations are bound to a key. Not all common operations apply to all kinds of objects. When an operation does not apply to an object, the Environment sometimes simply ignores the command and sometimes indicates it is not applicable.

Edit
Commit or discard changes
Create a command
Find defining occurrence or enclosing object
Find next or previous object
Complete, semanticize, or format object
Promote to a higher state or demote to a lower state
Explain in more detail
Copy or move
Insert or delete an object
Replay command history

Figure 3-1: Common Object Functionality

The common operations that are bound to the keyboard generally change the state of only one unit at a time due to the default parameter settings. The user cannot alter the default parameter settings for the common commands but can invoke appropriate procedures interactively (such as those in the compilation package) for such purposes.

3.1.2. Editing

Rational provides a comprehensive set of editing facilities. These can be divided into the following areas: general editor support, textual editing, and structural editing. They are presented in the following sections.

Unlike conventional environments, the user is always interacting with an editor. Editing functions are available at all times whether the context is editing a directory or preparing data for input/output to a program or creating a command or executing a program. The editing facilities are made even more useful with the assistance of the browsing capabilities.

In general, all editing commands apply to any object, but the actual effects of the commands are not always identical. One <edit> key initiates editing, although there are actually several type-specific editors. The type-specific editors know of the following object types: activity, Ada, command, debugger, help, job, library, link, list, search list, switch, text, venture, window, worklist, work order and cross reference list.

The Ada object editor supports both text and structure editing with a transparent transition between the two forms of editing. Other object editors support one or the other. (Rational does not document the differences.)

Structure editing takes the form of *direct* editing for Ada objects while *indirect* editing is used by the other type-specific editors supporting structure editing. The direct mode is where the cursor is positioned over a character and an operation is applied directly (such as delete the character). The indirect mode requires that the user set parameter values for a command. An example of indirect editing is editing an activity where the "insert" command needs to be invoked for changing a subsystem entry in the activity file. The differences between the modes can be annoying to the user. There is also an inconsistency in the timing of changes. Most editing involves editing the contents of a buffer (that is, the contents of a window). Changes are made by committing them via the <enter> or <promote> key as for activity files and Ada source respectively. But for links and directories, changes are immediate rather than buffered.

3.1.2.1. General Editor Support

Figure 3-2 summarizes general editor support, which consists of those operations that do not directly perform editing, but are auxiliary functions enhancing the editor's capabilities. These include operations for saving and recovering deleted text (kill-buffer management), managing macros, marking of portions of code, screen management, and window management.

Switches are provided to tailor some editor actions. Changing switches affects default actions such as overwriting characters or filling lines or defining delimiters for words. These switches are set external to the editor and affect the editor only after restarting it. The editor can also be tailored by binding any operations to the keyboard.

All keystrokes can be saved by the editor via logging them into a file. This allows the replaying of commands from the file. Note that even the timing of the commands will be replayed—this is helpful for presenting program demonstrations. This also allows stream editing for non-interactive programs.

To permanently change a source Ada object, changes must be "committed" via the <enter> key or the <promot> key, which implicitly saves and promotes the object. The editor makes sure that changes are not lost when the user logs out of the Rational Environment.

Marks can be set so that the editor records a certain position in the text or structure, such as a particular column and row location of a character. Unfortunately, these marks are not dynamically adjusted as text is changed. (A stack records all the marks.)

Screen management is a facility not commonly found in conventional environments. It allows

Retention of previous versions (programmer chooses number)
Tailoring of default editor actions via switches
Repetition of parameter for commands
Logging of keystrokes
Committing changes and prohibiting accidental logout with
uncommitted changes
Kill-Buffer manipulation:
  Recovery of deletions
  Stack of saves: copy, delete, next, previous, push, pop, rotation, swap
Macro management:
  Recording keystrokes
  Replaying macro
  Editing macro
  Binding macro to key
Marking:
  Record absolute line and character position
  Stack of marks: push, top, previous, next, copy, delete, rotate
Screen management:
  Record window organization on screen
  Screen stack: push, pop, next, previous, copy, delete, swap, rotate
  Dumping screen contents to file (includes graphics/font information)
  Repainting, clearing screen
Window management:
  Listing all windows and their attributes
  Finding window by name
  Forcing window to not be replaced
  Replacing window as soon as possible (limit of one request)
  Controlling number of windows on screen (programmer)
  Changing size and shape of windows
  Copying, joining, deleting, transposing windows
Image Management:
  Scrolling window up, down, left, right, beginning, end
  Moving to next or previous window

Figure 3-2: General Editor Functionality

users to save screens with their various windowpanes onto a stack, in effect taking a snapshot of the screen. This allows the user to set up different sets of windows for different tasks, and switch between them quickly; however, there is no consistent dependent window updating.

3.1.2.2. Text Editing

Figure 3-3 summarizes the text editing functionality. The functionality is very similar to that of conventional editors.

- Cursor movement (up, down, left and right)
- Character manipulation:
  - Case change, delete, quote, tab, transpose
  - Automatic insert or overwrite of characters
  - Automatic fill of line
  - Tab settings (modify, delete, set width of tab)
- Line manipulation:
  - Beginning, end of line
  - Next, previous line
  - Case change for line
  - Deletion and recovery of lines
  - Copying lines
  - Position: center, indent
  - Add, join line
- Word management:
  - Movement to beginning or end of a word
  - Programmer-defined delimiter for word break
  - Changing case
  - Deletion
  - Transposition
- Region (portion of text) management:
  - Beginning, end of region
  - Case change (only on first letter of word)
  - Fill, justify text
  - Create, move, copy or delete region
  - Select, deselect region
- Search facilities:
  - Regular expression pattern matching
  - Editing search string
  - Finding next or previous occurrence
  - Replacing next or previous occurrence of a string

Figure 3-3: Textual Editing Functionality

The user can add text in one of two modes, either by having newly typed characters inserted or by having newly typed characters overwrite existing text. A switch setting at editor startup time determines the mode.

A minor problem between text editing and text produced by Ada programs through Text_IO exists regarding end-of-line markers, those produced by the editor and those resulting from output of a program. This difference is documented in the Rational manuals.

3.1.2.3. Structure Editing

Figure 3-4 summarizes the structure editing functionality for Ada objects. Such editing deals with the manipulation of structures, about which the editor has some syntactic and semantic knowledge. It is the structure editing facilities, combined with the semantics-based browsing facilities and integrated with Ada code creation and smart compilation, that significantly aid the user in the Rational Environment. These are generally not present or are present in limited form in conventional environments.

Movement based on syntactic structure: child, parent, next, previous
Editing operations on Ada structures (delete, copy, move)
Ada language-sensitive search for Ada identifier or delimiter
Placement or removal of Ada comment delimiters around portion of structure
Semantics-based cursor movement (see browsing functionality)
"Smart" identifier replacement

Figure 3-4: Structural Editing Functionality

The editor provides construct completion such as "begin"..."end." This aids the user by filling in missing keywords of partially completed constructs. Unfortunately, the cursor is seldom placed at the next location of the completed construct to be filled in. The structure editor is also not able to tell the user which constructs are legal at a particular point in the program; i.e., users are expected to have knowledge of the Ada language.

A convenient and novel feature concerns source code generation. The editor provides a template of Ada code, such as matching specification/body parts or procedure signatures. Also, when requested, the editor makes default parameter settings for procedure calls, whether editing Ada source code or commands. The user can override some default template settings such as two parameters per procedure signature. These features add to the interactiveness of the Rational Environment.

The structure editor provides adequate default placement of comments. The user can, in a limited manner, override the editor's placement if necessary.

A very useful facility that integrates browsing and editing is that which globally replaces a name or expands a name. This ‘‘smart’’ identifier replacement results in the replacement of a name within the selected code. The replacement is not made in comments, nor are keywords replaced, as happens with a typical text editor's string replacement. Neither elision nor expansion of code for Ada editing exists. Such facilities could aid the user in viewing large program units by displaying them at different levels of detail in order to fit them into a window.

3.1.3. Browsing

Figure 3-5 summarizes the browsing facilities available. Browsing involves the Rational Environment's locating and displaying information pertaining to objects based on semantic information available in the Diana representation. This includes information such as the definition location and the usage location of objects. Browsing is extremely valuable, particularly in large systems since the user does not need to manually search and navigate through many structures. This applies to browsing through the static program structure during code creation as well as to navigation through error messages from semantic analysis and to browsing in the context of debugging. Also, it is invaluable for understanding foreign Ada code that is ported to the Rational Environment where a user may not be aware of the structure and relationships of objects. The browsing facilities add to the interactiveness of the Rational Environment. Since browsing is a frequent activity, all browsing functions are bound to function keys for single keystroke execution.

Find the definition of an object
Find enclosing object
Find next and previous object
Find the corresponding Ada "other" part
Find the home library
Show usages of name (given a certain scope)
Show any unused declarations (optionally including dependent units)

Figure 3-5: Browsing Functionality

Browsing is accomplished by finding an object and displaying it in the current window or a separate window. Finding the definition of an object results in the display of its content. In case of directories or program libraries it means traversing down their hierarchy, and in case of Ada objects it means the definition site (specification). Finding the enclosing object results in moving up the object hierarchy, e.g., moving from a program unit to its enclosing program library. The "find Ada other part" function provides a quick way to locate the complementary part of an Ada program unit, i.e., the specification if the body is selected and vice versa.

Two functions are provided for locating sets of objects. The "show usages" function shows all locations that use a selected object by displaying all usage sites in a window. The user can then browse through this information. The "show usages" function has parameters to specify the scope of the usage sites and the amount of semantic information to be displayed. The "show unused declarations" function highlights unused identifier definitions that can then be removed by the user if desired.

3.1.4. Ada Code Development

This section discusses facilities for producing, compiling, correcting errors, managing Ada libraries, executing Ada code, debugging, and testing. All these facilities make use of the semantics-based browsing facilities and the Ada object editor, which supports both text and structure editing. In addition, effective support is provided for source code generation, for incremental and minimal recompilation, for interactive error handling, for limiting change propagation, for dynamic linking, for execution of incomplete programs, and for debugging support for the full scope of the Ada language. The Ada code development support has reached a level of maturity such that Rational maintains all of the Rational Environment software, which amounts to over one million lines of Ada.

3.1.4.1. Code Creation

Figure 3-6 summarizes facilities for creating Ada code. These facilities are available for interactive use. They can be quite productive as they provide incremental syntax and semantic analysis, and aid the user in source code generation.

Unlike conventional Ada environments, the Rational Environment allows the user to create code directly in a program library. There is no need for the user to explicitly add a program unit to or remove it a program library. Program libraries are browsed in the same manner as are directories and Ada objects. The browser makes use of semantic information available in the program library.

Create program library
Compile and install incomplete Ada unit into library
Withdraw incomplete Ada unit from library
Create body from spec (insert template and "with" clauses)
Establish visibility to other libraries
Make a *separate* unit in-line (with appropriate syntactic changes)
  or vice versa
Automatic Ada "other part" source generator:
  Create spec from body
  Create private part (recursive for enclosed packages)
Delete or destroy units and dependents
Parse text files and create Ada objects from them
Keys:
  <format> for pretty printing, syntactic completion, and
    syntactic checking
  <complt> for semantic completion
  <semanticize> for checking static semantics of code
  <definition> for displaying the definition of an object

Figure 3-6: Code Creation Functionality

Several functions (as shown in Figure 3-6) significantly aid productivity of the user by correctly: completing portions of Ada code such as package bodies and "with" clauses; creating body stubs; creating syntactically correct but incomplete code (using placeholders); adding the procedure call template based upon the procedure's signature; and checking the syntax and static semantics of the code.

Functions are provided for turning subunits of a program unit into "separate" subunits and vice versa, alleviating unnecessary editing. Separate subunits become objects that can be explicitly named or selected and promoted and demoted independently of their parent unit as long as their state is not higher than that of their parents. Their names appear in the program library scoped within the name of the enclosing program unit.

Deletion of objects requires that bodies must be removed before specifications can be removed. Specifications cannot be removed until all (body) units dependent on them are demoted to the source state.

The Rational Environment will maintain consistency between Ada units. Whenever a unit is to be edited, the Rational Environment checks whether any dependent units need to be demoted as well. If demotion is required, the Rational Environment maintains dependency consistency by either not allowing the edit to occur and informing the user of obsolescence, or performing automatic demotion of the dependent units. The action taken by the Rational Environment depends on the command invoked by the user.

3.1.4.2. Compilation

Figure 3-7 summarizes the functionality available for compilation. An outstanding feature of the Rational Environment concerns its "smart" recompilation facilities. The Rational Environment attempts to minimize the amount of compilation and recompilation at any time via incremental compilation of upward compatible changes and by isolating code via subsystems. The Rational Environment automatically determines the compilation order.

COMPILATION
- Promote or demote a unit
- Show all errors and browse them
- List subprogram interdependencies
- Compile transitive closure of library or unit
- Compile code from an ASCII text file
- Pragma switches

MINIMAL AND SMART RECOMPILATION
- Minimum compilation for upward compatible changes
- No recompilation for comment inserts or changes
- No recompilation of (generic) bodies other than their own
- No recompilation when private parts change

Figure 3-7: Compilation Functionality

The <code all worlds> key will compile all Ada units and their transitive closure automatically. Parameters can be set by the user to control the scope of the compilation i.e., within the library or beyond, and the final state of the units. Only necessary compilations are made (i.e., no unit that is already coded will be recompiled).

Upward-compatible changes can be made to units in a subsystem without causing any recompilation of units importing the subsystem. Changes to subsystem interfaces have a global impact on the system and must be carefully managed. Changes internal to a subsystem can be made to any degree at any time with only local effect.

Closed private parts are an example of a compilation firewall within a subsystem. They are Ada "private" code parts but the Rational Environment will not recompile any dependent units if the contents of the private parts are changed since the dependents will not depend in any way on the representation of objects of these private types. Generally, conventional Ada language systems would recompile dependents. Incremental editing of private parts, along with minimal recompilation of derived types of private types, is provided. Closed private parts can speed up development tremendously. They are an example of how Rational bypasses major recompilation needs that seem inherent in the Ada language. This is added functionality that a conventional system would not have.

Incremental editing allows users to make additions, deletions, and changes to program units that have been compiled without requiring the whole unit to be demoted to source state. Only the demoted fragment has to be reprocessed when promoted. Such incremental editing saves a considerable amount of processing time that conventional systems would spend compiling. In order to optimally use these features, though, the user must cooperate with the Rational Environment by indicating the appropriate substructure of a program unit to be demoted for editing. The Rational Environment does not analyze demoted program units to determine whether reprocessing can be limited to substructures or avoided. Since this an important aspect of the Rational Environment that distinguishes it from conventional environments, the incremental editing capabilities should be presented in the Rational documentation and training courses in a systematic manner.

Comments can be inserted, deleted, and edited in coded Ada objects without any recompilation being required. Bodies, including bodies of generics, can be edited with no recompilation required other than their own. Generic specifications cannot be incrementally changed. When a body is brought to the installed state, all changes that do not affect existing dependencies can be made. This includes inserting context clauses, declarations, and statements and editing and deleting statements. Changes that affect existing dependencies, such as editing declarations or context clauses, require that the body be brought to the source state.

The following itemized list attempts to summarize the different incremental editing capabilities available when program units are either in installed or coded state. If program units are in coded state and the user wants to perform an incremental edit operation listed under installed units, the user is required to first explicitly demote the unit to the installed state.

- Editing installed units allows inserting incrementally:
    - New declarations that are upwardly compatible; existing declarations without dependents can be deleted or demoted and reinstalled.
    - New statements; existing statements can be deleted, demoted, edited, and reinstalled.
    - New context clause items that are upwardly compatible; existing context clauses without dependents can be deleted, demoted, edited, reinstalled and recoded.
    - New comments on lines by themselves; standalone comments can be deleted or demoted, edited and reinstalled.
- Editing coded units includes:
    - in a library unit specification, new declarations that are upwardly compatible can be inserted; existing declarations with no dependents can deleted, or edited and reinserted; due to Ada semantics for elaborations, the corresponding body is demoted to installed
    - new context clauses can be inserted if upwardly compatible and if the units named in the context clause are coded; existing context clauses with no dependents can be deleted or edited and reinserted; there is automatic demotion of any dependent main programs
    - can insert, delete, edit comments

The documentation does not state whether the compiler optimizes code and whether the optimizations can be enabled or disabled.

The compiler seems to be well tested, in that no bugs were found. The Rational compiler has, in fact, found several bugs with the Ada Compiler Evaluation Capability (ACEC) test suite. Only two compiler limitations were found when running all of the original Institute for Defense Analysis (IDA) ACEC suite: 15 static nestings and 256 records in a field.

A helpful option available to the user is to request the Rational Environment to indicate the amount of effort required to compile a system rather than actually doing it. Although what the cost value returned really means is not clear, useful cross-referencing information, such as missing units in the transitive closure, is provided.

3.1.4.3. Error Handling

Figure 3-8 summarizes the error-handling functionality available. Error handling involves error reporting, diagnostic display, and interactive support for correcting the errors. The browsing facilities assist by guiding the user through the erroneous points. Turnaround time for fixing errors is considerably shorter than that in conventional environments, since it is very easy to recompile after fixing errors.

Errors are indicated by underlined source code. With a keystroke, the user can request information about an error. By default, brief error diagnostics are initially displayed. The diagnostic levels are generally two levels of explanation deep. These levels are useful for the experienced user who may not need detailed explanation of an error versus the novice user who does.

Error Reporting:
  Interactive error highlighting
  Levels of detail on error messages
  Tailoring display of kinds of errors
Error Correction:
  Show next or previous error
  Common operations and editing and browsing

Figure 3-8: Error Handling Functionality

Browsing is very valuable during error repair when errors need to be fixed. A typical scenario follows. The user can use the <next item> key to locate the next error location. Using the browser, the user can examine the relevant part of the program to determine the location for an error correction. The Rational Environment encourages the user to fix an error and check syntax by pressing the <format> key. Note that the <next underline> key only locates errors and the <next prompt> key locates placeholders while the <next item> key locates both.

The user can tailor the error display for interactive or logged display. The user can request that certain errors be flagged in a particular way that is different from the default display in the error log. Also, the user can request program continuation despite certain kinds of errors, such as warnings or non-fatal errors.

Users requiring hard-copy compilation listings and error reports will get a fairly primitive listing that shows only the line number in which the error occurred, together with the error message.

3.1.4.4. Execution

Figure 3-9 summarizes the execution functionality.

  Suspend/resume/terminate execution
  Background and foreground job execution
  Program completion status indication
  Execution of incomplete programs (stubs or missing bodies)
  Dynamic/runtime linking (prelinking can be forced)

Figure 3-9: Execution Functionality

Programs can be suspended and resumed. They can also be terminated. Changes made to the program while the program is suspended are not reflected in the execution until execution is terminated and restarted. Post-mortem analysis (such as those done through memory dumps) is not available. The user is expected to obtain all information via interactive debugging or logging of results.

Unlike conventional Ada environments, the Rational Environment allows incomplete programs to be executed. Programs can be incomplete in the sense that body units are just stubs or that body units contain insertion points at the statement level. Code can be executed without all the Ada units being in the coded state. During execution, the program will execute until it encounters a program unit without code or an insertion point (i.e., a placeholder). Programs with missing bodies will get an error indication when attempting to run them. Programs containing stubs will execute up to the time that a call is made to an incomplete construct, at which time the system raises a Program_Error exception. Note that the Rational Environment does indicate which bodies do not exist until the program commences execution.

When a main program runs, the object code is linked at runtime. Prelinking can be forced by using the pragma Main in the main procedure, although prelinking will cause problems when subsystems are used, such as making releases of subsystems (this is not documented by Rational but is a problem known to Rational).

3.1.4.5. Library Management

Figure 3-10 summarizes the functionality of Ada program libraries in the Rational Environment.

LIBRARY MANAGEMENT
- Tailor library contents display through switches
- Create library
- Delete library
- Display objects in library
- Display dependencies (see browsing functionality)
- Copy or rename object
- Freeze and unfreeze object
- Remove object or clear program library
- Compile all units in library/libraries
- Compact library
- Move to another library
- Find pathname

LINKS
- Create links
- Delete and expunge links
- Copy or change links
- Display links

Figure 3-10: Library Management Functionality

The Rational Environment implements Ada library semantics. The browser allows users to navigate through program libraries in the same way as directories. The program units contained in a program library can be displayed with several levels of detail.

In Rational's terminology, an Ada library is a *world*. It contains Ada program units, i.e., Ada objects. Libraries can be shared via links. Ada units that are named in a "with" clause of another Ada unit either must be in the enclosing local context or must be visible via a link to another program library in the enclosing world. Different worlds can reside on different disk volumes. Worlds can be moved to a different disk volume. Moving worlds requires compilation of the closure of the units in the world.

Session switches are used to tailor the display of library information. The switches indicate which attributes of the objects are to be shown such as size. Changes to the switch settings only takes effect after the user reattaches to a session.

There are functions for freezing and unfreezing objects. When frozen program units are prevented from being modified. The user must have owner access to the world in which the objects are located. The main purpose of these functions is to provide primitives for the implementation of versioned subsystems (see Section 3.1.5.3).

Rational has integrated the Ada program library with version and configuration management facilities through subsystems. The implementation of subsystems makes use of some library management functions such as link operations. Users who make use of the version and configuration control facility are expected not to use links directly.

3.1.4.6. Debugging

Figure 3-11 summarizes the debugging functionality. Overall, the Rational Environment provides a debugger that supports the full scope of the Ada language, including tasks, exceptions, and generics. The debugger is integrated with the semantics-based browsing capabilities and the support for Ada code development. Users can transparently and quickly move between browsing, code development, and debugging.

The debugger is fairly easy to learn due to the available documentation, the nature of the interface, and on-line help facilities. The user is aided by the binding of commonly-used functions to the keyboard which minimizes keystrokes during debugging. The command language is that of Ada. Most debugger commands involve two keystrokes: one keystroke to select an object and another to execute a command.

Programs can be debugged without special preparation (such as compilation with a debug switch). Instead of pressing the <promote> key for normal execution of the program, the user presses <meta> <promote>. The debugger can also be called directly from the program. Furthermore, debugger command scripts can be prepared in files and read by the debugger.

The debugger runs in a separate process from the application program process. The first invocation of the debugger has a certain startup cost for this process. Unless requested by the user, the debugger process remains intact while the user is logged on. The same debugger process can be used to debug different programs. While the debugger process is intact, it maintains a record of all breakpoints and tracepoints set by the user. The user can re-enable them when the execution of the application program is restarted.

SET/RESET BREAKPOINTS/TRACEPOINTS ON
- Program unit entry/exit
- Exception
- Statement
- Breakpoint (only) on nth iteration of a loop
- Declarations
- Overloaded functions
- Rendezvous
- Breakpoint retention across debug sessions

CONTROL EXECUTION PATH
- Jump a number statements
- Enter a specified subprogram
- Exit the current subprogram
- Control over tasking execution
- Control over the scope of tracing
- Control over exception propagation

QUERY PROGRAM STATE
- Display source code
- Display breakpoints and tracepoints
- Display runtime stack
- Display history of commands
- Display task status
- Display standard and non-standard data types
- Display of source code at breakpoints and during stepwise execution

MODIFY PROGRAM STATE
- Modify variable values
- Change name resolution context

NUMERIC CONVERSION

Figure 3-11: Debugging Functionality

Breakpoints are points in the code where execution is stopped so that the user can examine specific details. Tracepoints are similar to breakpoints except that execution is not halted—the debugger merely displays that a point in the code has been passed. Breakpoints and tracepoints can be set or reset at any time. They can be set upon various events, such as entry or exit of a program unit, an exception, or any statement. The user has many features for controlling the execution of tasks, exceptions, and rendezvous. Execution of groups of tasks can be controlled, such as halting various tasks while allowing other tasks to proceed. The user can specify the granularity for enabling tracepoints and breakpoints down to an Ada scope level and select tasks in which they should be applied. Similarly, the granularity of single-stepping can be specified by the user.

The debugger automatically displays source code. It can display information about breakpoints, tracepoints, tasks, and runtime stacks and can keep a log representing the history of all statements executed. Program variables can be modified and debugging continued with the new value. The debugger in interaction with the browser can guide the user through executing and subsequently editing code. The browser can highlight the definition and usage points of objects and effectively guide the user through all the points in the source code requiring editing. Hence, the debugger/browser combination can considerably shorten the time needed for the debug/edit/compile/link/debug cycle. Browsing features available include show: definition and usage point of objects; enclosing object; Ada ''other part''; and errors. Changes made to the program through editing are not reflected in the executable image that is currently being debugged. The user must restart the execution of the program.

The debugger aids the user in displaying values. For standard Ada types, the debugger displays the appropriate scalar or structure or pointer information. The user can include special display routines in the program for non-standard types whose names can be registered with the debugger. As a result, the debugger will use those display routines for the specified types. These routines are recognized on a per program basis as they are part of the program.

There are comprehensive naming conventions allowing the debugger to interpret names within various contexts ranging from current working context, root of library system, immediately enclosing object or library or world, a specific task, or a component. Ada naming conventions are recognized by the debugger. Naming rules are extended to allow the user to disambiguate overloaded names and reference specific declarations and statements via debugger-generated numbers.

Some commands frequently used during debugging can be invoked without qualification from any window, whereas others require the cursor to be in the debugger window to execute without qualification. Some confusion, as to which case applies, could arise. This is due to the fact that functions provided by the debugger package of the Rational Environment are recognized without qualification only in a debugger window, while browsing and Ada editing commands are recognized in windows displaying program units as well. Also, browsing can cause the debugger window to disappear from the screen. However, it can be locked on the screen, can be located by selecting it from the window directory, or can be redisplayed in a window configuration using the screen management facility.

There are some restrictions in using the debugger:

- A user can only have one debug session active at any time.
- Post-mortem analysis is not supported in the sense that the debugger can examine a program whose execution has failed, but was not executed with the debugger active. Programs must be started from the debugger to have debugging available.
- No conditional breakpoints/tracepoints are supported. There are some limitations as to where breakpoints can be set for objects—none can be set on variables of access or task types, constants, *in* parameters, discriminants of variant records, and loop iteration variables.
- Tracepoints will not display the values of variables. Rather, messages indicate that a trace event has happened.

- Line numbering of the source is not automatically displayed; yet debugging display information is given referencing line numbers in many cases.
- Debugging capabilities are limited for code views due to minimal semantic information in this compact, executable representation of subsystems.

3.1.4.7. Testing

There are no specific tools for testing such as for performing unit and regression testing, or for performing analysis, such as identifying unreachable statements, control flow, stress testing based on program structure, test data coverage, and statistics gathering. However, the Diana representation is accessible from programs and such auxiliary tools can even be written by the Rational customer.

It is quite easy for the user to do interactive unit testing since any Ada subprogram can be run directly from the command window as a function. Hence, no test driver is needed. For regression testing and large-volume unit testing, the user can easily write a test driver as an Ada command procedure. The user must use the file utilities, such as comparison, for performing any test analysis tools. There are no specific tools for setting up a test harness that records expected output data and compares that with tests results, as with regression testing, or provides the ability to run tests in a particular order.

The facilities for executing incomplete code (as described in Section 3.1.4.4) make for quick interactive testing of program fragments.

3.1.5. Configuration Management

This section analyzes the configuration management facilities that the Rational Environment provides. The basic model is described in Section 2.3. The facilities can be divided into several areas: partitioning of systems, cooperative team code development with version tracking and releasing, coordination of changes across separate development paths, system-level composition, and database maintenance. These areas are discussed in the subsections below. We found problems in dealing with the configuration management facilities, mainly because of the newness of the implementation, the lack of environment-enforced policies, and because of the complexity of the issues.

In general, Rational's configuration management facilities represent an advancement in conventional version control and large-scale programming-in-the-large support. These facilities, though, are hampered by the lack of a production-quality user interface model that makes transparent the underlying primitives used to implement the configuration management facilities. We also found that a small portion of the facilities (such as Build and Destroy commands) did not work as documented, which indicates bugs in the implementation.

Note that there is considerable terminology with which the user must become familiar to use Rational's configuration management. This is not an easy task since related documentation is scattered throughout the manuals, and some of the most important explanations are only documented in the Rational *Training Notes*. In some circumstances we have tried to use commonly used terms rather than Rational-specific terms in order to reduce the confusion of the reader.

3.1.5.1. Partitioning Concepts

For large-scale software development it is necessary to partition the software into manageable portions for development by teams of users and for testing. In order to understand the facilities that the Rational Environment provides, it is necessary to understand several concepts. These concepts are basically an extension of the Ada package specification and implementation notion for supporting partitioning and interface checking. The notions themselves are not very complex, but the user interface of their implementation tends to frustrate the user due to the visibility of implementation layers in the user interface.

Rational allows the partitioning (or grouping) of objects into subsystems. (The subsystem notion is described in Section 2.1.2.4.) The following section discusses the contents of a subsystem in more detail. A subsystem is actually composed of a collection of views. A view is both a configuration of objects in that it represents a set of a particular generation of each object in the subsystem and an Ada program library in that Ada semantic consistency among the specified generations is enforced. Basically, the view represents the integration of the Ada library notion along with configuration management notions. The different variants of views can be summarized as follows.

There are several kinds of views:

- *Spec View*: A spec view is similar to an Ada package specification. It defines the set of implementation units that are potentially available, or visible, to units in other subsystems. Spec views have import/export lists representing the interface to other subsystems. Hence, any changes to spec views will require recompilation of corresponding load views, as well as views that import the spec view.
- *Load View*: A load view defines a single instance of a subsystem. It contains a full implementation of the program component that is encapsulated in the subsystem and includes all the spec view interface code. The load view basically corresponds to an Ada unit's body. There can be multiple load views per spec view which allows recombinant testing of code—this is how the Rational Environment provides for multiple Ada bodies per Ada specification. Units in a load view can be changed without requiring recompilation of any other views, provided that the view remains compatible with the spec view that defines its exports. Compatibility allows a load view to differ in certain ways from the spec view that represents it, such as with private parts.
- *Combined View*: A combined view is similar in contents to a load view but is a special case view. Unlike other views, it allows non-hierarchical importing, such as circular imports (and is also useful if the subsystem contains non-R1000 targets that require exporting of generics). A combined view has limitations that other views do not have, such as lack of compilation firewalls. Combined views can only contain combined views. It is recommended that use of combined views be restricted. (Information about combined views is not documented by Rational except in the *Training Notes*.)
- *Code View*: A code view is the same as a load view but is a special case view in that it minimizes the space used by the views since it does not include the Diana tree. Only state and history information, along with executable code, are incorporated. As a result, there are limitations such as neither browsing nor editing and only limited debugging are available for a code view.

These views can exist in one of two forms:

- *Release* view: A released view is "frozen," thereby representing a "baseline" release of the software. It is logically immutable in that it cannot be modified (although the Rational Environment does provide for such). Release views can be stored in two different ways that can affect the resultant functionality available:
    1. *Full-view release*: All information such as Diana representation, state, history, and generation indications are stored.
    2. *Configuration-only release*: Only the configuration information representing state, history, and generation indications are stored. This is the minimum information required in order to recreate the the source code (upon special request). Displaying of code, browsing, editing, or debugging cannot be performed.
- *Working* view: A working view, in contrast to a released view, is used for ongoing development work. There are no limitations on editing, browsing, or debugging.

3.1.5.2. Partitioning Management

Figure 3-12 summarizes the functionality available for managing views. The Rational Environment aids the user in: creating subsystems and views that can be based upon copying from other views and adjusting import and target information, deleting views or subsystems, managing the interface among the views, and displaying pertinent information about the views.

Create a subsystem or view (based upon a certain model)
Destroy a subsystem or view
Import/export a view into/from another view
Add or change imports
Remove unused imports from a view
Replace a model
Display of subsystem/view hierarchy

Figure 3-12: Partitioning Management Functionality

The Rational Environment uses a "template" known as the *model* upon which it creates a subsystem. This model has default information such as switch settings, initial links to commands and tools, access control lists, a target machine and parameters that indicate how the Rational Environment should generate names and version numbers for views. At any time the user can change the model, and the Rational Environment will use that model for subsequent operations on the view.

Creating a view involves copying information from another view. This enables the Rational Environment to "re-use" information in a sense, and provides a starting point for the user.

When a view (or subsystem) is created, predefined directories are generated. The user must become familiar with these and understand how the Rational Environment uses such. Command

descriptions in the manuals do not discuss these—only the Key Concepts section does. A view's four predefined directories are:

1. *Exports* in which users can create export restrictions files
2. *Imports* in which users can create import restrictions files
3. *State* which contains status information is kept
4. *Units* in which the Ada program units are created and edited

It is suggested by Rational that none of the directories will require frequent changing. The documentation does not make clear what changes can be made and how they are made.

Destroying a view destroys all related substructures. The user can request the Rational Environment to demote automatically importers of the view and have their imports automatically adjusted, such as removing imports. It is the user's responsibility to then promote those affected subsystems.

Destroying subsystems requires having all views first destroyed, whereas a view destruction automatically gets rid of subdirectory structure and units. The user can request that the configuration object be saved. Leaving the configuration object is a way of saving space (compared to leaving the Diana representation) without losing information. Otherwise, no reconstruction is possible.

Importing a view can result in adding new imports or changing existing imports (depending on a parameter setting). The latter has the effect of updating the imports of a view to reflect those of another view. The user can also request that the closure of views be imported. Through additional parameters, the user requests any necessary demotion and compilation to be done automatically. The Rational Environment does consistency checking to ensure that no view can directly or indirectly import more than one view from the same subsystem. The Rational Environment prohibits circular imports, unless combined views are involved.

The automatic copying and adjusting of information, such as that pertaining to links when views are created from other views, is extremely convenient for the user as it eliminates some of the "chores" related to copying code. Also, the automatic generation of unique names is convenient. For user-generated names, though, the Rational documentation cautions the user to be aware of possible confusion for the Rational Environment with using underscores in prefixes for these names (in case the Rational Environment needs to subsequently generate new versions of views). The user can specify the number of levels for automatic name generation for release and spec views. There are conventions that the Rational Environment uses for generating name suffixes. The user can indicate a name prefix at creation time.

For the special-case code view, since there is no Diana representation for the view, a special command for browsing is available (Display_code_view) that displays a mapping of the code segments and exceptions from the code view to the original view.

3.1.5.3. Controlled Configurations

Functionality for controlling versions of program units and their version development, and for views as configurations of program units is summarized in Figure 3-13.

Designate program units as controlled
Designate program units as uncontrolled
Create configuration object
Reconstruct view given its configuration object
Create a released view thereby freezing all program units
Query version history information
Append note to objects in database

Figure 3-13: Controlled Configurations Functionality

Program units that are part of a subsystem can be placed under version control. This is done by making program units *controlled*. In that case, a version history (*generations* in Rational terminology) of program units are kept. These program units are kept in archived state, and a delta technique is applied to store only differences. For controlled objects the Rational Environment coordinates modification by requiring them to be checked out (see Section 3.1.5.4). Unfortunately, newly created program units are not controlled—users have to do so explicitly. At any time the user can choose to make individual program units *uncontrolled* again, thereby suspending the recording of version history and coordination of modifications. This operation should be used carefully because it has side effects and program units in uncontrolled state are not easily detectable.

Subsystem views are program libraries that contain configurations of versioned program units. The configuration information is described in *configuration objects*. This is illustrated in Figure 3-14. A configuration object contains a list of program unit names and the selected version, as well as all the relevant state information (such as switch values, import/export lists and model name) necessary to (re)construct a subsystem view. If disk space is a concern, views can be destroyed and reconstructed when needed from the less space-consuming configuration object. Notice, however, that only controlled objects are recorded in a configuration object. Thus, only controlled program units are placed into the reconstructed view, even though the original view may have contained uncontrolled program units.

Subsystem views can be turned into release views, i.e., all program units in a view become frozen. Once frozen, no changes can be made to the view. The release function can compile program units if specified in a parameter. Note that views are frozen even if compilation fails. As a result, the units can be frozen in an "installed" state and cannot be promoted to ‘‘coded.’’ In this case the view will need to be unfrozen and fixed (only the owner can do so). Unfreezing should be used cautiously since it makes immutable code modifiable.

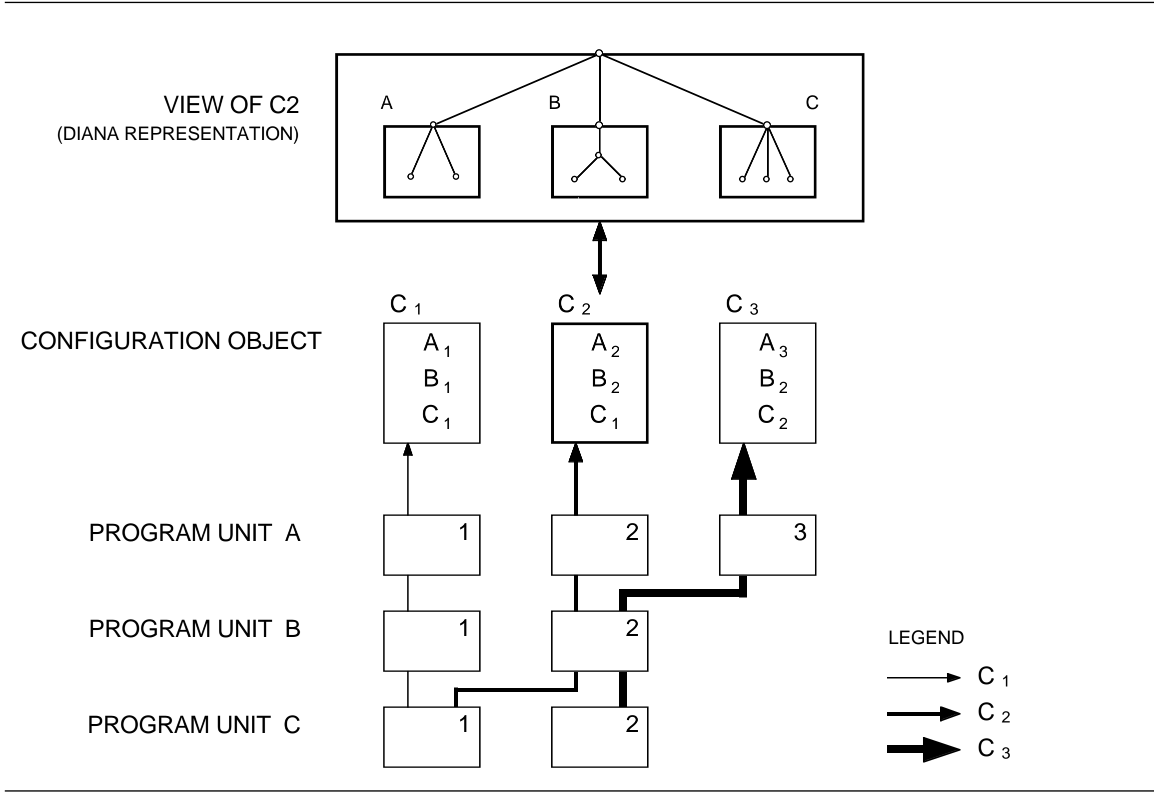

Figure 3-14: Versioning and Configuration in a Subsystem

The Rational Environment automatically constructs the names of released views and configuration objects, whereas code view names are entirely user-defined. A released view name has two components: a pathname prefix such as "rev1" and a set of release level numbers such as "_0_1." The prefix is taken from the basename of the working view, although it can be distinguished from the working view by a parameter in the Release command. The minor release level number is automatically updated by the Rational Environment unless the user requests that a different level be incremented (via a parameter in the Release command). The number of release levels that can be incremented is determined by the user-created file *Levels* in the subsystem's model.

There are commands for displaying typical database information, such as change history. There are no specific browsing facilities for navigating development history relationships of views, such as the view a particular view originated from.

3.1.5.4. Cooperative Code Development

Figure 3-15 summarizes the version tracking and product release capabilities for supporting sequential releases of subsystems by a team of developers.

Set up a cooperative development scenario (subpath)
Check-out objects (creating a new generation) or abandon check-out
Check-in objects
Update working view with changes by others
Revert objects to a specified "generation"
Query database status about objects

Figure 3-15: Cooperative Code Development Functionality

A team of users work cooperatively in a collection of working views, also referred to as "subpaths," each of them using a separate one. A set of cooperating working views is created via a series of *Make_Subpath* commands. The user specifies a name extension that the Rational Environment uses for constructing the names of the views. Naming conventions (related to paths and subpaths) can be bypassed using the basic command for creating a view (*CMVC.Copy*) rather than any path-related command.

Tracking and controlling versions of objects is similar to conventional check-in/check-out facilities. The Rational Environment will record the change history as long as the user designates an object as a controlled object. Once controlled, a reservation token is associated with the object. The user can supply a reservation token name or the Rational Environment will generate one, although why the name is needed is not evident in the documentation.

A user must check out a program unit into a working view in order for it to be modifiable. After editing and compiling the unit, the user checks it back in, thus making it available to other working views in a path. These working views can be updated with the just-released program unit by

explicitly invoking a command (Accept_Changes). The just-released unit will be copied to the view, but no recompilation will be needed, only recoding.

Note that the program unit is checked out to a view and not a particular user, which means anyone with access control to the view can proceed with editing. The default setting for access control is read and write access by everyone. Multiple program units can be checked out with a single command by using wildcards when naming the object. However, if one of the objects is already checked out, the command will abort. An "expected check-in time" can be indicated by the user at check-out time. Issuing the check-out command will result in the latest version of a program unit to be placed into the view. If the existing program unit version is out-of-date, dependent program units are demoted to source state unless the user requests automatic promotion to a particular state in a "goal" parameter.

Controlled objects can be browsed and compiled while checked-in. The documentation cautions the user from checking in an object in certain forms, such as one that contains insertion points—these points must be removed before check-in time; otherwise, compilation will subsequently not work. An Ada unit kind such as a procedure cannot be changed to another kind of unit while it is controlled. An Ada unit should only be controlled after its Ada name appears in the directory, not while it has its temporary name. In case these conventions are not followed, the user can recover by making the objects uncontrolled.

To delete and move a controlled object, first make it uncontrolled and then controlled after the move. The history information up to the move remains under the original object name. The object in its new location is made controlled, and its generation number starts from the beginning. In effect, the Rational Environment does not record the move in the history information.

With the revert function a user can make a particular program unit version the latest version. The user can also temporarily select an earlier version of a program unit—however, unless the revert function is used, check-out will overwrite the selected version by the latest version. A convenient parameter feature is the use of a negative number to indicate how many versions back to go. Thus, the user does not have to remember version numbers.

3.1.5.5. Independent Code Development

Figure 3-16 summarizes the functionality for coordination of changes between independent development paths with teams of developers. These facilities can be used in conjunction with the cooperative development ones.

Make a separate development path
Join (controlled) objects for cooperative development
Sever objects for independent development
Merge changes of controlled, joined objects

Figure 3-16: Parallel Code Development Functionality

A user creates an independent development path within a subsystem with the Make_Path function. Different paths of a subsystem use the same reservation tokens, i.e., program units cannot be checked out in different paths independently unless the units are severed. In effect, creation of a new path results in a development branch.

Coordination of modification to program units can be controlled individually. Program units can be *joined* across several views, which means that only one view can have the program unit checked out (i.e., the reservation token is shared). Program units can also be severed with respect to sets of views, which means that the program unit can be checked out to different sets of views independently. In effect, a development branch is created at the program unit level, i.e., program units can be updated independently in different paths, while views in the same path have all program units joined, i.e., update to program units is coordinated and serialized. Generally, views in different paths have all program units severed. If necessary, however, users can sever individual program units within a subpath to allow independent development, and they can join program units across paths to sequentialize their modifications even across otherwise independent development activities.

The merge facility (command Merge_Changes) takes two instances of a program unit and merges them into one, noting all the differences. These differences are noted in a report as well as highlighting them in the resultant (merged) object. The user may need to edit the resultant object to resolve any conflicts, or the user can strip away the conflict highlighters with one command. The comparison is merely a textual one, so any character difference is recognized, such as comment changes.

3.1.5.6. System Composition

Figure 3-17 summarizes the functionality for system-level composition of subsystems and their execution. This composition basically permits the user to compose a system from subsystems containing compatible sets of views. This information is recorded in a table structure. It is a very convenient device for allowing the user to test compositions of different subsystem versions.

Activity files represent system configuration as sets of views
Create, add, change, copy, delete an activity or its entries
Inherits changes from other activity files
Check compatibility between spec and load views
Check compatibility between views in activities
Write an activity into a file
Set the current default activity for a job
Browse activity items

Figure 3-17: System-Level Composition Functionality

An activity file is an object type and has a type-specific editor. An activity file records a set of subsystems making up a system composition, and indicates for each subsystem specification and

implementation which version is to be used. Each job has a current activity file. This activity file is used for dynamic linking and loading when the user invokes a program for execution. The user can check for compatibility between spec and load view pairs, and for consistent use of spec view versions across entries in the activity table.

An activity cannot contain a reference to another activity. A program cannot be executed by pointing to an activity file and applying the execute command. (Instead, users select the main program unit in the appropriate version of a subsystem.) An activity file can be created as a copy of another activity file, or relative to another activity file. In the second case, changes in the entries of the original activity file are visible to the other activity file. This can be used to maintain system compositions that are variants of a baseline system composition; upgrades to the baseline are immediately visible to the variants.

There is no version control for activity files. The user must explicitly arrange for such.

3.1.5.7. Database Maintenance

Figure 3-18 summarizes the functionality for maintaining configuration management databases. This involves supporting subsystems and development paths across multiple Rational machines and managing the consistency of the database information. These facilities seem rather complex and are not easy to understand, given the documentation.

Make secondary subsystem into a primary one or vice versa
Update a secondary database
Display subsystem's database
Check consistency of views between database and library system
Destroy or expunge database
Repair a database

Figure 3-18: Database Maintenance Functionality

Database maintenance is complex and requires an expert because this involves knowing the intricacies of configuration management objects. We expect only an expert to use such facilities.
There is no network file system across multiple Rational machines, so file transfer is used for moving subsystems to other Rational machines. When development occurs on multiple R1000s, a copy of each subsystem needs to reside on each machine so that the entire application can be executed. Two types of subsystems, primary and secondary, exist for different purposes. The primary subsystem represents the main development version, whereas the secondary ones are frozen and can only be used for local execution and testing. Each secondary subsystem is associated with exactly one primary subsystem via a unique identification number. A secondary subsystem has its own database that can be updated. Making a subsystem into a primary one effectively severs it from any other subsystem. A subsystem is automatically made primary if its database is deleted.

Moving the property of being the primary subsystem requires that conventions must be followed to avoid inconsistencies. It is not entirely clear from the documentation how these operations should be used and what their ramifications are.

Since both the database and library system record information about views, inconsistencies can occur. The Rational Environment provides functions that can check consistency of information between the two and make attempts to repair obvious discrepancies. Comparisons are textual so, for instance, different switch settings will be reported as a conflict due to textual comparison. Any missing imported views are reported. References to deleted models are removed. Information can become inconsistent due to the user's deleting information without following proper conventions. Repairs involve the creation of objects found missing. The documentation discusses the repairs that the Rational Environment attempts. We were unable to test out the database maintenance facilities since we only had one Rational machine.

Before destroying (or repairing) a database, the user can request the Rational Environment to display the "effort only" required (i.e., which units will become demoted as a side effect). When the database is destroyed, all compiled units in the subsystem are demoted to source state and all code views are deleted. Rational documents the situations in which it is useful for destroying the database. For repairing a database, the user can indicate the extent of repair, that is, whether to delete and rebuild the entire database or just deal with missing entries.

3.1.6. Operating System and System Administration Features

The Rational Environment provides some functionality that could be classified as similar to that of conventional operating systems and pertaining to system administration. This includes: tailoring of display and log information and terminal aspects; file handling and logging facilities; job control and program management; and administration relating to account management and archive facilities. These are discussed in the following sections.

3.1.6.1. Tailoring of Display and Logs

Figure 3-19 summarizes the tailoring facilities available for display and log information.

The Rational Environment provides the notion of a *switch* that a user can set which alters the Rational Environment's default treatment or display of information. Switches characterize the behavioral aspects of the Rational Environment, such as the nature of the display or system logging information. These switches are placed in a central location, that is, a switch file, rather than scattering the switches in different places. The switch file can be associated with structures such as a directory, subsystem, or session. Changes to some switches can take effect immediately (e.g., some library ones) while others take effect only after logging in again.

Library switches are associated with a library. A library switch file is associated with each library. This set affects how compilation is done, how links are managed, or how pretty-printing is done in that library. Library switches affect such facilities as whether Unchecked_Deallocation for access types should be enabled, or whether certain pragmas should be ignored, or the case of identifiers and keywords, or file transfer capabilities, or the amount of indentation.

SWITCHES
- Kinds of switches:
  - Library
  - Session affecting:
    - Editing images
    - Ada units
    - Debugging
    - Text I/O
    - Networking
    - Library display
    - Log and profile operations
    - Printing
  - Compiler (only cross-compiler)
  - Error reporting and display
- No user-defined switches, only environment-defined
- Operations:
  - Create a switch file
  - Type-specific (indirect) editing on switch files
  - Changes switches from within programs
  - Display switch file

PROFILE (filters)
- Log generation filter
- Error reaction filter

Figure 3-19: Display and Log Tailoring Functionality

A *session* refers to a particular environment profile that the user wants to set up through a collection of switches (called *session switches*). The user can define multiple sessions and name them. When logging in, the user selects a session, which has the effect of selecting a particular environment setup.

A *profile* provides a filter specification for error reaction and log generation. The error reaction filter specifies how to handle non-fatal errors. Choices include: ignore, abort, and propagate exception. Log generation can be tailored by specifying the types of system messages to be logged, their format, and files to be used for different logs. Profiles can also be used for remote sessions to aid networking operations.

3.1.6.2. File Handling and Logging

Figure 3-20 summarizes the file handling and logging functionality.

The file utilities provide a set of subprograms that allow any object that can be opened for text input/output to be compared, merged, and searched. Such objects include text files, directories

FILES

Append files or merge variant files
Strip merge conflict information from file
Compare files
Find patterns in files or count number of pattern matches

LOG

Create, redirect, copy, or append to log
Write operation success-status to log
Write user-defined messages to log
Use profiles for tailoring error response and activities used

Figure 3-20: File Handling and Logging Functionality

and program libraries, and Ada objects. Regular expressions can be used for pattern matching on names. The user can indicate whether subunits are to be included and can request case-sensitivity checks.

The Rational Environment allows the direct printing of text files, Ada units, and library listings. All other objects must be saved in a text file before they can be printed. The copy of the object committed to disk is printed—uncommitted changes through the editor are not reflected.

Logs are a mechanism for recording system messages as well as user-defined messages. The contents of logs can be tailored by specifying whether or not certain types of messages should be displayed (see also *profiles* in the previous section).

3.1.6.3. Job and Program Control

Figure 3-21 summarizes the job control and program management functionality. Typical functionality can be found. For running jobs, the activities and session switches are inherited from the initiating job. Jobs and scripts can be run after waiting a certain period of time. The user cannot designate a specific start time; however, a function is provided to determine the duration based on current and start times. The user can indicate which input, output, and error files the job is to use. From the documentation it is not clear whether the Rational Environment supports a job hierarchy based on job creation, or whether all jobs have to be managed individually, independent of their creation history.

3.1.6.4. System Administration

Figure 3-22 summarizes the user account management and archiving functionality. Since it is expected that Rational personnel provide all environment maintenance, there are few commands available to a user for altering system features. There seem to be adequate facilities for a user to determine system status by examining queues. Archiving operations can be performed on program libraries, directories, views, and subsystems.

JOBS
Connect, disconnect, stop, start, or kill a job
Place job into background mode
Query job status for all jobs
Program-callable procedure for forcing job into background mode
Privileged operator mode
PROGRAM
Compile and execute programs from other programs
Change access control identity of calling job to that specified
Run scripts or job
Wait for a specific job to complete

Figure 3-21: Job and Program Control Functionality

OPERATOR
Create user and manage groups for access control
Enable terminals for login or disable sessions
Change password
Query disk usage
Make, display, and cancel print requests
Display information about system status
ARCHIVING
Copy, save, restore single or multiple objects

Figure 3-22: System Administration Functionality

3.1.6.5. Access Control

The Rational Environment provides access control through access control lists. Access control lists can be attached to worlds, to program units, and to files. Access control lists specify the access rights different groups have for an object. Two predefined groups are *public* and *network_public*. Other groups can be defined by users with operator capability. Possible rights to files and program units are *read*, and *write*, while those for worlds are *create*, *delete*, *owner*, and *read*.

- *Write* rights allow modification or deletion of an object.
- *Create* rights allow creation of objects in a world.
- *Delete* rights permit deletion of the world.
- *Owner* rights permit changes to the access control list of an object in the world, to perform link operations on the world, to modify compiler switches defined in the world, and to freeze and unfreeze worlds.

- *Read* rights allow the display of the world content and the looking up of object names in the world.

The identity of each job is matched against entries in the access control list of an object to determine whether the job is granted access to the object. The default identity of a job is the user's login name, but it can be changed at runtime by privileged users, or with an appropriate password.

The access control facility has been designed as a general access control facility. It is implemented for the first time in release delta0. Primitives have been provided for creation of and modification to user-defined groups, for association of access control lists with objects and their change, and for setting job identity and checking its rights to an object. The adequacy of the access control model and the provided primitives will have to be determined after they have been used to provide user-level access control through an envelope of Ada functions that encode access control policies.

At the present time, the access control facility is not integrated into the other facilities of the Rational Environment. An example of the current level of integration and provided end-user functionality is the interaction between the CMVC facility and the access control facility. CMVC makes available a set of otherwise internal structures for the purpose of allowing access control lists to be attached by the user or system manager. These structures are only minimally documented in the user manuals. However, the semantics of these structures must be understood for the desired control effect to be achieved, e.g., where in those structures should an access control list be attached to get the effect of controlling the users who can change the exports of a subsystem. We expect that eventually an envelope of functions will be provided on top of the current primitives and that envelope will supply more suitable functionality for the system manager and end-user.

Currently, the default operation of the Rational Environment is to give everyone full rights to all objects. Users are expected to be cooperative rather than malicious. The documentation and provided training gives little guidance on the use of the access control facility.

3.2. Documentation

In general, the type of documentation facilities are similar to those of conventional environments. There is hardcopy documentation in the form of manuals and there are on-line information is procedural, whereas the manuals provide information that ranges from conceptual to procedure-level descriptions. Different documentation exists for different classes of users, such as novices versus experienced Rational users.

The documentation alone is not enough to train users about the intricacies of using the Rational Environment—experienced users recommend the training courses. Rational provides very good technical consultation via their representatives. They also have an 800 number hotline that is staffed during regular working hours and is quite responsive. Unfortunately, Rational does not make available to the customers a document listing known bugs of the Rational Environment.

3.2.1. Printed Documentation

Figure 3-23 summarizes the printed documentation that Rational makes available to users of the Rational Environment (see [7]).

---

1. *Basic Operations Manual*
2. *User's Guide*
3. *Training Manuals* (fundamental and advanced topics)
4. 11 Reference Volumes:
    a. *Reference Summary*
    b. *Editing Images and Editing Specific Types*
    c. *Debugging*
    d. *Sessions and Job Management*
    e. *Library Management*
    f. *Text Input/Output*
    g. *Data and Device Input/Output*
    h. *String Tools*
    i. *Programming Tools*
    j. *System Management Utilities*
    k. *Project Management*
5. *Ada Language Reference Manual*

Figure 3-23: Rational's Printed Documentation

The *Basic Operations Manual* is an concise, easy-to-use guide for novice users of the Rational Environment. It presents the commands and keys for the basic scenarios such as executing commands, editing, and traversing the libraries. It also includes a quick reference to the predefined key bindings.

The *User's Guide* is similar to the *Basic Operations Manual* although it discusses the features in more detail.

The training manuals are essentially copies of the slides that Rational uses in it training courses. These can serve as a reminder to users who went through the courses. For configuration management, we gleaned more information from these training manuals than was available at the time in the reference books.

The 11 reference manuals describe all the facilities of the Rational Environment. These are designed for users with a basic understanding of how to use the Rational Environment. Each of the eleven manuals are approximately 300 pages in length. They are consistently structured, making both perusal and searching for specific information easy. Organization of the documen-

tation reflects the Rational Environment's implementation. That is, chapters are organized by Ada package name such as Compilation or Activity. Although logically related procedures are grouped under a relevant package name, the user has difficulty locating a description if he or she does not know the package name. Then the index or online help facilities can assist the user in finding the package name.

Volume 1, the *Reference Summary*, contains the full Ada specification for each unit in the standard Rational Environment. These are organized by the pathname to the command. Cross-referencing information details the location in the other volumes of the pertinent documentation. The Keymap section presents the standard key bindings, organized by topic and by command name. The Master Index combines all the index information from each of the manuals. Volume 1 is intended to be used as a quick reference to the resources provided by the Rational Environment.

Volumes 2-11 typically contain the following sections: Table of Contents; Key Concepts, and Unit descriptions. The Key Concepts section is generally very comprehensive in covering the spectrum of functionality available and important concepts. It is an excellent starting point for novice users or users who do not have regular contact with the Rational Environment. The Unit section provides details about each command in the form of an Ada specification along with descriptive information and a discussion of the parameters. The sections are alphabetically organized within a package. The volumes have sections separated by tab pages making sections easily accessible.

Each volume contains a "how to use this book" as the first section. There is generally a one to two page overview of the sections within the manual, followed by several pages reiterating the basic structure of the eleven volumes and suggestions on how to locate information. Each volume ends with an index section pertaining to that volume. Although available in some chapters, generally there were no examples of using commands. Side effects of commands, i.e., objects accessed by commands but not passed by parameter, are usually not documented explicitly. Known errors are not documented. The documentation indicates in some cases when cautionary use is necessary.

There is repetition of conceptual information across manuals. This exists for reminding the user of basic concepts. In certain areas, such as workorder management or access control, documentation seems minimal and there was little distinction between the key concept description and unit descriptions.

3.2.2. Online Help

All the documentation is accessed by one of the four help keys. Users will see the same sort of information as that presented in the eleven reference volumes. Descriptions associated with the Ada procedure specifications are more explanatory in the on-line version. No on-line configuration management documentation was available at the time of our evaluation.

The user can ask for help on a key binding, a command, or a pathname to a command. If the user requests information based upon a partial name, the Rational Environment will raise a menu

showing a list of all possible names matching that query. The user must know at least a keyword from a command name in order to get assistance on a command from the Rational Environment.

In some cases not all the capabilities of commands are documented. For instance, the "run" command in the debugger allows single-stepping at about eight levels of granularity but the example in the printed documentation only shows the default value, i.e., statement level.

3.3. User Interface

At first sight, the user interface of the Rational Environment appears to be somewhat complex and overwhelming, especially to users of environments with VT100-type terminals who invoke text editors and tools through a conventional command language. The Rational terminal looks different; the keyboard has a large number of special function keys; the window system, editor, and command language differ from those of other environments; and the interaction model is highly interactive and responsive. However, the Rational Environment user interface has several characteristics that make it quite easy to use, even for the infrequent user. The user interface is uniform throughout the environment. Users interact with one editor as the user interface, one set of editing operations, one language for programming and command invocation. They view the information as objects and can navigate through it in an object-oriented manner, based on available structural and semantic information. The uniformity as a result of Diana as the primary representation has been discussed in Section 2.1.1. The effects of such a specialized system on the ease with which users learn to use the environment has been discussed in Section 2.2.2.1. In this section, we complement these discussions with some detailed information on the user interface functionality.

A user does not necessarily have to know Ada in order to interact with the command processor. Keys are available for helping the user form commands and code with the correct syntax and semantics. The most common commands are bound to function keys. A plastic template describing the functions assigned to the keys is provided to overlay the keyboard. In fact, the ability to interact with the Rational Environment by pressing a key (as opposed to typing out full command names) is very convenient. The binding of common operations to keys such as <promot> is very useful—the user presses <promot> to do many things, such as promote an object to a higher state, execute a program or a command, or force a window to remain on the screen.

There are a handful of keys that most users will constantly use. They are:

- <definition>—traverses structures and displaying the contents of objects.
- <enclosing object>—is the inverse of the Definition key (i.e., the parent item).
- <next item>—navigates through parameter lists or errors.
- <explain>—details messages or error indications.
- <complt>—performs command completion and so performs pretty-printing and syntactic checking.
- <format>—pretty prints an Ada unit, makes the structures available for object selection, and does syntax checking and completion.
- <semanticize>—checks the static semantics of Ada code.
- <promot>—serves many functions and is probably the most frequently used key. It can be used to: execute commands; promote an Ada object to a higher state; force a window not to be removed from the screen; and to run a program.
- <edit>—demotes the selected program unit or its substructure to source state.

Commands are bound to two types of key combinations:

- *Item-operation combinations* that involve an item key such as <region> or <word> (seven keys in all) and an operation such <d> for deleting the item.
- *Modified key combinations* that involve one or more of the modifier keys <shift>, <control>, and <meta> along with another key. Rational use particular combinations to represent certain functions, such as <control><shift> for screen cursor movement.

These combinations are considered "basic" ones by Rational. For the experienced user, "accelerated" bindings are available that enable the user to type as a touch-typist. Many commands also can have a numeric value prefixeded to them as an iteration count.

Parameters for commands can be found via "command completion." If the Environment cannot disambiguate in order to complete the command or statement, it raises a menu noting possible completions. For system names and some programmer-defined names, the Environment can perform name completion. Many procedures have long parameter lists, but default values are provided. The user can navigate through a parameter list and set parameter values. Parameters that have to be most frequently set or their default changed are listed first. Unfortunately, the default size of command windows is two lines and command parameters are listed in separate lines. Scrolling through the parameter list using such a small window can be annoying.

A type-ahead facility is available while a command or program is executing. There is no case sensitivity for commands. Audio feedback is given for incorrect key combinations. Commands can be re-executed with or without editing, providing a convenient history mechanism via commands that "undo." A history command list is maintained for each command window.

Some aggravating interface properties are:

- Cursor movement is allowed over blank space where there are no objects.
- In some cases changes to a displayed object, in particular program libraries, are not reflected in the corresponding window content. Examples are modification time of objects and display of a newly created spec view using the Make_Spec_View command. This can be very frustrating. Removing the window from the screen and redisplaying it does not update its content from the object content. The user has to destroy the window in order to get a correct display. Pressing the <format> key will update the displayed object in most cases. See also Common.Revert for additional information. Common.Revert will update in all cases.
- It is generally not clear as to when an object needs to be selected or when it is appropriate to just place the cursor at an object an invoke the command. It appears that objects must be selected for destructive or major operations such as demote, delete, or edit, whereas for check-out or Make_Uncontrolled, only cursor positioning is needed. Of course, the user can examine the command and look at the default parameter value of <cursor> or <selection> in order to determine such.
- A novice user will take a while to realize where the cursor is placed after the execution of a command. For instance, a <definition> command most helpfully places the cursor on the last visited object within that substructure when it opens the new window, whereas configuration management commands such as check-in or compilation command code unit leave the cursor in the same place as before command execution.
- A separate command window is attached to each window displaying an object (such as directory or Ada object). The user cannot reuse commands across command windows except by copying text across the windows.
- One system window is shared between all command windows. Any error message is displayed in the three-line system window which is located at the top of the screen. It can become confusing when the error message window is not cleared upon re-execution of the same command or another command since it is not immediately obvious that error message does not apply to that last command.
- Due to the algorithm Rational uses for replacing windows, the windows can get shuffled around. The user may force a particular window to stay in one position. However, the screen can comfortably accommodate only three object display windows and the result of locking one window is very frequent exchange of window content in the other windows.

3.4. Performance

Rational presents an environment in which performance measurements may not be comparable to those of conventional environments. This is due to the fact that the Rational Environment uses a different architecture and user interaction model. Therefore, quantitative numbers such as number of lines compiled per minute may not encapsulate its full power. On conventional Ada

environments, performance measurements, such as number of lines compiled per minute and size of source code and object code files, are normally taken. These numbers reflect the usage model of those environments. A comparison based on such numbers leads one to believe that the price of a Rational Environment is not easy to justify. Such numbers, however, do not take into account a variety of factors that better capture the new technology available in the Rational Environment and its effects on productivity.

Some factors are interleaving of syntax and semantic analysis with editing such that "compilation" is reduced to code generation, use of smart processing techniques and subsystems limit the scope of change propagation resulting in a (in some cases drastic) reduction of the number of lines of code to be recompiled, the provision of a version and configuration control model directly supporting the needs of cooperative and independent development, and the integration of these facilities with Ada program libraries. The effects of some of these factors only become apparent by examining large-scale Ada systems, measuring development activity at a granularity larger than execution of individual commands. Unfortunately, in the context of this evaluation we were not able to acquire measurements on productivity gain that would confirm or dispute the conjecture that the Rational Environment provides productivity gain over conventional Ada environments.

3.4.1. Timing Issues

Benchmark tests indicate that the Rational Environment compiles at a rate similar to that of Ada compilers on a DEC's MicroVAX II or a Sun 3/140. For execution, the R1000 seems to be 50% better than a Dec/MicroVAX II, which should be expected due to its specialized hardware. Notice, however, that the Rational Environment has not been optimized for bulk compilation of Ada code from text files, which is what the benchmark measurement reflects.

In general, the Rational Environment can be called highly responsive with many commands taking less than two seconds. For certain commands, such as semantic analysis or code generation of a complete program unit, creation or copying of a program library, invocation of a debugger on a program, users expect slower response in any Ada environment. The Rational Environment even shows good responsive behavior for these commands, i.e., faster than users are accustomed to from other environments. Execution of some of these commands naturally is dependent on the size of the objects being processed.

We have noticed that program libraries encounter a noticeable slowdown when they encounter a large number of entries. Lookup of entries in directories with 100 entries and more shows noticeable delay. Similarly, program libraries containing several hundred program units (e.g., placing all ACEC tests into one program library) affects the responsiveness.

The Rational Environment R1000 processor is intended to be used by several developers. For a Model 200-20 Rational recommends approximately ten simultaneous users. Such a number can be supported during code development. However, during system integration and testing of large systems, as well as bulk processing of Ada code (as is done when importing Ada code or compiling the ACEC suite) one or two such jobs result in an annoying sluggishness of the Rational Environment for the other users. Such sluggishness is more apparent to users of highly respon-

sive environments such as the Rational Environment, because such environments perform more processing interleaved with the user's editing activities (e.g., frequent semantic analysis of program unit fragments). In conventional environments, the user interactions separate more into pure editing activities and processing of program units. Thus, processing for keystroke interactions is limited to character manipulation.

An Effort_Only parameter on some of the commands aids the user in understanding the cost of performing the command. Unfortunately, the cost figure given is not documented so what it represents is not clear.

3.4.2. Space Issues

Overall, the disk space consumption of the Rational Environment is similar to that of conventional Ada environments. Due to the availability of semantic information and the maintenance of program units as separate objects, the Rational Environment is able to perform better than some other Ada environments.

The size of a program unit in installed or coded state is approximately eight to ten times that of its size in ASCII representation. The size of program units in archived state is approximately double the size of the program in a text file due to its representation as a token stream. Such numbers are similar to those of program units stored in program libraries of conventional Ada environments. Notice, that the Rational Environment only allocates space for program libraries that are actually used, and that the program unit in a program library the source code, the semantic information, and object code.

When under version control, the Rational Environment applies a delta technique to storing versions of program units in archived form. The delta technique is similar to those in UNIX/SCCS or DEC/CMS. Currently, the minimal cost for storing a delta is one Kbytes—thus, costlier than conventional version history mechanisms.

The load view of a subsystem can be converted into a code view, in which the subsystem representation consists of object code and limited symbol table information, resulting in space savings of a 9:1 ratio.

As a result of the dynamic linking capability, the Rational Environment does not require a linked execution image to be created and explicitly stored. Thus, space savings are made which should be noticeable, especially when large systems and several variants of systems are built.

4. Conclusions

This chapter summarizes the results of our analysis of the Rational Environment. The Rational Environment as evaluated is a language-centered environment for code development and maintenance of large-scale Ada systems using its own specialized hardware and software.

The Rational Environment as evaluated is described in detail in Section 1.2; this base environment does not include four separately packaged products which were not available to us at the time of our evaluation. These are: two products providing code cross-development support to targets other than the Rational R1000 processor; one product providing design support through Ada as a PDL with structured comments, generation of DoD-STD-2167 documentation from the design, and document formatting; and a product supporting electronic mail.

Our conclusions are organized into two parts. The first part summarizes the technological advances that the Rational Environment contributes to the environment and Ada technology, and distinguishes it from other Ada environments. These are a semantics-based interaction model and integrated support for large-scale Ada code development and maintenance. The second part discusses the Rational Environment from the product perspective. It examines the functionality provided by the Rational Environment, the learnability and maturity of this specialized system, its effectiveness relative to other Ada environments, and its integration into a full-scale software project organization.

4.1. Technological Advances in the Rational Environment

This section concentrates on technological advances that the Rational Environment makes in two areas. One area is that of support for semantics-based interaction and its effectiveness for code development and maintenance. The second area is that of support for large-scale Ada code development. The Rational Environment provides the concept of subsystem for cost-effective system composition, supports a version and configuration control model that caters to the needs of different development and maintenance teams, and integrates the team support well with the support for individual programmers.

4.1.1. Semantics-Based Interaction

As a language-centered environment, the Rational Environment provides a powerful and effective semantics-based interaction model that allows the user to browse Ada systems according to their syntactic structure and semantic dependencies, and to query semantic information (e.g., to determine the scope of changes). It allows the user to interact with the environment uniformly by using Ada as a command language and through a common editing paradigm applied throughout the environment, which supports transparent transition between textual and structural editing. The user interacts with logical units such as Ada packages. Instead of applying tools such as syntax checker and compiler to source code files and having to be concerned with the resulting output files, users of the Rational Environment promote Ada program units through different stages of compiledness. As a language-centered environment, the Rational Environment does limit its support for code development and maintenance to the Ada programming language.

In addition to the semantics-based browsing and editing facilities, the Rational Environment provides a highly responsive system by deploying smart compilation and dynamic linking techniques and through cooperative input from the user (by the user's selecting the part of a program unit to be changed) to reduce reprocessing after changes based on semantic information. As a result, potentially fewer lines of code have to be reprocessed as a result of changes. The debugging facility incorporates these code development facilities, and provides tracing and debugging support for the full Ada language.

This integration of code development and debugging capabilities is facilitated through the use of Diana as primary program representation. Though Diana is an inherent part of the Rational Environment architecture, the user's required knowledge of the Diana representation is limited to understanding that structural editing is done based on Ada constructs, and that semantic information such as display of definition or use of a name can be requested almost any time.

Such a semantics-based interaction model in combination with the use of smart processing techniques gives several benefits to the developer and maintainer. Syntax and semantic errors can be eliminated incrementally during editing by invoking the parser and semantic analyzer frequently on the edited portion of a program unit. Side effects of changes can be fixed quickly, by displaying all the sites affected by the change and navigating to them. These and other browsing capabilities (such as displaying definition sites of names and parameter templates for procedure calls) can be used effectively by developers familiar with the structure and interconnectivity of Ada code, together with the use of smart processing techniques, to minimize the amount of processing on changes, and can make the Rational Environment a powerful code maintenance tool. At the same time, the Environment's ability to generate code stubs, generate code for incomplete program units, and link systems dynamically support early testing during development and allow for rapid prototyping.

4.1.2. Large-Scale Code Development Support

The support for individual developers is integrated with facilities for large-scale development of Ada through multiple teams. In particular, the Rational Environment overcomes shortcomings in the Ada language for large-scale development through the introduction of the subsystem concept and integrates Ada development support with an advanced configuration and version control facility. However, since the Rational Environment concentrates its support to code development and maintenance, users will want to integrate it into their existing computing environments. This latter point is addressed in the second part of our conclusions.

The contributions of the Rational Environment to improving Ada's ability to effectively support large-scale development are in addressing the high cost of recompilation in large Ada systems, especially for changes to central components, and providing versioning and configuration support in an integral manner. This is done through the introduction of the subsystem concept. Subsystems permit large Ada systems to be partitioned into units larger than Ada packages, which contain manageable collections of Ada program units.

Following the philosophy of Ada packages, subsystems separate specification from implementation and require explicit import of facilities provided by others. Differing from Ada packages, subsystems do not require information that is essentially for code generation purposes to be provided as private part of a specification. The mechanisms of the subsystem eliminate some of recompilation that is otherwise necessary due to changes in the private part.
Subsystems represent independent program libraries. Subsystem implementations are compiled against subsystem specifications of imported subsystems, i.e., the interfaces are checked for semantic correctness, while the implementation of those other subsystems is not checked until runtime when dynamic linking is performed. As a result, subsystems act as propagation boundaries, i.e., program changes cause invalidation and recompilation only for units within a subsystem. In contrast, other Ada environments may provide the ability to partition the program units of an Ada system into multiple program libraries, but require the separate compilation property to be maintained across program library boundaries (i.e., changes in one program library invalidate program units in dependent program libraries).

Subsystem specifications and implementations can exist in multiple versions. An Ada system composition is described by enumerating the subsystems comprising the system and selecting the specification and implementation version for each. Based on such a composition description, the Rational Environment puts together the system at run-time and checks for compatibility of interfaces between the independently compiled contents of program libraries of subsystems. The benefits of such subsystem support are that it permits independent development to occur for each subsystem, and that system composition is a flexible operation permitting systems to be reconfigured at low cost. This can effectively reduce the processing cost at test and system integration, where a number of system configurations are created and executed. If the target is different from the Rational R1000 processor, independent development of subsystems is still possible, but composition cost may be higher due to potential lack of dynamic linking and interface checking capabilities in the target run-time system.

In addition to independent development of subsystems by different teams, the Rational Environment supports development and maintenance by multiple teams within one subsystem. The concept of development path allows different teams to independently work on different development threads, such as maintaining several field releases and further developing the product. Facilities for one development path to integrate changes to another development path, e.g., integration of bug fixes to a field release into the development thread, are provided. Within one development path explicit support is provided for cooperative development by individuals of a team working on the next version of a subsystem within one development path. The provided mechanism allows individual developers to work on program units in a coordinated manner by reserving program units into work areas and creating new versions of them, and by isolating developers from the changes of others, as well as providing an upgrade of changes released within the team in a controlled manner. Under normal circumstances the developer does not have to explicitly manipulate or compose versions of individual program units, a task that can be quite complex as the number of program units (Ada packages) reaches large proportions for large-scale systems.

In comparison, other Ada environments provide less integration of program libraries and version and configuration control facilities, usually a generic version and configuration control tool working with source code text files in a language-independent manner. The mechanisms supported by such tools often consist of version control of program components based on files with no distinction between visibility within one team and between teams, configuration management through component versions stored in text files or command scripts, and system build information maintained in "make" descriptions, the files of both of which can be submitted to the version control mechanism. Strict adherence to appropriate conventions is essential to maintaining consistency—and, in the case of Ada, often resulting in frequent and unnecessary compilations.

4.2. The Rational Environment as a Product

This section concentrates on the Rational Environment as a product. The product aspects addressed here do not represent a complete product evaluation, but concentrate on the technical issues. The product aspects addressed are the functionality coverage of the Rational Environment, the learnability and maturity of the environment, especially in light of its being a specialized hardware and software environment, the effectiveness of the Rational Environment as a code development and maintenance environment, and its integration into an existing project organization.

4.2.1. Functionality Coverage

We have classified the evaluated Rational Environment as a language-centered environment for Ada code development and maintenance. This is due to its emphasis in functionality on these activities. The Rational Environment is built on specialized hardware and has implemented its own operating system capabilities as needed for the implementation of a multi-user, time-shared, development environment. These operating system facilities are commented on in the context of the Rational Environment as a development environment, but are not compared to other operating systems.

The Rational Environment provides code development and maintenance functionality similar to that found in other Ada environments: including editing, compilation, linking, execution, and debugging for individual programmers, and version control and configuration management operations for the coordination of teams of developers and managing the version history of the software. It differs in several areas in the way code development is supported and how complete the support is. Two areas have already been highlighted in the previous section: a semantics-based interaction model and large-scale development support through subsystems as a system partitioning and composition mechanism and the integration of version and configuration control with Ada program library support.

As the semantics-based interaction model indicates, the user interacts with the Rational Environment in an object-oriented manner through its editor. This editor exists in several variants, differing in the kinds of objects it can manipulate, how much of the structure it understands, and whether a transparent transition between text and structure editing is supported. Because of the use of a common editing paradigm and the interaction with the Ada-based command processor

through the same editor, a uniform user interface is provided that extends from program manipulation to directory navigation and execution of operating system functions. This results in an highly responsive environment that is optimized to interactive use.

The general editor capabilities are similar to those commonly found in screen-oriented text editors supporting multiple windows. The Ada editing capabilities combine text editing with incremental parsing and syntactic and semantic completion as well as structural modification based on syntactic language constructs and semantic-based browsing.

Ada program units are created and maintained in program libraries, and semantic analysis and code generation are accomplished by promoting program units to different states of ''compiledness,'' avoiding the user's having to maintain separate source and compiled code files. Smart processing and dynamic linking techniques reduce the amount of processing necessary after changes compared to conventional Ada environments, given the user cooperates by selecting the portion of the program unit to be edited. Error reporting is integrated with the program browsing capabilities and provides several levels of explanation for errors. The Rational Environment supports the generation of stubs and the compilation and execution of incomplete programs. The debugger is also integrated with the program browsing capabilities and provides support for the full scope of the Ada language, including debugging of tasks and exceptions, which makes it stand out over many other Ada debuggers. The debugger's functionality could be improved by providing conditional tracing and breakpointing, as well as display of variables on tracepoints.

As is pointed out in the previous section, the Rational Environment enhances the notion of a program library through subsystems by allowing them to be processed independently and by providing separate specification and implementation. The version and configuration control facilities are advanced in that they are integrated with program libraries and provide support for independent development by different teams as well as cooperative development within a team. This is complemented by a simple workorder management facility, whose purpose is to provide task descriptions and automatic logging of certain version and configuration operations, and whose appropriateness will have to be shown in practical use.

The operating system facilities consist of a directory and file system whose access is limited to the local machine (network access to files is available through file transfer programs), user accounts, access control based on access lists, login session with user parameterization and job execution management as well as various system administration functions. The available functionality is similar to that commonly found in operating systems.

4.2.2. Learnability and Maturity

The uniformity of user interaction through a single editing paradigm and Ada as a programming and command language encourage learnability of the Rational Environment. Learnability and infrequent use is further eased by providing labelled binding frequently used functions to function keys. The syntactic and semantic completion capabilities, together with the retrieval of name definitions from the semantics-based editing support and the online help facility, also contribute to ease of use of the environment. Interaction at the command level requires little knowledge of Ada, while program construction and modification requires familiarity with Ada constructs, though some of the syntactic details are supplied by the editor.

The interaction paradigm and the presentation of the available functionality, however, is more complex than that commonly found on the Apple Macintosh. Users will have to learn the conceptual model supported in a particular functionality area, such as version control from the documentation and training material, before being able to use the Rational Environment. Due to the particular architecture of the Rational Environment, the available command set reflects the implementation in some of the functionality areas. This results in more difficult understanding of the available operations in those circumstances.

Learnability is also affected by how much the user interface and interaction model of the user's normal computing environment differs from that of the Rational Environment. Switching between the two models frequently is something to which many users may not be accustomed since they tend to work on one environment only. Users can, however, tailor the Rational Environment user interface to a certain degree.

A general problem with the user interface as far as the commands available to the user is that the way operations are implemented is reflected in user interface especially with respect to the configuration management facilities. We expect such will change as customer feedback is given to Rational.

The Rational Environment performs like a mature product. We found the hardware to be extremely reliable, despite the fact that it is custom-built hardware. Maintenance assistance for the hardware is excellent and includes dial-up diagnostics facilities. The facilities provided by the Rational Environment software were quite stable, although we found some ‘‘teething problems,’’ such as errors and inconsistencies. Many of the problems seemed minor and were repairable or could be bypassed. Rational's technical representatives were always helpful whenever software consultation was needed, and very responsive to problems or requests. The documentation is reasonable, but can be improved.

4.2.3. Effectiveness of a Specialized System

Due to the distinguishing characteristics of the Rational Environment's architecture, it is difficult to select adequate measures to compare its effectiveness to that of other Ada environments. For example, due to its semantics-based interaction model, users of the Rational Environment would rarely perform complete builds of systems from source text. Therefore, quantitative numbers, such as number of lines compiled per minute, may not encapsulate its full power.

We found that the Rational Environment is highly responsive, with many commands taking less than two seconds. Even semantic analysis and code generation of program units stayed at acceptable levels. The compilation speed was slightly better than that of Ada compilers on DEC/MicroVAX II or Sun 3/140. The reader is reminded, however, that due to incremental processing techniques the number of lines to be recompiled may be considerably less than that found in other Ada environments. Similarly, parsing and semantic analysis can be interleaved with editing such that processing of program units as one entity may be reduced to code generation. For execution on the Rational R1000 processor, the link step is reduced to dynamic linking of subsystems at run-time. The disk space consumption of the Rational Environment is similar to that of the better conventional Ada environments.

The Rational Environment is intended to be used and priced for use as a timesharing system (10 simultaneous users for a R1000 Model 200-20). This number of users can be supported during code development, though during system integration, testing, and certain maintenance activities a small number of users can at times slow down the Rational Environment considerably. The price per user is quite high and the system price competes with that for large timesharing systems. However, its support for large-scale development of Ada systems by multiple teams and its semantics-based interaction model and smart processing techniques may outperform the capabilities of other Ada environments in various phases of code development and in code maintenance. This conjecture on productivity gain will have to be confirmed through controlled experiments on large Ada systems, which was not possible in the context of this evaluation.

4.2.4. Integration into a Project Organization

Due to its characteristics, the Rational Environment is not intended to be used as a self-contained computing environment for general computing needs. The Environment, therefore, must be integrated into an organization on several levels: integration of the Rational hardware into the computing environment, portability of Ada code between Ada systems and support for different targets, and integration into a full life-cycle support environment.

The Rational Environment can be integrated into an existing computing environment through its support of the Ethernet throughout IP/TCP protocols with file transfer and remote terminal access capabilities. In addition to the Rational terminal, terminals supporting the VT100 standard can be used.

The Rational Environment supports both the porting of Ada code from one host to others, as well as cross-development to targets other than the R1000 processor. Problems in porting are similar to those commonly found when porting (Ada) software that makes use of facilities which are not part of the language standard or software packages commonly accepted as available on a range of machines.

A third aspect of integration is integration into a full life-cycle support environment. This may require integration with specification and design methods and their respective tools. Tools may be available on different hosts and require the construction of a heterogeneous machine environment architecture. Tools may have to be ported to the Rational Environment or built specifically for the Rational Environment to take advantage of its facilities. None of these is a simple undertaking.

References

[1] Dart, Susan A., Ellison, Robert, Feiler, Peter H., Habermann, A.N.
Software Development Environments.
IEEE Computer , Nov, 1987.

[2] Downey, Grace F., Bassman, Mitchell J., Dahlke, Carl.
Experiment Transcripts for the Evaluation of the Rational Environment.
Technical Report CMU/SEI-88-TR-21, Software Engineering Institute, Carnegie Mellon University, August, 1988.

[3] 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.

[4] Goos, G., Wulf, W.A., Evans, A. Jr., Butler, K.J.
Lecture Notes in Computer Science. Volume 161: *DIANA — An Intermediate Language for Ada.*
Springer-Verlag, Berlin, 1983.

[5] Graham, Marc H., Miller Daniel H.
ISTAR Evaluation.
Technical Report CMU/SEI-88-TR-3, Software Engineering Institute, Carnegie Mellon University, August, 1988.
Forthcoming.

[6] Leblang, David B., Chase, Robert P. Jr.
Computer-Aided Software Engineering in a Distributed Workstation Environment.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 104-112. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[7] 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

[8] Tichy, Walter F.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8(3):273-291, July, 1986.

[9] Weiderman, N.H., Habermann, A.N., Borger, M., Klein, M.
A Methodology for Evaluating Environments.
In *2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 199-207. ACM, December, 1986.

[10] Weiderman, N.H., et al.
Evaluation of Ada Environments.
Technical Report CMU/SEI-87-TR-1, Software Engineering Institute, Carnegie Mellon University, January, 1987.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (leave blank)	2. REPORT DATE July 1988	3. REPORT TYPE AND DATES COVERED Final

4. **TITLE AND SUBTITLE**
Evaluation of the Rational Environment

5. **FUNDING NUMBERS**
C — F19628-95-C-0003

6. **AUTHOR(S)**
Peter H. Feiler, Susan A. Dart, Grace Downey

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. **PERFORMING ORGANIZATION REPORT NUMBER**
CMU/SEI-88-TR-15

9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. **SPONSORING/MONITORING AGENCY REPORT NUMBER**
ESD-TR-88-16

11. **SUPPLEMENTARY NOTES**

12.a **DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified/Unlimited, DTIC, NTIS

12.b **DISTRIBUTION CODE**

13. **ABSTRACT** (maximum 200 words)

This report presents an analysis of the Rational R1000 Development System for Ada, also called the Rational Environment. The evaluation combined the use of the Software Engineering Institute (SEI) methodology for evaluation of Ada environments, an analysis of functionality not covered by that methodology, and an assessment of the novel environment architecture of the Rational Environment. In addition to this report, Experiment Transcripts for the Evaluation of the Rational Environment, by Grace Downey, Mitchell Bassman, and Carl Dahlke (CMU/SEI-88-TR-21, Software Engineering Institute, Carnegie Mellon University, 1988) contains support material for the experimental results. The support material is the result of performing experiments based on the SEI's environment evaluation methodology. It consists of transcripts of the experiments, the detailed answers to the evaluative questions, and the detailed performance results.

14. **SUBJECT TERMS**

15. **NUMBER OF PAGES**

16. **PRICE CODE**

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102