(2)

DRAFT

Requirements For IV&V of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Implementation

Diane E. Mularz
Jonathan D. Wood
Deborah M. Haydon

October, 1988

SPONSOR:
Ada Joint Program Office (AJPO)
Contract No.:
F19628-89-C-0001

~~This document was prepared for authorized distribution. It has not been approved for public release.~~

The MITRE Corporation
Washington C^3I Operations
7525 Colshire Drive
McLean, Virginia 22102

DRAFT

UNCLASSIFIED 21 FEB 1989

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETEING FORM
1. REPORT NUMBER / 2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Requirements for IV & V of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Implementation	5. TYPE OF REPORT & PERIOD COVERED Draft; 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Mularz, Diane; Wood, Johnathan; Haydon, Deborah	8. CONTRACT OR GRANT NUMBER(s) F19628089-C-0001
9. PERFORMING ORGANIZATION AND ADDRESS Mitre Corporation, 7525 Colshire Dr., McLean, VA 22102	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office, 3D139 (1211 S. FERN, C-107), The Pentagon, Washington, D.C. 20301-3081	12. REPORT DATE Oct 1988; 13. NUMBER OF PAGES 49
...fferent from Controlling Office)	15. SECURITY CLASS (of this report) UNCLASSIFIED; 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

UNCLASSIFIED

18. SUPPLEMENTARY NOTES

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

NATO; APSE; CAIS

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

This report outlines a strategy and specifies the requirements for the development of a test environment which included: a test administration function that will provide control over execution of the tests and management of the test results; and a test suite whose tests will be defined based on the syntax and semantics defined in the Special Working Group CAIS specification.

DD FORM 1473 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CLEARANCE RE[illegible]T FOR [illegible]UBLIC RELEASE OF DEPARTMENT OF DEFENSE INFORMATION

TO: Assistant Secrerary of Defense (Pu[illegible] Affairs)

ATTN: Director, Freedom of Information & S[illegible]curity Review, Rm 2C757, Pentagon

SEE INSTRUCTIONS ON REVERSE

(This form is to be used in requesting review and clearance of DoD information proposed for public release in accordance with DoDD 5230.9.)

1. DOCUMENT DESCRIPTION

a. TYPE REPORT	b. TITLE Requirements For IV&V of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Implementation
c. PAGE COUNT 50	d. SUBJECT AREA

a. NAME (*Last, First, MI*)	b. RANK	c. TITLE
	d. OFFICE	e. AGENCY

3. PRESENTATION/PUBLICATION DATA
Distribution To The General Public.

4. POINT OF CONTACT	5. PRIOR COORDINATION
a. NAME (Last, First, MI) CASTOR, Virginia, L. vc	a. OFFICE OSD(A)/CET/AJPO
b. TELEPHONE NUMBER (Include Area Code) (202) 694-0210	b. AGENCY R&AT

6. REMARKS
This document has been reviewed within the Ada Joint Program Office. It contains no sensitive or classified information.

7. RECOMMENDATION OF SUBMITTING OFFICE/AGENCY

a. The attached material has Department/Office/Agency approval for public realease (qualifications if any, are indicated in Remarks Section) and clearance for open publication is recommended under provisions of DoDD 5230.9 I am authorized to make this recommendation for release on behalf of:

Director, AJPO

b. Clearance is requested by 881121 (*YYMMDD*).

c. NAME (*Last, First, MI*)	d. TITLE	e. SIGNATURE
f. OFFICE	g. AGENCY	h. DATE (*YYMMDD*)

DD FORM 1910 82 MAR PREVIOUS EDITION IS OBSOLETE.

ABSTRACT

A NATO Special Working Group (SWG) on Ada Programming Support Environments (APSE) was established in October, 1986. Its charter is to develop a tool set that constitutes an APSE, to evaluate the APSE on both an individual component basis and on a holistic level, and to define a NATO interface standard for APSEs. A specific task within the associated MITRE work program is to develop the requirements to perform testing of the Common APSE Interface Set (CAIS) for the SWG. The SWG CAIS is the agreed upon tool interface set for the NATO effort, and is a variant of the CAIS standard, DOD-STD-1838. CAIS provides a standard set of kernel interfaces for APSE tools thus promoting portability of tools across disparate architectures.

The SWG CAIS is complex; there are over 500 unique interfaces defined in 29 Ada packages with over 1600 possible error conditions. This report outlines a strategy and specifies the requirements for the development of a test environment which include: a test administration function that will provide control over execution of the tests and management of the test results; and a test suite whose tests will be defined based on the syntax and semantics defined in the SWG CAIS specification. This test suite will focus on nominal functionality and completeness of critical interfaces. The test environment will be incrementally developed to correspond to phased deliveries of the SWG CAIS implementations. There will be two SWG CAIS implementations installed on two different host architectures. This report outlines the requirements to perform testing on either implementation. (KR)

ACKNOWLEDGMENTS

We would like to thank the MITRE peer reviewers for their thorough, conscientious, and timely critique of our paper. Much credit is also due to the U.S. Team members and Evaluation Review Board members who reviewed many preliminary drafts of this paper.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	ix
LIST OF TABLES	x
EXECUTIVE SUMMARY	xi
1.0 Introduction	1
1.1 Background	1
1.2 Objectives	3
1.3 Scope	3
1.4 Report Content	4
2.0 SWG CAIS Test Strategy	5
2.1 Informal Design Review	5
2.2 Test Approach	5
2.2.1 Establish a Test Environment	5
2.2.2 Reuse Existing Technology	13
2.2.3 Staged Testing Capability	15
2.3 Review of Installation Guide	15
3.0 Test Environment Requirements	17
3.1 SWG CAIS Test Execution Control Requirements	17
3.1.1 Test Administration	17
3.1.2 Test Configuration Management	17
3.1.3 Test Invocation	17
3.1.4 Run-time Control	18
3.1.5 Implementation Assessment	18
3.1.6 Environment Configuration Assessment	18

TABLE OF CONTENTS (Continued)

	Page
3.1.7 External Interface Requirements	19
3.1.8 Performance Requirements	19
3.2 Test Suite Requirements	20
3.2.1 Nominal Testing	20
3.2.2 Test Name Configuration Management	22
3.2.3 Reporting Results	23
3.3 Design Goals	23
3.3.1 Flexibility	23
3.3.2 Extensibility	23
3.3.3 Robustness	24
3.3.4 Portability	24
APPENDIX A: Overview of Verification & Validation Techniques	25
APPENDIX B: Available CAIS Validation Software	33
REFERENCES	41
GLOSSARY	45

LIST OF FIGURES

Figure Number		*Page*
1	Elements of the NATO SWG APSE	2

LIST OF TABLES

Table Number		Page
1	Scope of SWG CAIS Interface Level Functional Testing	6
2	SWG CAIS Critical Packages/Interfaces	8
3	SWG CAIS Package Dependencies	10
4	SWG CAIS vs. Ada Language Predefined Packages	14
B-1	Available CAIS Validation Software	34

EXECUTIVE SUMMARY

In October, 1986, nine NATO nations signed a Memorandum of Understanding (MOU) that established a Special Working Group (SWG) on Ada Programming Support Environments (APSE). The SWG's charter is to develop and evaluate a tool set using an agreed upon interface set that standardizes system support to the tools. The SWG agreed upon an interface set that is based upon the Common APSE Interface Set (CAIS), established as a Department of Defense (DOD) standard in February, 1988 and termed the SWG CAIS. The SWG CAIS serves as the portability layer in the APSE by providing a set of standard kernel level interfaces to a tool developer thus supporting system level functionality in an abstract, consistent manner. The United States (U.S.) is providing implementations of these interfaces on two different architectures. As a member of the U.S. team supporting the NATO SWG on APSEs as sponsored by the Ada Joint Program Office (AJPO), MITRE has responsibility for the Independent Verification & Validation of a SWG CAIS implementation. This document is intended to be both a plan for the IV&V task and a requirements definition for the technology to support it.

The SWG CAIS presents over 500 standard interfaces for use by a tool developer. These interfaces manipulate an underlying node model that manages relevant objects such as users, processes, files, and devices. A systematic approach must be defined to provide adequate testing of these tool interfaces prior to actual tool usage. The scope of the IV&V task has been defined to be an independent testing activity which includes an informal design review to understand SWG CAIS implementation features in preparation for test design, development of a test suite for a critical subset of the SWG CAIS interfaces, and review of the SWG CAIS Installation Guide through actual utilization and review of the installation procedures.

The test approach prescribed in this document will exercise the SWG CAIS interfaces using a test suite. The testing will be based on a functional, or "black-box" approach. The test environment will support the testing of critical tool interfaces, as well as procedures for providing control over test selection, input data selection, reporting of test results, and configuration of tests. The critical SWG CAIS interfaces will be exercised at the nominal level with initial testing of basic functionality as well as detailed testing which will include testing of overloaded subprograms, exception handling, adherence to the CAIS pragmatics, and handling of global exceptions.

This paper provides an overview of the proposed SWG CAIS IV&V effort. It sets forth the general framework and establishes technology requirements to support this effort.

1.0 Introduction

This document defines the requirements to support Independent Verification and Validation (IV&V) of the Common APSE Interface Set (CAIS) developed for the Ada Joint Program Office (AJPO) in conjunction with the NATO Special Working Group (SWG) on Ada Programming Support Environments (APSE), hereafter referred to as the SWG CAIS.

1.1 Background

In the early 1970's, the Department of Defense (DOD) determined that the proliferation of computer languages for embedded system software was consuming an increasing portion of the DoD software budget. To help address this problem, the Ada language was created and standardized in the early 1980s. However, it is recognized by DoD, the software engineering community, and our NATO counterparts that a standardized language alone is insufficient to address future large-scale development projects. To ensure the desired improvements in future software development projects, a language needs to be coupled with quality tools. The means to plan, analyze, design, code, test and integrate such systems on a common set of software is referred to as a programming support environment.

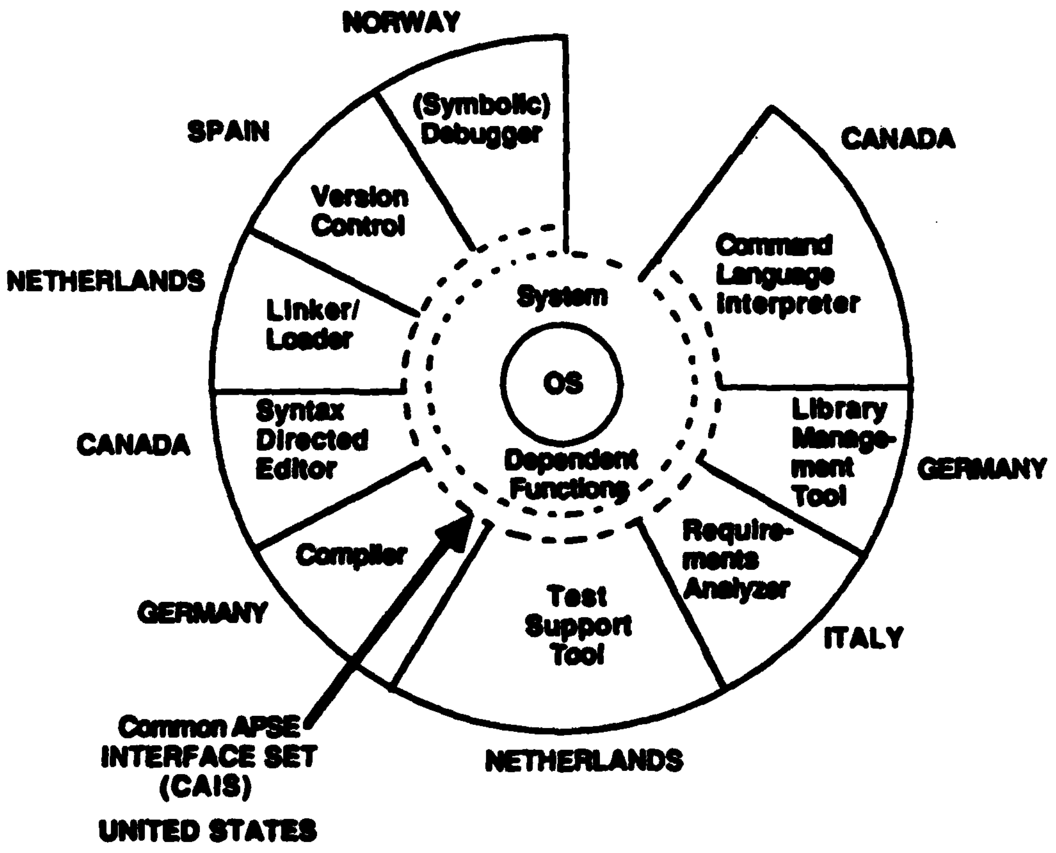
In October, 1986, nine NATO nations signed a Memorandum of Understanding (MOU) that established a SWG on APSEs. This SWG has several goals defined for it: development of an APSE on two different hosts using an agreed upon interface set; evaluation of the tools and the interface set as individual components; a holistic evaluation of the APSE (i.e, as an integrated entity rather than as individual components); and specification of a NATO interface standard for APSEs.

A STONEMAN-based APSE consists of a tool set and a system-level interface set. The interface set provides kernel level functionality in an abstract, consistent manner with specific system mapping embodied in a particular interface implementation. Use of these interfaces by a tool developer promotes transportability of APSE tools across disparate architectures.

The NATO SWG APSE is based upon a STONEMAN model. While the STONEMAN model assumes that tools will use an interface layer exclusively, the SWG APSE also allows direct access to the underlying system, where necessary. Figure 1 illustrates the NATO APSE and identifies the NATO participants responsible for the development of each component. DOD-STD-1838 defines a particular interface set named CAIS. The agreed upon interface set for the NATO effort is a variant of DOD-STD-1838 named the SWG CAIS. The SWG CAIS will be developed for two host architectures: DEC VAX/VMS and IBM VM/SP. Transportability of the NATO APSE will be demonstrated by a rehost of its component tools.

Four working boards were established to effect the SWG goals. Each board has an individual charter that defines its objectives and its deliverables. These four boards are: the Tools and Integration Review Board (TIRB), the Demonstration Review Board (DRB), the Interface Review Board (IRB) and the Evaluation Review Board (ERB).

Figure 1
Elements of the NATO SWG APSE



The TIRB coordinates the development and integration of the tools and the SWG CAIS within the NATO APSE. Each participant identified in Figure 1 is tasked with developing their respective APSE component. Specifically, the U.S. is responsible for implementation of the SWG CAIS on the two host architectures.

Since embedded systems support was a driver in the development of Ada and APSEs, the NATO project includes in its architecture an MC68020 processor as a target system. The DRB will employ the host APSE for development of two weapon system scenarios that are targeted to the MC68020. This demonstration will be used to evaluate the APSE from a holistic level.

The IRB is tasked with developing the requirements and specification of an interface standard for APSEs. To perform this task, the IRB will analyze existing interface standards such as CAIS, the planned upgrade, CAIS-A, a European standard known as the Portable Common Tool Environment (PCTE) and an upgrade called PCTE+. The results of their analysis will be an interface set specification that would define the recommended set to be used on future NATO APSEs.

The ERB will develop evaluation technology and will use it to assess the individual components of the APSE that are developed by the TIRB participants. Both the U.S. and the United Kingdom (UK) have specific tasks within this board. The U.S. is tasked with performing Independent Verification and Validation (IV&V) of the SWG CAIS implementations as well as their evaluation. The United Kingdom (UK) is responsible for evaluation of each of the tools within the NATO APSE.

The requirements outlined in this document identify the technology required to support IV&V of the SWG CAIS.

1.2 Objectives

The objective of this task is to define a cost-effective test effort that will nominally test the SWG CAIS and in so doing, provide a level of confidence to the SWG CAIS users (i.e., tool developers) that the SWG CAIS is sufficiently tested prior to tool integration.

1.3 Scope

The scope of this document is limited to defining the requirements for performing IV&V of the SWG CAIS. The scope of the IV&V task has been defined as an independent testing activity which includes an informal design review to understand SWG CAIS implementation features in preparation for test design, development of a test suite for a critical subset of the SWG CAIS interfaces, and review of the SWG CAIS Installation Guide through actual utilization of the installation procedures. This document represents the first of a set of deliverables to the NATO SWG on APSE effort related to this task.

1.4 Report Content

This document describes the requirements to perform testing of the SWG CAIS. The proposed strategy for testing of the SWG CAIS is given in Section 2. Section 3 specifies the requirements for a test environment that will support the testing of the SWG CAIS implementation. As background material, an overview of IV&V techniques and their applications is provided in Appendix A. Appendix B provides a tabular listing of all CAIS validation software that has been previously developed and is available for reuse.

2.0 SWG CAIS Test Strategy

The proposed strategy for testing of the SWG CAIS implementations takes into account current test techniques both in terms of methods and available technology along with considerations that are unique to the NATO effort.

The test strategy includes an informal design review, development of a test suite for a critical subset of the SWG CAIS interfaces, and review of the SWG CAIS Installation Guide.

2.1 Informal Design Review

To better understand the SWG CAIS implementation architecture and utilize this understanding in the design of more effective tests, the test team will review the SWG CAIS implementation design on an informal basis. This review should provide a basis for tailoring existing tests and developing additional tests. The design information (APSE MOU, NATO 1986) provided will be available for each review.

The user manual (APSE MOU, NATO 1986) provided with each version of the SWG CAIS implementation will be examined from the user viewpoint, to determine if the manual reflects a reasonable user interface and accurately describes user functionality.

2.2 Test Approach

The test approach for the actual testing will be developed for the SWG CAIS implementation. The test capability will be developed using a functional or black-box approach to test specification. Tests will be generated based on the specification of the SWG CAIS and not on the specific structural details unique to a given implementation.

The tests will be run to ensure simple or nominal functionality and completeness of the interfaces. The intent is to ensure to a tool writer that the expected SWG CAIS operations are present, and that these operations are syntactically and semantically equivalent to the SWG CAIS specification.

2.2.1 Establish a Test Environment

A test environment will be set up for the tests. This test environment will consist of a test suite and a set of manual procedures that will address the execution of Ada software. The specific requirements for this technology are provided in Section 3. The major components and rationale for the test environment are provided in the following Sections.

2.2.1.1 Test Suite. The scope of the SWG CAIS test effort currently includes only those interfaces determined to be critical to NATO tool writers. Table 1 lists the packages explicitly defined in the current SWG CAIS specification, the number of unique interfaces associated with each package, and the number of parameters and exceptions associated with each package. Note that the number of unique exceptions defined for the SWG CAIS is approximately 39. These can be raised by different interfaces for similar conditions resulting in over 1600 possible exception conditions.

Table 1
Scope of SWG CAIS Interface Level Functional Testing

Reference In SWG CAIS Specification	Package	Interfaces	Parameters	Possible Exceptions Raised
5.1.1	CAIS_DEFINITIONS	0	0	0
5.1.2	CAIS_NODE_MANAGEMENT	66	146	277
5.1.3	CAIS_ATTRIBUTE_MANAGEMENT	26	73	120
5.1.4	CAIS_ACCESS_CONTROL_MANAGEMENT	12	27	48
5.1.5	CAIS_STRUCTURAL_NODE_MANAGEMENT	4	26	36
5.2.1	CAIS_PROCESS_DEFINITIONS	0	0	0
5.2.2	CAIS_PROCESS_MANAGEMENT	38	114	166
5.3.1	CAIS_DEVICES	0	0	0
5.3.2	CAIS_IO_DEFINITIONS	0	0	0
5.3.3	CAIS_IO_ATTRIBUTES	16	16	60
5.3.4	CAIS_DIRECT_IO	16	44	28
5.3.5	CAIS_SEQUENTIAL_IO	11	34	28
5.3.6	CAIS_TEXT_IO	56	79	28
5.3.7	CAIS_QUEUE_MANAGEMENT	18	141	130
5.3.8	CAIS_SCROLL_TERMINAL_IO	42	58	64
5.3.9	CAIS_PAGE_TERMINAL_IO	49	73	82
5.3.10	CAIS_FORM_TERMINAL_IO	30	45	35
5.3.11	CAIS_MAGNETIC_TAPE_IO	19	32	43
5.3.12	CAIS_IMPORT_EXPORT	2	12	24
5.3.13	SWG_CAIS_HOST_TARGET_IO	6	10	10
5.4.1	CAIS_LIST_MANAGEMENT	29	55	98
5.4.1.21	CAIS_LIST_ITEM	10	33	63
5.4.1.22	CAIS_IDENTIFIER_ITEM	11	31	69
5.4.1.23	CAIS_INTEGER_ITEM	14	31	70
5.4.1.24	CAIS_FLOAT_ITEM	11	31	70
5.4.1.25	CAIS_STRING_ITEM	10	30	67
5.5	CAIS_STANDARD	0	0	0
5.6	CAIS_CALENDAR	15	29	5
5.7	CAIS_PRAGMATICS	0	0	0
	Totals	516	1610	1621

Determining the number of interfaces in a SWG CAIS package is not always as simple as counting the procedures and functions listed in the table of contents of DOD-STD-1838. For the CAIS I/O packages, some of the interfaces are "borrowed" from the Ada Language Reference Manual with both additions and deletions.

Since not all of the interfaces listed in Table 1 can be tested within the scope of this effort, only those SWG CAIS packages determined to be "critical" for the NATO tool writers will be tested. Critical SWG CAIS packages were selected based on the perception of anticipated usage by the tool writers. Ten of twenty-nine SWG CAIS packages were selected and within these packages the critical interfaces were determined. The five data definition packages will be tested indirectly through testing of these ten selected SWG CAIS packages. Table 2 lists the critical SWG CAIS packages and corresponding critical interfaces that will be tested.

Packages within the SWG CAIS are hierarchically defined. This means that successful use of an interface at one level in the SWG CAIS will in general depend on the successful execution of other SWG CAIS packages that it depends upon. Table 3 identifies the package dependencies. The test suite structure must account for these dependencies.

Given that validation of the SWG CAIS is based on a functional approach, the results of a test can only be analyzed from the inputs and outputs of a test; it cannot make use of any internal structures or logic of an implementation to evaluate the results. This implies that in some instances it will be necessary to execute one SWG CAIS interface to determine the validity of the output of another interface. For example, a test would be developed to test the OPEN file interface. To ensure that the OPEN function works correctly, the Boolean function IS_OPEN could be used. Therefore, the OPEN test depends on successful operation of the IS_OPEN interface. The order of interface tests therefore becomes an important test consideration. Lindquist in (Lindquist, 1984), *identifies* this test issue as a "hidden interface." Tests will be designed to exercise each interface independently. If there are hidden interfaces, the dependent test is performed first. The detailed requirements for this test suite are defined in Section 3.

2.2.1.2 Test Setup. In addition to the development of the test suite, a test support capability is needed in this test environment. Procedures to incorporate such a capability will be developed. These procedures will provide support for setting up the tests, controlling the execution of a test, determining the order of execution for a test set, managing the results generated during execution, and providing a user interface for the test environment. Configuration management techniques will be utilized to control the testing process. As well, the Ada code will be designed to capture and report test results.

The tests will focus on establishing the existence of the tested SWG CAIS interfaces and identifying whether the exceptions associated with each interface exist and are correctly utilized.

The specific requirements for this capability are provided in Section 3.

2.2.1.3 Tester. Some functions that need to be performed for testing will be provided by a person called a "tester." It will be necessary to have a person perform some manual functions as a part of a test exercise. These functions include: setup of a testing session, performance of recovery or restart procedures, performance of interactive tests, review and interpretation of test results, creation of test reports, resolution of discrepancies between the

Table 2
SWG CAIS Critical Packages/Interfaces

Critical SWG CAIS Packages	Critical Interfaces
CAIS_NODE_MANAGEMENT	DELETE_NODE OPEN CLOSE COPY_NODE CREATE_SECONDARY_RELATIONSHIP DELETE_SECONDARY_RELATIONSHIP SET_CURRENT_NODE GET_CURRENT_NODE
CAIS_ATTRIBUTE_MANAGEMENT	CREATE_NODE_ATTRIBUTE CREATE_PATH_ATTRIBUTE DELETE_NODE_ATTRIBUTE DELETE_PATH_ATTRIBUTE SET_NODE_ATTRIBUTE SET_PATH_ATTRIBUTE GET_NODE_ATTRIBUTE GET_PATH_ATTRIBUTE
CAIS_STRUCTURAL_NODE_MANAGEMENT	CREATE_NODE
CAIS_PROCESS_MANAGEMENT	SPAWN_PROCESS CREATE_JOB APPEND_RESULTS GET_RESULTS WRITE_RESULTS GET_PARAMETERS CURRENT_STATUS OPEN_NODE_HANDLE_COUNT IO_UNIT_COUNT ABORT_PROCESS DELETE_JOB
CAIS_DIRECT_IO	CREATE OPEN CLOSE RESET READ WRITE SYNCHRONIZE
CAIS_IMPORT_EXPORT	IMPORT_CONTENTS EXPORT_CONTENTS
CAIS_FORM(PAGE)_TERMINAL IO	N/A

Table 2
SWG CAIS Critical Packages/Interfaces (Concluded)

Critical SWG CAIS Packages	Critical Interfaces
CAIS_SEQUENTIAL_IO	CREATE
	OPEN
	CLOSE
	RESET
	READ
	WRITE
	SYNCHRONIZE
CAIS_TEXT_IO	CREATE
	OPEN
	CLOSE
	RESET
	PUT_LINE
	GET_LINE
	SYNCHRONIZE
CAIS_LIST_MANAGEMENT	SET_TO_EMPTY_LIST
	COPY_LIST
	CONVERT_TEXT_TO_LIST
	SPLICE
	CONCATENATE_LISTS
	EXTRACT_LIST
	REPLACE
	INSERT
	DELETE
	IS_EQUAL
	KIND_OF_LIST
	KIND_OF_ITEM
	NUMBER_OF_ITEMS
	GET_ITEM_NAME
	POSITION_BY_NAME
	POSITIONS_BY_VALUE
	TEXT_FORM
	TEXT_LENGTH
	EXTRACT_VALUE
	EXTRACTED_VALUE
	MAKE_THIS_ITEM_CURRENT
	MAKE_CONTAINING_LIST_CURRENT
	POSITION_OF_CURRENT_LIST
	CURRENT_LIST_IS_OUTERMOST
	CONVERT_TEXT_TO_TOKEN
	COPY_TOKEN

Table 3
SWG CAIS Package Dependencies

Package Name	Dependent on Package	Contains/Exports Package
CAIS_PRAGMATICS	N/A	None
CAIS_STANDARD	CAIS_PRAGMATICS	None
CAIS_LIST_MANAGEMENT	CAIS_STANDARD CAIS_PRAGMATICS	CAIS_LIST_ITEM CAIS_IDENTIFIER_ITEM CAIS_INTEGER_ITEM CAIS_FLOAT_ITEM CAIS_STRING_ITEM
CAIS_DEFINITIONS	CAIS_STANDARD CAIS_LIST_MANAGEMENT	None
CAIS_CALENDAR	CAIS_STANDARD	None
CAIS_NODE_MANAGEMENT	CAIS_STANDARD CAIS_DEFINITIONS CAIS_CALENDAR CAIS_LIST_MANAGEMENT	None
CAIS_ATTRIBUTE_ MANAGEMENT	CAIS_STANDARD CAIS_DEFINITIONS CAIS_LIST_MANAGEMENT	None
CAIS_ACCESS_CONTROL_ MANAGEMENT	CAIS_DEFINITIONS CAIS_LIST_MANAGEMENT	None
CAIS_STRUCTURAL_NODE_ MANAGEMENT	CAIS_DEFINITIONS CAIS_ACCESS_CONTROL_ MANAGEMENT CAIS_LIST_MANAGEMENT	None
CAIS_PROCESS_DEFINITIONS	CAIS_DEFINITIONS CAIS_LIST_MANAGEMENT	None

Table 9
SWG CAIS Package Dependencies (Continued)

Package Name	Dependent on Package	Contains/Exports Package
CAIS_PROCESS_MANAGEMENT	CAIS_STANDARD CAIS_CALENDAR CAIS_DEFINITIONS CAIS_LIST_MANAGEMENT CAIS_PROCESS_DEFINITIONS CAIS_ACCESS_CONTROL_ MANAGEMENT	None
CAIS_IO_DEFINITIONS	CAIS_STANDARD CAIS_DEFINITIONS CAIS_LIST_MANAGEMENT	None
CAIS_IO_ATTRIBUTES	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS	None
CAIS_DIRECT_IO (GENERIC)	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS CAIS_LIST_MANAGEMENT CAIS_ACCESS_CONTROL_ MANAGEMENT	Self
CAIS_SEQUENTIAL_IO (GENERIC)	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS CAIS_LIST_MANAGEMENT CAIS_ACCESS_CONTROL_ MANAGEMENT	Self

Table 3
SWG CAIS Package Dependencies (Concluded)

Package Name	Dependent on Package	Contains/Exports Package
CAIS_TEXT_IO (GENERIC)	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS CAIS_LIST_MANAGEMENT CAIS_ACCESS_CONTROL_MANAGEMENT	Self, INTEGER_IO, FLOAT_IO, FIXED_IO, ENUMERATION_IO
CAIS_QUEUE_MANAGEMENT	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS CAIS_LIST_MANAGEMENT CAIS_ACCESS_CONTROL_MANAGEMENT	None
CAIS_SCROLL_TERMINAL_IO	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS	None
CAIS_PAGE_TERMINAL_IO	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS	None
CAIS_FORM_TERMINAL_IO	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS	None
CAIS_MAGNETIC_TAPE_IO	CAIS_STANDARD CAIS_DEFINITIONS	None
CAIS_IMPORT_EXPORT	CAIS_DEFINITIONS CAIS_LIST_MANAGEMENT	None
SWG_CAIS_HOST_TO_TARGET_IO	CAIS_STANDARD CAIS_LIST_MANAGEMENT CAIS_IO_DEFINITIONS	None

tester's and the implementor's interpretation of the specification, etc. In general, the tester will provide the human interface to the test environment.

2.2.2 Reuse Existing Technology

Several technologies currently exist that can be incorporated to varying degrees into the SWG CAIS test environment specified in Section 3. The available technologies are presented here along with the strategy adopted for their reuse in the test environment. Appendix B lists the DOD-STD-1838 interfaces, identifies which interfaces have corresponding test software, and identifies the source for this test software.

2.2.2.1 ACVC Tests. As part of the specification of the Ada language (ANSI/MIL-STD-1815A, 1983), several packages are identified as predefined library packages of any Ada compilation system. In conjunction with the development of the Ada language, an Ada Compiler Validation Capability (ACVC) has been developed and is used to ensure compliance to the Ada language standard by a given vendor (ACVC User's Guide, Version 1.9). The ACVC includes tests for each of the predefined packages. A parallel exists between several of these predefined packages and a subset of the SWG CAIS packages. These packages are: TEXT_IO, DIRECT_IO, SEQUENTIAL_IO, and CALENDAR. The equivalent SWG CAIS packages either require compliance with the Ada standard for a given interface, identify a variation on the required functionality, or specify additional interfaces to be supported (See Table 4 for a comparison of the predefined Ada language standard input/output (I/O) packages and the SWG CAIS (I/O) packages. The package CALENDAR is virtually the same for both).

The ACVC tests for those Ada interfaces that are equivalent to SWG CAIS interfaces will be considered a sufficient test set for the SWG CAIS interfaces and will be incorporated into the test suite. For those SWG CAIS interfaces that replace the Ada language standard interfaces, a study of the available test software will be made. Reuse of these tests will be determined on a test by test basis. Those interfaces that are new to the SWG CAIS will, of course, require the development of new tests.

2.2.2.2 CAIS Prototype Tests. Several prototypes of the January, 1985 version of the CAIS (Proposed, MIL-STD-CAIS) have been developed, along with some validation software. Test software from the following two prototypes is available for reuse: an in-house MITRE implementation (Study of the Common APSE Interface Set (CAIS), 1985) and a CAIS Operational Definition (CAISOD), (CAISOD, 1986).

Note that the current CAIS standard (DOD-STD-1838) does not directly map to the January, 1985 version; packages have changed in both content and syntax, and subprograms have been modified syntactically as well as semantically. In addition, the SWG CAIS specification (SWGCAIS, 1987) introduces variations on DOD-STD-1838 based on the needs of the tool writers as identified in the NATO SWG on APSE Requirements document; some packages in the SWG CAIS will be subsets of the CAIS standard or will have degenerate behavior, and an additional package has been added to the SWG CAIS specification to support host/target communication. To actually determine the reuse potential for the MIL-STD-CAIS prototype test software, it will be necessary to analyze the tests with respect to the overall test case generation strategy identified for this test environment. If any tests can be reused or

Table 4
SWG CAIS vs. Ada Language Predefined Packages

Ada Standard Function/Procedure	CAIS_DIRECT_IO	CAIS_SEQUENTIAL_IO	CAIS_TEXT_IO
CREATE	R	R	R
OPEN	R	R	R
CLOSE	R	R	R
DELETE	N	N	N
RESET	R	R	R
MODE	S	S	S
NAME	N	N	N
FORM	N	N	N
IS_OPEN	S	S	S
READ	S	S	S
WRITE	S	S	S
SET_INDEX	S	-	-
INDEX	S	-	-
SIZE	S	-	-
END_OF_FILE	S	S	S
SET_INPUT	-	-	S
SET_OUTPUT	-	-	S
STANDARD_INPUT	-	-	N
STANDARD_OUTPUT	-	-	N
CURRENT_INPUT	-	-	S
CURRENT_OUTPUT	-	-	S
SET_LINE_LENGTH	-	-	S
SET_PAGE_LENGTH	-	-	S
LINE_LENGTH	-	-	S
PAGE_LENGTH	-	-	S
NEW_LINE	-	-	S
SKIP_LINE	-	-	S
END_OF_LINE	-	-	S
NEW_PAGE	-	-	S
SKIP_PAGE	-	-	S
END_OF_PAGE	-	-	S
SET_COL	-	-	S
SET_LINE	-	-	S
COL	-	-	S
LINE	-	-	S
PAGE	-	-	S
GET	-	-	S
PUT	-	-	S
GET_LINE	-	-	S
PUT_LINE	-	-	S
SYNCHRONIZE	A	A	A

R = replace	N = non-existent in CAIS
S = same in both	A = added for CAIS
- = not in Ada or CAIS	

modified to satisfy a needed test case, the tests will be added to the test suite. Likely candidates for reuse are those developed for the CAIS_LIST_MANAGEMENT package.

2.2.2.3 CAIS Terminal I/O Tests. A virtual terminal implementation in Ada along with the documentation and acceptance tests is available in the public domain. This capability formed the precursor to the three terminal packages currently defined in the CAIS. The tests have a high degree of reusability, are well documented, and should provide good coverage of these packages.

2.2.2.4 United Kingdom Evaluation Technology. The UK is contributing tool evaluation technology as an ERB deliverable to the NATO effort. This technology consists of a test harness and a test suite that will be used to evaluate individual tool components in the APSE. The test harness is written in Ada and is available for reuse in this NATO effort. The UK test harness will be evaluated for use in the initial test environment.

2.2.3 Staged Testing Capability

The TIRB plans to stage delivery of tools and the SWG CAIS implementations. A staged development approach will also be adopted for the SWG CAIS test capability. The initial test environment will focus on testing nominal functionality and will incorporate reusable existing tests. Later versions of the test environment will add nominal tests for interfaces that are not covered by any existing tests; and will provide enhancements to the test execution control and reporting component.

The SWG CAIS developer of the first implementation will provide staged releases of the SWG CAIS implementation during the development process. Each SWG CAIS implementation release will be regression tested. As the test environment is expanded to include additional tests, these additional tests will be performed against the most recent SWG CAIS implementation release.

2.3 Review of Installation Guide

The SWG CAIS Installation Guide will be reviewed from three perspectives: completeness, consistency, and correctness. The guide will be used to perform the initial setup of the SWG CAIS implementation in preparation for the test exercises. Any errors, omissions, or discrepancies in the Installation Guide will be noted in the final test report.

3.0 Test Environment Requirements

This section defines the technology required to support the SWG CAIS test strategy.

The primary goal of the SWG CAIS Test Environment is to support the test of a SWG CAIS implementation as much as possible. When necessary, this includes determining the extent of a partial implementation of the SWG CAIS and then tailoring the tests to test only those features implemented. As well, it is necessary to ensure that the SWG CAIS interfaces function in accordance with the DOD-STD-1838 specification as modified by the SWG. This technology must permit the construction and execution of a large number of repeatable, verifiable experiments on the SWG CAIS. Secondary goals for the Test Environment are to build a tool environment which is portable, flexible, extensible, and robust.

The remainder of this requirements section is organized into three sections, intended to outline the requirements for the Test Environment. First, the test execution control requirements are presented in detail. Second, the requirements for the Test Suite are discussed. Lastly, general design goals are discussed.

3.1 SWG CAIS Test Execution Control Requirements

Several functions are required to support the process of testing the SWG CAIS: administrative functions outside the actual testing process, management of the large volumes of data expected from executing the tests, configuring the test sequences, permitting control during the execution of the tests, reporting the results of the tests, assessing the extent of the SWG CAIS implementation undergoing test, determining the effectiveness of the tests, ensuring that differences in host implementations do not interfere with test results, and communicating with the user efficiently and reliably. Each major requirement is individually defined in the following sections.

3.1.1 Test Administration

Procedures shall be developed for establishing users, defining and maintaining SWG CAIS node model instances, and performing other administrative functions.

3.1.2 Test Configuration Management

Procedures shall be established that are capable of cataloging test results, enforcing an order to the tests for regression testing of a SWG CAIS implementation, and identifying Test Suite baselines. It should also support dynamic reconfiguration of tests. The tester should be able to select an individual test or tests for execution outside the context of an integrated test suite. This allows for user selection of a test execution order that is different from any predefined order.

3.1.3 Test Invocation

Many tests can be grouped together, such as the CAIS node management tests and the CAIS list management tests. Tests may be organized into groups which exercise interfaces from the same CAIS package or share the same type of testing approach. Grouping of tests will permit more confidence that no relevant tests have been overlooked, as well as decrease the time needed to construct a test set and allow the operator to rerun the same exact group of tests. The ability to form logical, hierarchical groups of tests will also be important, since the order in which tests are executed is critical to ensuring that only tested SWG CAIS interfaces will be used to test those SWG CAIS interfaces that have not been tested. Moreover, existing groups of tests (ACVC and prototype CAIS tests) will need to be easily integrated into the test suite. Groups of tests shall also be saved for future use in a permanent form. Each group shall have a unique name which can be referenced from the context of another group of tests. Naming groups and then using the names within the context of another group is the primary mechanism for forming hierarchical groups.

3.1.4 Run-time Control

The Test Environment will be designed to run in batch mode as much as possible. Interactive input and output will also be supported only where necessary. This will most likely be needed for testing of the terminal I/O packages.

3.1.5 Implementation Assessment

In addition to testing for the conformance of any SWG CAIS implementation, its extent (the part of the SWG CAIS specification which is actually implemented) shall also be discernible. Determining the extent of the SWG CAIS implementation is most important during the early phases of SWG CAIS development as it lends itself to a staged implementation.

3.1.6 Environment Configuration Assessment

It is necessary to determine the exact configuration of the underlying software and hardware environment. The implementation dependent characteristics of the Ada development environment must be understood so that they can be taken into account during testing. Different configurations of the underlying software and hardware may influence the test results. Some limitations and variations in the architecture and host operating system which may prevent some tests from reporting correct results are:

- task scheduling algorithm
- representations of data objects
- structure of the file system
- types of terminals and printing devices supported
- available Ada pragmas
- Ada attributes

The implementation of the SWG CAIS Test Environment will permit multiple user interfaces, according to the types of devices available.

3.1.7 External Interface Requirements

The Test Environment must interact with the host hardware and the host software development environment. The following two sections address these interfaces.

3.1.7.1 Hardware Interfaces. The Test Environment is required to execute on one of the two hardware configurations specified in the NATO SWG APSE Requirements document.

3.1.7.2 Software Interfaces. Some parts of the Ada development and execution environment must be present during SWG CAIS testing. The following list is a minimum set of resources which must be available for Test Environment execution:

- A validated Ada compilation system
- SWG CAIS implementation

3.1.8 Performance Requirements

Two types of performance are applicable to the design of the SWG CAIS Test Environment: timing and capacity. Timing performance refers to the ability of the system to execute tests in a timely manner. Capacity refers to the ability of an Ada compilation system to implement correctly a given number of Ada objects. For example, Ada compilers may have limitations on the number of enumeration literals supported before they exhaust symbol table space. The Ada compilation system may have difficulty handling the large numbers of functions and procedures in the SWG CAIS Test Suite.

Performance is also important during test suite construction. The test suite shall be designed so that tests can be independently compiled, linked and executed. It shall also be possible to incrementally add to or modify the test suite with the effort required to build a new suite localized to the tests being added or changed.

The speed of execution of the Test Environment is a secondary consideration to the more important issues of complete and reliable testing. Since the SWG CAIS implementation will likely have the greatest influence on total test time, and since the speed of the SWG CAIS implementation is not controllable, SWG CAIS tests should be designed to consume as little time as possible. The tests must also be designed to minimize execution time especially in the area of user interaction, but it is expected to be a secondary performance issue in the Test Environment as a whole.

3.2 Test Suite Requirements

The testing of the SWG CAIS is based on a black-box or functional testing approach. The goals of this testing effort are to ensure the existence and syntactic correctness of the SWG CAIS interfaces in a given implementation, and to ensure compliance to the semantic intent of the specification. The first goal can be met in a rather straightforward, brute force manner since the syntax of the interfaces is formally defined through Ada packages. The Ada compiler will do extensive static syntax checks as well as static semantic checks of the declared data types. Ensuring conformance to the semantics of the specification is more difficult, however, since the semantics of the SWG CAIS are not formally specified.

The tests consist of nominal tests which ensure simple or nominal functionality and completeness of the interfaces.

The details of the nominal tests are addressed in the Section 3.2.1. Items in the SWG CAIS specification which are global to the entire specification are addressed in Section 3.2.2. Section 3.2.3 specifies conventions to be followed for configuration management of the test names. The final results will be contained in a report as addressed in the Section 3.2.4.

3.2.1 Nominal Testing

In nominal testing, each "critical" SWG CAIS interface as defined in Table 2 shall be individually examined at the simplest or nominal level. An interface is a primary interface, an overload, or an additional interface. Each interface is unique even though some interfaces may share the same name. Although the focus of an interface test is the test of a particular interface, the actual test will often require the use of other SWG CAIS interfaces. Use of additional interfaces will be necessary to establish a SWG CAIS node model instance, to set the node(s) state, to traverse a part of the node model, to a specific node prior to test execution, or to ensure correctness of the interface under test. Therefore, the ability to execute tests for most interfaces will also depend on the success of other interface tests. The dependencies between interfaces will be derived during test generation for the individual interfaces. There are also dependencies among the SWG CAIS packages as previously identified in Table 4. Both forms of dependency will force a specific ordering of the nominal tests or force the combining of interface tests.

It is possible to identify sets of SWG CAIS interfaces which, for testing purposes, are mutually dependent, such as OPEN and IS_OPEN. If identification of a small "core set" of mutually dependent SWG CAIS interfaces upon which the rest of the SWG CAIS interfaces depend were possible, then the "core set" of interfaces could be tested by some other means (either by verification or mathematical proof, for example). The rest of the SWG CAIS interfaces could then be tested using only previously tested SWG CAIS interfaces. Testing would then be a process of using only already-tested SWG CAIS interfaces to produce other tested SWG CAIS interfaces.

However, it is not possible to identify a small "core set" of mutually dependent SWG CAIS interfaces upon which the rest of the SWG CAIS interfaces depend. This is probably because an interface set is designed not to be redundant in the first place.

Since the identification of a "core set" seems unlikely and requirements exist for black-box testing of the SWG CAIS interfaces on two architectures, the process of testing one SWG CAIS interface must depend on other SWG CAIS interfaces to both create the pre-conditions and evaluate the post-conditions for each test. Testing of the SWG CAIS must then proceed inductively, assuming for a given test that one set of SWG CAIS interfaces is correct when using that set to test another SWG CAIS interface.

Nominal tests must show that an implementation behaves as expected under both normal and exceptional conditions. To ensure this, the tests will set up illegal as well as legal input conditions to induce both normal and exceptional responses. These tests will examine each interface with simple test data sets to determine if the interface has been implemented beyond stubbing and to determine if each exception defined as relevant for that interface can be raised.

It is beyond the scope of the current test effort to perform exhaustive testing. This type of testing will determine the thoroughness of the interface implementation by aggressively attempting to locate errors in the semantics of each interface through more extensive test data sets.

3.2.1.1 Nominal Test Setup. A separate test will be written for each critical interface. This set of critical interfaces has been defined previously in Table 2. The purpose of this test is to demonstrate minimal functionality through the use of a set of simple test cases. Existence tests will check for the correct functioning of an interface using valid parameter values, and exception tests will check for the SWG CAIS implementation's ability to raise each exception.

Each test shall consist of a simple exercise of an interface. Input values will be defined for each parameter of mode "in" or "in out" as well as the actual node model instance needed (if any) prior to test execution. Expected values will be defined for each parameter of mode "in out" or "out" (if any), for each function return value (if any), for the expected exception to be raised (if any), and for the expected node model instance following test execution (if any).

The existence tests will check for the proper operation of an interface. Any exceptions raised will indicate that the expected conditions were not met. The exception tests will check for expected exceptions and expected exceptions that are not raised will indicate that the expected conditions were not met. The tests will consist of the following steps:

- Ensure that a correct SWG CAIS node model instance exists prior to execution of this test. This implies the creation of a SWG CAIS node model instance, the use of a predefined SWG CAIS node model instance, or the use of a previously generated node model instance created by a previously executed test. This will require the use of other SWG CAIS interfaces.

- Execute the interface with the predefined input values.

- Compare the actual profile values with the expected profile values.

- Compare the resultant SWG CAIS node model instance with the expected SWG CAIS node model instance. This may involve the use of other SWG CAIS interfaces.

- Compare the actual exception raised (if any) with the expected exception.

3.2.1.2 Nominal Test Success Criteria. A nominal test will be considered successful if the following criteria are met:

- The test reaches completion. Note that an interface used to establish the preconditions for this test could raise an exception. The test would then be considered invalid.

- The expected output profile matches the actual output profile for an existence test. For an exception test, the actual exception raised matches the expected exception.

- The expected SWG CAIS node model instance matches the actual SWG CAIS node model instance.

The status returned from each nominal test will be one of the following:

- Pass -- the nominal test was successful according to the stated success criteria.

- Fail/reason -- the nominal test was not successful according to the stated success criteria. The reason portion of the status will identify which of the criteria was not met (i.e., SWG CAIS node model instance was not correct, an unexpected exception was raised, a parameter was not correct, or the interface does not exist).

Note that some exceptions defined in the SWG CAIS cannot be raised directly through an external testing mechanism. For instance, the exception TOKEN_ERROR cannot be raised directly since a token is a limited private type that cannot be explicitly provided as input to a test. Such exceptions cannot and will not be explicitly tested.

3.2.1.3 Predefined Attributes and Relations. The SWG CAIS specification identifies two forms of predefined information: relations and attributes (See DOD-STD-1838, Appendix A). Tests shall be defined to ensure that the SWG CAIS implementation supplies the required predefined information for the critical interfaces. These tests shall be incorporated into the interface tests.

3.2.2 Test Name Configuration Management

Configuration management of the tests is needed to identify which tests are run against the SWG CAIS implementation. The suite of SWG CAIS tests is expected to be quite large. There are over 500 functions and procedures (SWG CAIS interfaces) in the thirty-two packages comprising DOD-STD-1838. The SWG CAIS has additional interfaces defined in the package SWG_CAIS_HOST_TARGET_IO. For each interface, a test must be constructed.

Since there is a large number of SWG CAIS tests, configuration management of the test names must be rigorous.

The method for generating test names must satisfy several requirements. The objective of these requirements is to identify the purpose and scope of a SWG CAIS test. Configuration management of the SWG CAIS Test Suite applies to both the different tests and their associated files (e.g., input and output files)

The specific requirements for generation of test names are:

- The test name shall uniquely identify the interface undergoing test. While several test case names may correspond to one SWG CAIS interface, in no circumstance will one test case name correspond to more than one SWG CAIS interface.
- The test names shall be methodically generated such that identification of the SWG CAIS interface being tested and the exact component of that interface being tested shall be related to DOD-STD-1838 and be simple to look up.
- The test names shall allow for overloaded versions of the same interface.
- The test names shall allow for identification of pragmatics being tested for that set of global tests defined in section 3.2. These names should also be related to DOD-STD-1838.

3.2.3 Reporting Results

The final test report will identify what the tests are supposed to accomplish, and include any relevant statistics regarding the number and type of tests run. A summary analysis of the test results will be presented.

3.3 Design Goals

This section describes additional design goals for the Test Environment. Where possible, the goals of flexibility, extensibility, robustness, and portability should be met.

3.3.1 Flexibility

Flexibility is the ability to change SWG CAIS test suite configurations with a minimum of delay. Flexibility can be accomplished by establishing and adhering to standards that will support future efforts in changing software to accommodate changes in requirements relating to the mission, function or associated data.

3.3.2 Extensibility

Extensibility is the ability to add new tests to the Test Suite at minimum cost. Extensibility of the Test Suite can be accomplished by establishing and adhering to syntactic and semantic standards for tests. One form of syntactic standard, the configuration management of test names, has already been mentioned. The configuration management of test names contributes to the overall extensibility as well as maintainability of the Test Suite by providing a taxonomy of tests. It makes clear which tests have been written, which tests are currently needed, and which intended purpose a SWG CAIS test serves.

Each test must set up the node model instance required for the test and verify its pre- and post- state (see Section 3.2). Therefore, each test can be executed independently of any other test. This philosophy supports ease of test suite extensibility.

The ability to extend the Test Environment to support SWG CAIS implementation testing should also be considered in its design. For instance, the tests should not preclude the inclusion and instrumentation of performance measurement functions if needed.

3.3.3 Robustness

Many tests in the Test Suite will raise exceptions. Two types of exceptions will be raised by these tests; those which are expected to be raised, and those which are not expected to be raised. Some tests will be written explicitly to observe the correct raising of an exception defined by the SWG CAIS specification. These exceptions are expected. If an unanticipated exception is raised in the SWG CAIS under test or if the test has an error which raises the wrong exception, then the exception will be unexpected.

In dealing with both types of exceptions, one overriding concern is to not have execution stop because of an exception that was not anticipated in the test's exception handler. Tests which are likely to raise exceptions shall have exception handlers embedded within them. Tests which may cause one of several exceptions to be raised shall explicitly handle the exceptions with relevant exception handlers embedded within their bodies. An unexpected exception shall also be trapped at the test level and its occurrence reported back.

3.3.4 Portability

Portability is the ability to move a software system from one computer to a dissimilar computer with few or no changes to the source code. Ideally, recompilation to produce new object code files and relinking are the only changes required to effect a move to another "Ada machine." Portability issues must be addressed at three levels: the computer, the host operating system, and the Ada compiler.

Portability can be achieved by limiting the use of implementation dependent features of the Ada language and the underlying machine. Some host dependencies will be required, such as access to the file system to store test results and configurations of tests to be run, but reliance on such features shall be justified in the design. The results database shall be portable. Individual tests shall also isolate nonportable features into separate portability packages. The Test Environment shall be written using Ada constructs that maximize its portability and minimize the rehosting effort.

APPENDIX A

Overview of Verification & Validation Techniques

A literature search of verification, validation, and testing techniques was performed as part of this effort. This appendix presents a summary of the findings for both the state-of-the-practice and the state-of-the-art. It is included here for completeness since the cited references influenced the SWG CAIS Independent Verification and Validation (IV&V) strategy.

In a software development life cycle, verification is the process used to determine correctness, completeness, and consistency in the transformation process from one phase to the other. Validation, on the other hand, has no regard for the intermediate phases and instead focuses on the degree to which the final product conforms to the original specification (See (Meyers, 1979); (Adrion et al, 1982); (FIPS-PUB-101, 1983); (Glass, 1979); (Lindquist, 1984)). The literature search revealed that the techniques available for performing IV&V of the software development process depend on the degree of formalism introduced into the software development activities. If the software is specified with a rigorous (mathematical) notation, a formal approach is said to be used in the development process and hence formal IV&V techniques can be applied; otherwise, an informal approach has been adopted. In general the use of formal representations of the software and corresponding validation and verification based on these representations is the exception rather than the rule in the current state-of-the-practice. Such formalism is usually only introduced into the development of life-critical or secure software.

A.1.0 Verification Techniques

In the development of a software system, two levels of verification are necessary:

- design verification --to show consistency, completeness, and correctness between the requirements and the design.
- code verification -- to show consistency, completeness, and correctness between the implementation and the design.

Techniques that can be employed at each of these verification steps are described here for both a formal and an informal approach.

A.1.1 Formal Verification Methods

Formal verification "is the process by which mathematical reasoning is used to show a system satisfies its requirements," (Nyberg et al, 1985). The use of formal verification requires the use of formal specifications at each stage of development. Formal verification is then used to evaluate the software at each stage using a consistency proof. This proving technique ensures correctness in the transformation process from one specification level to the next. Use of such techniques requires highly trained personnel and additional resources above and beyond the normal development resources. Therefore, such techniques are usually restricted to critical software such as a secure operating system (Walker et al, 1979).

A.1.2 Informal Verification Methods

If an informal approach has been adopted for the software development effort, there are no formal specifications available that rigorously show transformations from the requirements to the design to the code. Some informal verification techniques have evolved that are generally referred to as static analysis techniques. The techniques presented here correspond to the two verification activities of interest in this effort: design and code verification.

The generally accepted method of design verification is to conduct a design review upon completion of the design and prior to start of coding. Glass in (Glass, 1979) identifies the following items to be examined in a design review:

- External interfaces
- Internal interfaces
- Critical timing requirements
- Overall structure with respect to requirements allocation
- Human-software interactions

In (McCabe, 1980), a list of topics and detailed questions are posed for design review areas that include:

- System structure -- risk areas, functional dependencies
- Reliability -- areas that will be potentially unreliable
- Unknowns -- user interface, hardware characteristics
- Efficiency -- bottleneck components
- Modularity -- software engineering principles applied
- Hardware -- what effect failures will have on the software
- Algorithms -- complexity, performance
- Overall system concept

- Assumptions made -- hardware, user, other software deliveries
- Portability -- separation and degree of system dependent features
- Failure -- accommodation of failure detection/recovery in the design

Some combination of these review topics should be assessed for a given design as part of the design review process.

At the implementation level, two verification activities are normally performed: code inspections and unit testing. Code inspections are done for each module to ensure adherence to the design and to the coding standards. (Glass, 1979; Meyers, 1979). These inspections are performed as desk exercises and require a static review of the code prior to execution on the machine. A code inspection conducted by a developer and attended by an IV&V representative, constitutes an independent verification of the code for correctness, consistency and completeness. Developers normally unit test their software units in a development environment. This testing is designed to verify the structural logic of the unit. The adequacy of the unit testing is then independently verified by examination of the unit test data and results.

A.2.0 Validation Techniques

The generally accepted approach to validation is to use testing to dynamically analyze the code with respect to its intended functionality through execution of the implementation in a test environment. More specifically, a test suite is developed that exercises the software as completely and as rigorously as possible from a functional viewpoint; that is, the test suite focuses on the external view of the software as seen by the user rather than the internal or structural perspective. There are three items that need to be addressed in the development of a test suite: test case generation, test drivers, and test coverage.

A.2.1 Test Case Generation

The key issue in developing a validation capability is the generation of a minimally complete set of test cases. The completeness of the test cases is usually measured in terms of structural coverage achieved or through some statistical analysis techniques (See section A.2.4). A test case is defined to include both a set of inputs as well as the expected output(s). Since testing is an incomplete process by nature (unless exhaustive testing is performed over every possible value for each parameter) the subset of tests selected must be chosen in an intelligent, well-defined manner over the entire set of tests possible.

It is difficult to provide a structured view of test case generation techniques since many of the techniques can be applied at various testing levels and can be combined into an integrated strategy. As Beizer points out in (Beizer, 1983), a technique that may be considered a structural testing technique at one level of testing, may be viewed as a functional testing technique at another level. In an attempt to segment the discussion, the various techniques are examined from the degree of formality introduced at the specification level. If a formal specification of the semantics is available, numerous approaches exist for generating test cases from these formal semantics. If the specification is informally specified using a natural language, there are also numerous techniques available, but they are not rigorous and therefore lend themselves less to automation.

A.2.1.1 Artificial Intelligence Approaches

The use of artificial intelligence techniques has been successfully applied in the area of test case generation. Logic programming has been used in the validation of a set of UNIX kernel calls (Pesch et al. 1985). In the original specification of the kernel calls, the syntax of each system function was formally defined. However, the semantic intent of the function was specified in a natural language form. Rather than develop formal specifications for the system functions, test specifications were developed using a formal test specification language. Each test specification consists of two parts: facts about the system call and a set of test cases specified by rules. Each rule consists of a set of preconditions, the system call to be tested and the value it returns, and a set of post conditions. These test case specifications are then implemented in PROLOG.

A.2.1.2 Algebraic Specification Approaches

Another approach combines the use of an available algebraic specification and logic programming techniques for generation of test cases. This approach assumes the existence of an algebraic specification that formally defines the semantics of the intended implementation. These specifications are then used to automatically derive functional test data sets using a logic programming tool (Bouge et al, 1985). This technique is further refined by using a constraints-handling Prolog that limits the number of test data sets generated (Choquet, 1987).

In (Ostrand, 1985) the author advocates "transforming each equation of an algebraic specification into a procedure whose parameters are the equation's free variables." The procedure is then supplied with test cases that are executed. If the algebraic equations are satisfied by the test case parameters, a result of true is returned; otherwise, a result of false is supplied.

A.2.1.3 Assertion Testing

Another variant on the use of formal methods for testing software is to specify the intended behavior of a program via assertions. Assertions are logical expressions that state conditions that must be true at various steps in the program. The assertions are inserted into the actual implementation and processed into the base language by a preprocessor. These assertions are then evaluated to true or false during execution of a test. Two test data generation techniques that are examined in (Andrews, 1985) for use with assertions are grid and adaptive test data generation. Grid testing uses knowledge about the range of values that a parameter can have. This technique uses this information to generate input values for one or two parameters at fixed incremental points through their respective ranges. Assertion violations are monitored and reported on post-mortem. In adaptive test generation, the assertion violations are employed to generate and execute additional test cases. A feedback loop is used to generate new values for the input variables based on the assertion violations detected. This level of testing is primarily a structural level testing technique since assertions are introduced into the implementation and therefore an understanding of the internal logic is required.

For an Ada-based development, research is ongoing into the derivation of an assertion-based annotational language. This language (known as ANNA) augments an Ada package specification with assertions(Luckham & von Henke, 1984; von Henke et al, 1985). These assertions describe the intended behavior, or semantics, of the Ada package. The assertions are introduced as special comments into the package specification and are then preprocessed into executable Ada code.

A.2.1.4 Cause-effect graphs

Given an informal specification of the semantics of a software component, cause-effect graphs (Meyers, 1979; Collofello and Ferrara, 1984) can be used to transform the informal specification into a formal logic network representation. Causes represent input conditions while effects represent either a system transformation or an output. Causes and effects are connected using Boolean logic. The resultant graph is then used to derive test cases by selecting an effect and deriving the input conditions that must be satisfied to produce it. The use of these graphs is effective for small units but becomes unwieldy for large systems. One of the advantages to this technique is that it highlights inconsistencies in the original specification. A disadvantage is that it is not effective at analyzing boundary conditions.

A.2.1.5 Equivalence-class partitioning

Another way to derive test cases is by partitioning each input parameter's possible values into one or more equivalence classes (Meyers, 1979; Collofello and Ferrara, 1984). Both valid and invalid classes are defined. Test cases are then derived by selecting actual values in each equivalence class such that the maximum number of classes are covered with the minimum number of tests.

A.2.1.6 Boundary value analysis

It has been shown empirically that test cases that focus on boundary values provide a better rate of error discovery than those that explore intermediate values. Boundary value analysis builds on the equivalence partitioning technique. Equivalence partitioning creates equivalence classes for input parameters only. Boundary value analysis also partitions the output parameter space, then uses the derived partitions to select values at, above, and below the upper and lower bounds of each input and output parameter partition (Meyers, 1979; Collofello and Ferrara, 1985).

The Ada language tests some boundary level conditions itself. This is particularly true for declared subtypes (MIL-STD-1815A). An object that is defined as a subtype and passed as a parameter to a subprogram will have its values evaluated at subprogram execution time. Any attempt to go outside the bounds of the subtype constraints will result in a constraint error at run-time.

A.2.1.7 Random testing

The random test data generation technique attempts to minimize the number of cases that need to be executed by employing probability distributions to assist in test case selection. Meyers in his now classic book on software testing (Meyers, 1979) disregarded random testing as a viable technique. New studies (Ince, 1987) challenge this opinion, but the author suggests more empirical studies are needed before definitive conclusions can be drawn. In any case, random testing is strictly applicable at the structural test level where some degree of coverage of the control flow is required.

A.2.2 Test Driver Generation

In addition to techniques available for generating test cases, techniques also exist for automated test driver generation. One recent project has developed a method for creating test drivers for Ada packages (Besson and Queyras, 1987). The work provides a test environment generator composed of a package level driver, virtual bodies for non-implemented units, and a command language interpreter for setup and evaluation of test cases. Test drivers are derived from analysis of the Descriptive Intermediate Attributed Notation for Ada (DIANA) representation of the package specification. This intermediate form of an Ada program captures all of the syntactic and semantic intent of the code. Using this information, the software builds a test driver for the package that will execute the subprograms specified in the package. Since a package can have visibility into other packages through the use of context clauses, it is important to ensure consistency between package test drivers. This test environment generator ensures that if package specifications have changed, new drivers for that package and all its dependent packages are automatically generated.

A.2.3 Test Coverage Analysis

Validation test cases are derived from the functional specification and not from any knowledge of the structural components of the implementation. To determine how well the test suite is covering the internal representation of the software requires some form of test coverage measurement. Several test coverage techniques have been defined that determine measurement of internal coverage received during execution of the test suite. Coverage achieved can be measured in terms of control or data, or via an error seeding technique. Each of these measurement techniques is examined here.

Control flow coverage metrics analyze the coverage acquired over the logical structure of the software for a given set of test cases. Various measurements are defined for this form of coverage metric to include: program unit(s) executed, statements within a unit executed, branches executed, and paths through a unit (Meyers, 1979; Sneed, 1986; Collofello and Ferrara, 1984). To produce these metrics, the source is typically instrumented with probes that are used to maintain counts, identify the decision points exercised and the paths traversed during a test execution. However, empirical studies of test coverage versus errors detected (Sneed, 1986) show that such test coverage metrics do not tend to give confidence in the reliability of the software tested.

A more recently developed coverage metric analyzes coverage in terms of the data rather than the control flow (Sneed, 1986). This measurement technique requires that software specifications be formally developed based on an assertion method for each data item, function, and condition. These assertions are then used to derive the specified data usage. The subsequent code is then analyzed to determine the programmed data usage. During test case execution, actual data usage is determined. Measurements are given that compare actual to programmed usage, and programmed to specified usage. Initial studies indicate that "by requiring 90% data coverage twice as many program errors were discovered than with branch coverage, especially errors of omission and computing error as well as boundry [sic] errors." (Sneed, 1986).

A final technique introduced by Harlan Mills (Meyers, 1979) is termed error seeding or "bebugging." The approach introduces known errors into the software, executes the test software, and determines how many of the known errors were detected. Based on the number of errors discovered, a projection is made as to the number of errors remaining. This technique requires the knowledge of the internal structure of the code.

APPENDIX B

Available CAIS Validation Software

Table B-1 provides a summary of all the validation software currently available for reuse in the SWG CAIS validation effort. The table lists all packages defined in DOD-STD-1838. For each interface that has corresponding reusable test software, the source of the test software is indicated by the following codes:

- M -- MITRE prototype tests
- C -- CAISOD prototype tests
- A -- ACVC tests
- V -- Virtual terminal I/O tests

No reusable tests exist for the following packages: *access control*, *process management*, magnetic tape I/O, and import/export. These packages represent approximately 14% of the total number of SWG CAIS interfaces. It should be noted that the SWG CAIS specification only requires that discretionary access control be fully implemented in the access control package. Mandatory access control must be present with degenerate behavior. The SWG CAIS also defines a new package, SWG_CAIS_HOST_TARGET_IO, for which no test software exists.

Table B-1
Available CAIS Validation Software

CAIS Specification Packages vs. Available Validation Software	
CAIS - DOD-STD-1838	Available Tests
5.1.1 Package CAIS_DEFINITIONS	
5.1.2. Package CAIS_NODE_MANAGEMENT	
5.1.2.1 OPEN	M
5.1.2.2 CLOSE	M
5.1.2.3 CHANGE_INTENT	
5.1.2.4 IS_OPEN	
5.1.2.5 INTENT	
5.1.2.6 KIND_OF_NODE	
5.1.2.7 OPEN_FILE_HANDLE_COUNT	
5.1.2.8 PRIMARY_NAME	M
5.1.2.9 PRIMARY_KEY	M
5.1.2.10 PRIMARY_RELATION	M
5.1.2.11 PATH_KEY	
5.1.2.12 PATH_RELATION	
5.1.2.13 BASE_PATH	
5.1.2.14 LAST_RELATION	
5.1.2.15 LAST_KEY	
5.1.2.16 IS_OBTAINABLE	M
5.1.2.17 IS_SAME	M
5.1.2.18 INDEX	
5.1.2.19 OPEN_PARENT	M
5.1.2.20 COPY_NODE	M
5.1.2.21 COPY_TREE	M
5.1.2.22 RENAME	M
5.1.2.23 DELETE_NODE	
5.1.2.24 DELETE_TREE	M
5.1.2.25 CREATE_SECONDARY_RELATIONSHIP	
5.1.2.26 DELETE_SECONDARY_RELATIONSHIP	
5.1.2.27 SET_INHERITANCE	
5.1.2.28 IS_INHERITABLE	

Table B-1
Available CAIS Validation Software (Continued)

CAIS Specification Packages vs. Available Validation Software	
CAIS - DOD-STD-1838	Available Tests
5.1.2.29 Node iteration types and subtypes	N/A
5.1.2.30 CREATE_ITERATOR	
5.1.2.31 MORE	
5.1.2.32 APPROXIMATE_SIZE	
5.1.2.33 GET_NEXT	
5.1.2.34 SKIP_NEXT	
5.1.2.35 NEXT_NAME	
5.1.2.36 DELETE_ITERATOR	
5.1.2.37 SET_CURRENT_NODE	
5.1.2.38 GET_CURRENT_NODE	
5.1.2.39 TIME_CREATED	
5.1.2.40 TIME_RELATIONSHIP_WRITTEN	
5.1.2.41 TIME_CONTENTS_WRITTEN	
5.1.2.42 TIME_ATTRIBUTE_WRITTEN	
5.1.3. Package CAIS_ATTRIBUTE_MANAGEMENT	
5.1.3.1 CREATE_NODE_ATTRIBUTE	M
5.1.3.2 CREATE_PATH_ATTRIBUTE	M
5.1.3.3 DELETE_NODE_ATTRIBUTE	M
5.1.3.4 DELETE_PATH_ATTRIBUTE	M
5.1.3.5 SET_NODE_ATTRIBUTE	M
5.1.3.6 SET_PATH_ATTRIBUTE	M
5.1.3.7 GET_NODE_ATTRIBUTE	M
5.1.3.8 GET_PATH_ATTRIBUTE	M
5.1.3.9 Attribute iteration types	N/A
5.1.3.10 CREATE_NODE_ATTRIBUTE_ITERATOR	M
5.1.3.11 CREATE_PATH_ATTRIBUTE_ITERATOR	M
5.1.3.12 MORE	M
5.1.3.13 APPROXIMATE_SIZE	
5.1.3.14 NEXT_NAME	
5.1.3.15 GET_NEXT_VALUE	M
5.1.3.16 SKIP_NEXT	
5.1.3.17 DELETE_ITERATOR	
5.1.4 Package CAIS_ACCESS_CONTROL_MANAGEMENT	none
5.1.5 Package CAIS_STRUCTURAL_NODE_MANAGEMENT	
5.1.5.1 CREATE_NODE	M
5.2 CAIS process nodes	N/A
5.2.1 Package CAIS_PROCESS_DEFINITIONS	N/A
5.2.2 Package CAIS_PROCESS_MANAGEMENT	none

Table B-1
Available CAIS Validation Software (Continued)

CAIS Specification Packages vs. Available Validation Software

CAIS - DOD-STD-1838	Available Tests
5.3 CAIS input and output	N/A
5.3.1 Package CAIS_DEVICES	N/A
5.3.2 Package CAIS_IO_DEFINITIONS	N/A
5.3.3 Package CAIS_IO_ATTRIBUTES	none
5.3.4 Package CAIS_DIRECT_IO	
5.3.4.1 Definition of types	N/A
5.3.4.2 CREATE	A
5.3.4.3 OPEN	A
5.3.4.4 CLOSE	A
5.3.4.5 RESET	A
5.3.4.6 SYNCHRONIZE	
** Ada Language Subprogram equivalents **	
MODE	A
NAME	A
FORM	A
IS_OPEN	A
READ	A
WRITE	A
SET_INDEX	A
INDEX	A
SIZE	A
END_OF_FILE	A
5.3.5 Package CAIS_SEQUENTIAL_IO	
5.3.5.1 Definition of types	N/A
5.3.5.2 CREATE	A
5.3.5.3 OPEN	A
5.3.5.4 CLOSE	A
5.3.5.5 RESET	A
5.3.5.6 SYNCHRONIZE	
** Ada Language Subprogram equivalents **	
MODE	A
NAME	A
FORM	A
IS_OPEN	A
READ	A
WRITE	A
END_OF_FILE	A

Table B-1
Available CAIS Validation Software (Continued)

CAIS Specification Packages vs. Available Validation Software

CAIS - DOD-STD-1838	Available Tests
5.3.6 Package CAIS_TEXT_IO	
5.3.6.1 Definition of types	N/A
5.3.6.2 CREATE	M,A
5.3.6.3 OPEN	M,A
5.3.6.4 CLOSE	A
5.3.6.5 RESET	M,A
5.3.6.6 SYNCHRONIZE	
** Ada Language Subprogram equivalents **	
MODE	A
FORM	A
IS_OPEN	A
READ	A
WRITE	A
END_OF_FILE	A
SET_INPUT	A
SET_OUTPUT	A
CURRENT_INPUT	A
CURRENT_OUTPUT	A
SET_LINE_LENGTH	A
SET_PAGE_LENGTH	A
LINE_LENGTH	A
PAGE_LENGTH	A
NEW_LINE	A
SKIP_LINE	A
END_OF_PAGE	A
SET_COL	A
SET_LINE	A
COL	A
LINE	A
PAGE	A
GET	M,A
PUT	M,A
GET_LINE	A
PUT_LINE	A

Table B-1
Available CAIS Validation Software (Continued)

CAIS Specification Packages vs. Available Validation Software	
CAIS - DOD-STD-1838	Available Tests
5.3.7 Package CAIS_QUEUE_MANAGEMENT	none
5.3.8 Package CAIS_SCROLL_TERMINAL_IO	V
5.3.9 Package CAIS_PAGE_TERMINAL_IO	V
5.3.10 Package CAIS_FORM_TERMINAL_IO	V
5.3.11 Package CAIS_MAGNETIC_TAPE_IO	none
5.3.12 Package CAIS_IMPORT_EXPORT	none
5.3.13 Package SWG_CAIS_HOST_TARGET_IO	none
5.4.1 Package CAIS_LIST_MANAGEMENT	
5.4.1.1 Types, subtypes, constants, and exceptions	N/A
5.4.1.2 COPY_LIST	M,C
5.4.1.3 SET_TO_EMPTY_LIST	
5.4.1.4 CONVERT_TEXT_TO_LIST	M
5.4.1.5 TEXT_FORM	M
5.4.1.6 IS_EQUAL	M,C
5.4.1.7 DELETE	M,C
5.4.1.8 KIND_OF _LIST	M,C
5.4.1.9 KIND_OF_ITEM	M
5.4.1.10 SPLICE	M,C
5.4.1.11 CONCATENATE_LISTS	M,C
5.4.1.12 EXTRACT_LIST	M,C
5.4.1.13 NUMBER_OF_ITEMS	M,C
5.4.1.14 POSITION_OF_CURRENT_LIST	
5.4.1.15 CURRENT_LIST_IS_OUTERMOST	
5.4.1.16 MAKE_CONTAINING_LIST_CURRENT	
5.4.1.17 MAKE_THIS_ITEM_CURRENT	
5.4.1.18 TEXT_LENGTH	M,C
5.4.1.19 GET_ITEM_NAME	M,C
5.4.1.20 POSITION_BY_NAME	M
5.4.1.21 Package CAIS_LIST_ITEM	
5.4.1.21.1 EXTRACT_VALUE	M,C
5.4.1.21.2 REPLACE	M
5.4.1.18 INSERT	M
5.4.1.19 POSITION_BY_VALUE	M
5.4.1.20 Package CAIS_IDENTIFIER_ITEM	
5.4.1.20.1 TO_TOKEN	M,C
5.4.1.20.2 TO_TEXT	M,C
5.4.1.20.3 IS_EQUAL	M,C
5.4.1.20.4 EXTRACT	M,C
5.4.1.20.5 REPLACE	M,C
5.4.1.20.6 INSERT	M,C
5.4.1.20.7 POSITION_BY_VALUE	M,C

Table B-1
Available CAIS Validation Software (Concluded)

CAIS Specification Packages vs. Available Validation Software	
CAIS - DOD-STD-1838	Available Tests
5.4.1.21 Package CAIS_INTEGER_ITEM	
5.4.1.21.1 TO_TEXT	C
5.4.1.21.2 EXTRACT	C
5.4.1.21.3 REPLACE	C
5.4.1.21.4 INSERT	C
5.4.1.21.5 POSITION_BY_VALUE	C
5.4.1.22 Package CAIS_FLOAT_ITEM	
5.4.1.22.1 TO_TEXT	C
5.4.1.22.2 EXTRACT	C
5.4.1.22.3 REPLACE	C
5.4.1.22.4 INSERT	C
5.4.1.22.5 POSITION_BY_VALUE	C
5.4.1.23 Package CAIS_STRING_ITEM	
5.4.1.23.1 EXTRACT	M,C
5.4.1.23.2 REPLACE	M,C
5.4.1.23.3 INSERT	M,C
5.4.1.23.4 POSITION_BY_VALUE	C
5.5 Package CAIS_STANDARD	N/A
5.6 Package CAIS_CALENDAR	
5.6.1 Definition of types, subtypes and exceptions	N/A
5.6.2 CLOCK	A
5.6.3 YEAR	A
5.6.4 MONTH	A
5.6.5 DAY	A
5.6.6 SECONDS	A
5.6.7 SPLIT	A
5.6.8 TIME_OF	A
5.6.9 +	A
5.6.10 -	A
5.6.11 Comparison operators	A
5.7 Package CAIS_PRAGMATICS	N/A

REFERENCES

"Using the ACVC Tests ," ACVC Version 1.9.

"Ada Programming Language," ANSI/MIL-STD-1815A, Department of Defense, 22 January 1983.

"Guideline for Lifecycle Validation, Verification, and Testing of Computer Software," FIPS-PUB-101, U.S. Department of Commerce/National Bureau of Standards, June 6, 1983.

"Military Standard Common APSE Interface Set (CAIS)," Proposed MIL-STD-CAIS, Department of Defense, 31 January 1985.

Ada Programming Support Environments (APSEs) Memorandum of Understanding (MOU), NATO, 10 October, 1986.

"Common Ada Programming Support Environment (APSE) Interface Set (CAIS)," DOD-STD-1838, Department of Defense, 9 October 1986.

Terms of Reference for the Evaluation Review Board for the Special Working Group on Ada Programming Support Environments, NATO, December 11, 1986.

Terms of Reference for the Tools and Integration Review Board for the Special Working Group on Ada Programming Support Environments, NATO, December 11, 1986.

Introduction to the CAIS Operational Definition Documentation, Arizona State University, October, 1986.

"Specifications for the Special Working Group Common Ada programming Support Environment (APSE) Interface Set (CAIS) Implementations," US-Trondheim-002, NATO, 18 June 1987.

NATO SWG APSE Requirements, NATO, 25 August 1987.

Andrews, Dorothy M., "Automation of Assertion Testing: Grid and Adaptive Techniques," in *Proceedings of the Eighteenth Hawaii International Conference on System Sciences*, ed. Sprague, R.H., Jr., vol. 2, pp. 692-9, Western Periodicals Co., Honolulu, HI, USA, 1985.

Beizer, Boris, *Software Testing Techniques,* Van Nostrand Reinhold Company, 1983.

Benzel, T. V., "Analysis of a Kernel Verification," in *Proceedings of the 1984 Symposium on Security and Privacy,* pp. 125-131, IEEE Computer Society Press, 29 April - 2 May, 1984.

Besson, M. and Queyras, B., "GET: A Test Environment Generator for Ada," in *Ada components: libraries and tools, Proceedings of the Ada-Europe International Conference*, ed. Sven Tafvelin, pp. 237-250, Cambridge University Press, 26-28 May 1987.

Booch, Grady, *Software Engineering Components in Ada*, 1987.

Bowerman, Rebecca, Gill, Helen, Howell, Charles, Reagan, Tana, and Smith, Thomas, "Distributing the Common APSE (Ada Programming Support Environment) Interface Set (CAIS) ." MTR-86W00181, The MITRE Corporation, January 1987. Contract #: F19628-86C-0001

Bowerman, Rebecca E., "Study of the Common APSE Interface Set (CAIS)," WP-85W00537, The MITRE Corporation, 1 October 1985.

Carney, David J., "On The CAIS Implementation," IDA Memorandum Report M-481, Institute For Defense Analyses, June 1988.

Choquet, N., "Test Data Generation Using a Prolog with Constraints," in *Workshop on Software Testing, Banff, Canada*, 1985.

Collofello, Dr. James S. and Ferrara, Anthony F., "An automated Pascal multiple condition test coverage tool," in *Proceedings COMPSAC 84.*, pp. 20-26, IEEE Computer Society Press, 7-9 November 1984.

Glass, Robert L., *Software Reliability Guidebook*, Prentice Hall, 1979.

Henke, Friedrich W. von, Luckham, David, Krieg-Brueckner, Bernd, and Owe, Olaf, "Semantic Specification of Ada Packages," in *Ada in use: Proceedings of the Ada International Conference, Paris 14-16 May 1985*, ed. Gerald A. Fisher, Jr., vol. V, pp. 185 - 196, Cambridge University Press, September, October 1985.

Ince, D. C., "The Automatic Generation of Test Data," *The Computer Journal*, vol. 30, no. 1, pp. 63-69, February, 1987.

Lindquist, Timothy E., Facemire, Jeff, and Kafura, Dennis, "A Specification Technique for the Common APSE Interface Set," 84004-R, Computer Science Dept., VPI, April, 1984.

Lindquist, Timothy E., Freedman, Roy S., Abrams, Bernard, and Yelowitz, Larry, "Applying Semantic Description Techniques to the CAIS," in *the Formal Specification and Verification of Ada*, ed. W. Terry Mayfield, pp. 1-1 - 1-30, 14-16 May 1986.

Luckham, David and Henke, Friedrich W. von, "An Overview of Anna, A Specification Language for Ada," Technical Report No. 84-265, Computer Systems Laboratory, Stanford University, September 1984.

McCabe, Thomas J., *Structured Testing*, 1980.

McKinley, Kathryn L. and Schaefer, Carl F., "DIANA Reference Manual," IR-MD-078, Intermetrics, Inc., 5 May 1985. Contract N00014-84-C-2445

Myers, Glenford J., *The Art of Software Testing*, John Wiley & Sons, 1979.

Nyberg, Karl A., Hook, Audrey A., and Kramer, Jack F., "The Status of Verification Technology for the Ada Language," IDA Paper P-1859, Institute for Defense Analyses, July, 1985.

Osterand, T. J., "The Use of Formal Specifications in Program Testing," in *Third International Workshop on Software Specification and Design*, pp. 253-255, IEEE Computer Society Press, 26-27 August 1985.

Pesch, Herbert, Schnupp, Perter, Schaller, Hans, and Spirk, Anton Paul, "Test Case Generation Using Prolog," in *Proceedings of the 8th International Conference on Software Engineering*, pp. 252-258, 28-30 August, 1985.

Sneed, Harry M., "Data Coverage Measurement in Program Testing," *IEEE*, pp. 34-40, IEEE Computer Society Press, 1986.

W.R., Adrion and al, et, "Validation, Verification, and Testing of Computer Software," *ACM Computing Surveys*, vol. 14, no. 2, pp. 159-192, ACM, June, 1982.

Walker, B. J., Kemmerer, R.A., and Popek, G. J., "Specification and Verification of the UCLA UNIX Security Kernel," in *Proceedings of the Seventh Symposium on Operating Systems Principles*, pp. 64-65, ACM, New York, USA, 10-12 December, 1979.

Wu, Liqun, Basili, Victor R., and Reed, Karl, "A Structure Coverage Tool for Ada Software Systems," in *Proceedings of the Joint Ada Conference*, pp. 294-301., 1987.

GLOSSARY

Acronyms

ACVC	Ada Compiler Validation Capability
AIE	Ada Integrated Environment. An Ada Programming Support Environment project funded by the Air Force and contracted to Intermetrics Inc.
AJPO	Ada Joint Program Office. The office charged with the success of the Ada programming language.
ALS	Ada Language System. An Ada Programming Support Environment project funded by the Army and contracted to Softech, Inc.
APSE	Ada Programming Support Environment. The complete set of Ada development tools described by the "Stoneman" document, including the Ada compiler, linker, editor, debugger, etc.
CAIS	Common APSE Interface Set. The proposed standard (DOD-STD-1838) operating system interfaces for all Ada projects.
CAISOD	CAIS Operational Definition; a partial implementation of MIL-STD-CAIS, January 1985.
CIVC	CAIS Implementation Validation Capability.
CMS	Conversational Monitoring System. A trademark of International Business Machines, Inc.

DoD	Department of Defense. The organization which identified the need for a common, modern, high-order computer programming language.
DRB	Demonstration Review Board. One of four boards established by the NATO MOU. The main objective of the board is to coordinate and review the demonstration of an APSE capability through the use of two weapons systems scenarios, as the basis for the holistic APSE evaluation.
ERB	Evaluation Review Board. One of four boards established by the NATO MOU. The main objective of the work is to coordinate and review the specification and development of methods and tools for the evaluation of APSE tools and the demonstration of this technology, where possible, on the tools and the SWG CAIS.
IRB	Interface Review Board. One of four boards established by the NATO MOU. The main objective of the board is to coordinate and review the development of the requirements and specification of an interface standard for APSEs, based upon review of the evolutionary interface developments (including CAIS and PCTE), to be recommended for adoption and use by NATO and nations.
IV&V	Independent Verification and Validation
KAPSE	Kernel APSE. The level of an APSE that presents a machine independent portability interface to an Ada program.
KIT	KAPSE Interface Team.
MC68020	A 32-bit microprocessor produced by the Motorola Corporation.
MMI	Man Machine Interface.

MOU	Memorandum of Understanding. The form which NATO agreements take.
NATO	North Atlantic Treaty Organization.
SWG	Special Working Group
SWG CAIS	Title given to the specific CAIS implementation being developed for the NATO effort.
TIRB	Tools and Integration Review Board. One of four boards established by the NATO MOU. The main objective of the work is to coordinate and review the specification, developmnet and integration of a group of software tools representative of a usable APSE through their initial implementation on two distinct computer architectures using an agreed interface set.
UNIX	A widely-used operating system originally developed by Bell Telephone Laboratories.
VAX	Virtual Address eXtension. A trademark of Digital Equipment Company. The name of a widely-used computer system from Digital Equipment Company.
VMS	Virtual Memory System. A trademark of Digital Equipment Company. The operating system for a VAX computer.

Terms

Ada package	A program unit that allows for the specification of a group of logically related entities. A package normally contains a specification and a body.
Bebugging	The process of intentionally introducing errors into a program as a means of determining effectiveness of program testing.
Black-box testing	A testing approach that examines an implementation from an external or "black-box" perspective. The test cases are designed based on the functional specification and do not make use of any structural or internal knowledge.
Dynamic analysis	A validation technique that evaluates a product through actual execution of it.
Evaluation	The process used to quantify the fitness for purpose of an item in terms of its functionality, usability, performance and user documentation.
Exception	Error or other exceptional situation that arises during the execution of a program.
Formal specification language	A precise language used to convey the semantics or meaning of a computer program.
Formal verification	A process that employs formal mathematical proofs to show correctness of a specification or implementation with respect to its predecessor specification.
Functional testing	See black-box testing
Grey-box testing	A form of testing that blends techniques from both black-box and white-box testing.
Interface	A function or procedure defined in a CAIS package specification. It provides a tool writer with a standard mechanism for performing a system level service without knowledge or access to the underlying system architecture.

Metric	A quantifiable indication of the state of an entity.
Node Model Instance	A particular realization of nodes, relationships and attributes produced through execution of a set of CAIS interfaces.
Stoneman	The requirements document for an APSE; published by the Department of Defense.
Subprogram	A program unit that is executed by a subprogram call. The call can be in either the form of a function or a procedure.
Test case generation	The process of determining both the inputs to drive a test and the expected test results.
Test data	The set of inputs needed to execute a test.
Test driver	A software component that is used to exercise another software component under test.
Validation	The process used to determine the degree of conformance of an end product to its original specification.
Verification	The process used to determine the correctness of each transformation step in the development process.
White-box testing	A class of testing that examines the internal structure of software.