THE
ADA COMPILER VALIDATION CAPABILITY
(ACVC)
VERSION 2.2
USER'S GUIDE

September 25, 1998

Prepared for:

DISA/JIEO/CFCSE/JEXH
ATTN: Mr. Gary Shupe
5600 Columbia Pike
Falls Church, VA 22041

Prepared by:

Electronic Data Systems Corporation
Suite T-600
7900 Westpark Drive
McLean, VA 22102

Contents

1. INTRODUCTION..3

1.1 Definition of Terms 4

1.2 References 5

1.3 ACVC Purpose 6

2. CHANGES FOR ACVC 2.2 7

3. CONFIGURATION INFORMATION..9

3.1 Introduction 9

3.2 Structure 9
3.2.1 Physical 9
3.2.2 Logical..10
3.2.3 Legacy Tests..11
3.2.4 Foundation Code 11
3.2.5 Special Core Tests 12
3.2.6 Foreign Language Code 13

3.3 Test Classes 13
3.3.1 Class A 14
3.3.2 Class B..14
3.3.3 Class C..14
3.3.4 Class D 15
3.3.5 Class E..15
3.3.6 Class L..15
3.3.7 Foundation Code 16
3.3.8 Specialized Needs Annex Tests..16

3.4 Naming Convention..16
3.4.1 Legacy Naming 17
3.4.2 ACVC 2.2 Naming 19
3.4.3 Multiple File Tests..20

3.5 Test Program Format 21

3.6 General Standards..22

3.7 Test Structure 23

3.8 Delivery Directory Structure..24

3.9 File Format..25

4. USING THE ACVC 27

4.1 Introduction 27

4.2 Installation of the ACVC Test Suite 27
4.2.1 Contents of the ACVC Delivery 27
4.2.2 Guide to Decompressing Files 28
4.2.3 Files With Non-Graphic Characters 30

4.3 Tailoring the ACVC Test Suite 31
4.3.1 ImpDef Customization 31
4.3.2 Macro Defs Customization 32
4.3.3 Processing for Wide_Character Tests 33
4.3.4 Package SPPRT13 and Function FcnDecl 34
4.3.5 Modification of Package REPORT 35
4.3.6 Allowed Test Modifications 35

4.4 Processing the Support Files 36
4.4.1 Support Files 36
4.4.2 "CZ" Acceptance Tests 36

4.5 Establishing Command Scripts 37
4.5.1 Command Scripts 37
4.5.2 Dependencies 37

4.6 Processing ACVC Tests 38
4.6.1 Required Tests 38
4.6.2 Test Partitions 38
4.6.3 Bundling Test Programs 38
4.6.4 Processing That May be Omitted 39
4.6.5 Tests with Special Processing Requirement 39
4.6.6 Focus on Specific Areas 45

4.7 Grading Test Results 45
4.7.1 Expected results for Executable Tests 45
4.7.2 Expected Results for Class B 46
4.7.3 Expected Results for Class L 47
4.7.4 Inapplicable Tests 47
4.7.5 Withdrawn Tests 48

4.8 Addressing Problems or Issues 48
4.8.1 Typical Issues 48
4.8.2 Deviation from Expected Results - Petition & Review 49

4.9 Reprocessing and Regrading 49

APPENDIX A: VERSION DESCRIPTION 51

A.1 Core Test Files 53

A.2 Specialized Needs Annex Test Files 73

A.3 Foundation Code Files 75

A.4 Documentation Files 77

A.5 Other Files 79
A.51 List of ACVC 2.2 Files 79
A.52 Support Units Referenced by Many Tests 79
A.53 Preprocessing Tools and Data 79
A.54 Tests for Reporting Code 79

A.6 Tests With Special Requirements 81

A.7 Test Files Added In ACVC 2.2 83

A.8 Test Files Modified For ACVC 2.2 85

A.9 Support Files Modified For ACVC 2.2 87

A.10 Test Files Deleted Since ACVC 2.1 89

APPENDIX B: PARAMETERIZATION FILES 91

B.1. Macro Substitution File 93

B.2. Macro Substitution Tests 95

B.3. Package ImpDef and Its Children 97

APPENDIX C: OUTPUT OF CZ TESTS 99

C.1. Sample Output From CZ0004 101

C.2. Sample Output From CZ1101A 103

C.3. Sample Output From CZ1102A 105

C.4. Sample Output From CZ1103A 107
C.4.1 Output When External Files Are Supported 107
C.4.2 Output When External Files Are Not Supported 108

APPENDIX D: APPLICABILITY CRITERIA 109

D.1. Compile-Time Inapplicability 111
D.1.1 Type SHORT_INTEGER 111
D.1.2 Type LONG_INTEGER 111

D.1.3 Other Predefined Integer Types 111
D.1.4 Fixed Point Restrictions 112
D.1.5 Non-binary, Non-decimal Values of 'SMALL 112
D.1.6 Compiler Rejection of Supposedly Static Expression 112
D.1.7 Separate Compilation with Generics 112
D.1.8 Non-default Size Rep Clauses for Floating Point Types 113
D.1.9 Machine Code Insertions 113
D.1.10 Illegal External File Names 113

D.2. Reported Inapplicability 115
D.2.1 Value of MACHINE_OVERFLOWS is False 115
D.2.2 Value of MACHINE_OVERFLOWS is True 115
D.2.3 SYSTEM.MAX_DIGITS 115
D.2.4 Floating Point Overflow 116
D.2.5 Type DURATION 116
D.2.6 Text Files (Non-supported Features) 116
D.2.7 Text Files (Supported Features) 120
D.2.8 Sequential Files (Non-supported Features) 121
D.2.9 Sequential Files (Supported Features) 122
D.2.10 Direct Files (Non-supported Features) 123
D.2.11 Direct Files (Supported Features) 125
D.2.12 Stream Files (Non-supported Features) 126
D.2.13 Wide Text Files (Non-supported Features) 126
D.2.14 File I/O Tests 126

Preface

Although this document has been prepared under the sponsorship of the Ada Joint Program Office (AJPO), the responsibility for Ada conformity assessment (validation) has been transferred to the private sector. ACVC 2.2 is used in the conformity assessment system sponsored by the Ada Resource Association (ARA). Therefore, references to the conformity assessment system use the language of the ARA's procedures, which are compatible with language defined in the proposed International Standard for Ada Conformity Assessment. There is one notable exception to this usage: in the ARA-sponsored system, the test suite is called the Ada Conformity Assessment Test Suite, or ACATS. In this document, we have retained the identification of the test suite as ACVC 2.2.

1. Introduction

The Ada Compiler Validation Capability (ACVC) is the official test method used to check conformity of an Ada implementation with the Ada programming language standard (ANSI/ISO/IEC 8652:1995). The ACVC User's Guide is part of the ACVC and is distributed with the test programs and testing support packages. It explains the contents and use of the ACVC test suite.

This version of the ACVC is 2.2. ACVC versions based on ANSI/MIL-STD-1815A-1983, ISO/8652:1987 (Ada 83) were numbered 1.x where x ranged from 1 to 11. ACVC versions numbered 2.0, 2.0.1, and 2.1, based on ANSI/ISO/IEC 8652:1995 (Ada95) have previously been released. ACVC 2.2 contains test programs to check for conformity to new language features defined in [Ada95], as well as test programs to check for conformity to language features shared between Ada83 and Ada95. Subsequent maintenance or enhancement versions of the suite, if they are required, will be numbered 2.3, etc.

The ACVC User’s Guide describes the set of ACVC tests and how they are to be used in preparation for validation. The formal procedures for validation are described in [Pro98], and the rules in that document govern all validations, notwithstanding anything in this document that may be interpreted differently. Moreover, this guide does not discuss specific requirements on processing of the ACVC test suite, or submission and grading of results that an Ada Conformity Assessment Laboratory (ACAL) may impose.

The User's Guide is intended to be used by compiler implementers, software developers who maintain a version of the ACVC as a quality control or software acceptance tool, and third-party testers (e.g., Ada Conformity Assessment Laboratories).

Section 2 of the User’s Guide for ACVC 2.2 summarizes the changes between ACVC 2.1 and ACVC 2.2. Section 3 describes the configuration of the ACVC, including a description of the ACVC software and delivery files. Section 4 provides step-by-step instructions for installing and using the test programs and test support packages, and for grading test results. The appendices include other information that characterizes the ACVC 2.2 release.

Refer to Sections 1.1 and 4.7 for the definition of an acceptable result and the rules for grading ACVC 2.2 test program results. Section 4.8.2 provides instructions for submitting a petition against a test program if a user believes that a deviation from the acceptable results for a given test program is in fact conforming behavior.

The ACVC test suite is available from any ACAL and from the Ada Information Clearinghouse (sponsored by the ARA). See *http://www.adaic.org*.

1.1 Definition of Terms

Acceptable result : The result of processing an ACVC test program that meets the explicit grading criteria for a grade of "passed" or inapplicable.

ACVC Implementer's Guide (AIG) : A document describing the test objectives used to produce test programs for Ada83 ACVC versions (1.1-1.11). AIG section references are embedded in Ada83 test naming conventions.

Ada Conformity Assessment Authority (ACAA) : The part of the certification body that provides technical guidance for operations of the Ada certification system

Ada Conformity Assessment Laboratory (ACAL) : The part of the certification body that carries out the procedures required to perform conformity assessment of an Ada implementation. (Formerly AVF)

Ada implementation : An Ada compilation system, including any required run-time support software, together with its host and target computer systems.

Ada Joint Program Office (AJPO) : An organization within the U.S. Department of Defense that sponsored the development of ACVC 2.2 and formerly provided policy and guidance for an Ada certification system.

Ada programming language : The language defined by reference [Ada95].

Ada Resource Association (ARA) : The trade association that sponsors the Ada conformity assessment system.

Ada Validation Facility (AVF) : Former designation of an Ada Conformity Assessment Laboratory (which see).

Ada Validation Organization (AVO) : Organization that formerly performend the functions of the Ada Conformity Assessment Authority (which see)..

Certification Body : The organizations (ACAA and ACALs) collectively responsible for defining and implementing Ada conformity assessments, including production and maintenance of the ACVC tests, and award of Ada Conformity Assessment Certificates.

Certified Products List (CPL) : A published list identifying all certified Ada implementations. The CPL is available on the Ada Information Clearinghouse Intenet site (*www.adaic.org)*.

Challenge : A documented disagreement with the test objective, test code, test grading criteria, or result of processing an ACVC test program when the result is not PASSED or INAPPLICABLE according to the established grading criteria. A challenge is submitted to the ACAA.

Conforming implementation : An implementation that produces an acceptable result for every applicable test. Any deviation constitutes a non-conformity.

Core language : Sections 2-13 and Annexes A, B, and J of [Ada95]. All implementations are required to implement the core language. The validation tests for core language features are required of all implementations.

Coverage matrix : A document containing an analysis of every paragraph of [Ada95]. Each paragraph has an indication of whether is contains a testable Ada95 requirement, whether it is upwardly compatible from Ada83, or whether is testable in the ACVC suite (e.g. it contains an example). Paragraphs that contain testable requirements also indicate what ACVC test(s) specifically examine features described in the paragraph.

Deviation : Failure of an Ada implementation to produce an acceptable result when processing an ACVC test program.

Foundation Code : Packages used by multiple tests; foundation code is designed to be reusable. Generally a foundation is a package containing types, variables, and subprograms that are applicable and useful to a series of related tests. Foundation code is never expected to cause compile time errors. It may be compiled once for all tests that use it or recompiled for each test that uses it; it must be bound with each test that uses it.

Legacy Tests : Tests that were included in ACVC 1.12 that have been incorporated into later ACVC versions. The vast majority of these tests check for language features that are upwardly compatible from Ada83 to Ada95. Some of these tests have been modified from the ACVC 1.12 versions to ensure that Ada95 rules are properly implemented in cases where there were extensions or incompatibilities from Ada83 to Ada95.

Specialized Needs Annex : One of annexes C through H of [Ada95]. Conformity testing against one or more Specialized Needs Annexes is optional. There are validation tests that apply to each of the Specialized Needs Annexes. Results of processing these tests (if processed during a validation) are reported on the certificate and in the Validated Compilers List.

Test Objectives Document (TOD) : A document containing the test objectives used for new ACVC tests that focus on Ada95-specific features.

Validated Compilers List (VCL) : Former designation of the Certified Products List (which see).

Withdrawn Tests : A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language. Withdrawn tests are not applicable to any implementation. Withdrawn tests are often modified and restored to subsequent ACVC releases.

Withdrawn Test List : A list maintained by the ACAA documenting the currently withdrawn tests. The withdrawn test list is updated from time to time as protests from implementers are received and processed, or as other technical information is received.

1.2 References

[Ada83] ANSI/MIL-STD-1815A-1983, ISO 8652:1987, FIPS 119 Reference Manual for the Ada Programming Language--superseded by ISO-8652:95)

[Ada95] ANSI/ISO/IEC 8652:1995, FIPS 119-1 The Reference Manual for the Ada Programming Language, February 1995

[Pro98] Ada Resource Association: Ada Conformity Assesment Operating Procedures, August 1998

1.3 ACVC Purpose

The purpose of the ACVC is to check whether an Ada compilation system is a conforming implementation, i.e., whether it produces an acceptable result for every applicable test.

A fundamental goal of conformity assessment (validation) is to promote Ada software portability by ensuring consistent processing of Ada language features as prescribed by [Ada95]. ACVC tests use language features in contexts and idioms expected in production software. While they exercise a wide range of language feature uses, they do not and cannot include examples of all possible feature uses and interactions.

It is important to recognize that the ACVC tests do not guarantee compiler correctness. A compilation system that correctly processes the ACVC tests is not thereby deemed error-free, nor is it thereby deemed capable of correctly processing all software that is submitted to it.

The ACVC tests do not check or report performance parameters (e.g., compile-time capacities or run-time speed). They do not check or report for characteristics such as the presence and effectiveness of compiler optimization. They do not investigate or report compiler or implementation choices in cases where the standard allows options.

2. Changes for ACVC 2.2

Version 2.2 of the ACVC is a maintenance version. Some tests known to have problems have been modified, and others have been permanently removed, but no new tests have been added to the ACVC. (Two new tests appears, consisting of code removed from existing tests for the purpose of resolving incompatibilities between two sub-objectives.) See Appendix A for lists of deleted and modified tests.

3. Configuration Information

3.1 Introduction

This section describes the physical and logical structure of the ACVC delivery, and it describes the test classes, naming conventions used, test program format, test structure, delivery structure, and file format.

ACVC 2.2 is a revision of ACVC 2.1, and has the same delivery structure. The support tools are essentially unchanged, except for updating header comments and version identification.

The test suite does not provide tools or scripts that can be used to manage complete test processing, since such tools are normally site specific.

3.2 Structure

The ACVC 2.2 test software includes test code that exercises specific Ada features, foundation code (used by multiple tests), support code (used to generate test results), and tool code (used to build tools necessary to customize ACVC tests). The suite includes tests for the core language and tests for the Specialized Needs Annexes. Table 1 summarizes the number of tests and files in the ACVC suite.

	Total	Core Tests	SNA Tests	Foundations	Docs	Other
Number of Files	4222	3897	246	44	12	23
Number of Tests	3650	3464	186	0	0	0

Table 1.

The delivery structure of the test suite is described in Section 3.8.

3.2.1 Physical

Table 1 summarizes the number of files that compose ACVC 2.2. In addition to files containing test code proper, the ACVC 2.2 test suite includes various support files:

Others consists of

1 List of all files

14 Code that is referenced by tests

4 Code and data used for preprocessing tests to insert implementation specific information

4 Test routines for reporting code ("CZ" tests)

Note that the number of files containing test code is larger than the number of tests in the ACVC suite because several tests use code included in separate files.

A file name consists of a name plus an extension. Multiple files that contain code used by a single test have related names. File names are the same as that of the test contained in the file when possible. File names conform to MS-DOS naming conventions, therefore they may be shorter than the software name because of file name length restrictions (e.g., enumchek rather than enumcheck). File (and test) names follow conventions that indicate their function in the test suite; naming conventions are explained in Section 3.4. The files are organized into distinct directories and subdirectories based on their function in the test suite. The directory organization is explained in Section 3.8.

The ACVC is available to the general public from an ACAL or on the Internet via FTP:

ftp adaic.org

Login as user **anonymous**, and use your email address as password. Look for an "acvc" subdirectory in a "compilers" directory.

For World-Wide Web users, start with this URL:

http://www.adaic.org

and look for "compilers".

Note that the ACVC files are available in both compressed Unix tar and DOS zipped formats. Section 4.2.2 provides a discussion of techniques to convert these files to a usable format.

Always read the README file.

3.2.2 Logical

Table 1 summarizes the number of tests that check the conformance of an Ada implementation to the core language and conformance to the Specialized Needs Annexes of [Ada95].

Core tests apply to all implementations. Specialized Needs Annex tests are not required for any implementation. Tests for a given Specialized Needs Annex may be processed by implementations that claim implementation of that annex.

In general, no test result depends on the processing or the result of any other test. Exceptions are noted in Section 4.5.2. No annex test depends on the implementation of any other annex, except possibly in cases where one annex specifically depends on another in Ada95 (e.g., no test for the Information Processing Annex uses features from any other annex, however Real Time Annex and Distributed Processing tests may depend on Systems Programming Annex features). [There is a single exception to this rule: see Section 4.6.5.2.] Annex tests may use any core feature.

Tests may be created from one or more compilation units. If a test consists of a single compilation unit (a main subprogram only), the test code will be contained in a single file. Tests built from more than one compilation unit may require multiple files. Moreover, some compilation units, called foundation code, may be used by more than one test. Even in these cases, the resulting tests are strictly independent: if test A and test B use the same foundation code, the results of processing (and running, if appropriate) A have no effect on the results of processing (and running, if appropriate) B. Foundation code is more fully explained in Section 3.2.4.

Tests are named using conventions that provide (limited) information about the test. The test naming conventions are explained in Section 3.4. Each test belongs to a single test class that indicates whether it is or is not an executable test. Test classes are explained in Section 3.3.

In addition to test code and foundation code, there is code on which many or all of the executable tests in the suite depend (e.g., package Report, package ImpDef, package TCTouch). Some of this code must be customized to each implementation. There is also code that must be used to build support tools used to customize the suite of tests to an implementation. The customization process is described in Section 4.3.

3.2.3 Legacy Tests

Many tests check only language features that are common to Ada83 and Ada95. The vast majority of these tests came unmodified from the ACVC 1.12 suite. Some tests were modified to check for the correct implementation of Ada95 rules in cases where language rules changed from Ada83.

3.2.4 Foundation Code

Some tests use foundation code. Foundation code is reusable across multiple tests that are themselves independent of each other. It is intended to be compiled and included in an environment as part of the compilation process of a test. If the test is executable, the foundation code must be bound with all other code for the test prior to execution.

Foundation code is always expected to compile successfully; it is never expected to be run by itself. Foundation code is not, in and of itself, a test, and is therefore not characterized by a test class (see 3.3). One may think of it as providing some utility definitions and routines to a number of different tests. Names of foundation units (and therefore names of files containing foundation code) are distinguished as described in Naming Convention, Section 3.4.

3.2.5 Special Core Tests

This section identifies tests that appear in the Core (since their requirements are enunciated there) but that may be graded as non-supported for implementations not claiming support of certain Specialized Needs Annexes.

3.2.5.1 Annex C Requirements

Section 13 includes implementation advice paragraphs. The ACVC does not require implementations to conform to those paragraphs unless they claim support for Annex C, Systems Programming (cf. C.2(2): "The implementation shall support at least the functionality defined by the recommended levels of support in Section 13.")

Tests that check conformance to the implementation advice are listed below:

CD10001	CD30005	CD40001
CD20001	CD33001	CD72A01
CD30001	CD33002	CD72A02
CD30002	CD30004	CD90001
CD30003		

Implementations that claim support for Annex C are required to process and pass the tests listed above. (Passing of CD10001 is required only if the implementation also claims support for Annex F.)

Implementations that do not claim support for the appropriate Annexes are still required to process these tests. Such implementations may reject the lines marked with the special comment "-- ANX-C RQMT", in which case the test will be graded as "unsupported". If an implementation accepts such lines in one of these tests, then the test must be bound (linked) and executed, with a passed or not_applicable result.

3.2.5.2 Annex F Requirements

There are also Core tests that may be rejected by implementations not claiming support for Annex F. These tests require support for decimal fixed point types; such support is not required except for implementations claiming support for Annex F.

Core tests that require support for decimal fixed point types are listed below:

CD10001
CXAA010

Implementations that claim support for Annex F are required to process and pass the tests listed above. (Note that passing CD10001 is required only if the implementation also claims support for Annex C.)

Implementations that do not claim support for the appropriate Annexes are still required to process these tests. Such implementations may reject the lines marked with the special comment "-- ANX-F RQMT", in which case the test will be graded as "unsupported". If an implementation accepts all such lines in one of these tests, then the test must be bound (linked) and executed, with a passed or not_applicable result.

3.2.6 Foreign Language Code

Several tests for Annex B features (and one Section 13 test) include files containing non-Ada code (Fortran, C, Cobol). These tests must be compiled, bound, and run by implementations that support foreign language interfaces to the respective non-Ada language. The foreign language code uses only the most basic language semantics and should be compilable by all Fortran, C, and Cobol compilers, respectively. In cases where a foreign language does not accept the code as provided, modifications are allowable. See Section 4.3.6.

Files that contain foreign code are identified by a special file extension. See Section 3.4.2.

The tests which include Fortran code are: CXB5004 and CXB5005

The tests which include C code are: CXB3013 and CD30005

The test which includes Cobol code is: CXB4009

3.3 Test Classes

There are six different classes of ACVC tests, reflecting different testing requirements of language conformity testing. Each test belongs to exactly one of the six classes, and its membership is encoded in the test name, as explained later. The purpose and nature of each test category is explained below. The test classifications provide an initial indication of the criteria that are used to determine whether a test has been passed or failed.

3.3.1 Class A

Class A tests check for acceptance (compilation) of language constructs that are expected to compile without error.

An implementation passes a class A test if the test compiles, binds, and executes reporting "PASSED". Any other behavior is a failure.

Only legacy tests are included in this class.

3.3.2 Class B

Class B tests check that illegal constructs are recognized and treated as fatal errors. They are not expected to successfully compile, bind, or execute. Lines that contain errors are marked "-- ERROR:", and generally include a brief description of the illegality on the same or following line. (The flag includes a final ":" so that search programs can easily distinguish it from other occurrences of the word "error" in the test code or documentation.) Some tests also mark some lines as "-- OK", indicating that the line must **not** be flagged as an error. Lines so marked are often, but not always, constructs that were errors in Ada83 but are correct in Ada95.

An implementation passes a class B test if each indicated error in the test is detected and reported, and no other errors are reported. The test fails if one or more of the indicated errors are not reported, or if an error is reported that cannot be associated with one of the indicated errors. If the test structure is such that a compiler cannot recover sufficiently to identify all errors, it may be permissible to "split" the test program into separate units for re-processing (see Section 4.3.6 for instructions on modifying tests).

In some cases and for some constructs, compilers may adopt various error handling and reporting strategies. In cases where the test designers determined that an error might or might not be reported, but that an error report would be appropriate, the line is marked with "-- OPTIONAL ERROR:" or a similar phrase. In such cases, an implementation is allowed to report an error or fail to report an error without affecting the final grade of the test.

3.3.3 Class C

Class C tests check that executable constructs are implemented correctly and produce expected results. These tests are expected to compile, bind, execute and report "PASSED" or "NOT-APPLICABLE". Each class C test reports "PASSED", "NOT-APPLICABLE", or "FAILED" based on the results of the conditions tested.

An implementation passes a class C test if it compiles, binds, executes, and reports "PASSED". It fails if it does not successfully compile or bind, if it fails to complete execution (hangs or crashes), if the reported result is "FAILED", or if it does not produce a complete output report.

The tests CZ1101A, CZ1102A, CZ1103A, and CZ00004 are treated separately, as described in Section 4.4.2.

3.3.4 Class D

Class D tests check that implementations perform exact arithmetic on large literal numbers. These tests are expected to compile, bind, execute and report "PASSED". Each test reports "PASSED" or "FAILED" based on the conditions tested. Some implementations may report errors at compile time for some of them, if the literal numbers exceed compiler limits.

An implementation passes a class D test if it compiles, binds, executes, and reports “PASSED”. It passes if the compiler issues an appropriate error message because a capacity limit has been exceeded. It fails if does not report “PASSED” unless a capacity limits is exceeded. It fails if it does not successfully compile (subject to the above caveat) or bind, if it fails to complete execution (hangs or crashes), if the reported result is "FAILED", or if it does not produce an output report or only partially produces one.

Only legacy tests are included in this class.

3.3.5 Class E

Class E tests check for constructs that may require inspection to verify. They have special grading criteria that are stated within the test source. They are generally expected to compile, bind and execute successfully, but some implementations may report errors at compile time for some tests. The "TENTATIVELY PASSED" message indicates special conditions that must be checked to determine whether the test is passed.

An implementation passes a class E test if it reports "TENTATIVELY PASSED", *and* the special conditions noted in the test are satisfied. It also passes if there is a compile time error reported that satisfies the special conditions. Class E tests fail if the grading criteria in the test source are not satisfied, or if they fail to complete execution (hang or crash), if the reported result is "FAILED", or if they do not produce a complete output report.

Only legacy tests are included in this class.

3.3.6 Class L

Class L tests check that all library unit dependencies within a program are satisfied before the program can be bound and executed, that circularity among units is detected, or that pragmas that apply to an entire partition are correctly processed. These tests are normally expected to compile successfully but not to bind or execute. Some implementations may report errors at compile time; potentially illegal constructs are flagged with "-- ERROR:" Some class L tests indicate where bind errors are expected. Successful processing **does not** require that a binder match error messages with these indications.

An implementation passes a class L test if does **not** successfully complete the bind phase. It passes a class L test if it detects an error and issues a compile time error message. It fails if the test successfully binds and/or begins execution. An L test need not report "FAILED" (although many do if they execute).

This result is acceptable if the test has flagged errors and the compiler has identified them.

As with B-tests, the test designers determined that some constructs may or may not generate an error report, and that either behavior would be appropriate. Such lines are marked with "-- OPTIONAL ERROR:" In such cases, an implementation is allowed to report an error or fail to report an error. If an error is reported at compile time, the binder need not be invoked. If no errors are reported at compile time, the binder must be invoked and must not successfully complete the bind phase (as indicated by the inability to begin execution).

3.3.7 Foundation Code

Files containing foundation code are named using the regular test name conventions (see Section 3.4). It may appear from their names that they represent class F tests. There is no such test class. Foundation code is only used to build other tests, so foundation units are not graded. However, if a foundation unit fails to compile, then the tests that depend on it cannot be compiled, and therefore will be graded as failed.

3.3.8 Specialized Needs Annex Tests

Specialized Needs Annex tests have no separate classifications and are classified in the same way as all other tests. There are Class B, Class C, and Class L SNA tests.

3.4 Naming Convention

This section describes the naming conventions used in ACVC 2.2, specifically as they apply to files. All file names are of the form <name>.<type>, where <type> is a one, two, or three character extension. File names indicate test class, compilation order (if applicable), and whether the test is implementation dependent or requires customization. When a test is included in a single file, <name> duplicates the test name. The same is true of a foundation. In multiple file tests, the first 7 characters of the file <name> are normally the same as the name of the test, however in some cases, the structure of the test requires that the file name be different from the Ada unit. The application of the conventions to tests is straightforward.

There are two different but similar naming conventions used in ACVC 2.2. Legacy tests use the naming conventions of previous ACVC versions. Tests new since ACVC 1.12 use the new convention. The conventions are consistently distinguishable at the 7th character of the name: legacy names have a letter in the 7th position, whereas newer names have a digit.

3.4.1 Legacy Naming

The name of a legacy test is composed of seven or eight characters. Each character position serves a specific purpose as described in the table below. The first column identifies the character position(s) starting from the left, the second column gives the kind of character allowed, and the third gives the corresponding meaning:

Position		
1	Letter	Test class (cf. Section 3.3)
2	Hexadecimal	AIG chapter containing the test objective
3	Hexadecimal	Section within the above AIG chapter
4	Alpha-numeric	Sub-section of the above AIG section
5-6	Decimal	Number of the test objective within the above sub-section
7	Letter	Letter identifier of the sub-objective of the above objective.
8	Alpha-numeric	*optional* - Compilation sequence identifier -- indicates the compilation order of multiple files that make up a single test. This position is used only if the test comprises multiple files.

The convention is illustrated in Figure 1.

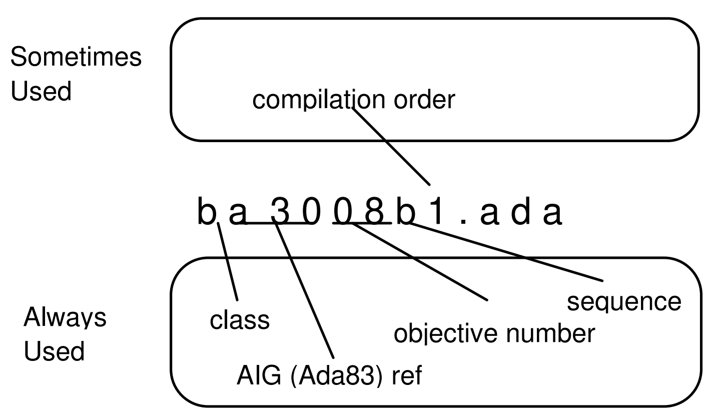

Figure 1. Legacy File Name Convention

In multiple file tests, the intended order of compilation is indicated by a numeral at position 8. The first file to be compiled has '0', the second has '1', and so forth.

Note: The use of a ninth character ('m') to indicate the file containing the main subprogram has been discontinued. The following table lists the files containing the main subprograms of the legacy multiple file tests.

AD7001C0	BA1010J0	BA3006A6	CA2004A0
AD7001D0	BA1010K0	BA3006B4	CA2007A0
B38103C3	BA1010L0	C38108C1	CA2008A0
B38103E0	BA1010M0	C38108D0	CA5003A6
B63009C3	BA1010N0	C39006C0	CA5003B5
B73004B0	BA1010P0	C39006F3	CA5004B1
B83003B0	BA1010Q0	C64005D0	CC3019B2
B83004B0	BA1011B0	C83022G0	CC3019C2
B83004C2	BA1011C0	C83024E1	LA5001A7
B83004D0	BA1020A0	C83F01C2	LA5007A1
B83024F0	BA1020B6	C83F01D0	LA5007B1
B83E01E0	BA1020C0	C83F03C2	LA5007C1
B83E01F0	BA1020F2	C83F03D0	LA5007D1
B86001A1	BA1101B0	C86004B2	LA5007E1
B95020B2	BA1101C2	C86004C2	LA5007F1
BA1001A0	BA1109A2	CA1011A6	LA5007G1
BA1010A0	BA1110A1	CA1012B4	LA5008A1
BA1010B0	BA2001F0	CA1013A6	LA5008B1
BA1010C0	BA2003B0	CA1014A0	LA5008C1
BA1010D0	BA2011A1	CA1020E3	LA5008D1
BA1010E0	BA3001A0	CA1022A6	LA5008E1
BA1010F0	BA3001B0	CA1102A2	LA5008F1
BA1010G0	BA3001C0	CA2001H3	LA5008G1
BA1010H0	BA3001E0	CA2002A0	
BA1010I0	BA3001F0	CA2003A0	

The file name extension is three characters long. There are four extensions:

.ada A file that contains only Ada code. It does not require any pre-processing to create a compilable test. It will be submitted directly to the implementation for determination of test results. All implementations must correctly process these tests.

.dep A file that has a test involving implementation-dependent features of the language. These tests may not apply to all implementations.

.tst A file that has "code" that is not quite Ada; it contains "macro" symbols to be replaced by implementation-dependent values, and it must be customized (macro expanded) to prepare it for compilation (see Section 4.3.2). Once customized, the resulting test must be processed as indicated by its class.

.adt A file that has been modified by the macro processor. It contains only Ada code and may be submitted to the implementation for results. All implementations must correctly process these tests. There are no files in the ACVC distribution with this extension; they are only produced as the output of the macro processor.

Tests developed since ACVC 1.12 use different file name extensions.

> Note that legacy tests have **not** been renamed for ACVC 2.2. Since [Ada95] includes some organizational differences from [Ada83], this means that the name of a legacy test sometimes will not correspond to the clause of [Ada95] in which the tested feature is described.

3.4.2 ACVC 2.2 Naming

The name of an Ada95 test is composed of seven or eight characters. Foundation code has a name composed of seven characters. The use of each character position is described below. The first column indicates the character position(s) starting from the left, and the second column indicates the kind of character allowed, and the third column gives the corresponding meaning:

Position		
1	Letter	Test class; foundations are marked 'F'
2	Alpha-numeric	If other than an 'x', the section of [Ada95] describing the feature under test. An 'x' indicates that the test includes one or more features from an annex of [Ada95]
3	Alpha-numeric	Core clause or annex letter identifier (either core or Specialized Needs Annex)
4	Hexadecimal	Sub-clause (if a core test), or clause (if an annex test)
5	Alpha-numeric	Foundation identifier (alphabetic, unless no foundation is required, in which case a '0')
6-7	Decimal	Sequence number of this test in a series of tests for the same clause; foundation code will have "00" .
8	Alpha-numeric	*optional* - Compilation sequence identifier -- indicates the suggested or required compilation order of multiple files that make up a single test (0 is compiled first). This position is used only if the test comprises multiple files.

This convention is illustrated in Figure 2.

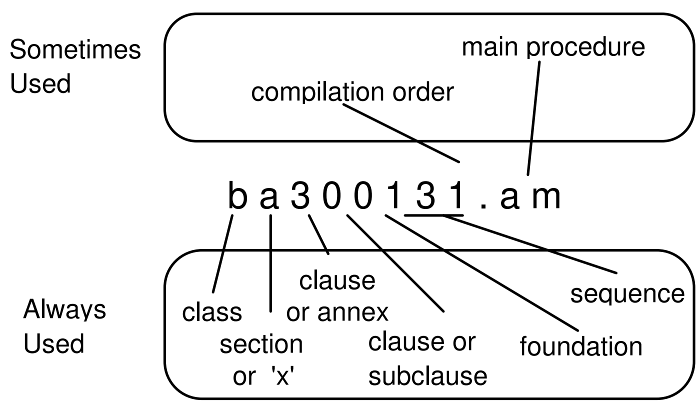

Figure 2. Naming convention in ACVC 2.2

The file name extension is a one or two character file name extension. There are six extensions:

`.a`	A file that contains only Ada code (except for configuration pragmas in the case of some Specialized Needs Annex tests). It does not require any processing to prepare it for compilation (unless configuration pragmas must be handled separately). It is normally submitted directly to the implementation for determination of test results.
`.am`	A file that contains the main subprogram for a multi-file test. Generally, this extension is used for one only file of a test. In rare cases (some Annex E tests), a multi-file test may have more than one file containing a "main" subprogram; in such cases, the correct testing procedure is described in the Special Requirements section of the test prologue.
`.aw`	A file that has "code" that is not quite Ada; it contains one or more designated strings that must be replaced by a character from the upper half of ISO8850-1 (Latin-1) (see Section 4.3.3). The resulting test must be compiled and run as all other class C tests.
`.ftn`	A file that contains Fortran language code and must be compiled by a Fortran compiler. These files are used by tests that check a foreign language interface to Fortran.
`.c`	A file that contains C language code and must be compiled by a C compiler. These files are used by tests that check a foreign language interface to C.
`.cbl`	A file that contains Cobol language code and must be compiled by a Cobol compiler. These files are used by tests that check a foreign language interface to Cobol.

A test that depends on foundation code has an alphabetic character in the fifth position of its name. The required foundation will have the same characters in the second through fifth positions of its name. For example, C**123A**xx depends on F**123A**00.

3.4.3 Multiple File Tests

When tests are contained in multiple files (i.e., compilation units are contained in different files), the file names are related. The first seven positions of the names of all the files (other than foundation files) comprised by a single test will be identical. The eighth position will provide a distinguishing alpha-numeric which indicates the suggested or required compilation order. In legacy tests, the ninth position is used to indicate the file that contains the main program. For newer tests, the extension ".am" indicates the file with the main program.

All tests apply the convention of naming the main subprogram the same as the file (excluding the file extension) plus the letter 'm' (for legacy tests only). For example, the legacy test, C39006F, is contained in four files, named `c39006f0.ada`, `c39006f1.ada`, `c39006f2.ada`, and `c39006f3m.ada`. The main sub-program of the test is contained in `c39006f3.ada` and is named "C390006F3M". The test C390006 is also contained in four files, named `c3900060.a`, `c3900061.a`, `c3900062.a`, and `c3900063.am`. The main subprogram of the test is contained in `c3900063.am` and is named "C3900063".

There are a small number of Specialized Needs Annex tests for the Distributed Processing Annex that require two active partitions and have two main subprograms. These tests have two files with the .am extension to signify the location of the (multiple) main subprograms.

3.5 Test Program Format

Each test file is composed of a test prologue, documenting the test, and the test code proper. All prologue lines are marked as comments. [The prologue in files containing non-Ada code is marked according to the comment conventions of the foreign language.]

The prologue for all tests is based on that of legacy tests. Legacy tests are generally, but not entirely, consistent in their use of the prologue. The format of the prologue between test files and foundation files is slightly different.

The general format of the prologue is as follows:

<file name> - The distribution name of the file containing this prologue.

DISCLAIMER - Use restrictions for ACVC tests; included in all tests.

OBJECTIVE - A statement of the test objective; included in all tests.

TEST DESCRIPTION - A short description of the design or strategy of the test or other pertinent information. Included in all newer tests but not generally included in legacy tests.

SPECIAL REQUIREMENTS - *optional* - Included if the test has any special requirements for processing. Normally, this section will be found only in Specialized Needs Annex tests. For example, an Annex E test may check for the correct implementation of partitions; the requirements for test partitioning and what to use as a main subprogram in each partition would be documented in this section.

TEST FILES - *optional* - Included if the test depends on multiple files; identifies the component files of a multi-file test.

APPLICABILITY CRITERIA - *optional* - Specifies the conditions under which the test can be ruled inapplicable.

PASS/FAIL CRITERIA - *optional* - Explains how to interpret compilation, binding, and/or run-time results for grading the test.

MACRO SUBSTITUTIONS - *optional* - Identifies the macro symbol(s) in the file that must be replaced and provides a brief description of what the replacement(s) represent.

CHANGE HISTORY- History of the test file. Included in all tests.

All tests have the line immediately after the disclaimer marked "--*". The newer tests have the line after the last prologue line (before the first line of executable code) marked "--!" No other comment lines are marked with those conventions, so the next line after the disclaimer and the first line of code may be found quickly with an editor search.

Some tests are composed of multiple files (other than foundation code). Rather than repeating the complete prologue in each file, an alternate approach has been used. The file containing the main program has the complete prologue; the other, related files have those sections that apply to files (TEST FILES, CHANGE HISTORY) and refer to the main file for the other sections.

3.6 General Standards

ACVC tests were developed to a general set of standards. To promote a variety of code styles and usage idioms in the tests, standards were not necessarily rigorously enforced but were used as guidelines for test writers. A maximum line length of 79 characters was used to enhance electronic distribution of tests (except when specific testing requirements dictated otherwise, usually in .dep and .tst files). Tests tend to be about 120 executable lines long, though many tests deviate from this norm (either longer or shorter) to achieve a design that focuses on the objective and a readable, maintainable test. Sometimes complex objectives have been divided into sub-objectives to achieve complete coverage in comprehensible, maintainable tests. Some tests check multiple sub-objectives; in other cases, sub-objectives are checked in separate tests.

Legacy tests use only the basic 55-character set (26 capital letters, 10 digits, and 19 punctuation marks). Unless there is a specific test requirement, numeric values are in the range (-2048..2047), which can be represented in 12 bits. Numeric values are generally in the range (-128..127). Tests new to ACVC 2.x use both upper and lower case letters and may use larger numeric values (but within the range (-65536..65535) except in rare cases).

Legacy tests tend to use as few Ada features as necessary to write a self-checking executable test that can be read and maintained. Newer tests tend to exhibit a usage oriented style, employing a rich assortment and interaction of features and exemplifying the kind of code styles and idioms that compilers may encounter in practice.

In the newer tests, Ada reserved words are entirely in lower case. Identifiers normally have their initial letter capitalized. Every attempt has been made to choose meaningful identifiers. In B class tests, identifier names often provide a clue to the specific case or situation under test. In C class tests, identifiers are normally chosen to help document the test design or the intent of the code.

The newer executable tests generally provide some visual separation of those test elements which focus on conformance issues from those which govern the flow of a test. For example, there is frequently a need to establish preconditions for a test and examine post-conditions after a section of test code has executed. To distinguish between constructs (types, objects, etc.) that are part of the test code and those that are artifacts of the testing process (e.g., pre-, post-conditions), the latter have "TC_" prefixed to the identifier name. This prefix is shorthand for "Test_Control".

3.7 Test Structure

Executable tests (class A, C, D, E) generally use the following format:

```
with Report;
procedure Testname is
     <declarations>
begin
     Report.Test ("Testname", "Description ...");
     ...
     <test situation yielding result>
     if Post_Condition /= Correct_Value then
          Report.Failed ("Reason");
     end if;
     ...
     Report.Result;
end Testname;
```

The initial call to Report.Test prints the test objective using Text_IO output. After each section of test code, there is normally a check of post conditions. The **if** statement in this skeleton is such a check; unexpected results produce a call to Report.Failed. The sequence of test code / check of results may be repeated several times in a single test. Finally, there is a call to Report.Result that will print the test result to Text_IO output. Often, but not always, this structure in enclosed in a declare block.

One or more calls to Report.Failed will report a result of "FAILED" and a brief suggestion of the likely reason for that result.

More complex tests may include calls to Report.Failed in the code other than in the main program, and therefore exhibit the following format for the main procedure:

```
with Report;
procedure Testname is
     <declarations>
begin
     Report.Test ("Testname", "Description ...");
     ...
     Subtest_Call;
     ...
     Report.Result;
end Testname;
```

Fail conditions are detected in subprograms (or tasks) and Report.Failed is called within them.

Occasionally, as a test is running, it will determine that it is not applicable. In such a case, it will call Report.Not_Applicable which will report a result of "NOT_APPLICABLE" (unless there is also a call to Report.Failed).

Often, a test calls one of the functions Report.Ident_Int or Report.Ident_Bool to obtain a value that could be provided as a literal. These functions are intended to prevent optimizers from eliminating certain sections of test code. The ACVC suite has no intention of trying to discourage the application of optimizer technology, however satisfactory testing of language features often requires the presence and execution of specific lines of test code. Report.Ident_Int and Report.Ident_Bool are structured so that they can be modified when needed to defeat optimizer advances.

Class B tests may be structured differently. Since they are not executable, they normally do not include calls to Report.Test or Report.Result (since those lines of code would have no output effect). Instead, intentional errors are coded that invoke specific legality rules. The source code includes comments that document expected compiler results. Legal constructs may also be included in B class tests. Constructs that are allowed by the legality rules are marked "-- OK"; constructs that are disallowed are marked "-- ERROR:". There is usually a brief indication of the nature of an intentional error on the same line or the line following a comment. The indications of expected results are approximately right justified to the code file margin, about column 79, for quick visual identification.

Class L tests are multifile tests with illegalities that should be detected at bind time. They are generally structured like class C tests, often with calls to Report.Test and Report.Result, but they are not expected to execute.

3.8 Delivery Directory Structure

The delivery of ACVC tests is structured into a directory tree that reflects the organization of the test suite and support code. See Fig. 3.

The top level directory contains the support subdirectory, the docs subdirectory, and a subdirectory for each major grouping of tests. The support subdirectory contains all support packages (Report, ImpDef, TCTouch) and the source code for all test processing tools (Macro expander, Wide Character processor). Each of the other subdirectories contains all tests that begin with the indicated prefix. For example, all of the B2* tests are in the b2 subdirectory; all of the CXH* tests are in the cxh subdirectory. Note that all of the A* tests are in the a directory, all of the D* tests are included in the d subdirectory, and all of the E* tests are included in the e subdirectory. The l directory contains the L tests for the core; other L tests are in directories named with three letters, indicating the class (l) and the Specialized Needs Annex to which the tests apply.

Subdirectories that would be empty are not stubbed.

Figure 3 sketches this scheme, but does not show complete detail. A list of all subdirectories is included in Section 4.2.2.

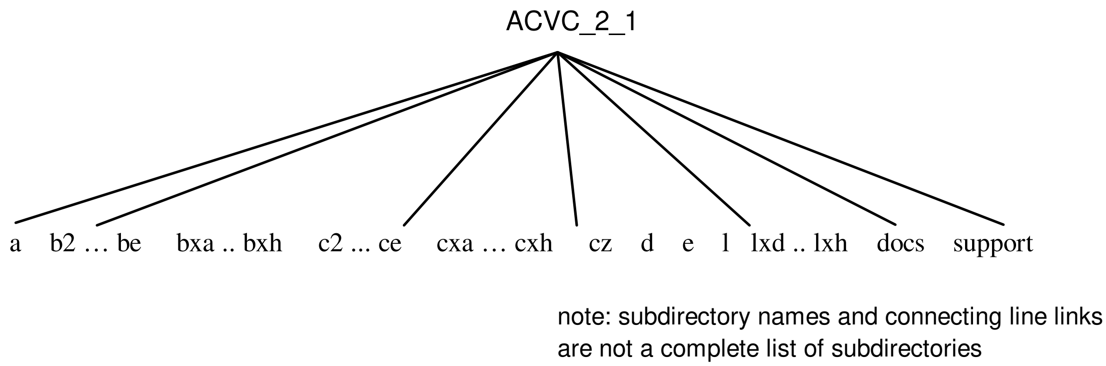

Figure 3. Delivery Directory Structure

3.9 File Format

To conserve space, all files in the delivered ACVC 2.2 (including test files, foundation files, and support files) have been compressed, except for three informational files. Decompressed files (see Section 4.2.2) use only ASCII characters. No formatting control characters, rulers or other information intended for word processors or editors is included in the files.

Files with the .zip extension have been compressed using a DOS zip utility; files with the .Z extension have been first put in Unix tar format and then compressed with Unix compress.

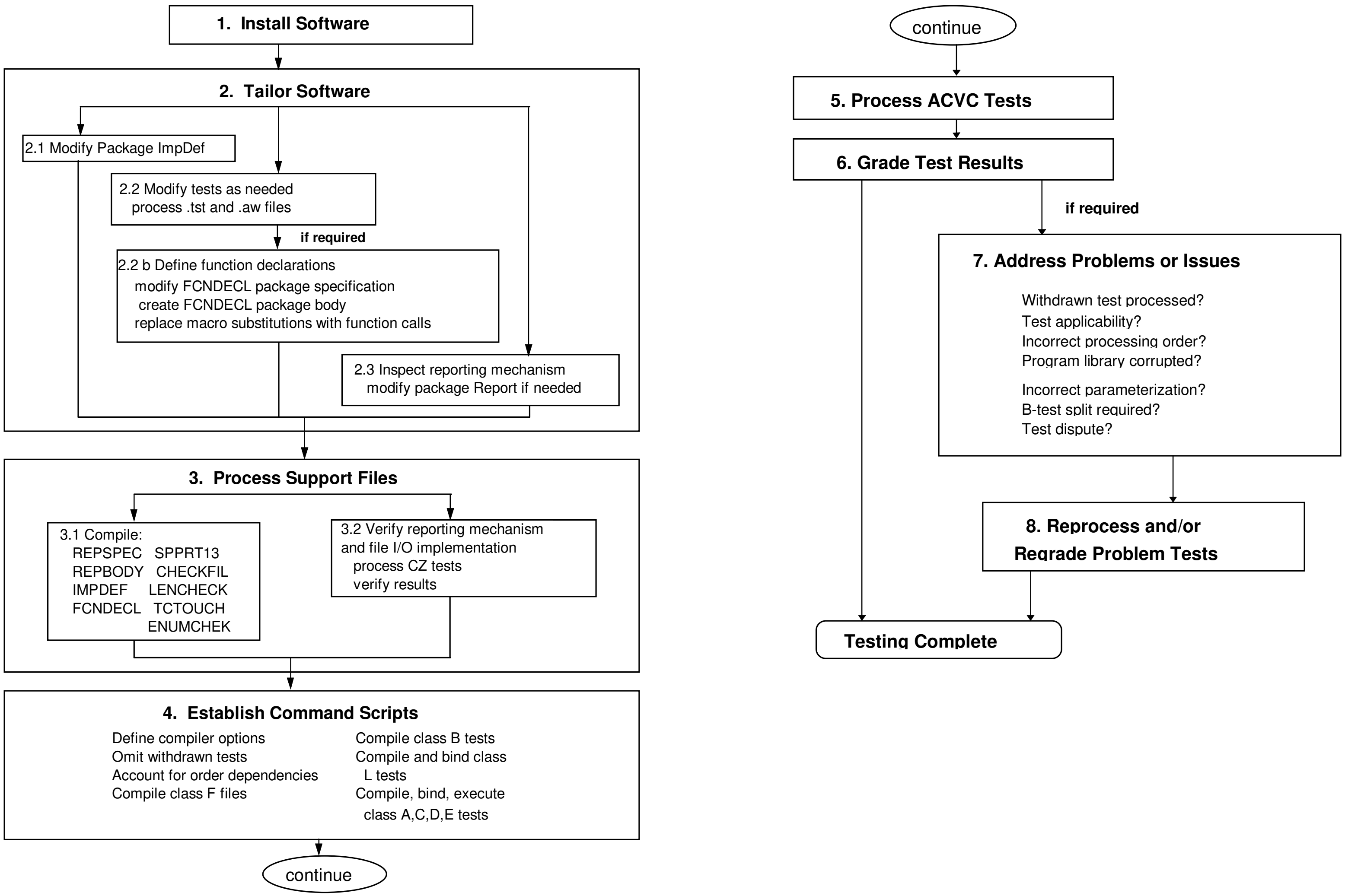

Figure 4 (Cont.) Using the ACVC

4. Using The ACVC

4.1 Introduction

There are eight major steps involved in using the ACVC test suite; two of them are sometimes not required. The steps are: installing the software, tailoring the software, processing the support files, establishing command scripts, processing the ACVC tests, grading the test results, addressing problems (if necessary), and reprocessing problem tests (if necessary). The first six of these tasks must be completed successfully to accomplish a test run. The first four normally need be completed only once for each ACVC release. Each step is explained in the following sections. The flow from one to the next is illustrated in figure 4.

4.2 Installation of the ACVC Test Suite

The ACVC test suite must be unloaded from the delivery medium or downloaded from a delivery site before it can be unpacked, customized for an implementation, run, and graded.

4.2.1 Contents of the ACVC Delivery

The delivery consists of 9 archives (sets of compressed files) or a single compressed tar file. Each archive or compressed tar file contains compressed versions of ACVC software (test, foundation, and/or support code) structured into a directory tree. Files must be extracted from the archives. Archive contents are described later in this section.

The following uncompressed files are included in the delivery:

```
README
acvc2_2.lst
MULNAMES.TXT
ug.txt
```

The file named README contains a description of the distribution and extraction procedures. The file named `acvc2_1.lst` contains the names of all test files (including foundations) and support files in ACVC 2.2. The file named MULNAMES.TXT lists the files containing the main units of legacy multi-file tests. The file named ug.txt is an ASCII version of this User’s Guide. Since none of these files is compressed, they may be read in an editor, listed to the screen of a PC, or printed to standard print output. All are ASCII files.

It is anticipated that two other uncompressed ASCII files will appear in the Internet distribution directory, as follows:

modifiedyymmdd.txt

withdrawnyymmdd.txt

The file whose name begins with "modified" is a list of permitted and required test modifications as approved by the ACAA. The "yymmdd" portion of the name is replaced by digits representing the date of issue. The ACAA approves modifications when tests are found to have flaws that are amenable to correction or deletion by making minor modifications in the code.

The file whose name begins with "withdrawn" is a list of tests that have been withdrawn from the test suite (not to be used in subsequent validation testing). The "yymmdd" portion of the name is replaced by digits representing the data of issue. The ACAA withdraws a test when it is found to have flaws so serious that extensive modifications would be required to salvage the test.

4.2.2 Guide to Decompressing Files

The ACVC files are provided in two forms: compressed in zip format and compressed in Unix compress format. Zipped files are included in 9 zip archives (files) with the file extension .zip. The Unix compressed file, with extenxion .Z, contains a Unix tar file. This section provides generic instructions for uncompressing them. These instructions are not the only ways to uncompress the files; sophisticated users may wish to use their own procedures.

If the instructions below are used, the following subdirectories will have been created and populated with test files after all uncompression:

./acvc2_2/a	./acvc2_2/c4	./acvc2_2/bxd
./acvc2_2/b2	./acvc2_2/c5	./acvc2_2/bxe
./acvc2_2/b3	./acvc2_2/c6	./acvc2_2/bxf
./acvc2_2b4	./acvc2_2/c7	./acvc2_2/bxg
./acvc2_2/b5	./acvc2_2/c8	./acvc2_2/bxh
./acvc2_2/b6	./acvc2_2/c9	./acvc2_2/cxc
./acvc2_2/b7	./acvc2_2/ca	./acvc2_2/cxd
./acvc2_2/b8	./acvc2_2/cb	./acvc2_2/cxe
./acvc2_2/b9	./acvc2_2/cc	./acvc2_2/cxf
./acvc2_2/ba	./acvc2_2/cd	./acvc2_2/cxg
./acvc2_2/bb	./acvc2_2/ce	./acvc2_2/cxh
./acvc2_2/bc	./acvc2_2/cz	./acvc2_2/lxd
./acvc2_2/bd	./acvc2_2/d	./acvc2_2/lxe
./acvc2_2/be	./acvc2_2/e	./acvc2_2/lxh
./acvc2_2/bxa	./acvc2_2/l	./acvc2_2/docs
./acvc2_2/bxb	./acvc2_2/cxa	./acvc2_2/support
./acvc2_2/c2	./acvc2_2/cxb	
./acvc2_2/c3	./acvc2_2/bxc	

Note that the names are given here in all lowercase; some systems may create lowercase names. The path separator, shown here as '/', may also differ.

You may wish to copy the three uncompressed files into the acvc2_2 directory.

4.2.2.1 Decompressing Zipped Files

All ACVC files other than those noted in Section 4.2.1 have been compressed (zipped) into compressed archives (zip-files) that have the MS-DOS file extension "`.zip`". A DOS utility was used to compress them. They must be decompressed before they can be further processed. All ACVC 2.2 files may be decompressed using the following steps. Approximately 23 MB of free space on a DOS machine hard drive will be required to accomplish the decompression using this technique.

Copy each archive (file with `.zip` extension) to the hard disk. Decompress it using the default settings (no options). For example, if the archive name is `acvc2.zip`, type

unzip acvc2 or **pkunzip -r acvc2**

The files were compressed on a Unix system, where <LF> is used as a line terminator. If they are decompressed on a system that uses <CR><LF> as a line terminator, the above unzip command should be modified to

unzip -a acvc2

After all files have been extracted from the archive, delete the archive file from the hard disk if you wish to conserve space.

As it decompresses files, unzip will restore the directory structure of the files, creating all needed subdirectories.

Some users may prefer to work with ACVC files in an alternate directory structure or none at all. If the unzip utility is invoked with the "-j" option, all files in the archive will be decompressed and placed in the local working directory. In other words, none of the above subdirectories will be created. Since there are too many ACVC files to fit into a root DOS directory, if you wish to put all files in a single directory, you *must* first create a subdirectory (e.g., **mkdir \ACVC**) and unzip all archives there.

4.2.2.2 Decompressing Unix Compress Files

All ACVC files have been included in a Unix tar format file and then compressed using the Unix compress utility. To create a set of ACVC files, first copy the compressed file `acvc2_2.tar.Z` from the distribution source to a hard drive. Uncompress the file with the Unix command

uncompress acvc2_2.tar.Z

(note that particular Unix implementations may have different formats or require specific qualifiers.) After the acvc file has been uncompressed, it must be untarred. Move to the directory where you want the ACVC2_2 directory to be created and then untar the ACVC file

tar -xvf <path>/acvc2_2.tar

where <path> is the location of the uncompressed tar file.

Please note that these are generic instructions and may need to be customized or modified for specific systems.

4.2.3 Files With Non-Graphic Characters

Four ACVC test files contain non-graphic (control) characters that may be lost or corrupted in the file transfer and decompression process. The user must ensure that the proper characters are restored as necessary. The following paragraphs describe the four tests.

4.2.3.1 A22006C

This test checks that format effectors can appear at the beginning of a compilation. At the beginning of the file, the first line is empty (indicated by the system's end-of-line marker, which may be a sequence of one or more characters or may be indicated by some other means). The second line contains 20 characters: 6 control characters followed by the comment delimiter, a space, and the file name (A22006C.ADA). The control characters are:

Common Name	Ada Name	ASCII Value	
		Decimal	Hex
Carriage return	ASCII.CR	13	0D
Carriage return	ASCII.CR	13	0D
Vertical tab	ASCII.VT	11	0B
Line feed	ASCII.LF	10	0A
Line feed	ASCII.LF	10	0A
Form feed	ASCII.FF	12	0C

4.2.3.2 B25002A

This test checks that control characters (other than format effectors) are not permitted in character literals. The expected characters are documented in source code comments, using the customary 2- or 3-letter mnemonics. The 28 characters are used in their ASCII order, and have ASCII values 0 through 8, 14 through 31, and 127.

4.2.3.3 B25002B

This test checks that the five format effector characters cannot be used in character literals. There are two groups of code containing the illegal characters; in each group, the characters appear in the order given below:

Common Name	Ada Name	ASCII Value	
		Decimal	Hex
Horizontal tab	ASCII.HT	9	09
Vertical tab	ASCII.VT	11	0B
Carriage return	ASCII.CR	13	0D
Line feed	ASCII.LF	10	0A
Form feed	ASCII.FF	12	0C

4.2.3.4 B26005A

This test checks the illegality of using control characters in string literals. Each string literal (ASCII codes 0 through 31 and 127) is used once, and the uses appear in ASCII order. Each use is also documented in a source code comment, which identifies the character by its common 2- or 3-character mnemonic.

4.3 Tailoring the ACVC Test Suite

There are some files in the delivery that require modification before ACVC 2.2 is ready for processing by an Ada implementation. Package ImpDef (impdef.a) must be edited to include values suitable for proper testing of an implementation if the defaults are not acceptable. ImpDef is a package that is new to the 2.X suite, and all users will have to do this modification. The macros.dfs file must similarly be edited to include values suitable for testing. This file is slightly different from previous ACVC suites, so all users will have to modify it, but most changes can be retained from previous versions. All .tst files (including package Spprt13 (spprt13s.tst)) must have their macro symbols replaced by implementation specific values. A body for FcnDecl (fcndecl.ada) must be provided if necessary. Finally, Package Report (repbody.ada) must be modified if necessary; previous modifications can generally be carried forward. The required customization is described in the following sections.

4.3.1 ImpDef Customization

All implementations *must* customize impdef.a for ACVC 2.2 unless they wish to rely on the defaults provided. ImpDef *must* be part of the environment whenever a test that depends on it is processed. Note that in ACVC 2.2, ImpDef uses child libraries for the Specialized Needs Annexes. The only ImpDef children that need be modified are those associated with the SNAs that the implementation uses for validation.

ACVC tests use the entities in ImpDef to control test execution. Much of the information in ImpDef relates to the timing of running code; for example, the minimum time required to allow a task switch may be used by a test as a parameter to a delay statement. The time to use is obtained as an ImpDef constant.

impdef.a was added as a new feature to ACVC 2.0 suite. It is related to macro.dfs in that it must be customized with values specific to an implementation and ACVC tests will rely on these values. ImpDef is different in the following respects:

- Defaults are provided. Some implementations may be able to rely entirely on the default values and subprograms, so no customization would be necessary.
- Some implementations may choose to provide bodies for one procedure and/or one function. Bodies so provided must satisfy requirements stated in ImpDef.
- It is not used for macro expansion of tests. Instead, ImpDef must be available at compile time (i.e., included in the environment) for tests that rely upon it.

There are child packages of ImpDef for each of the Specialized Needs Annexes. An implementation that uses one or more of the Specialized Needs Annexes in its validation must customize the associated ImpDef child packages (or rely on their defaults) and must set the appropriate booleans in `impdef.a`. Specific instructions for the values required by ImpDef and its children are included in `impdef.a`, `impdefc.a`, `impdefd.a`, `impdefe.a`, `impdefg.a`, and `impdefh.a`. (Note that impdefc, for example, refers to Annex C.) A copy of ImpDef is included in Appendix B.

4.3.2 Macro Defs Customization

> The details of macro.dfs have not changed from ACVC 2.1 to ACVC 2.2. A version of macro.dfs that was tailored for ACVC 2.1 should be valid for ACVC 2.2 unless some implementation characteristics have changed.

Tests in files with the extension ".tst" contain symbols that represent implementation dependent values. The symbols are identifiers with a initial dollar sign ('$'). Each symbol must be replaced with an appropriate textual value to make the tests compilable.

The Macrosub program distributed with the ACVC can automatically perform the required substitutions. This program reads the replacement values for the symbols from the file macro.dfs and edits all the ".tst" tests in the suite to make the needed changes. It writes the resulting, compilable programs into files with the same name as the original but with the extension .adt. A sample macro.dfs is included with the ACVC, and is included in Appendix D; it contains descriptions of all the symbols used in the test suite.

Substitutions using the Macrosub program may be made as follows:

1. Edit the file macro.dfs using values appropriate for the implementation. Symbols that use the value of MAX_IN_LEN are calculated automatically and need not be entered.

2. Create a file called tsttests.dat that includes all of the .tst test file names, and their directory locations if necessary. A version of this file (without directory information) is supplied.

3. Compile and bind MacroSub.

4. Run MacroSub.

The program will replace all symbols in the .tst files with values from macro.dfs. Test files with the original test name but the extension .adt will contain the processable tests. The original .tst files will not be modified.

4.3.3 Processing for Wide_Character Tests

There are two tests in ACVC 2.2 that **require** preprocessing. They must be processed with the Wide Character tool; the macro expander tool will not work with them. Information for these tests is **not** included in macro.dfs.

There are two tests in ACVC 2.2 that check an implementation's ability to process characters drawn from the full set of graphic symbols of ISO 10646 BMP (See [Ada95] 2.1). Since such characters cannot be included in the distribution media in a way that can reliably be read by an arbitrary implementation, they contain character sequences that must be replaced by the intended character. A special tool, the WideChr program, which will automatically perform the required substitutions, has been included with this distribution.

The affected tests are contained in files with the extension .aw. Each such test contains a six or eight character sequence of the form

"[ab]"

or

"[abcd]"

Note that double quotes make up part of the special sequence (acting as part of the escape sequence). The processor will replace the string with a character that is designated by 16#abcd#, where the alphanumeric characters ‘a’, ‘b’, ‘c’, ‘d’, are hexadecimal digits. Note that the strings to be replaced do not start with ‘$’, and the replacement is synthetic, not substitution. Therefore, the macro expander tool will not work with these tests.

The WideChr tool takes the designated tests as input. The names of the required tests are included in the WideChr tool code as constants. It reads path names for the tests from ImpDef. The tool reads the tests, synthesizes the necessary replacements, and writes the resulting, compilable programs into files with the same name as the original but with the extension .a.

Substitutions using the WideChr program may be made as follows:

1. Edit the file impdef.a to indicate the path where the tests are located. This value will be concatenated with the test name to form the complete name of a file.

3. Compile and bind WideChr.

4. Run WideChr.

The program will replace all special sequences in the .aw files with synthesized characters. Test files with the original test name but the extension .a, in the same path location as the original .aw files, will contain the processable tests. The original .aw files will not be modified.

4.3.4 Package SPPRT13 and Function FcnDecl

Package SPPRT13 declares six constants of type System.Address that are primarily used by tests of Section 13 features. It is in the file spprt13s.tst. As distributed, the package uses macro symbols which must be replaced. In most cases, the substitution can be accomplished by the macro substitution described in the preceding section. If appropriate literals, constants, or predefined function calls can be used to initialize these constants, they should be supplied in macro.dfs. Otherwise, the package FCNDECL must be modified.

The version of SPPRT13 distributed with ACVC 2.2 is slightly different from the version distributed with ACVC 1.11. A body is not required for this package (and would, therefore, be illegal in Ada95).

> All implementations should verify that package SPPRT13 can be properly customized using the macro substitution technique. Note that in Ada95, a body for SPPRT13 is illegal.

The specification for package FCNDECL is in the file fcndecl.ada. SPPRT13 depends on FCNDECL (in a context clause that both "with"s it and "use"s it). As supplied with the ACVC, FCNDECL is an empty package specification. If appropriate literals, constants, or predefined function calls cannot be used to customize the constants declared in SPPRT13, the implementer must declare appropriate functions in the specification of FCNDECL and provide bodies for them in a package body or with a pragma Import.

Modifications to FCNDECL must receive advance approval from the ACAL (and, if necessary, the ACAA) before use in a validation attempt.

4.3.5 Modification of Package REPORT

All executable tests use the Report support package. It contains routines to automate test result reporting as well as routines designed to prevent optimizers from removing key sections of test code. The specification of package Report is in the file repspec.ada; the body is in repbody.ada.

Under some conditions, the body of package Report may need to be modified. For example, the target system for a cross-compiler may require a simpler I/O package than the standard package Text_IO. In such a case, it may be necessary to replace the context clause and the I/O procedure names in the body of Report.

Modifications to Report must receive advance approval from the ACAL (and, if necessary, the ACAA) before use in a validation attempt.

4.3.6 Allowed Test Modifications

Class B tests have one or more errors that implementations must identify. These tests are structured such that, normally, implementations can report all included errors. Occasionally, an implementation will fail to find all errors in a B-test because it encounters a limit (e.g., error cascading, resulting in too many error reports) or is unable to recover from an error. In such cases, a user may split a single B-test into two or more tests. The resulting tests must contain all of the errors included in the original test, and they must adhere as closely as possible to the style and content of the original test. Very often, the only modification needed is to comment out earlier errors so that later errors can be identified. In some cases, code insertion will be required. An implementation **must** be able to demonstrate that it can detect and report **all** intended B-test errors.

Splits may also be required in executable tests, if, for example, an implementation capacity limitations is encountered (e.g., a number of generic instantiations too large for the implementation). In very exceptional cases, tests may be modified by the addition of a length clause (to alter the default size of a collection), or by the addition of an elaboration Pragma (to force an elaboration order).

Tests that use configuration pragmas (see 4.6.5.4) may require modification since the method of processing configuration pragmas is implementation dependent.

Some tests include foreign language code (Fortran, C, or Cobol). While the features used should be acceptable to all Fortran, C, and Cobol implementations, respectively, some implementations may require modification to the non-Ada code. Modifications must, of course, preserve the input-output semantics of the (foreign language) subprogram; otherwise, the ACVC test will report a failure.

All splits and modifications must be approved in advance by the ACAL (and, if necessary, the ACAA) before they are used in a validation attempt. It is the responsibility of the user to propose a B-test split that satisfies the intention of the original test.

Modified tests should be named by appending an alphanumeric character to the name of the original test. When possible, line numbers of the original test should be preserved in the modification.

All tests must be submitted to the compiler as distributed (and customized, if required). If a test is executable (class A, C, D, E) and compiles successfully, then it must be run. Modified tests or split tests may be processed next. Only the results of the modified tests will be graded.

If the ACAA has issued a Modified tests list (see Section 4.2.1), then the required modifications must be made. The permitted modifications may be made if desired (or if necessary for the particular implementation).

4.4 Processing the Support Files

After all the files identified in Section 4.3 have been customized as needed and required, the support files can be processed and the reporting mechanism can be verified.

4.4.1 Support Files

The following files are necessary to many of the ACVC tests. Implementations that maintain program libraries may wish to compile them into the program library used for validation:

`repspec.ada`	`repbody.ada`
`impdef.a`	`impdefc.a` (If validating Annex C)
`fcndecl.ada`	`impdefd.a` (If validating Annex D)
`checkfil.ada`	`impdefe.a` (If validating Annex E)
`lencheck.ada`	`impdefg.a` (If validating Annex G)
`enumchek.ada`	`impdefh.a` (If validating Annex H)
`spprt13s.adt` (after macro substitution)	
`tctouch.ada`	

(Depending on local requirements and strategy, it may also be convenient to compile all foundation code into the program library as well.)

4.4.2 "CZ" Acceptance Tests

Four tests having names beginning “CZ” are part of the ACVC suite. Unlike other tests in the suite, they do not focus on Ada language features. Instead, they are intended primarily to verify that software needed for the correct execution of the test suite works as expected and required. They check, for example, to see that package Report and package TC_Touch work correctly.

All CZ tests must execute correctly and exhibit the prescribed behavior for a successful validation.

The acceptance test CZ1103A must be processed and run as the first step of a validation attempt to ensure correct operation of CHECKFILE. (Some of the executable file I/O tests use a file checking procedure named CHECKFILE that determines an implementation's text file characteristics. The source code for this procedure is in the file checkfil.ada.) CZ1103A checks whether errors in text files are properly detected, therefore, CZ1103A will print some failure messages when it is executed. The presence of these messages does **not** necessarily mean the test has failed. A listing of the expected output for CZ1103A is included in Appendix D (times and dates in the actual output will differ).

The acceptance test CZ00004 produces output that verifies the intent of the validation. It relies on ImpDef having been correctly updated for the validation and produces output identifying the annexes (if any) that will be included as part of the validation.

4.5 Establishing Command Scripts

Users will often find it convenient to run large numbers of ACVC tests with command scripts. This section discusses some of the issues to be considered in developing a script.

4.5.1 Command Scripts

All compiler options and switches that are appropriate and necessary to run the ACVC tests must be identified and included in commands that invoke the compiler. The same is true for the binder or any other post-compilation tools. Any implementation dependent processing of partitions, configuration pragmas, and strict mode processing must be part of the scripts for running tests that rely on these features.

A script should compile (only) all class B tests. It should compile and bind all class L tests; if link errors are not explicitly given, the script should attempt to execute the L tests. It should compile all class F files. It should compile, bind, and execute all class A, C, D, and E tests.

4.5.2 Dependencies

A command script must take account of all required dependencies. As noted earlier, some tests are composed of multiple test files. Also, some tests include foundation code, that may be used by other tests. If a foundation is not already in the environment, it must be compiled as part of building the test. All files that are used in a test must be compiled in the proper order, as indicated by the file name. For implementations that require the extraction individual compilation units from test files before submission to the compiler, the individual units must be submitted to the compiler in the same order in which they appear in the file.

4.6 Processing ACVC Tests

After the ACVC tests and support code have been installed and all required modifications and preliminary processing has been completed, the suite can be processed by an implementation. This section describes the tests required for validation, required partitioning, how tests may be bundled for efficiency, and certain processing that may be streamlined. It also describes how the suite has been organized to allow a user to focus on specific development needs.

4.6.1 Required Tests

An implementation may be validated against the core language only or the core language plus one or more Specialized Needs Annexes. All core tests (except as noted in 4.6.4) must be processed with acceptable results for validation against the core language. All legacy tests, as well as all newer tests for clauses 2-13 and annexes A and B are core tests. Validation including one or more Specialized Needs Annexes requires that all tests for the annex(es) in question be correctly processed in addition to all core tests

Tests that are not applicable to an implementation (e.g., because of size limitations) and tests that report "NOT APPLICABLE" when run by an implementation must nevertheless be processed and demonstrate appropriate results.

Tests that are on the current Withdrawn List as maintained by the ACAA need not be processed.

4.6.2 Test Partitions

Unless otherwise directed by the Special Requirements section of a test, all tests are to be configured and run in a single partition. The method of specifying such a partition is implementation dependent and not determined by the ACVC. The only tests that must be run in multiple partitions are those which test Annex E, Distributed Systems.

4.6.3 Bundling Test Programs

In some situations, the usual test processing sequence may require an unacceptable amount of time. For example, running tests on an embedded target may impose significant overhead time to download individual tests. In these cases, executable tests may be bundled into aggregates of multiple tests. A set of bundled tests will have a driver that calls each test in turn; ACVC tests will then be called procedures rather than main procedures. No source changes in the tests are allowed when bundling; that is, the only allowed change is the method of calling the test.

All bundles must be approved by the ACAL (and, if necessary, the ACAA) to qualify for a validation attempt. It is the responsibility of the user to identify the tests to be bundled and to write a driver for them.

4.6.4 Processing That May be Omitted

A user may streamline processing of the ACVC tests to the greatest degree possible consistent with complete processing of all tests.

Many Ada95 tests rely on foundation code. A foundation need not be compiled anew each time a different test uses it. In a processing model based on a program library, it is reasonable to compile the code into the library only once and allow the binder to use the processed results for each test that "with"s the foundation.

A user may determine, with ACAL concurrence, that some tests require support that is impossible for the implementation under test to provide. For example, there are tests that assume the availability of file I/O whereas some (embedded target) implementations do not support file I/O. Those tests need not be processed during on-site testing; however, the implementer must demonstrate that they are handled in accordance with the language standard. This demonstration may be performed before on-site testing, in which case it need not be repeated.

Annex B tests that require foreign language code (Fortran, C, Cobol) to be compiled and bound with Ada code need not be processed if an implementation does not support a foreign language interface to the respective language.

Tests for the Specialized Needs Annexes of [Ada95] need not be processed except by implementations that wish to have Annex results documented. In that case, only the tests for the annex in question (in addition to all core tests) need be processed. If any tests for a particular Annex are processed, then all tests for that Annex must be processed. If an implementation does not support a feature in a Specialized Needs Annex test, then it must indicate the non-support by rejecting the test at compile time or by raising an appropriate exception at run time. (See [Ada95] 1.1.3(17).)

No test on the withdrawn list need be processed.

4.6.5 Tests with Special Processing Requirement

Some tests may require special handling. These are primarily SNA tests, but some core tests are affected. For example, distributed processing tests may require an executable image in multiple partitions, where partitions are constructed in an implementation specific manner. Real-time processing tests may have configuration pragmas that have to be handled in an implementation specific way. Numeric Processing tests require strict mode processing to be selected. Each such test has a Special Requirements section in the test header describing any implementation specific handling that is required for the test.

A list of all such tests is included in Appendix A.

4.6.5.1 Foreign Language Interface Tests

Annex B, Interface to Other Languages, is part of the Ada95 core language. Any implementation that provides one or more of the packages Interfaces.C, Interfaces.COBOL, or Interfaces.Fortran **must** correctly process, and pass, the tests for interfaces to C, Cobol, and/or Fortran code respectively, with the possible exception of tests containing actual foreign code.

An implementation that provides one or more of these Interfaces child packages must successfully compile the Ada units of tests with actual foreign language code. If the implementation does not support the actual binding of the foreign language code to Ada, these tests may report binding errors, or may reject the pragma Import, in which case they may be graded as inapplicable. If the implementation supports the binding and an appropriate compiler is available, the tests must execute and report "Passed". If the implementation supports the binding, but it is not feasible to have an appropriate compiler available, then the tests may be graded as inapplicable by demonstrating that they fail to bind.

If one of the Interfaces child packages is not provided, then the corresponding tests may be graded as inapplicable, provided they reject the corresponding "with" clause.

The tests involving interfaces to foreign code are listed below.

The foreign language code included in ACVC tests uses no special or unique features, and should be accepted by any standard (C, Cobol, or Fortran) compiler, there may be dialect problems that prevent the code from compiling correctly. Modifications to the foreign language code are allowable; the modifications must follow the code as supplied as closely as possible, and the result must satisfy the requirements stated in the file header. Such modifications must be approved in advance by the ACAL (and, if necessary, the ACAA). The method for compiling foreign code is implementation dependent and not specified as part of the ACVC. Ada code in these tests must be compiled as usual. The Ada code includes Pragma Import which references the foreign language code. The link name of foreign language object code must be provided in ImpDef. When all code has been compiled, the test must be bound (including the foreign language object code) and run. The method for binding Ada and foreign language code is implementation dependent and not specified as part of the ACVC. The test must report “PASSED” when executed.

4.6.5.1.1 C Language Interface

If the implementation provides the package Interfaces.C, the tests identified below must be satisfactorily processed as described above.

The starred tests contain C code that must be compiled and linked if possible, as described above. The C code is easily identifiable because the file has the extension ".C". The C code may be modified to satisfy dialect requirements of the C compiler. The C code files must be compiled through a C compiler, and the resulting object code must be bound with the compiled Ada code. Pragma Import will take the name of the C code from ImpDef.

CD30005*	CXB3004	CXB3008	CXB3012	CXB3016
CXB3001	CXB3005	CXB3009	CXB3013*	
CXB3002	CXB3006	CXB3010	CXB3014	
CXB3003	CXB3007	CXB3011	CXB3015	

4.6.5.1.2 Cobol Language Interface

If the implementation provides the package Interfaces.COBOL, the tests identified below must be processed satsifactorily, as described above.

The starred test contains Cobol code that must be compiled and linked if possible, as described above. The Cobol code is easily identifiable because the file has the extension ".CBL". The Cobol code may be modified to satisfy dialect requirements of the Cobol compiler. The Cobol code files must be compiled through a Cobol compiler, and the resulting object code must be bound with the compiled Ada code. Pragma Import will take the name of the Cobol code from ImpDef.

CXB4001	CXB4003	CXB4005	CXB4007	CXB4009*
CXB4002	CXB4004	CXB4006	CXB4008	

4.6.5.1.3 Fortran Language Interface

If the implementation has a Fortran language interface, the tests identified below must be processed satisfactorily, as described above.

The starred tests contain Fortran code that must be compiled and linked if possible, as described above.. The Fortran code is easily identifiable because the file has the extension ".FTN". The Fortran code may be modified to satisfy dialect requirements of the Fortran compiler. The Fortran code files must be compiled through a Fortran compiler, and the resulting object code must be bound with the compiled Ada code. Pragma Import will take the name of the Fortran code from ImpDef.

CXB5001	CXB5003	CXB5005*
CXB5002	CXB5004*	

4.6.5.2 Tests for the Distributed Processing Annex

The ACVC tests for the Distribution Annex are applicable only to implementations that wish to validate this SNA. Not all of these tests apply to all implementations, since the annex includes some implementation permissions that affect the applicability of some tests.

The principal factors affecting test applicability are:

1. whether the Remote_Call_Interface pragma is supported;
2. whether a Partition Communication System (PCS) is provided (i.e., whether a body for System.RPC is provided by the implementation);
3. f a body for System.RPC is provided, whether it can be replaced;
4. whether the Real-Time Annex is also supported.

An implementation may validate against the annex without providing a PCS. An implementation may disallow replacement of language-defined compilation units like System.RPC. However, in order to validate against the Distribution Annex, an implementation **must either** provide a PCS **or** allow a body for System.RPC to be compiled.

4.6.5.2.1 Remote_Call_Interface pragma

[Ada95] allows explicit message-based communication between active partitions as an alternative to RPC [E.2.3(20)]. If an implementation does not support the Remote_Call_Interface pragma then the following tests are not applicable:

BXE2009	CXE2001	CXE4004	CXE5003
BXE2010	CXE4001	CXE4005	LXE3001
BXE2011	CXE4002	CXE4006	
BXE4001	CXE4003	CXE5002	

4.6.5.2.2 Partition Communication System

An implementation is not required to provide a PCS [E.5(27)] in order to validate the Distribution Annex. If no PCS is provided then the following tests are not applicable:

CXE1001	CXE4001	CXE4003	CXE4005
CXE2001	CXE4002	CXE4004	CXE4006

4.6.5.2.3 System.RPC

Two tests provide a body for System.RPC. If an implementation does not allow the compilation of a body for System.RPC then the tests identified below are not applicable.

An implementation may include a private part that includes declarations, such as additional procedures and functions, that impose additional requirements on System.RPC. If an implementation allows the compilation of a body for System.RPC and includes additional declarations, then the same declarations (and implementations) may be added to the body of System.RPC in the tests identified below. Declarations in the private part of the implementation's System.RPC do not affect the applicability of the tests in this group.

CXE5002	CXE5003

4.6.5.2.4 Real-Time Annex Support

Many implementations that support the Distribution Annex will also support the Real-Time Annex. Test CXE4003 is designed to take advantage of Real-Time Annex features in order to better test the Distribution Annex.

For implementations that do not support the Real-Time Annex, test CXE4003 must be modified. This modification consists of deleting all lines that end with the comment "--RT".

4.6.5.2.5 Configuring Multi-Partition Tests

Some Distribution Annex tests require multiple partitions to run the test, but no more than two partitions are required for running any of them. All multi-partition tests contain a main procedure for each of the two partitions. The two partitions are referred to as "A" and "B" and the main procedures for these partitions are named *<test_name>*_A and *<test_name>*_B respectively. Each test contains instructions naming the compilation units to be included in each partition. Most implementations will be primarily concerned with the main procedure and RCI packages that are to be assigned to each partition; the remainder of the partition contents will be determined by the normal dependency rules. The naming convention used in multi-partition tests aid in making the partition assignments. If the name of a compilation unit ends in "_A<optional_digit]>" then it should be assigned to partition A. Compilation units with names ending in "_B<optional_digit>" should be assigned to partition B.

The following tests require that two partitions be available to run the test:

CXE1001	CXE4002	CXE4005	CXE5003
CXE2001*	CXE4003	CXE4006	LXE3001
CXE4001	CXE4004	CXE5002	LXE3002*

(*) Tests CXE2001 and LXE3002 contain a Shared_Passive package and two active partitions. They may be configured with either two or three partitions. The two partition configuration must have two active partitions and the Shared_Passive package may be assigned to either one of the active partitions. The three partition configuration consists of two active partitions and a single passive partition, and the passive partition will contain the single Shared_Passive package.

4.6.5.2.6 Running Multi_Partition Tests

All of the multi-partition tests include the package Report in both of the active partitions. In order for the test to pass, **both** partitions must produce a passed message (except for LXE3002 - see special instructions for that test). If either partition produces a failed message, or if one or both partitions do not produce a passed message, the test is graded "failed".

When running the multi-partition tests it is not important which partition is started first. Generally, partition A acts as a server and partition B is a client, so starting partition A first is usually best.

In the event a test fails due to the exception Communication_Error being raised, it is permissible to rerun the test.

4.6.5.3 Tests for the Numerics Annex

Many of the tests for Annex G, Numerics, **must** be run in strict mode. The method for selecting strict mode is implementation dependent and not specified by the ACVC. (Note that the tests for numerical functions specified in Annex A may, *but need not*, be run in strict mode.) The following tests must be run in strict mode:

CXG2003	CXG2010	CXG2016	CXG2022
CXG2004	CXG2011	CXG2017	CXG2023
CXG2006	CXG2012	CXG2018	CXG2024
CXG2007	CXG2013	CXG2019	
CXG2008	CXG2014	CXG2020	
CXG2009	CXG2015	CXG2021	

4.6.5.4 Tests that use Configuration Pragmas

Several of the tests in Annex D, Real Time Processing, Annex E, Distributed Processing, and Annex H, Safety and Security, use configuration pragmas. The technique for applying a configuration pragma to a test composed of multiple compilation units is implementation dependent and not specified by the ACVC. Every implementation that uses any such test in a validation must therefore take the appropriate steps, which may include modifications to the test code and/or post-compilation processing, to ensure that such a pragma is correctly applied. The following tests require special processing of the configuration pragma:

BA15001	CXD2002	CXD4010	LXH4002
BXC5001	CXD2003	CXD5002	LXH4003
BXH4001	CXD2004	CXD6002	LXH4004
BXH4002	CXD2005	CXD6003	LXH4005
BXH4003	CXD2006	CXDA003	LXH4006
BXH4004	CXD2007	CXDB005	LXH4007
BXH4005	CXD2008	CXH1001	LXH4008
BXH4006	CXD3001	CXH3001	LXH4009
BXH4007	CXD3002	CXH3003	LXH4010
BXH4008	CXD3003	LXD7001	LXH4011
BXH4009	CXD4001	LXD7003	LXH4012
BXH4010	CXD4003	LXD7004	LXH4013
BXH4011	CXD4004	LXD7005	
BXH4012	CXD4005	LXD7006	
BXH4013	CXD4006	LXD7007	
CXD1004	CXD4007	LXD7008	
CXD1005	CXD4008	LXD7009	
CXD2001	CXD4009	LXH4001	

4.6.6 Focus on Specific Areas

The ACVC test suite is structured to allow compiler developers and testers to use parts of the suite to focus on specific compiler feature areas.

Both the legacy tests and the newer tests tend to focus on specific language features in individual tests. The name of the test is generally a good indicator of the primary feature content of the test, as explained in the discussion of naming conventions. Beware that legacy test names have not changed, but the Ada Reference Manual organization has changed from [Ada83] to [Ada95], so some legacy test names point to the wrong clause of [Ada95]. Further, note that the general style and approach of the newer tests creates user-oriented test situations by including a variety of features and interactions. Only the primary test focus can be indicated in the test name.

ACVC 2.2 tests are divided into core tests and Specialized Needs Annex tests. Recall that annexes A and B are part of the core language. All annex tests (including those for annexes A and B) have an 'X' as the second character of their name; Specialized Needs Annex tests have a letter between 'C' and 'H' (inclusive) corresponding to the annex designation, as the third character of the test name.

4.7 Grading Test Results

Although a single test may examine multiple language issues, ACVC test results are graded "passed", "failed", or "not applicable" as a whole.

All customized, applicable tests must be processed by an implementation. Results must be evaluated against the expected results for each class of test. Results that do not conform with expectations constitute failures. The only exceptions allowed are discussed above in test splitting and modification; in such cases, processing the approved modified test(s) must produce the expected behavior. Any differences from the general discussion of expected results below for executable or non-executable tests are included as explicit test conditions in test prologues.

Warning or other informational messages do not affect the pass/fail status of tests.

Expected results for executable and non-executable tests are discussed in Sections 4.7.1 - 4.7.3. Tests that are non-applicable for an implementation are discussed in 4.7.4. Withdrawn tests are discussed in 4.7.5.

4.7.1 Expected results for Executable Tests

Executable tests (classes A, C, D, E) must be processed by the compiler and any post-compilation steps (e.g., binder, partitioner) without any errors. They must be loaded into an execution target and run. Normal execution of tests results in an introductory message that summarizes the test objective, possibly some informative comments about the test progress, a final message giving pass / fail status, and graceful, silent termination. They may report "PASSED", "TENTATIVELY PASSED", "FAILED", OR "NOT APPLICABLE".

A test that fails to compile and bind, including compiling and binding any foundation code on which it depends is graded as "failed", unless the test includes features that need not be supported by all implementations. For example, an implementation may reject the declaration of a numeric type that it does not support. Allowable cases are clearly stated in the Applicability Criteria of tests. Annex L of [Ada95] requires implementations to document such implementation-defined characteristics.

A test that reports "FAILED" is graded as "failed" unless the ACAL, and possibly the ACAA, determine that the test is not applicable for the implementation.

A test that reports "PASSED" is graded as "passed" unless the test produces the pass message but fails to terminate gracefully (e.g., crashes, hangs, raises an unexpected exception, produces an earlier or later "FAILED" message). This kind of aberrant behavior may occur, for example, in certain tasking tests, where there are multiple threads of control. A pass status message may be produced by one thread, but another thread may asynchronously crash or fail to terminate properly.

A test that reports "NOT APPLICABLE" must be run by the implementation and is graded as "not applicable" unless it produces the not-applicable message and then fails to terminate gracefully.

A test that reports "TENTATIVELY PASSED" is graded as "passed" if the test results satisfy the pass/fail criteria in the test. Normally, verification requires manual inspection of the test output.

A test that fails to report, or produces only a partial report, will be graded as "failed" unless the ACAL, and possibly the ACAA, determine that the test is not applicable for the implementation.

4.7.2 Expected Results for Class B

Class B tests are expected to be compiled but are not subject to any more processing and are not intended to be executable. An implementation must correctly report each clearly marked error (the notation "-- ERROR:" occurs at the right hand side of the source). A multiple unit B test file generally will have errors only in one compilation unit. Error messages must provide some means of specifying the location of an error, but they are not required to be in direct proximity with the "-- ERROR:" marking of the errors.

Some B-tests also include the notation "-- OK" to indicate constructs that must **not** be identified as errors. **This is especially important since some constructs were errors in Ada83 that are legal in Ada95**.

Some B-tests exercise constructs whose correctness depends on source code that is textually separated (e.g., a deferred constant and its full declaration). In these cases, it may be reasonable to report an error at both locations. Such cases are marked with "-- OPTIONAL ERROR". These lines may be flagged as errors by some, but not all, implementations. Unless an optional error is marked as an error for the wrong reason, an error report (or lack of it) does not affect the pass/fail status of the test.

A test is graded as "passed" if it reports each error in the test. The content of error messages is considered only to determine that they are indeed indications of errors (as opposed to warnings, e.g.) and that they refer to the expected errors. The Reference Manual does not specify the form or content of error messages. In particular, a test with just one expected error is graded as "passed" if the test is rejected at compile time.

A test is graded as "failed" if it fails to report on each error in the test or if it marks legal code as erroneous.

4.7.3 Expected Results for Class L

Class L tests are expected to be rejected before execution begins. They must be submitted to the compiler and to the linker/binder. If an executable is generated, then it must be submitted for execution. Unless otherwise documented, the test is graded as "failed" if it begins execution, regardless of whether any output is produced.. (Twenty-eight L tests contain documentation indicating that they may execute. See below.)

In general, an L test is expected to be rejected at link/bind time. Some tests contain "-- ERROR:" indications; an implementation that reports an error associated with one of these lines is judged to have passed the test (provided, of course, that the link attempt fails).

The following tests are exceptions to the general rule that an L test must not execute:

> Test LXE3002, for the Distributed Systems Annex, is a test that has two partitions, each of which may execute. As documented in the source code, this test is graded "failed" if both partitions report "TENTATIVELY PASSED". Other outcomes are graded as appropriate for Class L tests.
>
> Tests LA24001..27 (twenty-seven core language tests), as documented in the source code, may execute if automatic recompilation is supported. These tests are graded as "passed" if they execute and report "PASSED". Other outcomes are graded as appropriate for Class L tests.

4.7.4 Inapplicable Tests

Each ACVC test has a test objective that is described in the test prologue. Some objectives address Ada language features that need not be supported by every Ada implementation (e.g., "check floating-point operations for digits 18"). These test programs generally also contain an explicit indication of their applicability and the expected behavior of an implementation for which they do not apply.

A test may be inapplicable for an implementation given:

- appropriate ACVC grading criteria; or
- an ACAA ruling on a petition to accept a deviation from expected results.

Appropriate grading criteria include:

a. whether a test completes execution and reports "NOT APPLICABLE";

b. whether a test is rejected at compile or bind time for a reason that satisfies grading criteria stated in the test program.

All applicable test programs must be processed and passed.

4.7.5 Withdrawn Tests

From time to time, the ACAA determines that one or more tests included in a release of the ACVC should be withdrawn from the test suite. Tests that are withdrawn are not processed during a validation attempt and are not considered when grading an implementation.

Usually, a test is withdrawn because an error has been discovered in it. A withdrawn test will not be reissued in the same test suite, although it may be revised and reissued in a later suite.

The ACAA maintains a list of withdrawn tests.

4.8 Addressing Problems or Issues

After all tests have been processed and graded, any remaining problems should be addressed. Test failures must be identified and resolved. This section discusses issues that are not due to implementation errors (bugs).

4.8.1 Typical Issues

Here are some typical causes of unexpected ACVC test failures (often resulting from clerical errors):

Processing a test that is on the ACAA list of withdrawn tests;

Processing a test that is not applicable to the implementation (as explained in Section 4.7.4;

Processing files (or tests, see Section 4.5.2) in an incorrect order;

Processing tests when units required in the environment are not present.

Test result failures resulting from technical errors may include:

Incorrect values in ImpDef, which provide inappropriate values to tests at run-time;

Incorrect values in macro.dfs which result in incorrectly customized tests;

Incorrect substitutions in wide_character tests;

Need to modify a test (e.g., split a B-test).

Finally, occasionally a user discovers an error in a new ACVC test. More rarely, errors are uncovered by compiler advances in tests that are apparently stable. In either case, if users believe that a test is in error, they may file a dispute with the ACAL. The dispute process is described in the next section.

4.8.2 Deviation from Expected Results - Petition & Review

Each test indicates in its prologue what it expects from a conforming implementation. The result of processing a test is acceptable if and only if the result is explicitly allowed by the grading criteria for the test.

A user may challenge an ACVC test on the grounds of applicability or correctness. A challenger should submit a petition against the test program to an ACAL or to the ACAA, following the procedure and the format presented in [Pro98]. A petition must clearly state whether it is a claim that the test does not apply to the implementation or that the test is erroneous. The petition must indicate the specific section of code that is disputed and provide a full explanation of the reason for the dispute.

ACALs will forward petitions from their customers to the ACAA for decisions. The ACAA will evaluate the petitioner's claims and decide whether

- the test is applicable to the implementation (i.e., deviation is *not* allowed);
- the test is not applicable to the implementation (i.e., deviation *is* allowed);
- the test should be withdrawn (deviation is allowed and the test is not considered in determining validation results).

A deviation is considered to be a test failure unless a petition to allow the deviation has been accepted by the ACAA.

4.9 Reprocessing and Regrading

After all problems have been resolved, tests that failed can be reprocessed and regraded. This step completes the ACVC testing process.