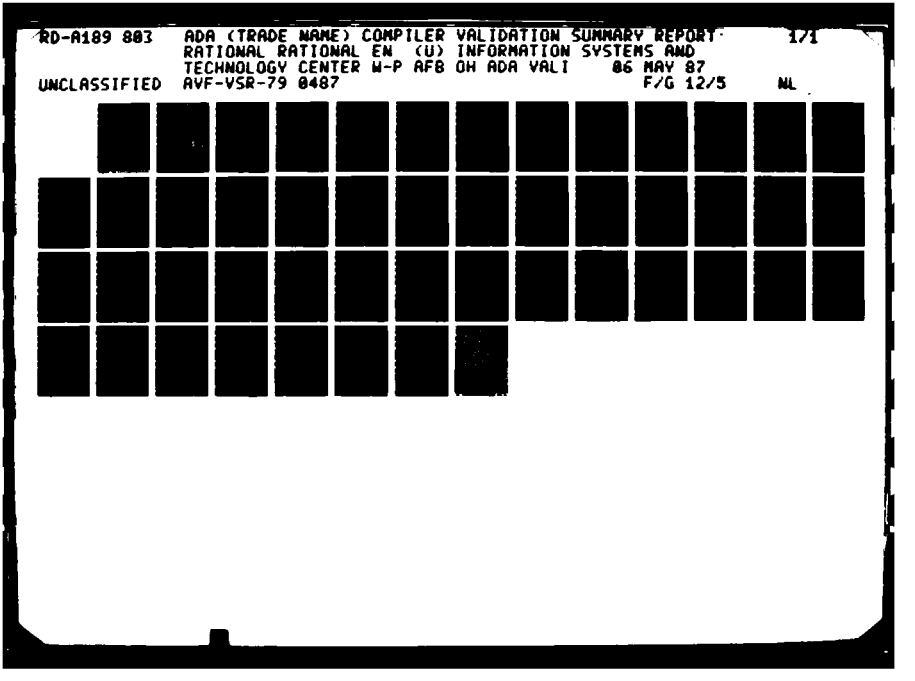
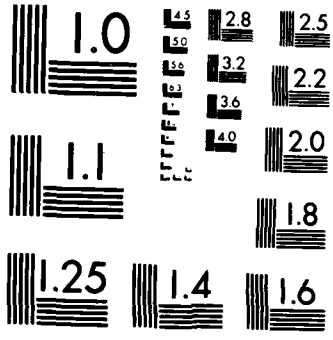

AD-A189 803 ADA (TRADE NAME) COMPILER VALIDATION SUMMARY REPORT 1/1
RATIONAL RATIONAL EN (U) INFORMATION SYSTEMS AND
TECHNOLOGY CENTER W-P AFB OH ADA VALI 06 MAY 87
UNCLASSIFIED AVF-VSR-79 0487 F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(When Data Entered)

DTIC FILE COPY

(2)

AD-A189 803

NTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Rational. Rational Environment A_9_5_2. Rational Arthitecture (R1000 Model 200)	5. TYPE OF REPORT & PERIOD COVERED 6 May 1987 to 6 May 1988 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Wright-Patterson AFB	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION AND ADDRESS Ada Validation Facility ASD/SIOL Wright-Patterson AFB OH 45433-6503	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	12. REPORT DATE 6 May 1987 13. NUMBER OF PAGES 45
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Wright-Patterson	15. SECURITY CLASS (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20. If different from Report)

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

See Attached

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Rational Environment, A_9_5_2, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The Rational Environment is hosted on a Rational Architecture (R1000 Model 200) operating under Rational Environment, Release A_9_5_2. Programs processed by this compiler may be executed on a Rational Architecture (R1000 Model 200) operating under Rational Environment, Release A_9_5_2.

On-site testing was performed 4 May 1987 through 6 May 1987 at Mountain View CA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The nineteen tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 43 of the processed tests determined to be inapplicable. The remaining 2171 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	250	333	242	160	97	136	260	124	32	218	217	2171
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	75	87	5	1	0	3	2	6	0	0	16	209
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

AVF Control Number: AVF-VSR-79.0487
87-02-06-RAT

Ada® COMPILER
VALIDATION SUMMARY REPORT:
Rational
Rational Environment™, A_9_5_2
Rational Architecture (R1000® Model 200)

Completion of On-Site Testing:
6 May 1987

Prepared By:
Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington, D.C.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

™Rational Environment is a trademark of Rational

®Rational and R1000 are registered trademarks of Rational


Place NTIS form here

Ada® Compiler Validation Summary Report:

Compiler Name: Rational Environment, A_9_5_2

Host:
Rational Architecture (R1000 Model 200)
under Rational Environment,
Release A_9_5_2

Target:
Rational Architecture (R1000 Model 2
under Rational Environment,
Release A_9_5_2

Testing Completed 6 May 1987 Using ACVC 1.8

This report has been reviewed and is approved.

Georgeanne C Chitwood

Ada Validation Facility
Georgeanne Chitwood
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

John F Kramer

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA

Virginia L Castor

Ada Joint Program Office
Virginia L. Castor
Director
Department of Defense
Washington DC

®Ada is a registered trademark of the United States Government
(Ada Joint Program Office).

EXECUTIVE SUMMARY

This Validation Summary Report (VSR) summarizes the results and conclusions of validation testing performed on the Rational Environment, A_9_5_2, using Version 1.8 of the Ada® Compiler Validation Capability (ACVC). The Rational Environment is hosted on a Rational Architecture (R1000 Model 200) operating under Rational Environment, Release A_9_5_2. Programs processed by this compiler may be executed on a Rational Architecture (R1000 Model 200) operating under Rational Environment, Release A_9_5_2.

On-site testing was performed 4 May 1987 through 6 May 1987 at Mountain View CA, under the direction of the Ada Validation Facility (AVF), according to Ada Validation Organization (AVO) policies and procedures. The AVF identified 2210 of the 2399 tests in ACVC Version 1.8 to be processed during on-site testing of the compiler. The nineteen tests withdrawn at the time of validation testing, as well as the 170 executable tests that make use of floating-point precision exceeding that supported by the implementation, were not processed. After the 2210 tests were processed, results for Class A, C, D, and E tests were examined for correct execution. Compilation listings for Class B tests were analyzed for correct diagnosis of syntax and semantic errors. Compilation and link results of Class L tests were analyzed for correct detection of errors. There were 43 of the processed tests determined to be inapplicable. The remaining 2171 tests were passed.

The results of validation are summarized in the following table:

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	250	333	242	160	97	136	260	124	32	218	217	2171
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	75	87	5	1	0	3	2	6	0	0	16	209
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

The AVF concludes that these results demonstrate acceptable conformity to ANSI/MIL-STD-1815A Ada.

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from characteristics of particular operating systems, hardware, or implementation strategies. All of the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent but permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do no, conform to the Ada Standard
- To attempt to identify any unsupported language constructs required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc., under the direction of the AVF according to policies and procedures established by the Ada Validation Organization (AVO). On-site testing was conducted from 4 May 1987 through 6 May 1987 at Mountain View CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983.

2. Ada Validation Organization: Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1984.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. A set of programs that evaluates the conformity of a compiler to the Ada language specification, ANSI/MIL-STD-1815A.
Ada Standard	ANSI/MIL-STD-1815A, February 1983.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. In the context of this report, the AVF is responsible for conducting compiler validations according to established policies and procedures.
AVO	The Ada Validation Organization. In the context of this report, the AVO is responsible for setting procedures for compiler validations.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	A test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.

Inapplicable test	A test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	A test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	A test found to be incorrect and not used to check conformity to the Ada language specification. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce link errors.

Class A tests check that legal Ada programs can be successfully compiled and executed. However, no checks are performed during execution to see if the test objective has been met. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers

permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of these units is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of the tests in the ACVC follow conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation.

Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: Rational Environment, A_9_5_2

ACVC Version: 1.8

Certificate Number: 870504W1.08057

Host Computer:

Machine:	Rational Architecture (R1000 Model 200)
Operating System:	Rational Environment, Release A_9_5_2
Memory Size:	32 megabytes

Target Computer:

Machine:	Rational Architecture (R1000 Model 200)
Operating System:	Rational Environment, Release A_9_5_2
Memory Size:	32 megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. This compiler is characterized by the following interpretations of the Ada Standard:

- Capacities.

  The compiler correctly processes tests containing loop statements nested to 65 levels and recursive procedures separately compiled as subunits nested to 10 levels. Block statements nested to 65 levels could not be processed. It correctly processes a compilation containing 723 variables in the same declarative part. (See tests D55A03A..H (8 tests), D56001B, D64005E..G (3 tests), and D29002K.)

- Universal integer calculations.

  An implementation is allowed to reject universal integer calculations having values that exceed SYSTEM.MAX_INT. This implementation rejects such calculation. (See tests D4A002A, D4A002B, D4A004A, and D4A004B.)

- Predefined types.

  This implementation supports the additional predefined type LONG_INTEGER in the package STANDARD. (See tests B86001C and B86001D.)

- Based literals.

  An implementation is allowed to reject a based literal with a value exceeding SYSTEM.MAX_INT during compilation, or it may raise NUMERIC_ERROR or CONSTRAINT_ERROR during execution. This implementation rejects the test during compilation. (See test E24101A.)

- Array types.

  An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises STORAGE_ERROR when the array objects are declared. (See test C52103X.)

A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the length of a dimension is calculated and exceeds INTEGER'LAST. (See test C52104Y.)

A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises no exception and accepts the declaration. (See test E52103Y.)

In assigning one-dimensional array types, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. In assigning two-dimensional array types, the expression does not appear to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Discriminated types.

  During compilation, an implementation is allowed to either accept or reject an incomplete type with discriminants that is used in an access type definition with a compatible discriminant constraint. This implementation rejects such subtype indications during compilation. (See test E38104A.)

  In assigning record types with discriminants, the expression appears to be evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

- Aggregates.

  In the evaluation of a multi-dimensional aggregate, all choices appear to be evaluated before checking against the index type. (See tests C43207A and C43207B.)

  In the evaluation of an aggregate containing subaggregates, all choices are not evaluated before being checked for identical bounds. (See test E43212B.)

  All choices are evaluated before CONSTRAINT_ERROR is raised if a bound in a nonnull range of a nonnull aggregate does not belong to an index subtype. (See test E43211B.)

- Functions.

  An implementation may allow the declaration of a parameterless function and an enumeration literal having the same profile in the same immediate scope, or it may reject the function declaration. If it accepts the function declaration, the use of the enumeration literal's identifier denotes the function. This implementation rejects the declaration. (See test E66001D.)

- Representation clauses.

  The Ada Standard does not require an implementation to support representation clauses. If a representation clause is not supported, then the implementation must reject it. While the operation of representation clauses is not checked by Version 1.8 of the ACVC, they are used in testing other language features. This implementation accepts 'SIZE clauses; it rejects 'STORAGE_SIZE for tasks; it accepts 'STORAGE_SIZE for collections only if the value is expressed as a static expression and lies within specified limits; it rejects 'SMALL clauses. Enumeration representation clauses appear not to be supported. (See tests C55B16A, C87B62A, C87B62B, C87B62C, and BC1002A.)

- Pragmas.

  The pragma INLINE is not supported for procedures or for functions. (See tests CA3004E and CA3004F.)

- Input/output.

  The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants. (See tests AE2101C, CE2201D, and CE2201E.) The package DIRECT_IO cannot be instantiated with unconstrained array types and record types with discriminants witnout defaults. (See tests AE2101H and CE2401D.)

  An existing text file can be opened in OUT_FILE mode, can be created in OUT_FILE mode, and can be created in IN_FILE mode. (See test EE3102C.)

  More than one internal file can be associated with each external file for text I/O for reading only. (See tests CE3111A..E (5 tests).)

  More than one internal file can be associated with each external file for sequential I/O for reading only. (See tests CE2107A..F (6 tests).)

More than one internal file can be associated with each external file for direct I/O for reading only. (See tests CE2107A..F (6 tests).)

An external file associated with more than one internal file cannot be deleted. (See test CE2110B.)

Temporary sequential files are given a name. Temporary direct files are given a name. Temporary files given names are not deleted when they are closed. (See tests CE2108A and CE2108C.)

. Generics.

Generic subprogram declarations and bodies can be compiled in separate compilations. (See test CA2009F.)

Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C and BC3205D.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.8 of the ACVC contains 2399 tests. When validation testing of Rational Environment was performed, nineteen tests had been withdrawn. The remaining 2380 tests were potentially applicable to this validation. The AVF determined that 209 tests were inapplicable to this implementation, and that the 2171 applicable tests were passed by the implementation.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	68	864	1170	14	11	44	2171
Failed	0	0	0	0	0	0	0
Inapplicable	1	3	198	3	2	2	209
Withdrawn	0	7	12	0	0	0	19
TOTAL	69	874	1380	17	13	46	2399

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER												TOTAL
	2	3	4	5	6	7	8	9	10	11	12	14	
Passed	102	250	333	242	160	97	136	260	124	32	218	217	2171
Failed	0	0	0	0	0	0	0	0	0	0	0	0	0
Inapplicable	14	75	87	5	1	0	3	2	6	0	0	16	209
Withdrawn	0	5	5	0	0	1	1	2	4	0	1	0	19
TOTAL	116	330	425	247	161	98	140	264	134	32	219	233	2399

3.4 WITHDRAWN TESTS

The following nineteen tests were withdrawn from ACVC Version 1.8 at the time of this validation:

C32114A	C41404A	B74101B
B33203C	B45116A	C87B50A
C34018A	C48008A	C92005A
C35904A	B49006A	C940ACA
B37401A	B4A010C	CA3005A..D (4 tests)
		BC3204C

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 209 tests were inapplicable for the reasons indicated:

- C34001D, B52004E, B55B09D, and C55B07B use SHORT_INTEGER which is not supported by this compiler.
- C34001F and C35702A use SHORT_FLOAT which is not supported by this compiler.

- C34001G and C35702B use LONG_FLOAT which is not supported by this compiler.

- D4A004B uses literals which are outside of the bounds of LONG_INTEGER (i.e., less than SYSTEM.MIN_INT or greater than SYSTEM.MAX_INT).

- C55B16A makes use of an enumeration representation clause containing noncontiguous values which is not supported by this compiler.

- D56001B is inapplicable because it attempts to process 65 levels of block nesting which exceeds the capacity of the compiler.

- D64005G is inapplicable because it makes use of nested procedures as subunits to a level of 17 which exceeds the capacity of the compiler.

- B86001D requires a predefined numeric type other than those defined by the Ada language in package STANDARD. There is no such type for this implementation.

- C87B62B..C (2 tests) use length clauses which are not supported by this compiler. The length clause is rejected during compilation. In C87B62B the length clause for the collection is rejected because it uses a non-static value.

- C92005B is inapplicable because this system raises NUMERIC_ERROR when an attept is made to converts the value of a task's STORAGE_SIZE attribute to the type STANDARD.INTEGER.

- C96005B checks implementations for which the smallest and largest values in type DURATION are different from the smallest and largest values in DURATION's base type. This is not the case for this implementation.

- CA3004E, EA3004C, and LA3004A use INLINE pragma for procedures which is not supported by this compiler.

- CA3004F, EA3004D, and LA3004B use INLINE pragma for functions which is not supported by this compiler.

- AE2101H and CE2401D use an instantiation of package DIRECT_IO with unconstrained array types which is not supported by this compiler.

- CE2107B..E (4 tests), CE2110B, CE2111D, CE2111H, CE3111B..E (4 tests), CE3114B, and CE3115A are inapplicable because multiple internal files can be associated with the same external file for reading only. The proper exception is raised when multiple access is attempted.

- CE2401B is inapplicable because this implementation does not support creating a Sequential I/O or Direct I/O file for a type containing access types.

- The following tests require a floating-point accuracy that exceeds the maximum of 15 supported by the implementation:

C24113L..Y (14 tests)
C35705L..Y (14 tests)
C35706L..Y (14 tests)
C35707L..Y (14 tests)
C35708L..Y (14 tests)
C35802L..Y (14 tests)
C45241L..Y (14 tests)
C45321L..Y (14 tests)
C45421L..Y (14 tests)
C45424L..Y (14 tests)
C45521L..Z (15 tests)
C45621L..Z (15 tests)

3.6 SPLIT TESTS

If one or more errors do not appear to have been detected in a Class B test because of compiler error recovery, then the test is split into a set of smaller tests that contain the undetected errors. These splits are then compiled and examined. The splitting process continues until all errors are detected by the compiler or until there is exactly one error per split. Any Class A, Class C, or Class E test that cannot be compiled and executed because of its size is split into a set of smaller subtests that can be processed.

Splits were required for 43 Class B tests and 1 Class E test:

B22003A	B26005A	B51001A
B22004A	B29001A	B51003A
B22004B	B2A003A	B53003A
B22004C	B2A003B	B55A01A
B23004A	B2A003C	B64001A
B23004B	B32103A	B67001A
B24001A	B33202B	B67001B
B24001B	B33203B	B67001C
B24001C	B35101A	B67001D
B24005A	B37201A	B95001A
B24005B	B37307B	B95079A
B24204A	E38104A	BB3005A
B24204B	B41202A	BC3003A
B24204C	B44001A	BC3013A
B26002A	B32103A	

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.8 produced by the Rational Environment was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and that the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the Rational Environment using ACVC Version 1.8 was conducted on-site by a validation team from the AVF. The configuration consisted of a Rational Architecture (R1000 Model 200) operating under Rational Environment, Release A_9_5_2.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring splits during the prevalidation testing were included in their split form on the magnetic tape.

The contents of the magnetic tape were loaded directly onto the host computer using a special program developed by Rational for that purpose. This program read each test file and placed it into a directory based on the file name. Directories were organized according to chapter so that the tests for a given chapter were placed into the same directory. Some test files were loaded into special directories because of special requirements for running those tests. One special directory contained a subset of chapter 14 tests that use both CURRENT_OUTPUT and STANDARD_OUTPUT and required a special version of the package REPORT (see below). A second special directory contained tests that have side-effects on the program library during compilations, such as the one that redefines the package SYSTEM.

Once all tests had been loaded to disk, three parallel batch streams were started. The Rational Environment includes both an interactive Ada editing facility and support for incremental change to semantically consistent units. Since the ACVC is structured only for testing batch compilers, Rational constructed a batch facility that invokes components of the interactive and incremental compilation system. Tests were run using this batch facility, and thus indirectly using the Rational Environment compilation system, but no explicit testing of these facilities was attempted.

In the special batch environment constructed for testing purposes, it was necessary to change package body REPORT so that writing was done to CURRENT_OUTPUT rather than STANDARD_OUTPUT because the file STANDARD_OUTPUT was identified with the terminal used to start up the batch environment.

The test team verified that this was the only change to the package. However, two versions of REPORT were required because some of the chapter 14 tests check the use of STANDARD_OUTPUT and CURRENT_OUTPUT. These tests were run using the standard ACVC version of the REPORT package with the console designated as STANDARD_OUTPUT. Execution results for these tests were copied to the printer from the console.

After the test files were loaded to disk, the full set of tests was compiled on the Rational Architecture (R1000 Model 200), and all executable tests were linked and run. Results were printed from the host computer.

The compiler was tested using command scripts provided by Rational and reviewed by the validation team. All switches were set to default, except for the following:

Option	Effect
Directory - Create_Subprogram_Specs : Boolean := False	

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

The validation team arrived at Mountain View CA on 4 May 1987, and departed after testing was completed on 6 May 1987.

APPENDIX A

DECLARATION OF CONFORMANCE

Rational has submitted the following declaration of conformance concerning the Rational Environment.

DECLARATION OF CONFORMANCE

Compiler Implementor: Rational
Ada® Validation Facility: ASD/SCOL, Wright-Patterson AFB, OH
Ada Compiler Validation Capability (ACVC) Version: 1.8

Base Configuration

Base Compiler Name: Rational Environment Version: A_9_5_2
Host Architecture ISA: OS&VER #:
Rational Architecture (R1000) Rational Environment, Release A_9_5_2
Target Architecture ISA: OS&VER #:
Rational Architecture (R1000) Rational Environment, Release A_9_5_2

Implementor's Declaration

I, the undersigned, representing Rational, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Rational is the owner of record of the Ada language compiler listed above and, as such, is responsible for maintaining said compiler in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler listed in this declaration shall be made only in the owner's corporate name.

Walter Wallach Date: 5/5/87
Rational
Walter Wallach, Manager, Software Test and Release

Owner's Declaration

I, the undersigned, representing Rational, take full responsibility for implementation and maintenance of the Ada compiler listed above, and agree to the public disclosure of the final Validation Summary Report. I further agree to continue to comply with the Ada trademark policy, as defined by the Ada Joint Program Office. I declare that all of the Ada language compilers listed, and their host/target performance are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.

Walter Wallach Date: 5/5/87
Rational
Walter Wallach, Manager, Software Test and Release

®Ada is a registered trademark of the United States Government (Ada Joint Program Office).

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of MIL-STD-1815A, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the Rational Environment, A_9_5_2, are described in the following sections which discuss topics in Appendix F of the Ada Language Reference Manual (ANSI/MIL-STD-1815A).

Appendix F for the R1000 Target

This section of the *Reference Manual for the Ada Programming Language* is Appendix F for the Rational Environment, the Rational architecture, and the R1000 target. This appendix describes the following implementation-dependent features:

- Compilation
- The predefined language environment
- Attributes
- Pragmas
- Representation clauses
- Chapter 14 I/O
- Limits

Compilation

The following sections introduce some of the concepts that underlie the Rational Environment compilation system and provide a summary of the separate compilation rules for Ada units in the Environment.

Unit States

The Rational Environment provides an integrated representation of programs, independent of their compilation state. In the Environment, no distinction is made between source code, object code, or other implementation-dependent representations.

In the Environment, each Ada unit can be in one of four basic states, ranging from archived, the lowest state, to coded, the highest state. Transforming a program to the state in which it can be executed consists of promoting all of its units from the source state (or from the archived state) to the coded state; finally, promoting a command that references the program will execute it. Each of the states is described in more detail below:

- *Archived*: The image of the unit cannot be edited. Units in this state also do not have the definition capability and structure-oriented highlighting that is available to units in the source, installed, and coded states. Units can be put in the archived state to save space.

- *Source*: The image of the unit can be edited. Other units that reference it (in the Ada sense) cannot be in a state higher than the source state.
- *Installed*: The unit has been syntactically and semantically checked according to the definition of the Ada language. Other units can now reference it (in the Ada sense); that is, they can be promoted from the source state to higher states.
- *Coded*: Code has been generated for the unit, and the unit can be executed from a Command window (if the unit is R1000 code).

Treatment of Generics

Because the Rational Environment and the Rational architecture do not depend on macro expansion approaches to compile generics, the specification and the body of a generic are not required to be compiled at the same time. Bodies of generics can be changed without making the instantiations of these generics obsolete.

If the formal part of a generic contains private (or limited private) types, certain additional implicit dependencies among the specification, body, and instantiations of a generic may be introduced (see Section 13.3.2 of the *Reference Manual for the Ada Programming Language*). The effect of these implicit dependencies is described more fully in "Installation," below, and in the discussion of the Must_Be_Constrained pragma in "Pragmas," later in this section.

Installation

Installation ordering rules follow Ada's separate compilation rules. Specs must be installed before their corresponding bodies are installed. Subunits must be installed after their parents are installed. A unit spec must be installed before another unit that refers to it can be installed. Bodies can be changed without making other units that refer to their specification obsolete.

If the formal part of a generic contains private (or limited private) types, certain additional implicit installation dependencies among the specification, body, and instantiations of a generic may be introduced (see Section 13.3.2 of the *Reference Manual for the Ada Programming Language*).

If the specification and body of such a generic are installed, and if the body contains language constructs that would require constrained actuals for the formal private (or limited private) types, instantiations that do not provide constrained actuals for these formals cannot be installed after this point (semantic errors will be generated). If, on the other hand, the specification for such a generic and at least one instantiation with unconstrained actuals for the formals have been installed, the body for the generic cannot then be installed if it contains language constructs that would require constrained actuals (semantic errors will be generated).

The Environment supports the Must_Be_Constrained pragma, which can be used to provide more explicit control over the treatment of generics with formals that are private (or limited private). More information is available in the description of the Must_Be_Constrained pragma in "Pragmas," later in this section.

It is always legal for a generic actual parameter to be a type with discriminants if the discriminants have default values. In generic unit instantiation, the Rational Environment treats such actual parameters as if they were constrained types. This conforms to the requirements of AI–00037 (a ruling by the Ada board on the interpretation of the LRM).

Literal declarations outside the bounds of the Long_Integer type are rejected at installation time. The bounds of Long_Integer are System.Min_Int .. System.Max_Int.

A parameterless function having the same name and type as an enumeration literal (declared in the same scope) is rejected at installation time. This conforms to AI–00330 (a ruling by the Ada board on the interpretation of the LRM).

Incremental Operations on Installed Units

The Rational Environment supports the following incremental changes to units in the installed state:

- New declarations that are upwardly compatible (based on Ada semantics) can be inserted. Existing declarations with no dependents can be deleted or demoted from installed state to source state, edited, and then reinstalled.
- New statements can be inserted. Existing statements can be deleted or demoted to source state, edited, and then reinstalled.
- New context clause items can be inserted if they are upwardly compatible (based on Ada semantics). Existing context clause items with no dependents can be deleted or demoted from installed state to source state, edited, and then reinstalled.
- New stand-alone comments (on lines by themselves) can be inserted. Existing stand-alone comments can be deleted or demoted from installed state to source state, edited, and then reinstalled.

Incremental insertion, deletion, and editing of stand-alone comment lines is always allowed.

Incremental operations are not allowed for two-part types, generic formal parts, or generic specifications with installed instantiations. Incremental operations for declarations are also supported only for manipulations of the entire declaration, not for component parts.

Coding

Code is generated for a unit when the body of the unit is promoted to the coded state. Promoting a specification to coded does not result in the generation of any code. Code is generated to elaborate declarations in a specification when the corresponding body is promoted to coded. Promoting a specification to coded results in information being computed about the specification that allows clients to be coded.

Coding order differs in some respects from installation order. A library unit specification must be coded before its body can be coded. Package, generic package, and task subunits are coded before their parents are coded. Subprogram and generic subprogram subunits are coded after their parents are coded. Library unit specifications must be coded before any clients can be coded. A main program body can be coded only after every specification and body in the closure of the main program has been coded. The system may optimize these strict ordering rules when it can make use of information from previous promotions.

Incremental Operations on Coded Units

The Rational Environment supports the following incremental changes to units in the coded state:

- In a library unit specification, new declarations that are upwardly compatible (based on Ada semantics) can be inserted. Existing declarations with no dependents can be deleted, or they can be edited and reinserted. Because the elaboration code for the declarations in a specification is associated with the corresponding body, incremental insertions or deletions in a library unit specification result in the demotion of the corresponding body to the installed state.
- In a library unit specification, pragmas can be incrementally inserted, deleted, or edited only if all declarations to which the pragma refers are simultaneously inserted, deleted, or edited within the same insertion point.
- New context clauses that are upwardly compatible (based on Ada semantics) can be inserted only if the units named in the context clause are coded. Existing context clauses with no dependents can be deleted, or they can be edited and then reinserted. Incremental insertion or deletion of context clauses results in the demotion of any dependent main programs.
- Insertion, deletion, and editing of comments are allowed in all coded units.

All restrictions on incremental insertions, deletions, and editing of units in the installed state also apply to units in the coded state.

The Predefined Language Environment

The following material describes the predefined library units (all in the *Rational Environment Reference Manual*, PT): package Standard, package System, the Unchecked_Deallocation procedure, and the Unchecked_Conversion function.

Package Standard

Package Standard defines all of the predefined identifiers in the language.

```
package Standard is

    type Boolean is (False, True);
    for Boolean'Size use 1;

    type Integer       is range -2**31-1 .. 2**31-1;

    type Long_Integer  is range (-2**62 - 2**62) .. (2**62 - 1 + 2**62);
                            --      -2**63       ..       2**63-1

    type Float is digits 15 range (2.0**1023) - (2.0**97) + (2.0**1023)..
                                - ((2.0**1023) - (2.0**97) + (2.0**1023));
                                 --  -1.7977E308 .. 1.7977E308;

    type Character is (Nul, ..., Del);
    for Character use (0, ..., 127);
    for Character'Size use 8;

    package Ascii is ... end Ascii;

    subtype Natural  is Integer range 0 .. Integer'Last;
    subtype Positive is Integer range 1 .. Integer'Last;

    type String is array (Positive range <>) of Character;

    type Duration is delta 2.0**(-15)
                      -- -3.051757812500E-05
                    range -(2.0**32) .. (2.0**32) - (2.0**(-15));
                      -- -4.294967296000E+09  .. 4.294967296000E+09

    Constraint_Error : exception;
    Numeric_Error    : exception;
    Program_Error    : exception;
    Storage_Error    : exception;
    Tasking_Error    : exception;

end Standard;
```

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, package Standard.

Package System

Package System defines various implementation-dependent types, objects, and subprograms.

Other declarations defined in package System are reserved for internal use and are not documented. These declarations should not be required for users of the Rational Environment.

```
package System is

    type Name     is (R1000);

    System_Name   : constant Name := R1000;

    Bit           : constant :=    1;
    Storage_Unit  : constant :=    1 * Bit;

    Word_Size     : constant :=  128 * Bit;
    Byte_Size     : constant :=    8 * Bit;
    Megabyte      : constant := (2 ** 20) * Byte_Size;
    Memory_Size   : constant := 32 * Megabyte;


    -- System-Dependent Named Numbers

    Min_Int       : constant := Long_Integer'Pos (Long_Integer'First);
    Max_Int       : constant := Long_Integer'Pos (Long_Integer'Last);

    Max_Digits    : constant := 15;
    Max_Mantissa  : constant := 63;
    Fine_Delta    : constant := 1.0 / (2.0 ** 63);
    Tick          : constant := 200.0E-9;

    subtype Priority is Integer range 0 .. 5;


    type Byte is new Natural range 0 .. 255;

    type Byte_String is array (Natural range <>) of Byte;
    -- Basic units of transmission/reception to/from IO devices
    -- The following exceptions are raised by Unchecked_Conversion or
      . Unchecked_Conversions

    Type_Error : exception;
    Capability_Error : exception;
    Assertion_Error: exception;
end System;
```

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, package System.

For additional information on the exceptions, see the reference entries in the *Rational Environment Reference Manual*, PT, Unchecked_Conversion function and package Unchecked_Conversions.

Unchecked_Deallocation Procedure

The Unchecked_Deallocation procedure is used to perform unchecked storage deallocation for values designated by access types that are not tasks and do not contain components that are tasks or pointers to tasks.

Its formal parameter list is:

```
generic
  type Object is limited private;
  type Name   is access  Object;
procedure Unchecked_Deallocation(X  : in out Name);
```

The Unchecked_Deallocation procedure assigns null to X and reclaims storage for the object it designates.

Deallocation is not allowed if the designated type of the access type is a task type or an access to a task type or if it contains such subcomponents. When the designated type or its subcomponents is a generic formal or a private type exported from a subsystem specification having a closed private part, it is not possible to determine at compilation time whether deallocation will be performed. The Allows_Deallocation function in !Tools can be used at run time to determine whether deallocation will be performed.

For additional information, see the reference entries in the *Rational Environment Reference Manual*, PT, package Unchecked_Deallocation.

Unchecked_Conversion Function

The Unchecked_Conversion generic function converts objects of one type to objects of another type.

Its formal parameter list is:

```
generic
  type Source is limited private;
  type Target is limited private;
function Unchecked_Conversion (S : Source) return Target;
```

The Source type is the type of the source object bit pattern that is to be converted to the Target type.

A faster, package version of the Unchecked_Conversion function can be found in the *Rational Environment Reference Manual*, PT, package Unchecked_Conversions.

For additional information and examples, see the reference entries in the *Rational Environment Reference Manual*, PT, Unchecked_Conversion function and package Unchecked_Conversions.

Package Machine_Code

Package Machine_Code is not currently supported.

Attributes

The Environment supports no implementation-dependent attributes other than those defined in Appendix A of the *Reference Manual for the Ada Programming Language*. The following clarifications and restrictions complement the descriptions provided in Appendix A:

- 'Address: This attribute is not supported; any number returned is meaningless.
- 'First_Bit: This attribute is not supported.
- 'Last_Bit: This attribute is not supported.
- 'Position: This attribute is not supported.
- 'Storage_Size: 'Storage_Size is meaningful only when applied to access types or access subtypes, in which case it returns the number of storage units reserved for the collection associated with the base type for the access type or subtype. The value returned by 'Storage_Size is meaningless for task types or task objects.

Pragmas

The Environment supports pragmas for application software development in addition to those defined in Appendix B of the *Reference Manual for the Ada Programming Language*. They are described below, along with additional clarifications and restrictions for the pragmas defined in Appendix B:

- Controlled: Because the implementation does not support automatic garbage collection, this pragma is always implicitly in effect for the R1000 target.
- Disable_Deallocation (X): This pragma is used to disable deallocation for type X, where X is the name of the type for which you want to disable deallocation.
- Enable_Deallocation (X): This pragma is used with the Unchecked_Deallocation generic to enable deallocation for type X, where X is the name of the access type for which you want to reclaim storage. This pragma can also be used on a generic formal to indicate that it should be deallocatable.
- Inline: This pragma currently has no effect for the R1000 target.
- Interface: The Environment does not currently support the execution of other languages on the Rational architecture. To support development of target-dependent software containing this pragma, however, the Environment recognizes the pragma. The effect of this pragma is that a body is implicitly built that will raise the Program_Error exception if the subprogram is executed when the Ignore_Interface_Pragmas library switch is false.
- List: This pragma currently has no effect.
- Loaded_Main: This pragma is generated by the Environment to specify that a unit is a code-only unit. When package Archive (*Rational Environment Reference Manual*, LM) is used to generate a code-only unit, a Main pragma is converted to a Loaded_Main pragma automatically.

- Main: This pragma is used to cause the Environment to preload the object code for the compilation units referenced by a main program. Normally this loading is done when a Command window referencing these units is promoted.

  The pragma takes no parameters and should be placed immediately after the declaration for the specification or the body of the main subprogram. Note that there is a restriction that the parameters to subprograms containing this pragma must be of types defined in package Standard, package System, or any other predefined package in the Environment directory structure provided by Rational.

  The pragma can be placed only after library units. The loading takes place when the body of the main program is promoted to the coded state. For this to occur, all compilation units referenced by the main program must be in the coded state.

  When subsystems are used, the loading of subprograms containing a Main pragma will use the current activity to determine the actual subsystem implementations that will compose the main program. Once the loading has taken place, the execution of the main program can occur without requiring an activity.

  Executing a main program containing this pragma first causes the closure of the library units referenced by the main program to be elaborated. The program is then executed. If there are references in the Command window to units in the closure of the main program other than within the main program, these references will cause their own copy of these units to be elaborated. These elaborated instances will be separate from those of the main program's elaboration.

- Memory_Size: This pragma has no effect.

- Must_Be_Constrained: This pragma is used in a generic formal part to indicate that formal private (and limited private) types must be constrained or need not be constrained.

  This pragma allows programmers to declare explicitly how they intend to use the formals in the specification for a generic. Then the Environment can check that any instantiations of the generic that are installed before the body of the generic is installed are legal.

  The pragma's syntax is:

  ```
  pragma Must_Be_Constrained ([<cond> =>] <type_id>, ...);
  ```

  The condition can be either yes or no and defaults to the previous value (which is initially yes) if omitted. The type identifier must be a formal private (or limited private) type defined in the same formal part as the pragma.

  If the condition value of no is specified, any use in the body that requires a constrained type will be flagged as a semantic error. If yes is specified, any instantiations that contain actuals that require constrained types will be flagged with semantic errors if the actuals are not constrained.

- Open_Private_Part: This pragma is used in conjunction with subsystems to indicate that a subsystem interface has an open private part.

- Optimise: This pragma currently has no effect.

- Pack: All records and arrays are stored packed in the minimum number of bits that they require, unless explicitly overridden by a length representation clause (see "Representation of Objects," below). Thus, this pragma has no effect.

- Page: This pragma is used by the print spooler to cause a new page. The pragma will be the last line on the page. The next line will be printed on the next page.
- Page_Limit (X): This pragma specifies that the page limit for the current job should be no less than X, where X is a number. This pragma overrides the library switch Page_Limit, which overrides the session switch Default_Job_Page_Limit. For a more detailed description, see the reference entries in the *Rational Environment Reference Manual*, SMU, System_Utilities.Get_Page_Counts and System_Utilities.Set_Page_Limit procedures.
- Priority: Priorities can be specified only inside a task or a library main program. If multiple priorities are specified, only the first priority specified is used. The default priority is 2.
- Private_Eyes_Only: This pragma is used in conjunction with subsystems to indicate that items following the pragma in a context clause are required only in the private part of the subsystem interface.
- Shared: This pragma currently has no effect.
- Storage_Unit: The only legal storage unit value for the Rational architecture is 1.
- Suppress: This pragma currently has no effect.
- System_Name: The only legal system name is R1000.

Representation Clauses

The Rational Environment does not currently provide a complete implementation for representation specifications. To facilitate host/target development of target-dependent code containing representation clauses, however, the Environment will optionally compile unsupported representation clauses when the Ignore_Unsupported_Rep_Specs library switch is set to true.

Representation of Objects

The Environment follows some simple rules for representing objects in virtual memory, and these rules can be used to create objects with arbitrary bit images without using representation clauses.

For discrete types as components of structures (records and arrays), the Rational architecture representation will allocate the minimum amount of space to represent the range imposed by the (possibly dynamic) constraints of the applicable subtype, using a two's complement representation that is zero based.

For example:

```
subtype Binary is Integer range 0 .. 1;    -- uses 1 bit

subtype A is Integer range -3 .. 120;      -- uses 8 bits

type B is new Natural range 0 .. 63;       -- uses 6 bits

type C is new Natural range 1022 .. 1023; -- uses 10 bits

type D is (X, Y, Z);                       -- uses 2 bits
                                           --   X => 0
                                           --   Y => 1
                                           --   Z => 2
type E is (X);                             -- uses 0 bits
```

Size representation clauses are supported for all enumeration types that are not declared with two-part declarations. Thus, the above rules can be overridden. A specific example is the representation for the Standard.Character type, which takes 8 bits instead of 7 because of a size representation clause.

For records without discriminants, the Rational architecture stores the fields in the order specified in the type declaration, using the minimum space required for each field, with no additional Environment-generated fields.

```
type R1 is              -- uses 8+6+1 = 15 bits
    record
        Field_1 : A;
        Field_2 : B;
        Field_3 : Boolean;
    end record;

type R2 is              -- uses 15+1 = 16 bits
    record
        Field_1 : R1;
        Field_2 : Boolean;
    end record;
```

For constrained array types, the Rational architecture stores the elements packed, using the minimum space for each element, with no additional fields.

```
type A1 is array (1..N) of R1;          -- uses 15*N bits
                                        -- N need not be static

type A2 is array (0..10) of Boolean; -- uses 11 bits

type R3 is                              -- uses 15+11+2 = 28 bits
    record
        Field_1 : R1;
        Field_2 : A2;
        Field_3 : D;
    end record;
```

Length Clauses

- 'Size: The Rational architecture supports the 'Size attribute for discrete types only. These types are further limited in that they can have only a single declaration point (that is, they cannot be incomplete or private types). The size specified must be less than or equal to 64.
- 'Storage_Size for collections: The default collection size is 2**24 bits. The storage size for a collection can range from 2**8 to 2**32 bits. The storage size for a collection determines the number of bits required to represent access types for the collection (for example, for collections of the default 2**24 bit size, the number of bits required to store objects of the access type that is associated with this collection is 24). Only types with single declaration points can have storage size specified (that is, they cannot be incomplete or private types).

  Storage sizes for collections must be specified as static expressions.
- 'Storage_Size for tasks: Because each task in the Rational architecture gets its own virtual address space, storage size specifications for tasks are meaningless and, consequently, are not supported.
- 'Small: This length clause is not currently supported.

Enumeration Representation Clauses

No enumeration representation clauses are currently supported.

Record Representation Clauses

No record representation clauses are currently supported.

Address Clauses

No address clauses are currently supported.

Interrupts

Because interrupts do not exist in the Rational architecture, these representation clauses are not needed and, consequently, are not supported.

Chapter 14 I/O

The Environment supports all of the I/O packages defined in Chapter 14 of the *Reference Manual for the Ada Programming Language*, except for package Low_Level_Io, which is not needed. The Environment also provides a number of other I/O packages. The packages defined in Chapter 14, as well as the other I/O packages supported by the Environment, are more fully documented in the *Rational Environment Reference Manual*, Text Input/Output (TIO) and Data and Device Input/Output (DIO).

The following list summarizes the implementation-dependent features of the Chapter 14 I/O packages:

- Filenames: Filenames must conform to the syntax of Ada identifiers. They can, however, be keywords of the Ada language.
- Form parameter: Depending on the external file being written to, this parameter affects the way terminals and Ada units are read. For example, it can specify whether to have the Page pragma read with the Page_Pragma_Mapping option.
- Instantiations of package Direct_Io and package Sequential_Io with access types: Such instantiations are allowed. If files are created or opened using such instantiations, the Use_Error exception is raised.
- Count type: The Count type for package Text_Io and package Direct_Io is defined as:

```
package Text_Io is
  ...
  type Count is range 0 .. 1_000_000_000;
  ...
end Text_Io;

package Direct_Io is
  ...
  type Count is new Integer
              range 0 .. Integer'Last/Element_Type'Size;
  ...
end Direct_Io;
```

- *Field subtype: The Field subtype for package* Text_Io *is defined as:*

```
subtype Field is Integer range 0 .. Integer'Last;
```

- Standard_Input and Standard_Output files: When a job is run from a Command window, these files are the interactive input/output windows provided by the Rational Editor. When a job is run from package Program, options allow the user to specify what Standard_Input and Standard_Output will be.
- Internal and external files: More than one internal file can be associated with a single external file for input only. Only one internal file can be associated with a single external file for output or inout.
- Sequential_Io and Direct_Io packages: Package Sequential_Io can be instantiated for unconstrained array types or for types with discriminants without default discriminant values. Package Direct_Io cannot be instantiated for unconstrained array types or for types with discriminants without default discriminant values.
- Terminators: The line terminator is denoted by the character Ascii.Lf, the page terminator is denoted by the character Ascii.Ff, and the end-of-file terminator is implicit at the end of the file. A line terminator directly followed by a page terminator is compressed to the single character Ascii.Ff. The line and page terminators preceding the file terminator are implicit and do not appear as characters in the file. For the sake of portability, programs should not depend on this representation, although it can be necessary to use this representation when importing source from another environment or exporting source from the Rational Environment.

- Treatment of control characters: Control characters, other than the terminators described above, are passed directly to and from files to application programs.
- Concurrent properties: The Chapter 14 I/O packages assume that concurrent requests for I/O resources will be synchronized by the application program making the requests, except for package Text_Io, which will synchronize requests for output.

Limits

The following package specifies the absolute limits on the use of certain language features:

```
with system;
package Limits is

  Large : constant := <some very large number>;

   -- Scanner
  Max_Line_Length                          : constant := 254;

  -- Semantics
  Max_Discriminants_In_Constraint          : constant := 256;
  Max_Associations_In_Record_Aggregate     : constant := 256;
  Max_Fields_In_Record_Aggregate           : constant := 256;
  Max_Formals_In_Generic                   : constant := 256;
  Max_Nested_Contexts                      : constant := 250;
  Max_Nested_Packages                      : constant := Large;
  Max_Units_In_Transitive_Closure_Of_With_Lists
                                           : constant := Large;
                  -- (limited by virtual memory stack size)
  Max_Number_Of_Libraries                  : constant := Large;

  -- Code Generator
  Max_Indices_In_Array_Aggregate           : constant := 64;
  Max_Parameters_In_Call                   : constant := 255;
  Max_Expression_Nesting_Depth             : constant := Large;
                  -- (limited by virtual memory stack size)
  Max_Number_Of_Fields_In_Records          : constant := 255;
  Max_Number_Of_Entries_In_A_Task          : constant := 255;
  Max_Number_Of_Dimensions_In_An_Array     : constant := 63;
  Max_Nesting_Of_Subprograms_Or_Blocks_In_A_Package
                                           : constant := 14;

  -- Execution
  Max_Number_Of_Tasks                      : constant := Large;
                  -- (limited by available disk space)
  Max_Object_Size                          : constant := (2**32)*System.Bit;

end Limits;
```

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..253 => 'A', 254 => '1')
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..253 => 'A', 254 => '2')
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..125 => 'A', 126 => '1', 127..254 => 'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..125 => 'A', 126 => '2', 127..254 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..251 => '0', "298")

Name and Meaning	Value
$BIG_REAL_LIT A real literal that can be either of floating- or fixed-point type, has value 690.0, and has enough leading zeroes to be the size of the maximum line length.	(1..248 => '0', "69.0E1");
$BLANKS A sequence of blanks twenty characters fewer than the size of the maximum line length.	(1..234 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	1_000_000_000
$EXTENDED_ASCII_CHARS A string literal containing all the ASCII characters with printable graphics that are not in the basic 55 Ada character set.	"abcdefghijklmnopqrstuvwxyz!$%?@[\]^`{}~"
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$FILE_NAME_WITH_BAD_CHARS An illegal external file name that either contains invalid characters, or is too long if no invalid characters exist.	BAD_CHARACTERS&<>=
$FILE_NAME_WITH_WILD_CARD_CHAR An external file name that either contains a wild card character, or is too long if no wild card character exists.	WILDCARDS@
$GREATER_THAN_DURATION A universal real value that lies between DURATION'BASE'LAST and DURATION'LAST if any, otherwise any value in the range of DURATION.	5.0E09
$GREATER_THAN_DURATION_BASE_LAST The universal real value that is greater than DURATION'BASE'LAST, if such a value exists.	5.0E09

Name and Meaning	Value
$ILLEGAL_EXTERNAL_FILE_NAME1 An illegal external file name.	BAD_CHARACTERS&<>=
$ILLEGAL_EXTERNAL_FILE_NAME2 An illegal external file name that is different from $ILLEGAL_EXTERNAL_FILE_NAME1.	(1..100 => 'A') & (1..100 => 'A') & (1..100 => 'A')
$INTEGER_FIRST The universal integer literal expression whose value is INTEGER'FIRST.	-2_147_483_647
$INTEGER_LAST The universal integer literal expression whose value is INTEGER'LAST.	2_147_483_647
$LESS_THAN_DURATION A universal real value that lies between DURATION'BASE'FIRST and DURATION'FIRST if any, otherwise any value in the range of DURATION.	-5.0E09
$LESS_THAN_DURATION_BASE_FIRST The universal real value that is less than DURATION'BASE'FIRST, if such a value exists.	-5.0E09
$MAX_DIGITS The universal integer literal whose value is the maximum digits supported for floating-point types.	15
$MAX_IN_LEN The universal integer literal whose value is the maximum input line length permitted by the implementation.	254
$MAX_INT The universal integer literal whose value is SYSTEM.MAX_INT.	9_223_372_036_854_775_807

TEST PARAMETERS

Name and Meaning	Value
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER if one exists, otherwise any undefined name.	NO_SUCH_TYPE
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFFFFFFFFFE#
$NON_ASCII_CHAR_TYPE An enumerated type definition for a character type whose literals are the identifier NON_NULL and all non-ASCII characters with printable graphics.	(NON_NULL)

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following nineteen tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form "AI-ddddd" is to an Ada Commentary.

- C32114A: An unterminated string literal occurs at line 62.
- B33203C: The reserved word "IS" is misspelled at line 45.
- C34018A: The call of function G at line 114 is ambiguous in the presence of implicit conversions.
- C35904A: The elaboration of subtype declarations SFX3 and SFX4 may raise NUMERIC_ERROR instead of CONSTRAINT_ERROR as expected in the test.
- B37401A: The object declarations at lines 126 through 135 follow subprogram bodies declared in the same declarative part.
- C41404A: The values of 'LAST and 'LENGTH are incorrect in the if statements from line 74 to the end of the test.
- B45116A: ARRPRIBL1 and ARRPRIBL2 are initialized with a value of the wrong type--PRIBOOL_TYPE instead of ARRPRIBOOL_TYPE--at line 41.
- C48008A: The assumption that evaluation of default initial values occurs when ar exception is raised by an allocator is incorrect according to AI-00397.
- B49006A: Object declarations at lines 41 and 50 are terminated incorrectly with colons, and end case; is missing from line 42.
- B4A010C: The object declaration in line 18 follows a subprogram body of the same declarative part.

- B74101B: The begin at line 9 causes a declarative part to be treated as a sequence of statements.

- C87B50A: The call of "/=" at line 31 requires a use clause for package A.

- C92005A: The "/=" for type PACK.BIG_INT at line 40 is not visible without a use clause for the package PACK.

- C940ACA: The assumption that allocated task TT1 will run prior to the main program, and thus assign SPYNUMB the value checked for by the main program, is erroneous.

- CA3005A..D (4 tests): No valid elaboration order exists for these tests.

- BC3204C: The body of BC3204C0 is missing.

END
DATE
FILM
4-88
DTIC