(1)

Design Specification for Test and Evaluation of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Implementation

Deborah M. Haydon
Annette Englehart
Mike McClimens
Jonathan D. Wood
Phil Yu

23 February 1989

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

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

890910

UNCLASSIFIED

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) Design Specification for Test & Evaluation of the NATO Common Ada Programming Support Environment INTERFACE SET IMPLEMENTATION	5. TYPE OF REPORT & PERIOD COVERED'
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) HAYDON, D. ENGLEHART, A McClimens, M. Wood, J. Yu, P.	8. CONTRACT OR GRANT NUMBER(s) F19628-89-C-0001
9. PERFORMING ORGANIZATION AND ADDRESS MITRE CORP. 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 23 Feb 89
	13. NUMBER OF PAGES 83
(If different 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)

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

DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

A North Atlantic Treaty Organization (NATO) Special Working Group (SWG) on Ada Programming Support Environments (APSEs) 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 design to perform test and evaluation of SWG Common APSE Interface Set (CAIS) implementations. 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 1,600 possible error conditions. This report outlines an approach and specifies the design for the development of the test and evaluation environment. The design outlines the tests to be developed and discusses attributes of the test environment that influence the design of the test suite. This test suite will include two categories of tests. The first category will test for nominal functionality and completeness of the interfaces by exercising each of the interfaces deemed critical in the Requirements for Test and Evaluation of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) (Mularz, 1988). The second category will test for overall usability of SWG CAIS capabilities by exercising combinations of the critical interfaces typically found in the APSE tools. There will be two SWG CAIS implementations installed on two different host architectures. This report provides the design for the proposed test and evaluation of the SWG CAIS implementations.

ACKNOWLEDGMENTS

We would like to thank the MITRE peer reviewers, Robbie Hutchison and Tom Smith, 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 preliminary drafts of this paper.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	xiii
LIST OF TABLES	x
EXECUTIVE SUMMARY	xi
1.0 INTRODUCTION	1
1.1 Background	1
1.2 Objective	3
1.3 Scope	4
1.4 Document Organization	4
2.0 SWG CAIS TEST AND EVALUATION APPROACH	5
2.1 Develop Test Environment	5
2.2 Develop Test Suite	5
2.3 Reuse Existing Technology	6
2.4 Support Staged Test and Evaluation Capability	13
2.5 Incorporate Design Goals	13
2.5.1 Flexibility	13
2.5.2 Extensibility	14
2.5.3 Robustness	14
2.5.4 Portability	14
3.0 TEST ENVIRONMENT	17
3.1 Test Administration Procedures	18
3.2 Test Configuration Management Procedures	19

TABLE OF CONTENTS (Continued)

	Page
3.3 Test Invocation	19
3.4 Run-time Control	19
3.5 Implementation Assessment	19
3.6 Environment Configuration Assessment	20
3.7 User Requirements	20
3.8 External Interface Requirements	20
3.8.1 Hardware Interfaces	20
3.8.2 Software Interfaces	20
3.9 Performance Requirements	21
3.10 Test Support	21
3.10.1 Timing	22
3.10.2 Tree Walkers	22
4.0 TEST SUITE	23
4.1 General	23
4.1.1 Nominal Testing	23
4.1.2 Usability Testing	24
4.1.3 Test Name Standards	25
4.1.4 Input/Output Standards	26
4.1.5 Test Design Standards	26
4.1.6 Test Success Criteria Standards	31
4.2 Node Management Tests	31
4.2.1 Test Strategy	32
4.2.2 Specific Support Packages	32
4.2.3 Organization of Node Model	33

TABLE OF CONTENTS (Continued)

	Page
4.2.4 Nominal Tests	33
4.2.5 Usability Tests	35
4.3 Attribute Management Tests	35
4.3.1 Test Strategy	36
4.3.2 Specific Support Packages	36
4.3.3 Organization of the Node Model	36
4.3.4 Nominal Tests	38
4.3.5 Usability Tests	39
4.4 Structural Node Management Tests	40
4.4.1 Test Strategy	40
4.4.2 Specific Support Packages	40
4.4.3 Organization of the Node Model	40
4.4.4 Nominal Tests	40
4.4.5 Usability Tests	42
4.5 Process Management Tests	44
4.5.1 Test Strategy	44
4.5.2 Specific Support Packages	44
4.5.3 Organization of Node Model	44
4.5.4 Nominal Tests	45
4.5.5 Usability Tests	48
4.6 Direct IO/Sequential IO/Text IO Tests	51
4.6.1 Test Strategy	51
4.6.2 Specific Support Packages	51
4.6.3 Organization of Node Model	52
4.6.4 Nominal Tests	52
4.6.5 Usability Tests	54
4.7 Import_Export Tests	54
4.8 Page_Terminal_IO Tests	54

TABLE OF CONTENTS (Concluded)

	Page
4.8.1 Nominal Tests	54
4.8.2 Usability Tests	54
4.9 List Management Tests	55
4.9.1 Test Strategy	55
4.9.2 Specific Support Packages	56
4.9.3 Organization of Node Model	56
4.9.4 Nominal Tests	56
4.9.5 Usability Tests	59
APPENDIX : SWG CAIS Test Suite Traceability Matrix	61
REFERENCES	67
GLOSSARY	71

LIST OF FIGURES

Figure Number		*Page*
1	Elements of the NATO SWG APSE	2
2	SWG CAIS Input Data File Sample	27
3	SWG CAIS Output Data File Sample	28
4	SWG CAIS Procedure Documentation Sample	29
5	SWG CAIS Attribute Management Test Node Models	37
6	SWG CAIS Node Model Structures	41

LIST OF TABLES

Table Number		*Page*
1	Scope of SWG CAIS Interface-Level Functional Testing	7
2	SWG CAIS Critical Packages/Critical Interfaces	8
3	SWG CAIS Package Dependencies	10

EXECUTIVE SUMMARY

In October 1986, nine North Atlantic Treaty Organization (NATO) nations signed a Memorandum of Understanding (MOU) that established a Special Working Group (SWG) on Ada Programming Support Environments (APSEs). 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 enhancement of the Common APSE Interface Set (CAIS), which was established as a Department of Defense (DOD) standard in October 1986. This enhancement, termed the SWG CAIS, was subsequently baselined on 25 August 1988. 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 sponsored by the Ada Joint Program Office (AJPO) and supporting the NATO SWG on APSEs, MITRE has responsibility for the test and evaluation of the SWG CAIS implementations.

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 integration of the SWG CAIS implementation with the SWG APSE. There are two categories of testing activities. The first category includes development of a test suite to perform nominal testing (i.e., testing of individual interfaces) for a critical subset of the SWG CAIS interfaces. The second category of testing activities includes enhancement of the test suite to incorporate SWG CAIS usability testing (i.e., testing combinations of interfaces typical of APSE tools).

This paper provides the design for the proposed test and evaluation of the SWG CAIS. The design is based on a functional, or "black-box," approach that precludes any knowledge of an implementation's internal operation. The design reflect the requirements put forth in Requirements for Test and Evaluation of the NATO Common Ada Programming Support Environment (APSE) Interface Set (Mularz, 1988). This paper also discusses attributes of the test environment that influence the design of the test suite, including the following: test configuration management, reporting of test results, documentation of procedures for executing the tests, the need to assess partial SWG CAIS implementations, and efficient running of the test suite. General design goals and standards are described for the tests, and each test to be developed is described in terms of the test objective, basic test approach, and results to be reported.

1.0 INTRODUCTION

This document defines the design of the test suite and the approach for a test environment to support test and evaluation of the Common Ada Programming Support Environment (APSE) Interface Set (CAIS) implementation developed for the Ada Joint Program Office (AJPO) in conjunction with the North Atlantic Treaty Organization (NATO) Special Working Group (SWG) on APSE, hereafter referred to as the SWG CAIS implementation.

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 1980's. However, it is recognized by the Department of Defense, 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, integrate, and maintain such systems on a common set of software is referred to as a programming support environment.

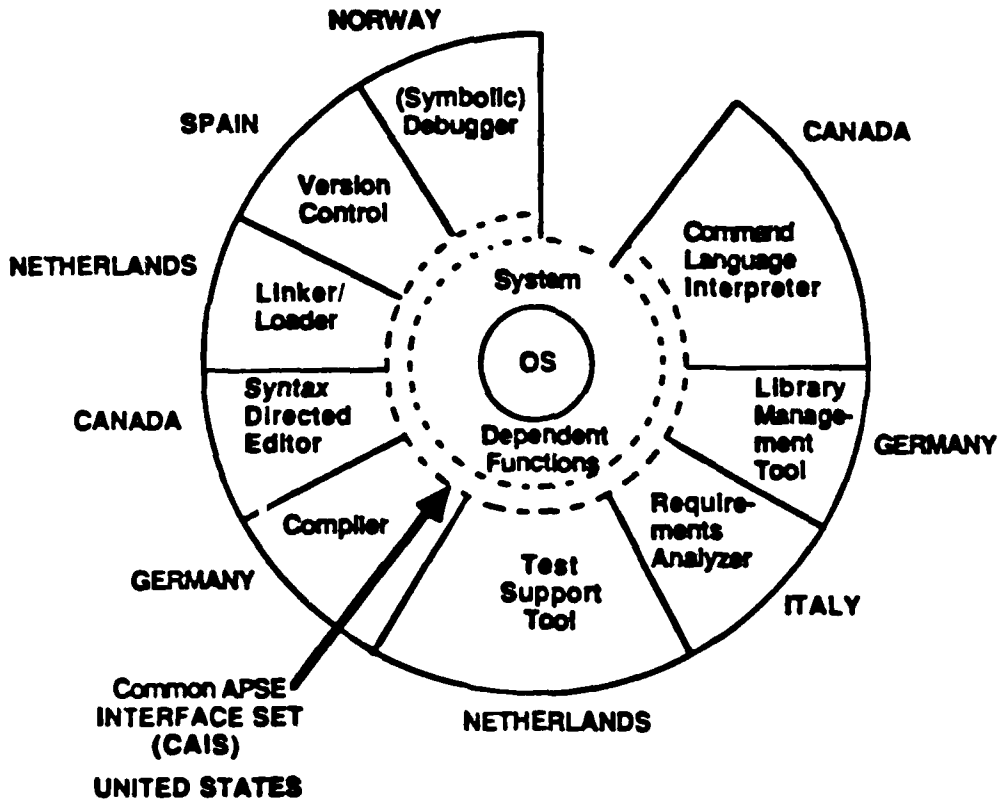
A STONEMAN-based APSE consists of a tool set and a system-level interface set (STONEMAN, August 1980). The interface set provides kernel-level functionality in an abstract, consistent manner with specific system mapping embodied in a particular interface implementation. Use of these interfaces by a tool developer promotes transportability of APSE tools across disparate architectures.

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 host architectures 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.

The NATO SWG APSE is based upon a STONEMAN model. While the goal of the STONEMAN model is 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 (DOD,1986). The SWG CAIS will be developed for two host architectures: Digital Equipment Corporation (DEC) VAX/VMS and a yet-to-be-determined architecture. Transportability of the NATO APSE will be demonstrated by a porting 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

Figure 1
Elements of the NATO SWG APSE



Tools and Integration Review Board (TIRB), the Demonstration Review Board (DRB), the Interface Review Board (IRB), and the Evaluation Review Board (ERB).

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 its respective APSE component. Specifically, the United States is responsible for implementation of the SWG CAIS on the two host architectures.

Since embedded systems support was a primary concern in the development of Ada and APSEs, the NATO project includes APSE support for 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 participants will develop evaluation technology and will use it to assess the individual components of the APSEs that are developed by the TIRB participants. Both the United States and the United Kingdom (UK) have specific tasks within this board. The United States is tasked with performing test and evaluation of the SWG CAIS implementations. The United Kingdom is responsible for evaluation of tools within the NATO APSE.

The "Requirements for Test and Evaluation of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Implementation" document identified the technology required to support test and evaluation of the SWG CAIS implementation (Mularz, 1988). This document presents the design for those tests that compose the test and evaluation activities.

1.2 Objective

The objective of this task is to design a limited set of tests that will test the SWG CAIS implementation for functionality and usability and also provide a level of confidence to the SWG CAIS users (i.e., tool developers), that the SWG CAIS implementation is sufficiently tested prior to tool integration.

Evaluation is the process used to quantify the fitness for purpose of an item in terms of its functionality, usability, performance, and user documentation. The test suite for the test and evaluation of the SWG CAIS implementations comprises two categories: nominal tests and usability tests. Nominal tests check for proper functionality of individual interfaces, and usability tests check sequences of interface calls for both usability and performance. The sequences of interface calls are chosen to reflect typical usage by tool writers. Procedures for a detailed evaluation of documentation have not been developed because DOD-STD-1838 and the SWG Addendum provide detailed descriptions of the functionality of SWG CAIS

implementations. Evaluation of users manuals and installation guides for SWG CAIS implementations, however, will be performed using standard check lists.

1.3 Scope

The scope of this document is limited to defining the design for the tests that will be used in test and evaluation activities for the SWG CAIS implementation. The scope of the test and evaluation task has been defined as a set of testing activities that is independent of the SWG CAIS developer. There are two categories of testing activities. The first category, nominal testing, includes the testing for correctness in the functionality of critical interfaces. These critical interfaces are defined in Section 2.2. The second category, usability testing, includes an enhancement of the nominal test activities to test combinations of SWG CAIS interfaces typical of APSE tools. Exhaustive testing is outside the scope of the current effort. This design of the test and evaluation environment and test suite is specifically targeted to the DEC VAX/VMS SWG CAIS implementation.

1.4 Document Organization

This document describes the design for the tests that will be used to perform test and evaluation of the SWG CAIS implementation. Section 2 gives the approach for evaluation of the SWG CAIS implementation. Section 3 identifies the test environment that will support the testing of the SWG CAIS implementation. Section 4 includes design information on the test suite used for both nominal and usability testing. The appendix provides a traceability matrix showing the names of the interfaces exercised by each test.

2.0 SWG CAIS TEST AND EVALUATION APPROACH

The approach for test and evaluation of the SWG CAIS implementations takes into account current test and evaluation techniques both in terms of methods and available technology along with considerations that are unique to the NATO effort. The test and evaluation approach includes development of a test environment, which will consist of a set of manual and automated procedures that will address the execution of tests and the collection of results, as well as the development of a test suite of Ada programs. The reuse of existing technology and support of a staged test and evaluation capability is considered, as is the incorporation of design goalare such as flexibility, extensibility, robustness, and portability.

The approach for the development of the test environment is described further in Section 2.1. The approach for the development of the test suite is further described in Section 2.2. (Specific design information on the test environment is provided in Section 3, and information on the test suite is provided in Section 4.) Reuse of existing technology is described in Section 2.3, support of a staged test and evaluation capability is described in Section 2.4, and incorporation of design goals is described in Section 2.5.

2.1 Develop Test Environment

A test environment consisting of a set of test procedures (both automated and manual) and guidelines in controlling the execution of the tests will be developed for the test and evaluation process. The primary goal of the SWG CAIS test procedures is to support the test and evaluation of a SWG CAIS implementation in a controlled manner and to be able to support test and evaluation of partially completed SWG CAIS implementations. The design of the tests takes into account the requirement that the SWG CAIS interfaces function in accordance with the DOD-STD-1838 specification as modified by the SWG. The design of the test procedures takes into consideration the need for the construction and execution of a large number of repeatable tests on the SWG CAIS implementation.

2.2 Develop Test Suite

The evaluation 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.

Some tests will be run to determine simple or nominal functionality and completeness of the interfaces; other tests will be run to determine the usability of the SWG CAIS. The intent is to determine that the tested SWG CAIS operations are syntactically and semantically equivalent to the SWG CAIS specification and that the SWG CAIS implementation is usable.

The scope of the SWG CAIS test and evaluation effort currently includes only those interfaces determined to be critical to NATO SWG APSE 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. The unique exceptions defined for the SWG CAIS can be raised

by different interfaces for similar conditions resulting in over 1,600 possible exception conditions.

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 a priori to be "critical" for the tool writers will be tested. Critical SWG CAIS packages were selected based on the perception of anticipated usage by the NATO SWG APSE tool writers. Ten of thirty-four 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 on which it depends. Table 3 identifies the package dependencies. The design of the test suite structure will account for these dependencies.

Given that test and evaluation of the SWG CAIS implementation 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 evaluate 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 evaluation consideration. Lindquist identifies this evaluation issue as a "hidden interface" (Lindquist, 1984).

Tests will be designed to exercise each interface independently. If there are hidden interfaces, whenever possible, the dependent test is performed first. The evaluation tests will measure interface timing and capacity limits. Some evaluation tests will mimic specific SWG APSE tools/SWG CAIS interactions in assessing the usability of SWG CAIS interfaces and strategies for making best use of SWG CAIS interfaces. The detailed design for this test suite is defined in Section 4.

2.3 Reuse Existing Technology

Several technologies currently exist that were considered for incorporation into the SWG CAIS test environment. The available technologies were presented in the Requirements for Test and Evaluation of the NATO Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Implementation document (Mularz, 1988) and included Ada Compiler Validation Capability (ACVC) tests, CAIS prototype tests, CAIS Terminal Input/Output (I/O) test, Naval Ocean Systems Center (NOSC) List Management tests, and the United Kingdom test harness and test suite evaluation technology.

These existing tests and the UK test harness were all reviewed for possible reuse in the SWG CAIS test and environment capability. All of the List Management tests originally developed by NOSC and the CAIS Terminal I/O test were reused and composed the List

Table 1
Scope of SWG CAIS Interface-Level Functional Testing

SWG CAIS Spec. Ref.	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.

Table 2
SWG CAIS Critical Packages/Critical 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/Critical 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 9
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 CAIS_ACCESS_CONTROL	None
CAIS_ACCESS_CONTROL_MANAGEMENT		
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 3
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 CAIS_DEVICES	None
CAIS_IO_ATTRIBUTES	CAIS_STANDARD CAIS_DEFINITIONS CAIS_IO_DEFINITIONS CAIS_DEVICES	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

Management and CAIS Terminal I/O portions of the test suite. The NOSC tests were modified to allow batch processing instead of interactive processing, and to produce output files that contain the test results.

The remaining tests were not directly usable, however, because they overlapped the coverage provided by the NOSC list management tests, they focused on Ada issues rather than SWG CAIS issues, they were heavily implementation-dependent, or they were packaged to exercise large groups of interfaces rather than individual interfaces. The UK test harness was not used originally because of risks in using it while developing tests on a partially completed SWG CAIS implementation. The SWG CAIS test suite and test environment have been designed to allow evolution towards use of the UK test harness once the test suite and SWG CAIS implementation are complete. The test suite also incorporates methods of testing exceptions and of performing automatic checking of test results that were used in the existing test technology.

2.4 Support Staged Test and Evaluation Capability

The TIRB plans to stage delivery of tools and the SWG CAIS implementations. A staged development approach was considered for the SWG CAIS test and evaluation capability, but not incorporated into the final design. Designs for all tests are included in this document. Included are test designs for testing nominal functionality as well as for usability.

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 independently using the test and evaluation process. The test configuration management procedures outlined in Section 3.3 address the issue of regression testing.

2.5 Incorporate Design Goals

Additional design goals of flexibility, extensibility, robustness, and portability were considered and where possible incorporated into the design. In each of the following sections, examples of how each design goal was incorporated into the design are given.

2.5.1 Flexibility

Flexibility is the ability to change SWG CAIS test suite configurations with a minimum of effort. Flexibility is 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. Naming conventions, configuration management guidelines that facilitate identification and collection of test results, and systematic techniques in designing tests that exercise, where possible, only one interface all support this goal of flexibility. As well, use of common reporting packages will make the tests more flexible by improving modularity.

2.5.2 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.

The test design included, where possible, the setup of the state of the node model instance required for the test and verification of its pre- and post-state. Therefore, each test can be executed independently of any other test. This philosophy supports ease of test suite extensibility.

Some of the tests will use input data files, allowing test cases to be added or modified by editing a data file.

The ability to extend the test environment to support SWG CAIS implementation evaluation further was considered in its design. For instance, the tests do not preclude the inclusion and instrumentation of performance measurement functions if needed.

2.5.3 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 implementation under test or if the test has an error which raises the wrong exception, the exception will be unexpected

In dealing with both types of exceptions, one overriding concern is to not have test execution terminate because of an exception that was not anticipated in the test's exception handler. Rather, there should be orderly error recovery procedures from the SWG CAIS when limits are exceeded or exceptions are encountered. Tests that are likely to raise exceptions shall have exception handlers embedded within them. Tests that 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.

2.5.4 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 are 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 data base that contains the results shall be portable. Both the test environment and 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. The test design takes this design goal into account. Tests may also use packages that translate the logical names referenced in the tests to host specific file names. The TG_Names package is an example. Use of a single user node, specifically "current_user" will permit other users to execute a test without modifying the source code of the test. The test suite and test environment are designed to be portable; however, implementation-dependent features (e.g., Ada I/O statements and host command procedures) will be utilized. To the extent possible, these implementation-dependent features will be isolated.

3.0 TEST ENVIRONMENT

The test environment is the set of manual and automatic procedures required to control the execution of a large number of repeatable verifiable experiments on the SWG CAIS implementations. Each test of the SWG CAIS implementation is realized as a sequence of Ada statements. A test may be implemented as a main program, or several tests may be grouped together as individual procedures under the control of the main Ada program. In either case, the test environment must support the design goals of flexibility, extensibility, robustness, and portability across the following processes required for running the tests:

- Compilation and linking of test code
- Installation of executable tests into the CAIS environment
- Initiation of test execution
- Collection of test results
- Generation of test reports

Furthermore, the test environment must support the requirement to test partially completed SWG CAIS implementations. That is, each test must focus on a single SWG CAIS interface or a group of closely related SWG CAIS interfaces, and a test must not depend upon other tests being executed prior to its own execution (i.e., they must not rely on any resultant state of the SWG CAIS).

The use of a comprehensive test harness normally offers the best approach for providing a good test environment and was initially considered. However, the requirement to test partial implementations of the SWG CAIS involves risks in integrating the implementation under test with a large test harness. Therefore, environment support for the SWG CAIS test suite will be provided first as a collection of manual and automated procedures, design considerations, and organizational decisions that together form an environment for the efficient management of the test suit.

The test environment will be designed to allow migration to the use of a test harness as that technology, the SWG CAIS implementation, and the SWG CAIS test suite itself matures. The fundamental requirements for this migration are that the tests be modularized for individual execution and that the conventions for test naming and organization ensure unique identification of tests and test results. The use of a test harness will provide for automatic report generation, increase the automation of the test procedures, and further the design goals of flexibility, extensibility, and portability. Therefore, the initial test environment will contain the following:

- Main Ada programs, where appropriate, that collect together several tests as independent subprograms (thereby reducing the number of executable load modules).

- Host command files to compile and link tests.
- Host command files to execute tests.
- Templates to assist the manual generation of test reports.

This initial test environment will then be enhanced, as the test suite, SWG CAIS implementation, and test harness mature by undertaking the following:

1) Use of the test harness to collect test results and generate test reports.

2) Replacement of host command files with test harness abstract control files, i.e., test control sequences.

The support required for testing SWG CAIS implementations includes test administration procedures for establishing node model states, test configuration management, test invocation, execution (i.e., run-time) control, the ability to assess partial SWG CAIS implementations, identification of how the operational environment is affected by system configuration, user documentation of test inputs and expected outputs, limitation of hardware and software dependencies in the test, and test suite performance issues. These elements of test support are individually defined in the following sections.

3.1 Test Administration Procedures

Many tests will be grouped together in sequences, such as the CAIS node management tests and the CAIS list management tests. Tests will be organized into sequences that exercise interfaces from the same CAIS package or share the same type of testing approach. This grouping approach of sequences 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 sequence of tests. The ability to form logical sequences of tests will also be important since the order in which tests are executed is critical to ensuring that SWG CAIS interfaces used to implement the test of another interface will have been tested prior to the test that is dependent upon them. There will be a means of saving and re-executing test sequences. Test sequences shall be saved either in job control files or in main Ada programs.

The procedures for test administration also require that each test include methods for creating and controlling the state of the node model as required for proper execution of each particular test. The test setup includes establishing user-independent pathnames; defining and maintaining an appropriate set of SWG CAIS node model instances (so that tests may be run in either a single-user or multi-user mode); and creating the expected set of nodes, relationships, and attributes prior to the actual invocation of series of tests and then deleting any modifications to the node model upon test completion. The test setup and test cleanup functions will be kept separate from the actual test code either as independent procedures or as consequences of command procedure instructions.

3.2 Test Configuration Management Procedures

The purpose of the test configuration management procedures for the SWG CAIS test and evaluation effort is to keep track of all of the information relevant during testing: test source files, test libraries, test executables, input data files, formal output files, trace files, logged session files, baselines of various versions of the SWG CAIS, and test results.

The configuration management scheme to support the tracking of this information may be simple, but should be consistently enforced. Naming conventions will be adopted to associate the various types of files that apply to the same test, and directories will be used to maintain the separateness of these files and to group files of the same type. Source files, test input data files, and test output report files should be maintained in separate directories. All tests will obtain their test data from the input directory and will output formal reports and trace files to the output directory. Tests may also output status messages to the screen. Templates for reports will list all test names, and test output files will list the versions of both the test source code and the test input file.

3.3 Test Invocation

The tests must be able to be run within the environment created by the SWG CAIS implementation. The general approach is to set up a SWG CAIS node instance by creating a user whose Default_Tool relation points to a file node that contains an executable that is either a single SWG CAIS test or a controller that allows individual tests to be run. The test or controller is then invoked by logging into the CAIS.

3.4 Run-time Control

The test environment is designed to run in batch mode as much as possible. Interactive input and output are supported only where necessary, such as for testing of the TERMINAL I/O package and for debugging purposes.

3.5 Implementation Assessment

As mentioned in Section 2.4, a staged development approach was considered for the SWG CAIS test and evaluation capability. The actual requirement is to be able to test partial SWG CAIS implementations. This requirement does not relieve the SWG CAIS test and environment capability from addressing the full scope of the SWG CAIS. Rather, it increases the difficulty by constraining the test construction and requiring greater flexibility within the test environment. Thus, tests must be targeted to individual interfaces or to small groups of closely related interfaces. Each SWG CAIS implementation release will be regression tested independently using the test and evaluation process. During regression testing, a new release is tested against the standard test suite to determine whether previously failed tests now pass and to verify that previously passed tests still pass. Thus, the extent of the SWG CAIS implementation completion and conformance is determined.

3.6 Environment Configuration Assessment

Because each test must be compiled and linked with SWG CAIS implementations under test, it is necessary to determine the exact configuration of the underlying software and hardware environment. Installation of a SWG CAIS implementation will undoubtedly involve the setting of various system capacities, access authorities, and other configuration options. The setting of such system parameters will affect both the capacities and the efficiency of the implementation. Proper and consistent setting of these variables will be required for correct execution of the tests. The implementation-dependent characteristics of the Ada development environment must be understood so that they can be taken into account during testing. The behavior and results of tests (especially process management tests) will be affected by task or process scheduling algorithms (priority allocation schemes).

As well, the SWG CAIS tests should not depend on all SWG CAIS interfaces during the testing process since all SWG CAIS interfaces may not be fully operational. Therefore, some host-dependent services must be used. These host-dependencies will be limited to dependencies arising from the use of Ada I/O facilities. In particular, SWG CAIS tests will depend upon the host compiler's implementation of Text_Io and upon the syntax of host filenames.

3.7 User Requirements

The procedures for test setup, test execution, and interpretation of test results will be documented in a user's manual.

3.8 External Interface Requirements

The test environment will interact with the host hardware and the host software development environment. The following two sections address these interfaces.

3.8.1 Hardware Interfaces

There are no explicit hardware requirements in the SWG CAIS implementation test environment. All host dependencies are limited to the Ada compilation system, the SWG CAIS implementation, and the host operating system. The test environment does assume the ability to print files and to interact via a terminal. Disk and memory capacities required by the test environment correspond to those required by the SWG CAIS implementation and are within the limits specified in the NATO SWG APSE requirements document (NATO, 1987).

3.8.2 Software Interfaces

The following list is a minimum set of resources that will be available for test environment execution:

- A validated Ada compilation system with Text_Io that compiles the SWG CAIS.
- A SWG CAIS implementation (possibly a partial implementation).

- Operating system support for command procedures, directory structures, and logical pathnames.

Some parts of both the Ada development environment and the Ada run-time environment will be present during SWG CAIS testing. SWG CAIS implementations are also likely to incorporate the use of shared files and will likely require special capacity limits and host privileges.

3.9 Performance Requirements

Two types of performance are applicable to the design of the SWG CAIS test environment: timing and capacity. The tests must be simple, including only the complexity required by the test objective. Therefore, both control and data structure complexity will be avoided whenever possible.

The primary performance requirement for the test suite is that a tester be able to run the entire test suite within a reasonable period of time, nominally one week. This is affected by both the speed of the tests and the amount of user interaction required. Performance is also important during test suite maintenance. The test suite is designed so that tests can be independently compiled, linked, and executed. The design also allows the incremental addition to or modification of the test suite. The minimal effort required to build a new suite is 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. SWG CAIS tests are designed to minimize execution time especially in the area of user interaction, but speed of execution is a secondary performance issue in the test environment as a whole.

Capacity refers to the ability of an Ada compilation system to generate correct code for a given number of Ada objects. For example, the test design takes into consideration that Ada compilers may have limitations on the number of enumeration literals supported before they exhaust symbol table space. For tests involving multiple instances of objects or loops to obtain average execution times, the number of objects or loops will be abstracted within the test so that these tests may be calibrated for the SWG CAIS implementation under test.

3.10 Test Support

Some test support packages were developed for the tests for the SWG CAIS implementation. The first category of test support includes timing packages. The second category of test support includes a Tree_Walkers package, which is used to determine the current state of the node model instance. Other test support packages were also developed for one or more sets of tests. These are described fully in the section(s) where the tests that utilized them are described.

3.10.1 Timing

SWG CAIS interfaces will have their timing characteristics measured. The elapsed time for some SWG CAIS interfaces will be measured. The Ada package Time_Recorder provides these timing services through interfaces that start the measured time, stop the measured time, and report the average of several timings.

3.10.2 Tree Walkers

The package Tree_Walkers is intended to be used as a resource by testers to determine the current state of the node model. The Tree_Walkers package contains the following utilities: Walk_Tree, Do_Something, Delete_Tree, Print_Tree, and Print_Node.

Walk_Tree is a generic Ada procedure that implements a general-purpose iterator for identifying all descendants of any node. The Do_Something procedure provides a means of encapsulating actions to be performed at each node of the tree. Delete_Tree and Print_Tree are other examples of useful instantiations of Walk_Tree. The procedure Print_Node is the action performed at each node of the tree when Print_Tree is invoked and is described fully in the next paragraph.

Procedure Print_Node prints out the contents of a node, including the primary name, node kind, all of the attributes, all of the primary relationships, and all of the secondary relationships. While some SWG CAIS implementations may have implementation-dependent services for printing the contents of a node model, this procedure is constructed specifically to be independent of the implementation. The intended use of this procedure, Print_Node, and the associated procedure Print_Tree is to "see" the state of the current node model. This is necessary when running tests that require a method for determining "what else" may be hidden in the node model.

4.0 TEST SUITE

This section defines the test suite required to support the SWG CAIS test and evaluation approach. The tests are organized into sets that correspond to the functional sections of DOD-STD-1838. While the design and style of the individual tests vary considerably across these sets, the tests within each functional set are quite similar. Section 4.1 describes the general design criteria that apply to all tests. Each of the sections 4.2 through 4.9 describes the design for a set of related tests, which includes information on test strategy, specific support packages, organization of the node model, as well as specific designs for both the nominal and usability classes of testing. For each test, the interface(s) that will be exercised will be identified along with the corresponding section number in the "Common Ada Programming Support Environment (APSE) Interface Set (CAIS)," DOD-STD-1838.

4.1 General

This section consists of general information regarding the design of both the nominal and usability categories of testings, as well as design standards with regard to test names, input/output, actual test design, and test success criteria.

4.1.1 Nominal Testing

In nominal testing, each "critical" SWG CAIS interface as defined in table 2 shall be individually examined at the simplest level, i.e. the results of a single call to an individual interface will be compared against expected results. Nominal testing can be used to determine the set of interfaces supported (i.e., existence testing) by the SWG CAIS implementation under test. An interface is a primary interface, an overload, or an additional interface. Each interface is unique even though some interfaces may be overloaded and therefore share the same name. Although the focus of a nominal test is the test of a particular interface, the actual test will often require the use of other SWG CAIS interfaces. For some tests the use of several interfaces will be necessary to establish an initial state of a SWG CAIS node model instance, to set a node(s) state, to traverse a part of the node model to a specific node prior to test execution, to ensure correctness of the interface under test, or to remove any modifications made to the state of the node model instance. Therefore, the ability to execute tests for most interfaces will also depend on the success of other interface tests. However, all actions required for test set up or for post-test clean up will be kept separate from the actual test code. 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 3. 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 that, 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, 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 remaining SWG CAIS interfaces depend.

The approach adopted for nominal testing, therefore, is to test mutually dependent interfaces in a single test, to identify a small set of dependent interfaces for each set of related tests, and as much as possible to order the tests so that dependent interfaces are tested prior to their use in other tests.

Each test shall consist of a simple exercise of an interface. The purpose of each 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 ability of the SWG CAIS implementation to raise each exception under the conditions specified in DOD-STD-1838. Input values will be defined for each parameter of mode "in" or "in out" as well as the actual state of the 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 state of the node model instance following test execution (if any).

It is beyond the scope of the current test and evaluation effort to perform exhaustive testing. Exhaustive testing would determine the thoroughness of the interface implementation by aggressively attempting to locate errors in the semantics of each interface through even more extensive test data sets and through more thorough exception testing. The current test suite has been limited to the set of interfaces identified as critical and has concentrated primarily on existence testing with only minimal coverage of exception handling.

4.1.2 Usability Testing

Usability testing builds on the tests included in the nominal testing category. This usability category of tests will make use of predefined usability scenarios that chain SWG CAIS interface calls together and the ability of the SWG CAIS implementation to meet the needs of tool developers. These test sequences would reflect normal actions, as well as actions that stress the limits of the implementation. Thus, the usability tests both mimic probable usage by tool writers and test the robustness of the implementation. Guidelines for performing a minimal evaluation of the SWG CAIS implementation capabilities were developed by Carney (1988). These guidelines were helpful in the design of the SWG CAIS implementation usability tests. As in nominal testing, only the set of critical interfaces in the critical packages, which has been defined previously in table 2, will be considered. It is outside the scope of the current testing effort to develop more than a minimal subset of possible scenarios.

The purpose of the usability tests is to exercise the SWG CAIS implementation based on typical individual tool usage. The design of the tests will support single-user or multi-user testing and include measurement of both capacity and timing measurements. This usability test software will transform the state of the node model instance but will not actually emulate the real tools and their functionality. The important factor is that nodes can be accessed. The actual contents of nodes is not important.

The primary objective of the usability tests is to determine the extent to which typical usage of SWG CAIS implementation interfaces by SWG CAIS tool users can be expected to meet performance expectations. Whereas a nominal test might exercise one interface one time, a usability test would exercise one or more related interfaces many times. Usability tests often contain a scenario of several interface calls that would typically be performed by one or more tools in the SWG APSE. For example, one nominal test might determine if a node can be created and another nominal test might determine if an already-created node can be deleted. A usability test would determine if a number of nodes could be created, determine the time required to create the node(s), and then delete them and determine the time required to delete the node(s). As well, a usability test might determine an implementation's capacity limits for nodes by creating nodes until no more could be created, or by alternately creating and deleting nodes and thereby testing for proper collection of unused space.

4.1.3 Test Name Standards

Configuration management of the test suite is needed to associate tests with the section of DOD-STD-1838 to which they apply. For each critical interface identified in Section 2.2, a test will be constructed. Although there are a large number of SWG CAIS tests, a simple naming convention, consistently applied, allows easy correlation of tests to corresponding sections of DOD-STD-1838.

In accord with a naming convention originally adapted from conventions required by the United Kingdom Test Harness, tests will be named using the format "T*nn.ada". To represent test numbers greater than 99, the "T*nnn.ada" format will be used. The "*" represents a group letter, and "nn" or "nnn" represents the number of the test, which can span from "01" through a maximum of "999". The groups and their corresponding packages are as follows:

- B - List Management
- C - Node Management
- D - Attribute Management
- G - Process Management (only nominal tests)
- I - Direct IO
- J - Sequential IO
- K - Text IO
- P - Page Terminal IO
- S - Structural Node Management
- V - Process Management (only usability tests)

Input files will be named "T*nn.IN". Output files will be named "T*nn.OUT". Trace files will be named "T*nn.TRACE". Note that "nnn" will be used when the corresponding test file is of the form "T*nnn.ada".

4.1.4 Input/Output Standards

Input/output standards were adopted for all three kinds of data files: input, output, and trace files. In each data file, information in the form of comments precedes the data lines. All data files associated with a test will be designed with a standard header. All input files will contain version numbers and the date of the last change. Multiple data sets within an input file will be separated by comments. A primary output file always will be produced and will follow a standardized format that will include the name and version of the input file that was used by the test. Tests will also be designed to produce trace files to print intermediate results for use during debugging of tests. The format of trace file will vary to accommodate the design of each particular test. Figure 2 includes a sample of an input data file. Figure 3 includes a sample of an output data file.

4.1.5 Test Design Standards

The test and evaluation of the SWG CAIS implementation 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, to ensure compliance to the semantic intent of the specification, and to ensure usability by tool developers. The first goal can be met in a straightforward manner since the syntax of the interfaces is formally defined through Ada packages. The Ada compiler performs static syntax checks (as well as some static semantic checks) of the SWG CAIS interface implementations. Ensuring conformance to the semantics of the specification is more difficult, however, since the semantics of the SWG CAIS are not formally specified. The last goal, that of usability, is subjective in nature; however, objective statistics from the results of the usability tests will form the basis for later determinations of the usability of SWG CAIS implementations with respect to specific tool requirements.

The tests consist of two categories: nominal tests that exercise single interfaces and usability tests that exercise sequences of interfaces similar to those required by tools. The second category uses tool usage scenarios that test the overall usability of the SWG CAIS. These usability tests will use the interfaces previously tested during the nominal testing to check for erroneous interactions and artificial boundaries. In the design of both the nominal and usability tests, all tests will adhere to the following design standards. Figure 4 includes an example for documenting a procedure.

4.1.5.1 Minimize Use of Implementation-Dependent Interfaces. Minimize use of implementation-supplied interfaces other than those specified in the SWG CAIS specification. The top-level interfaces are those that are intended for tool writer use. An implementation may make other, lower level interfaces visible, but these should not be used by tests as this would jeopardize the portability of the code to another system.

Figure 2
SWG CAIS Input Data File Sample

```
!=====================================================================
!
!
!           Input test data for testing COPY_LIST: TB01
!
!   These data are in 1-tuples.  The expected result is always
!   the same as the input.  No attempt is made to check for ill-formed
!   lists:  See TB17 for the CONVERT_TEXT_TO_LIST function which
!   checks for ill-formed lists and lists exceeding normal capacity.
!
!=====================================================================
(())
(a,())
((2),(3))
(("a","b","c"))
(A=>a)
(A=>(),B=>(C=>()))
(A=>2,B=>(3))
(A=>"a",B=>("b","c"))
((A=>2),B=>3)
("ab cd ef")
( "ab")
("ab" )
_quit

--file to test copy function
--modified to remove test cases which raise exceptions like syntax error
--this file needs more test cases.
```

Figure 9
SWG CAIS Output Data File Sample

```
CAIS Version 3.2                                                    10/24/88


    TG30: Tests the following interfaces in
          CAIS_PROCESS_MANAGEMENT

          CAIS 5.2.2.14 - Open_Node_Handle_Count

==========================================================================
Test Case 1 : Open 'Current_Job and get open node handles


    Interface name tested: Open_Node_Handle_Count
    Interface number tested: 5.2.2.14
    Interface overload tested: A


Test Inputs:
    No inputs

Expected Results:
    Initially 1, then 6, open node handles


Actual Results:
    Success - 1, then 6, node handles returned

Result of Test: PASSED

(3.2 10/24/88 TG30 1 5.2.2.14 A PASSED)

--------------------------------------------------------------------------
                                   .
                                   .
                                   .
--------------------------------------------------------------------------

Summary of Test Procedures:

     Number of Test Cases Passed:  4
     Number of Test Cases Failed:  0
     Total Number of Test Cases:   4

    Result of Test Procedure: PASSED

--------------------------------------------------------------------------
```

Figure 4
SWG CAIS Procedure Documentation Sample

```
-------------------------------------------------------------------------
--  Test Procedure:          T G 0 4
--  ---------------
--
--  Author:  Dave Hough      MITRE Washington Software Center
--  ------
--
--  Revision History:
--  -----------------
--    <Date>          <#nn>    <Name>     <Description>
--
--  Purpose:
--  --------
--  Provide functional testing of the SWG CAIS, a variant
--  of DOD-STD-1838, dated 9 October 1986.  This work has been
--  performed in support of the NATO SWG on APSE project, sponsored by
--  AJPO.
--
--  Interface(s) Tested:
--  --------------------
--    5.2.2.1 (B)           Spawn_Process           Cais_Process_Management
--
--  Implementation:
--  ---------------
--
--  Exceptions Raised, Handled, Propagated:
--  ---------------------------------------
--
--  I/O: None
--  Machine/Compiler Dependencies: None
-------------------------------------------------------------------------
```

4.1.5.2 Use Test Utilities. Implementation-specific code should be isolated into test utilities packages and made available across all tests.

4.1.5.3 Use Directed Input/Output. Use named input and output files in test input/output versus using defaults. This practice not only allows better control over input/output but also is more portable. The host specific names should be collected in a package and declared as constant strings.

4.1.5.4 Use a Single User Node. Utilize only one user node. That is, create all node model instance structures under the current user node only. Implementation-dependent restrictions governing the login process and other practical considerations lead to this decision. Furthermore, use the predefined constant "current_user" or Root_Process to stand for the user node, rather than using an existing user node such as "user(smith)". This allows other users to execute the test with minimal trouble.

4.1.5.5 Ensure the Initial State of the Node Model Instance. Ensure that a correct state of the SWG CAIS node model instance exists prior to execution of this test. This implies the creation of a SWG CAIS node model instance and generation of an initial test state, or the use of a predefined SWG CAIS node model instance with a predefined test state, or the use of a previously generated node model instance with a test state as created by a previously executed test. This will require the use of other SWG CAIS interfaces.

4.1.5.6 Use Predefined Input Values. Execute the interface with predefined input values, where practical. Tests that have an input data file can be expanded to include other test cases by editing a source file.

4.1.5.7 Indicate Pass/Fail Results. All tests will contain their own expected results, either in the input data file, or embedded in the test code. This data will be used to determine and explicitly report the success or failure of the test. No test will merely print results to be verified by the tester.

4.1.5.8 Handle All Exceptions. Each test will assume that unexpected exceptions may occur at any time. Tests will contain handlers to check for the presence for unexpected exceptions as well as expected exceptions.

4.1.5.9 Limit Test to Determining That an Error Exists. Do not attempt to determine the cause of an error, only that an error according to the SWG CAIS specification exists. Determining the cause of a SWG CAIS error is the responsibility of the SWG CAIS implementer. Tests will be designed to spot errors only. The tests will provide as much information as required to have confidence in the discrepancy between the SWG CAIS specification and the test results.

4.1.5.10 Protect against Unexpected Exceptions. Continue gathering data if an unexpected exception condition happens. This can be achieved by liberal use of exception blocks. Each test may be decomposed into several test cases, each relatively independent of the other.

4.1.5.11 Compare States of Node Model. Compare the resultant state of the SWG CAIS node model instance with the expected state of the SWG CAIS node model instance. This may involve the use of other SWG CAIS interfaces. The Tree_Walkers package, described in section 3.10.2, provides interfaces that can determine the state of the node model within the current user's bounds.

4.1.6 Test Success Criteria Standards

A 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 matches the actual output for an existence test. For an exception test, the actual exception raised matches the expected exception.
- The expected state of the SWG CAIS node model instance matches the actual state.

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

- Pass--the test was successful according to the stated success criteria.
- *Fail/reason--the test was not successful* according to the stated success criteria. The reason portion of the status will identify which criterion was not met (i.e., state of the 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 in black-box testing. 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.

4.2 Node Management Tests

These tests exercise the interfaces defined in the CAIS_NODE_MANAGEMENT package (5.1.2). CAIS_NODE_MANAGEMENT provides the general primitives for manipulating, copying, renaming, and deleting nodes and their relationships. The operations tested here are generally applicable to all kinds of nodes and relationships. These tests also vary the intents and pathnames associated with the nodes upon which are being operated. Although these tests require that nodes be created, operations for creation of nodes reside separately in packages specific to the node kind (structural, process, and file) and are explicitly covered within those test sets.

The node management tests begin with the prefix "TC" and are divided into nominal tests and usability tests. The nominal tests address specific operations and exercise all overloaded forms of an operation. Usability tests are intended to test efficiently many

interfaces acting together in a realistic way. Some of them also time the interfaces or measure their capacity. In addition, the nominal, rather than the usability, tests may include exception testing.

4.2.1 Test Strategy

The node management tests are designed to perform black-box testing. However, each test necessarily relies on node management interfaces to build the set of nodes upon which the subtests are performed. The CAIS_NODE_MANAGEMENT interfaces used by other node management tests are Create, Open, Close, and Is_Open. All three types of nodes (structural, process, and file) are created, and they are opened with intents appropriate for the tests being performed. Proper implementation of these four node management interfaces is essential for proper testing of CAIS_NODE_MANAGEMENT using this test set. Therefore, the tests exercising these four interfaces should be run prior to running the other node management tests.

The node management tests follow a design that allows them to be driven by input files. Because the number of subtests is generally small, however, the subtests are normally coded directly in the test source rather than read from input files. Each test exercises a specific operation and generally includes tests for all alternate interfaces associated with that operation. Each test calls setup routines that create a set of nodes needed to test the operation and then deletes this set of nodes upon completion of the tests. The tests are data driven even to the extent that control flow (i.e., the procedures to be executed) is governed by the input data. Data read from the input file not only determines calls to the appropriate procedures that then read additional data and invoke the specified CAIS interfaces, but also controls which setup and which trace procedures to call.

The nominal node management tests create both a log and a trace output file. For each subtest, a pass/fail message is printed based upon a comparison of actual and expected results. At the end of the test, the number of failed subtests and the number of passed subtests are printed. Whereas the trace file contains a complete list of inputs, expected results, and actual results, as well as trace commentary, the log file contains only a summary of the test results.

Exhaustive testing is outside the scope of the current effort. Therefore, not all exceptions will be tested, although the tests will be designed to accommodate future additions of exception testing.

4.2.2 Specific Support Packages

The node management tests make use of three formatting support packages:

- REPORT--Provides procedures for reporting test names, indicating pass/fail status for sub-tests, and reporting an overall test result (that indicates passage only if all sub-tests pass).

- PRINT_MODEL--Provides procedures to format and print descriptions of nodes, relationships, and attributes.

- PRINT_OUTPUT–Provides standardized data structures and print procedures for test headers, subtest results, and summary results.

All node management tests use a set of common procedures that standardize the reading of input data and the formatting of output data. These procedures perform a particular I/O function and also place messages in the trace file that describe the actions taking place. Procedures to read node numbers, strings, intent arrays, intent specifications, etc., are available to each test. For output, procedures are provided for initializing a header as well as the formatting routines described previously.

4.2.3 Organization of Node Model

Each node management test requires a set of related nodes so that the specific operation can be tested via a variety of pathnames and via a variety of indicators made up of a base node, a relationship, and a key. At the start of each test an appropriate set of nodes is created, and these nodes are then deleted at the close of the test. The set of nodes created will typically contain multiple levels of primary relationships, a few secondary relationships, and nodes of all three types (structural, process, and file). For simplicity, the set of nodes is hard coded into the test, rather than being read from the test input file.

4.2.4 Nominal Tests

Nominal tests were created for 17 of the more important interfaces. The interfaces tested are 5.1.2.1 through 5.1.2.11, 5.1.2.19, 5.1.2.23, 5.1.2.25, 5.1.2.26, 5.1.2.37, and 5.1.2.38.

4.2.4.1 TC01. Exercise the Open interfaces (5.1.2.1). Perform tests using all four interfaces for Open specifying nodes by pathname and by base-key-relation triplets and specifying intentions both singly and via arrays. Use all three types (structural, process, and file) of nodes in the TC01 subtests. Subtests are read from file TC01.IN.

4.2.4.2 TC02. Exercise the Close interface (5.1.2.2). The Close operation only has a single interface that operates on a node handle. Exception tests need not be performed. Subtests are read from file TC02.IN.

4.2.4.3 TC03. Exercise the Kind_of_Node interfaces (5.1.2.6). Check all three kinds of nodes (process, structural, and file). Include subtests to check for proper propagation of Status_Error exceptions.

4.2.4.4 TC04. Exercise the Is_Open interfaces (5.1.2.4). Perform tests against both opened and closed nodes. There are no exceptions associated with the Is_Open interface.

4.2.4.5 TC05. Exercise the Open_Parent interfaces (5.1.2.19). Perform tests against Open_Parent interfaces with both single intentions and arrays of intentions. Exception tests need not be performed.

4.2.4.6 TC06. Exercise the Delete_Node interfaces (5.1.2.23). Check the Delete_Node interfaces using both a node handle and a pathname to designate the node being deleted. Exception tests need not be performed.

4.2.4.7 TC07. Exercise the Create_Secondary_Relationship interfaces (5.1.2.25). Check the Create_Secondary_Relationship operation using both a node handle and a pathname to designate the object of the relationship. The inheritance parameter may be defaulted and never tested. Exception tests need not be performed.

4.2.4.8 TC08. Exercise the Delete_Secondary_Relationship interfaces (5.1.2.26). Check the Delete_Secondary_Relationship operation using both an indicator (base, key, and relation) and a pathname to designate the relationship being deleted. Deletion using the default relationship need not be tested. Exception tests need not be performed.

4.2.4.9 TC09. Exercise the Set_Current_Node interfaces (5.1.2.37). The Set_Current_Node operation is checked using both a node handle and a pathname to designate the node being specified as current. Exception tests need not be performed.

4.2.4.10 TC10. Exercise the Get_Current_Node interfaces (5.1.2.38). Check the Get_Current_Node interface using both single intents and arrays of intents. Exception tests need not be performed.

4.2.4.11 TC11. Exercise the Change_Intent interfaces (5.1.2.3). Check the Change_Intent interface using both single intents and arrays of intents. Exception tests need not be performed.

4.2.4.12 TC12. Exercise the Intent interface (5.1.2.5). Perform tests over a wide range of intents. Test for proper propagation of the Status_Error exception.

4.2.4.13 TC13. Exercise the Open_File_Handle_Count interfaces (5.1.2.7). Exception tests need not be performed.

4.2.4.14 TC14. Exercise the Primary_Name interfaces (5.1.2.8). Perform tests on all three kinds of nodes (structural, process, and file). Also perform tests for proper propagation of Status_Error and Intent_Violation exceptions. Because of the small number of possible subtests, the subtests may be hard coded rather than driven by an input file.

4.2.4.15 TC15. Exercise the Primary_Key interfaces (5.1.2.9). Perform tests on all three kinds of nodes (structural, process, and file). Also perform tests for proper propagation of Status_Error and Intent_Violation exceptions.

4.2.4.16 TC16. Exercise the Primary_Relation interfaces (5.1.2.10). Perform tests on all three kinds of nodes (structural, process, and file). Tests are also performed for proper propagation of Status_Error and Intent_Error exceptions.

4.2.4.17 TC17. Exercise the Path_Key interfaces (5.1.2.11). Also perform tests for proper propagation of Status_Error exceptions.

4.2.5 Usability Tests

For the node management package, there are four usability test, TC101 through TC104. The usability test TC101 measures the elapsed time it takes to create a node, open it, test if it is open, close it, and delete it. The usability tests TC102 through TC104 test the 41 node management interfaces. TC102 will test interfaces 5.1.2.1 through 5.1.2.17. TC103 will test interfaces 5.1.2.18 through 5.1.2.28, and 5.1.2.37 through 5.1.2.42. TC104 will test interfaces 5.1.2.30 through 5.1.2.36.

4.2.5.1 TC101. Determine the elapsed time required by each of the five basic interfaces whose performance is critical to the SWG CAIS implementation. They are Create_Node, Open, Close, Is_Open, and Delete_Node.

Loop through each of these operations several times so that an average execution time for each individual interface can be computed. Use an input file so that the top-level node name, the intents, and the number of passes through the timing loop may be varied. Read in the data and create nodes one at a time. In a second part of the test, create an array of nodes, leaving them open. For all tests, report interim timings every 20 passes through the loop as well as at the end of the test.

4.2.5.2 TC102. Test the 17 interfaces (5.1.2.1 through 5.1.2.17). Perform tests on all three kinds of nodes and using all alternate interfaces.

4.2.5.3 TC103. Test the 17 interfaces (5.1.2.18 through 5.1.2.28 and 5.1.2.37 through 5.1.2.42). Perform tests on all three kinds of nodes and using all alternate interfaces. Report pass/fail status when possible, and when a pass/fail decision cannot be made (as is the case for operations that return time stamps) explicitly report the information obtained from the test.

4.2.5.4 TC104. Test the iterator interfaces (5.1.2.30 through 5.1.2.36). Build iterators over both primary and secondary relationships. Use nodes of all kinds (structural, file, and process). Also use a variety of selection patterns. After building each iterator, traverse it, printing out the names of all nodes in the iterator. These names should be accompanied by enough information (e.g., node name, kind of relationships selected, and pattern) so that the results can be verified.

4.3 Attribute Management Tests

The CAIS_ATTRIBUTE_MANAGEMENT package provides a set of interfaces that support manipulation of attributes on either nodes or relationships. The package contains a total of 17 unique interfaces--of which only eight are tested at the nominal level. The interfaces that are not tested at the nominal level are those used to create and manipulate attribute iterators. These interfaces are exercised in a more elaborate test, the Tree_Walker, which is described in section 3.10.2.

4.3.1 Test Strategy

The nominal tests for attribute management examine a subset of the interfaces at both the existence and exception level. All exceptions corresponding to the interfaces are tested with the exception of the Status_Error and Security_Violation. Testing for the Status_Error exception is considered to be a secondary concern because it will be raised only if the node to be used for attribute manipulation is not opened. The Security_Violation exception is raised only in the event of mandatory access violations (which are not implemented for the SWG CAIS), so this exception is also ignored in the test designs.

The usability tests for the attribute management package perform scenario type tests using one or more of the interfaces in a single test. The tests have several primary objectives: to determine if the limits defined by the CAIS_PRAGMATICS are indeed achievable; if they are not, to determine what the actual implementation limits are and if these limits are different depending on initial conditions; and finally, to determine the robustness of the implementation when a limit is reached. As a secondary objective, timing statistics are gathered for each CAIS interface used in the execution of the test. The total test time, number of times each interface is executed, and the average time per interface are reported. These times are currently wall clock times and can be used to gauge relative execution times for the interfaces.

4.3.2 Specific Support Packages

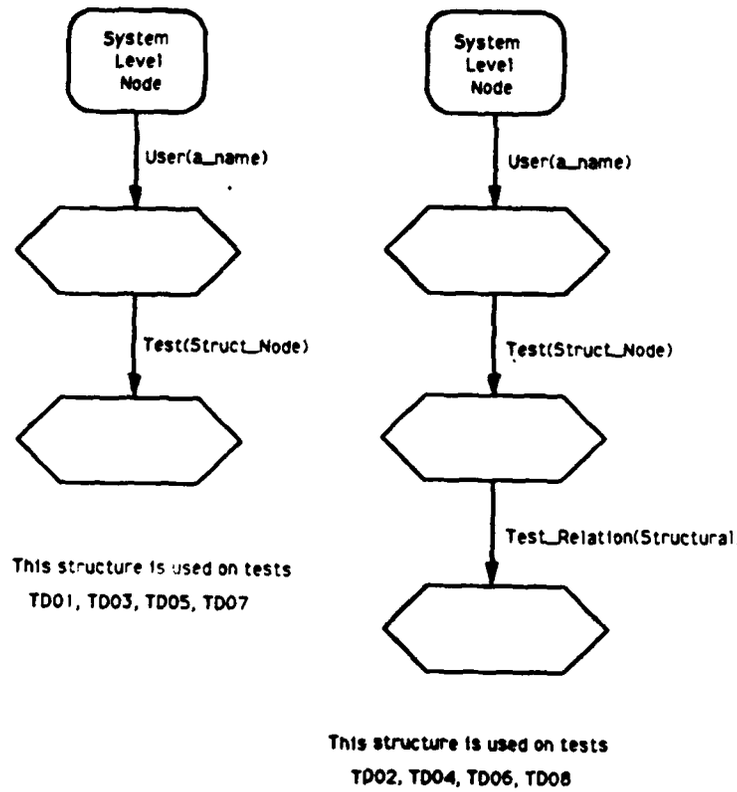
Three support packages developed to supplement the performance of the individual tests are listed as follows:

- CREATE_AN_INSTANCE--Creates an instance of the CAIS node model based on user-supplied test information.
- DATA_MGMT_PKG--Provides the common procedures and data structures to support the attribute management nominal tests (TDxx).
- STD_TEST_RESULTS_PKG--Defines a set of messages to be recorded for each exception in the CAIS for both unexpected as well as expected instances.

4.3.3 Organization of the Node Model

Each attribute management test uses a simplistic node model for attribute manipulation. In the case of node attribute tests, the interfaces are exercised against a structural node that emanates from the current_user node along a path 'test(struct_node). For path attribute tests, another structural node is created from the test(struct_node) node along the path path. Figure 5 provides an illustration of the two node models and a mapping of tests to the appropriate node model.

Figure 5
SWG CAIS Attribute Management Test Node Models



4.3.4 Nominal Tests

4.3.4.1 TD01. Exercise the Create_Node_Attribute interface (5.1.3.1). First, attempt to create a node attribute that has been previously created in an earlier test case--this is to verify that an "Attribute_Error" exception will be appropriately raised. Then, test the creation of a node for the first time. To exercise the Create_Node_Attribute, read and convert an input character string to a list format, and use that input data as parameters of the "Create_Node_Attribute" interface. The node is then either created for the first time, or denied creation, depending on the input data.

4.3.4.2 TD02. Exercise the Create_Path_Attribute interface (5.1.3.2) by verifying that valid path attributes can be created, and verifying that the exception handling accompanying the Create_Path_Attribute performs as expected.

Read attribute information entries from an input file into the Create_Path_Attribute interface to create a path attribute. After the attribute information is accepted, perform a Get_Path_Attribute to verify that the attribute exists. Use the same input data to inject invalid attribute fields into the Create_Path_Attribute interface to verify that the correct exceptions will be raised.

4.3.4.3 TD03. Exercise the Delete_Node_Attribute interface (5.1.3.3) by attempting to create and then delete an attribute named Delete_an_Attribute. Perform this test by creating and deleting an attribute using the Create_Node_Attribute and Delete_Node_Attribute interfaces. Then attempt a Get_Node_Attribute of the deleted attribute. The test is successful in deleting the attribute if an Attribute_Error is raised by the Get_Node_Attribute operation.

4.3.4.4 TD04. Exercise the Delete_Path_Attribute interface (5.1.3.4) by verifying that all created path attributes can be deleted using the Delete_Path_Attribute interface. Achieve this by first establishing a structural node and a child node so that a user-defined path may exist on which to create and delete attributes. Then test this interface by first creating the path attribute using the Create_Path_Attribute interface. Immediately delete the created path attribute using the Delete_Path_Attribute. The deletion is verified when the test attempts to perform a Get_Path_Attribute of the path and an Attribute_Error exception is raised.

4.3.4.5 TD05. Exercise the Set_Node_Attribute interface (5.1.3.5) to verify that the Set_Node_Attribute interface performs properly under normal conditions, and that the exceptions Syntax_Error, Predefined_Attribute_Error, Attribute_Error, and Intent_Violation are raised appropriately. Achieve this by first creating a node attribute and then attempting to set the value for the attribute under various input conditions.

4.3.4.6 TD06. Exercise the Set_Path_Attribute interface (5.1.3.6) to verify that the Set_Path_Attribute interface performs properly under normal conditions, and that the following exceptions corresponding to the interface are appropriately raised: Pathname_Syntax_Error, Relationship_Error, Syntax_Error, Predefined_Relation_Error,

Predefined_Attribute_Error, and Attribute_Error. Achieve this by first creating a path attribute and then attempting a Set_Path_Attribute to set the value for the path attribute under various input conditions.

4.3.4.7 TD07. Exercise the Get_Node_Attribute interface (5.1.3.7) to verify that the Get_Node_Attribute performs properly under normal conditions, and that the following exceptions corresponding to the interfaces are raised when appropriate: Syntax_Error, Attribute_Error, and Intent_Violation. Achieve this by creating several node attributes using the Create_Node_Attribute interface, then attempting to retrieve the various node attributes using the Get_Node_Attribute.

4.3.4.8 TD08. Exercise the Get_Path_Attribute interface (5.1.3.8) to verify that the Get_Path_Attribute performs properly under normal conditions, and that the following exceptions corresponding to the interfaces are raised when appropriate: Pathname_Syntax_Error, Syntax_Error, Attribute_Error, and Intent_Violation. Achieve this by creating several path attributes using the Create_Path_Attribute interface, then attempting to retrieve the various path attributes using the Get_Path_Attribute.

4.3.5 Usability Tests

4.3.5.1 TD101. Examine the limits imposed on the Create_Node_Attribute (5.1.3.1) interface by determining the maximum number of node attributes that may be created at one time. Achieve this by creating a node and then creating as many node attributes as possible. Timing tests accompanying test case TD101 determine the amount of time to execute the Create_Node_Attribute interface.

4.3.5.2 TD102. Examine the limits imposed on the Create_Path_Attribute interface (5.1.3.2) by determining the maximum number of attributes that can be created on a path. Achieve this by creating a path and then creating as many path attributes as possible. Timing tests accompanying test case TD102 determine the amount of time to execute the Create_Node" and Create_Path_Attribute interfaces.

4.3.5.3 TD103. Examine the limits imposed on the Set_Node_Attribute (5.1.3.5) and Get_Node_Attribute (5.1.3.7) interfaces by creating a node attribute and sequencing through a series of Set_Node_Attribute interfaces to set the value of the attribute and Get_Node_Attribute interfaces which retrieves that attribute value. In addition to the interface performance tests, test case TD103 provides timing tests to determine the amount of time required to execute the Set_Node_Attribute and Get_Node_Attribute interfaces.

4.3.5.4 TD104. Examine the limits imposed on the Create_Path_Attribute (5.1.3.2), the Create_Node_Attribute (5.1.3.1), and the Create_Node" (5.1.5.1) interfaces to determine the total number of attributes that can be created on several nodes at one time. Achieve this by creating a node model instance of 20 structural nodes. Then walk the tree adding attributes to each node and to each path of each node until no new attributes can be added. In addition to the interface performance tests, test case TD104 provides timing tests to determine the time required to execute the Create_Node, Create_Node_Attribute, Create_Path_Attribute, and Open interfaces.

4.4 Structural Node Management Tests

The CAIS_STRUCTURAL_NODE_MANAGEMENT package provides a set of interfaces that support the creation, deletion, and manipulation of structural nodes.

4.4.1 Test Strategy

The nominal tests for structural node management consist of one test package with four test procedures. Each one tests one format of the Create_Node interface.

The usability tests for structural node management determine if varying node model structures result in different limits being achieved by an implementation. For this type of test, three node model instances are used: a "breadth-first" structure, a "depth-first" structure, and a "binary-tree" structure all built from the "current_user" node. Some tests are executed against all three structures. Figure 6 illustrates the three structures.

4.4.2 Specific Support Packages

Three support packages developed to support the performance of the individual tests are listed as follows:

- CREATE_AN_INSTANCE--Creates an instance of the CAIS node model based on user-supplied test information.
- DATA_MGMT_PKG--Provides the common procedures and data structures to support the Structural Node Management tests.
- STD_TEST_RESULTS_PKG--Defines a set of messages to be recorded for each exception in the CAIS for both unexpected as well as expected instances.

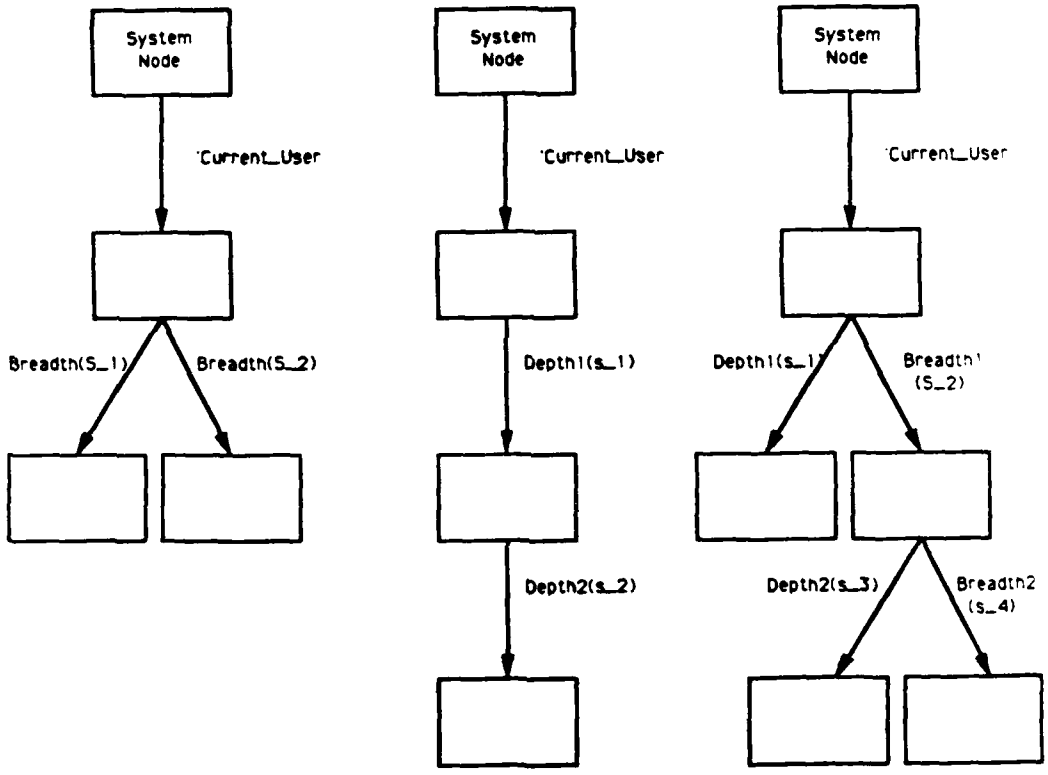
4.4.3 Organization of the Node Model

Each structural node management test uses a node model like the one described for the attribute management tests (Section 4.3). All interfaces are exercised against a structural node that emanates from the current_user node along a path 'test(struct_node). Here nodes and relationships are created, deleted , or copied depending on the test objective.

4.4.4 Nominal Tests

The only nominal test is "TS01". Exercise the four format types of the Create_Node (5.1.5.1) interface by attempting to create one structural node using each of the formats provided for Create_Node. When a structural node is created from each of these formats, a result is recorded to an output file.

Figure 6
SWG CAIS Node Model Structures



4.4.5 Usability Tests

4.4.5.1 TS101. Examine the limits imposed on the Create_Node (5.1.5.1) interface by attempting to create as many nodes as possible in a breadth-first manner up to the pragmatic limit, emanating from the current_user node. In addition to the functional evaluation of the tests, test case TS101 provides timing tests to determine the amount of time required to execute the Open and Create_Node interfaces.

4.4.5.2 TS102. Examine the limits imposed on the Create_Node (5.1.5.1) interface by attempting to create as many nodes a possible in a depth-first manner, beginning at the current_user node. Achieve this by creating a node and setting the current node to the newly created node. Repeat until no more nodes can be created. In addition to the functional tests, TS102 provides timing tests to determine the amount of time required to execute Create_Node interface.

4.4.5.3 TS103. Exercise the limits imposed on the Create_Node (5.1.5.1) interface by attempting to create as many nodes as possible using a binary tree structure up to the pragmatic limit beginning at the current_user node. In addition to the functional tests, TS103 provides timing tests to determine the amount of time required to execute Open and Create_Node interfaces.

4.4.5.4 TS104. Exercise the Create_Node (5.1.5.1) interface by examining the actual limits of creating structural nodes in a breadth-first manner from the current_user node. Achieve this by creating and closing a node, for as many nodes as possible up to the pragmatic limit. In addition to these functional tests, TS104 provides timing tests to determine the amount of time required to execute Open, Create_Node, and Close interfaces.

4.4.5.5 TS105. Exercise the Create_Node (5.1.5.1) interface by examining the actual limits of creating structural nodes in a depth-first manner from the current_user node. Achieve this by creating, closing, then reopening a node, for as many nodes as possible up to the pragmatic limit. In addition to these functional tests, TS105 provides timing tests to determine the amount of time required to execute the Open, Create_Node, and Close interfaces.

4.4.5.6 TS106. Examine the limits imposed on the Create_Node (5.1.5.1) and the Close (5.1.2.2) interfaces by creating, opening, and closing as many nodes as possible. In addition to the functional tests, TS106 provides timing tests to determine the amount of time required to execute the Open and Create_Node interfaces.

4.4.5.7 TS107. Exercise the Create_Node (5.1.5.1) and the Delete_Node (5.1.2.23) interfaces by creating and deleting nodes, then verifying that the node's primary relationship has been deleted by using the CAIS_NODE_MANAGEMENT Boolean function Is_Obtainable. Do this for as many nodes as possible in a breadth-first manner. In addition to the functional tests, TS107 provides timing tests to determine the amount of time required to execute the Open, Create_Node, and Delete_Node interfaces.

4.4.5.8 TS108. Exercise the Create_Node (5.1.5.1), the Create_Secondary_Relationship (5.1.2.25), and the Delete_Node (5.1.2.23) interfaces in order to determine whether deletion of one or more secondary relationships will permit more nodes to be created. Achieve this by creating a node in a breadth-first manner from the 'current_user, then adding a secondary relationship between that node, and then deleting the node. Repeat this sequence until no more nodes can be created. In addition to the functional tests, TS108 provides timing tests to determine the amount of time required to execute the Open, Create_Node, Delete_Node, and Create_Secondary_Relationship interfaces.

4.4.5.9 TS109. Examine the performances of the Create_Node (5.1.5.1), Open (5.1.2.1), and Close (5.1.2.2) interfaces to determine the average time to open and close a node handle. Achieve this by creating a node model of 20 nodes and closing the nodes as they are created. Then walk through the established node structure, opening and closing each node handle. In addition to the functional tests, TS109 provides timing tests to determine the amount of time required to execute the Open, Create_Node, and Close interfaces.

4.4.5.10 TS110. Exercise the limits imposed on the Create_Node (5.1.5.1) and the Copy_Node (5.1.2.20) interfaces by creating a node and then copying it until no new nodes may be made. In addition to the functional tests, TS110 provides timing tests to determine the amount of time required to execute the Open, Create_Node, and Copy_Node interfaces.

4.4.5.11 TS111. Exercise the Create_Node (5.1.5.1) and the Create_Secondary_Relationship (5.1.2.25) interfaces by creating a node from the current_user node and then creating secondary relationships on that path until no more can be created. This verifies the number of secondary relationships that may be created. In addition to the functional tests, TS111 provides timing tests to determine the amount of time required to execute the Open, Create_Node, and Create_Secondary_Relationship interfaces.

4.4.5.12 TS112. Examine the limits imposed on the Create_Node (5.1.5.1) and the Create_Secondary Relationship (5.1.2.25) interfaces by creating 20 nodes related as parent and child, then creating secondary relationships throughout the tree until either a predefined limit is set, or no more relationships can be added. In addition to the functional tests, TS112 provides timing tests to determine the amount of time required to execute the Open, Create_Node, and Create_Secondary_Relationship interfaces.

4.4.5.13 TS113. Examine the limits imposed on the Create_Node (5.1.5.1), the Create_Secondary_Relationship (5,1,2,25), and the Delete_Secondary_Relationship (5.1.2.26) interfaces by creating two nodes related as parent and child, then creating and deleting secondary relationships on that path up to a predefined limit. In addition to the functional tests, TS113 provides timing tests to determine the amount of time required to execute the Open, Create_Node, Create_Secondary_Relationship, and Delete_Secondary_Relationship interfaces.

4.4.5.14 TS114. Exercise the Set_Current_Node (5.1.2.37) and Get_Current_Node (5.1.2.38) interfaces by creating a small number of unrelated nodes, then calling the Set_Current_Node and Get_Current_Node interfaces for each. In addition to the functional

tests, TS114 provides timing tests to determine the amount of time required to execute the Open, Close, Set_Current_Node, Get_Current_Node, and Create_Node interfaces.

4.5 Process Management Tests

Process management standardizes the initiation, running, and termination of host processes. The nominal tests are found in the TG series of tests, and the usability tests are found in the TV series.

4.5.1 Test Strategy

Most of the TG tests use the login process current_user node as the process node which is acted upon by the various process management interfaces.

All processes, when initiated through Spawn_Process, Invoke_Process, or Create_Job, must designate a target file node, which contains the executable image that process is to manage. Some of the earlier TV tests spawn processes that designate a particular file node. TVLOOP periodically sends messages to the screen announcing its presence and stops only when aborted with Abort_Process. To reduce the effort in configuring file nodes, later TV tests contain within themselves the TVLOOP function, and the test designates itself as the target. These tests assume that the Get_Parameters interface and some List Management interfaces work flawlessly.

The execution of a process management test is accomplished by first creating a file node under the current user node, importing its executable contents from the host file system's corresponding link file, creating target file node if necessary, importing its contents in like manner, and setting the Default_Tool relationship of the user node to designate the file node that contains the test. The SWG CAIS login procedure and Default_Tool relation are not specified in any standard, but the result of logging in is that the executable contained in the file node at the end of the Default_Tool relationship is executed

4.5.2 Specific Support Packages

Four packages and two additional functions support process management testing. Package TG_Names translates logical file names found in the tests into host specific file names. Package Report Package Print_Output formats output into standard formats. Package Time_Recorder provides timing utilities. Function CAIS_Version determines the version of the CAIS being tested. Function Date returns a string that contains the date. This date is contained in the formal output files.

4.5.3 Organization of Node Model

Each process management test is contained in a file node directly under the current user node with relationship formed from relation "test" and a key that is the same as the name of the test or test target. For example, the test TV01 is contained in the file node TVLOOP is normally contained in 'current_user'test(tvloop). All processes spawned by the tests have

names that, in part, reflect the name of the test, making it possible to simultaneously initiate more than one test without pathname conflicts.

4.5.4 Nominal Tests

Each of the TG tests exercises a single overload of each interface in the package CAIS_Process_Management.

4.5.4.1 TG04. Exercise the Spawn_Process interface (5.2.2.1). Determine if the simplest form of process spawning works. Spawn a null process. The spawned process source code is in a separate file.

4.5.4.2 TG05. Test the Await_Process_Completion interface (5.2.2.2a). That interface is tested both with and without a time limit. The raising of several exceptions is also attempted, including Node_Kind_Error, Status_Error, and Intent_Violation. A process is spawned and Await_Process_Completion is then called.

4.5.4.3 TG06. Test the overload of the Await_Process_Completion interface (5.2.2.2b). That interface is tested both with and without a time limit. The raising of several exceptions is also attempted, including Node_Kind_Error, Status_Error, and Intent_Violation. A process is spawned, and Await_Process_Completion is then called.

4.5.4.4 TG07. Exercise the Invoke_Process interface (5.2.2.3a). The target process is to create and write to a file. Exceptions are not tested.

4.5.4.5 TG08. Exercise the overload of the Invoke_Process interface (5.2.2.3b). The target process is to create and write to a file. Exceptions are not tested.

4.5.4.6 TG09. Exercise the Create_Job interface (5.2.2.4). The target process is to create and write to a file. Exceptions are not tested.

4.5.4.7 TG11. Exercise the Delete_Job interface (5.2.2.5a). Create a legitmate job and immediately delete that job and check the status of the deleted job with Current_Status. Attempt to raise Name_Error, Predefined_Relation_Error, Status_Error, Lock_Error, and Intent_Violation.

4.5.4.8 TG12. Exercise the first overload of the Delete_Job interface (5.2.2.5b). Create a legitmate job and immediately delete that job and check the status of the deleted job with Current_Status. Attempt to raise Name_Error, Predefined_Relation_Error, Status_Error, Lock_Error, and Intent_Violation.

4.5.4.9 TG13. Exercise the Append_Results interface (5.2.2.6). The interface Get_Results is used to verify that the results list has been appended. The only specific exception defined for this interface, Lock_Error, is general enough such that it is not tested.

4.5.4.10 TG14. Exercise the Write_Results interface (5.2.2.7) on the current job. Attempt to raise a Lock_Error.

4.5.4.11 TG15. Exercise the Get_Results interface (5.2.2.8a) on the current job. Results of the current job should be an empty list. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.12 TG16. Exercise the first overload of the Get_Results interface (5.2.2.8b) on the current job. Results of the current job should be an empty list. Attempt to raise a Node_Kind_Error, a Status_Kind error, and an Intent_Violation.

4.5.4.13 TG17. Exercise the second overload of the Get_Results interface (5.2.2.8c) on the current job. Results of the current job should be an empty list. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.14 TG18. Exercise the third overload of the Get_Results interface (5.2.2.8d) on the current job. Results of the current job should be an empty list. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.15 TG19. Exercise the Current_Status interface (5.2.2.9a) on the current job. The current job should be Ready. Attempt to raise a Node_Kind_Error, a Status_Kind error, and an Intent_Violation.

4.5.4.16 TG20. Exercise the first overload of the Current_Status interface (5.2.2.9b) on the current job. The current job should be Ready. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.17 TG21. Exercise the Get_Parameters interface (5.2.2.10) on the current job. The parameter list of the current job should be empty. Attempt to raise a Lock_Error.

4.5.4.18 TG22. Exercise the Abort_Process interface (5.2.2.11a) on a spawned process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.19 TG23. Exercise the first overload of the Abort_Process interface (5.2.2.11b) on a spawned process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.20 TG24. Exercise the second overload of the Abort_Process interface (5.2.2.11c) on a spawned process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.21 TG25. Exercise the third overload of the Abort_Process interface (5.2.2.11d) on a spawned process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.22 TG26. Exercise the Suspend_Process interface (5.2.2.12a) on a spawned process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.23 TG27. Exercise the first overload of the Suspend_Process interface (5.2.2.12b) on a spawned process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.24 TG28. Exercise the Resume_Process interface (5.2.2.13a) on a spawned then suspended process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.25 TG29. Exercise the first overload of the Resume_Process interface (5.2.2.13b) on a spawned, then suspended process. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.26 TG30. Exercise the Open_Node_Handle_Count interface (5.2.2.14a) on the current job. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.27 TG31. Exercise the first overload of the Open_Node_Handle_Count interface (5.2.2.14b) on the current job. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.28 TG32. Exercise the Io_Unit_Count interface (5.2.2.15a) on the current job. Determine if the interface correctly counts the number of lines of input and output generated by the Text_IO, Get_Line, and Put_Line interfaces. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.29 TG33. Exercise the first overload of the Io_Unit_Count interface (5.2.2.15b) on the current job. Determine if the interface correctly counts the number of lines of input and output generated by the Text_IO, Get_Line, and Put_Line interfaces. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.30 TG34. Exercise the Time_Started interface (5.2.2.16a) on the current job. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.31 TG35. Exercise the first overload of the Time_Started interface (5.2.2.16b) on the current job. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.32 TG36. Exercise the Time_Finished interface (5.2.2.17a).

4.5.4.33 TG37. Exercise the first overload of the Time_Finished interface (5.2.2.17b).

4.5.4.34 TG38. Exercise the Machine_Time interface (5.2.2.18a) on the current job. This interface should return 0.0 to indicate that the current job has not finished. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.35 TG39. Exercise the first overload of the Machine_Time interface (5.2.2.18b) on the current job. This interface should return 0.0 to indicate that the current job has not finished. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.36 TG40. Exercise the Process_Size interface (5.2.2.19a) on the current job. This interface should return a nonzero value. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.4.37 TG41. Exercise the first overload of the Process_Size interface (5.2.2.19b) on the current job. This interface should return a nonzero value. Attempt to raise a Node_Kind_Error, a Status_Kind_Error, and an Intent_Violation.

4.5.5 Usability Tests

The following usability tests will be attempted. The list is extensive, but the critical nature of the process management portion of the SWG CAIS merits full attention. No attempt will be made at this time to instrument the tests for timing, although each test will be connected to the test harness and have the standard output format. Each test is presented as a goal with a possible implementation.

4.5.5.1 TV01. Exercise the Spawn_Process interface (5.2.2.1). Find the limit of sequentially spawned processes. Sequentially spawn new processes until no new processes can be spawned. Determine which exception is raised after the last process is spawned and then check the status of the previously spawned processes.

4.5.5.2 TV02. Exercise the Spawn_Process interface (5.2.2.1). Find the limit of recursively spawned processes. Recursively spawn new processes until no new processes can be spawned. Determine which exception is raised after the last process is spawned and then check the status of the previously spawned processes.

4.5.5.3 TV03. Exercise the Spawn_Process interface (5.2.2.1). Determine if predefined attributes, inherited relationships, and the current status of spawned processes are correct. Spawn 5 to 10 processes and check the above.

4.5.5.4 TV04. Exercise the Invoke_Process interface (5.2.2.3). Determine if one process can invoke another process for some small number (5 to 10) processes. Recursively invoke processes 5 to 10 times and determine if all processes except the last process are blocked and if the last process is executing.

4.5.5.5 TV08. Exercise the Create_Job interface (5.2.2.4). Find the limit of sequentially created jobs. Sequentially create new jobs until no new jobs can be created. Determine which exception is raised after the last job is created, and then check the status of the previously created jobs.

4.5.5.6 TV09. Exercise the Create_Job interface (5.2.2.4). Find the limit of recursively created jobs. Recursively create new jobs until no new jobs can be created. Determine which exception is raised after the last job is created, and then check the status of the previously created jobs.

4.5.5.7 TV11. Exercise the Abort_Process interface (5.2.2.11). Make sure that a large number of spawned processes can be aborted. Spawn 50 to 100 processes, and abort each. Determine if the spawned processes are aborted.

4.5.5.8 TV12. Exercise the Abort_Process interface (5.2.2.11). Make sure that a large number of created jobs can be aborted. Create 50 to 100 jobs, and abort each. Determine if the jobs are aborted.

4.5.5.9 TV13. Exercise the Abort_Process interface (5.2.2.11). Determine if aborting a spawned process aborts all descendant processes. Recursively spawn some small number (10 to 20) of processes. Abort the most ancient process, and determine if all descendant processes are aborted, as they should be.

4.5.5.10 TV14. Exercise the Abort_Process interface (5.2.2.11). Determine if aborting a job does not abort any other job. Recursively create some small number (10 to 20) of jobs. Abort the most ancient job, and determine if all subsequent jobs are executing.

4.5.5.11 TV15. Exercise the Invoke_Process interface (5.2.2.3). Determine if an invoked process, when aborted, does not block execution of the calling process. Invoke a process, abort it, and determine if the calling process is again executing.

4.5.5.12 TV16. Exercise the Abort_Process interface (5.2.2.11). Determine if a process that contains a task that aborts its containing process aborts the containing process. Create a process that contains a task that aborts its containing process. Determine if the containing process has been aborted.

4.5.5.13 TV17. Exercise the Delete_Process interface (5.2.2.5). Determine if Delete_Job deletes an entire tree of recursively spawned processes. Recursively spawn some small number of processes and call Delete_Job on the root process node. Determine if all descendant process nodes have been deleted and if the main calling process is now executing.

4.5.5.14 TV18. Exercise the Delete_Process interface (5.2.2.5). Determine that Delete_Job deletes only the job passed as a parameter and no others. Create several jobs, delete them one at a time, ensuring that the remaining jobs have not been deleted.

4.5.5.15 TV19. Exercise the Await_Process_Completion interface (5.2.2.2). Determine if Await_Process_Completion indeed blocks the main process until the spawned process is finished. From some main process, spawn a process, and then call Await_Process_Completion. Determine if the main process is blocked while waiting for the spawned process to complete. Determine if the actual time spent waiting for process completion is at least equal to the time limit parameter of the function. Accomplish this with and without the time limit.

4.5.5.16 TV20. Exercise the Await_Process_Completion interface (5.2.2.2). Determine if Await_Process_Completion blocks a main process until the created job completes. From some main process, create a job, and then call Await_Process_Completion. Determine if the main process is blocked while waiting for the created process to complete. Ensure that the actual time spent waiting for process completion is at least equal to the time limit parameter of the function.

4.5.5.17 TV21. Exercise the Suspend_Process (5.2.2.12) and Resume_Process (5.2.2.13) interfaces. Determine if processes can be directly suspended and resumed. Spawn some large (50 to 100) number of processes. Suspend each, and determine that all are suspended; resume each and determine if all are executing.

4.5.5.18 TV22. Exercise the Suspend_Process (5.2.2.12) and Resume_Process (5.2.2.13) interfaces. Determine if processes can be indirectly suspended and resumed. Spawn some small (10 to 20) number of processes, each of which spawns another process. Suspend the grandchild processes, and determine if the child processes are still executing. Resume the grandchild processes, then suspend the child processes. Determine if the child and grandchild processes are suspended.

4.5.5.19 TV23. Exercise the Suspend_Process (5.2.2.12) and Resume_Process (5.2.2.13) interfaces. Determine if trees of processes are suspended and resumed correctly. Recursively spawn some small number (5 to 10) of processes, and suspend the most ancient one. Determine if all descendant processes are suspended. Resume the most ancient one, and determine if all descendant processes are resumed.

4.5.5.20 TV24. Exercise the Spawn_Process (5.2.2.1) and Get_Parameters (5.2.2.10) interfaces. Demonstrate a pop-up alarm clock. Create a process and pass it a time in seconds to delay, then send output to the screen.

4.5.5.21 TV32. Exercise the Write_Results (5.2.2.7) and Get_Results (5.2.2.8) interfaces. Determine if results can be consistently written to and read back from a process. This is to be attempted a large number (1000) of times.

4.5.5.22 TV33. Exercise the Append_Results (5.2.2.6) and Get_Results (5.2.2.8) interfaces. Determine that the null string is the first result from a process. Determine what exception is raised after appending results repeatedly

4.5.5.23 TV34. Exercise the Time_Started (5.2.2.16), Time_Finished (5.2.2.17), and Machine_Time (5.2.2.18) interfaces. Determine that when several processes are spawned, their Time_Started attributes reflect the order of process initiation. Also, determine that reasonable results are returned for the Time_Finished attribute when the processes are completed.

4.5.5.24 TV35. Exercise the Get_Parameters interface (5.2.2.10) by spawning processes, invoking processes, and creating jobs with and without parameters and then determining that the parameters that were passed arrived intact.

4.5.5.25 TV36. Exercise the Current_Status interface (5.2.2.9). Spawn processes, invoke processes, and create jobs and determine their status just after initiation, suspension, resumption, abortion, and termination. Recursively spawn several processes, abort the most ancient, and determine that the Current_Status of all descendant processes is correct.

4.5.5.26 TV37. Exercise the Io_Unit_Count interface (5.2.2.15). Invoke some process that makes a known number of Get/Put operations. After termination, determine that the value of the Io_Unit_Count attribute of the proces node is correct. Invoke some process that makes a known number of Get/Put operations and that periodically suspends itself. During periods of suspension determine that the value of the Io_Unit_Count attribute is correct. Invoke some process that opens a known number of node handles and that periodically suspends itself. During periods of suspension, determine that the value of the Open_Node_Handle_Count is currently correct.

4.5.5.27 TV38. Exercise the Process_Size interface (5.2.2.19). Determine that the size of a spawned process and the size of an aborted process are nonzero, and that the size of two spawned processes with the same target is the same size.

4.6 Direct IO/Sequential IO/Text IO Tests

The CAIS_DIRECT_IO and CAIS_SEQUENTIAL_IO packages provide facilities for directly accessing and sequentially accessing data elements in SWG CAIS files. The CAIS_TEXT_IO package provides facilities for accessing textual data elements in SWG CAIS files.

4.6.1 Test Strategy

The SWG CAIS input/output nominal tests are designed such that the same test strategy is employed for each of the direct input/output and sequential input/output tests. text input/output nominal tests are handled separately. The direct input/output test names begin with TI followed by a two-digit number. The sequential input/output test names begin with TJ followed by a two-digit number. All tests with the same two digits have identical test executions. The text input/output nominal tests have a TK prefix.

Usability tests are intended to test efficiently many interfaces working jointly in a realistic manner. However, because of the interdependencies in the interfaces, failure of an early interface may leave later interfaces untested. The usability tests are limited to CAIS_TEXT_IO interfaces.

4.6.2 Specific Support Packages

There are two support packages required for the proper execution of the SWG CAIS I/O tests. The first is a generalized report writer that formats expected outputs and reports anomalous results. The second support package involves the ability to measure the delta time between events.

4.6.3 Organization of Node Model

The node model will be a dynamic list of file node handles that may be accessed by an index.

4.6.4 Nominal Tests

The tests with prefix TI and TJ use identical logic and structure in their tests. They will be discussed as one test with TI representing CAIS_DIRECT_IO tests and with TJ representing CAIS_SEQUENTIAL_IO tests.

In all test cases, each interface is executed with the following instantiations to check for proper execution: "Integers", "Strings", "Enumeration", "Fixed", "Float", "Boolean", "Character", "Array", "Records". If all tests pass without errors, the test is said to pass. Otherwise, the test fails.

4.6.4.1 TI01 and TJ01. Exercise the Create interfaces (5.3.4.2 and 5.3.5.3). Additional tests are executed to determine that a file of mode In_File can be created, that a file of mode Out_File can be created, and that a file of mode Inout_File can be created.

4.6.4.2 TI02 and TJ02. Additional tests are executed to determine that a true is returned when Is_Open is applied to an open file of mode Out_File, that a true is returned when Is_Open is applied to an open file of mode In_File, that a true is returned when Is_Open is applied to an open file of mode Append_File, and that a false is returned when Is_Open is applied to a closed file of mode Out_File.

4.6.4.3 TI03 and TJ03. Additional tests are executed to determine that mode Out_File is returned when mode is applied on a file of mode Out_File, that mode In_File is returned when mode is applied on a file of mode In_File, that mode Append_File is returned when mode is applied on a file of mode Append_File. and that exception Status_Exception is returned when Mode is applied to a file already closed.

4.6.4.4 TI04 and TJ04. Exercise all instantiations of the Close interfaces (5.3.4.4 and 5.3.5.4). Additional tests are executed to determine that an open file is closed when Close is applied, and that no exception is raised when Close is applied to an already closed file.

4.6.4.5 TI05 and TJ05. Exercise the Open interfaces (5.3.4.3 and 5.3.5.3). Additional tests are executed to determine that a file with mode In_File can be successfully opened, that a file can be opened with mode Out_File, that a file can be opened with mode Append_File, and that a Status_Error exception is raised when opening an already opened file.

4.6.4.6 TI06 and TJ06. Exercise the Write interfaces (5.3.4 and 5.3.5). Additional tests are executed to determine that an element can be written to a file. Also tests the Synchronize interface (5.3.4.6 and 5.3.5.6), to determine that the exception Mode_Error is raised when a Write is applied to an open file of mode In_File, and that the exception Status_Error is raised when Write is applied to a closed file.

4.6.4.7 TI07 and TJ07. Exercise the Reset interfaces (5.3.4.5 and 5.3.5.5). Additional tests are executed to determine that a file can be reset to mode In_Mode, that a file can be reset to mode Out_File, that a file can be reset to mode Append_File, and that the exception Status_Error is raised when resetting a closed file.

4.6.4.8 TI08 and TJ08. Exercise the End_of_File interfaces (5.3.4 and 5.3.5). In addition to the complete set of instantiations being performed, tests are conducted to determine that End_of_File returns true when applied to an open file at End_of_File as expected, that false is returned when an End_of_File test is made on a file containing additional elements, that Mode_Error is raised when an End_of_File test is made on a file of mode Out_File, and that Status_Error is raised when a closed file is tested for End_of_File.

4.6.4.9 TI09 and TJ09. Exercise the Read interfaces (5.3.4 and 5.3.5). In addition to the usual instantiation tests, tests are run to determine that elements in a file can indeed be read, that the exception End_Error is raised when if Read is applied to a file at the End_of_File as expected, that the exception Mode_Error is raised when Read is applied to a file not of mode In_File as expected, and that a Status_Error is raised when Read is applied to a closed file as expected.

4.6.4.10 TK01. This package tests the proper execution of the Create, Open, Close, Is_Open, Mode, and End_of_File (5.3.6) for text files. The test strategy is to create a text file node, attempt to apply Open to the text file, determine the proper operation of the Open with Is_Open, apply Mode to the created file, determine that an open file can be closed, verifying the proper execution of Close with Is_Open exception, and determine that a file can be re-opened after a Close operation, and verify that an exception is raised when an End_of_File check is made on a file of mode Out_File. Any exceptions raised during the execution of the above tests will be reported. Execution will then proceed with the next test in sequence.

4.6.4.11 TK02. This package executes test cases for Get, Put, New_Line, Skip_Line, and End_of_File interfaces for text files. The test strategy employed is to create and open a text file node for output, write predefined text to the output file and close it, reopen the file and apply a Get to it, verifying for the same text that was previously written, apply Skip_Line to the file and perform a Get_Line and compare to predefined results to verify proper execution, verify that the file was properly constructed and accessed by performing a Skip_Line and then another Get_Line and comparing to expected results, and execute End_of_File and verify results. If any exceptions are raised during test execution, an error message is reported with execution continuing at the next test in sequence.

4.6.4.12 TK03. This package executes test cases for the Set_Input and Set_Output interfaces for text files. The test strategy is to create and open two text files with Inout intent, apply Is_Open to verify proper execution of the Open, write predefined text into each file and reset both files to input, apply Set_Input to the first file and Set_Output to the second file, apply Mode to determine the proper execution of Set_Input and Set_Output, and verify the integrity of the file by performing Get_Line and comparing to the previously written predefined text. Any unexpected exceptions raised cause an error to be reported with execution continuing with the next test in sequence.

4.6.5 Usability Tests

Three usability tests will be executed to determine the performance characteristics of the Put_Line, Get_Line, and Synchronize interfaces.

4.6.5.1 TK101. This package invokes the Put_Line interface for an open file node up to the pragmatic limit. Output is made to the report file each 100 executions along with the time required for the execution. The average time for each Put_Line is also calculated using the interval timer results.

4.6.5.2 TK102. This package executes Get_Line interface for an open file node 1,000 times and computes the average time for each incidence. Output is made to the report file to record interim results.

4.6.5.3 TK103. This package executes the Synchronize interface 100 times and calculates the average time required for each execution. Interim results are recorded every 10 iterations.

4.7 Import_Export Tests

The IMPORT_EXPORT package interfaces are tested indirectly by all other tests. Some tests will call the IMPORT_EXPORT interfaces, and other tests will rely on the implementation-dependent SWG CAIS administrative services tools to import and export host files directly. No explicit tests will be written for these interfaces.

4.8 Page_Terminal_IO Tests

The CAIS_PAGE_TERMINAL_IO package provides the capability to communicate with page terminal devices. Positions on a display are directly addressable and are arranged into horizontal rows and vertical columns. Each position on the display is identifiable by the combination of a positive row number and a positive column number. The test strategy involves the development of just one usability test.

4.8.1 Nominal Tests

No nominal tests were developed for this interface.

4.8.2 Usability Tests

An open file node handle with In_Out intent is required for the successful execution of the Page_Terminal_Io test called "TP01". This test is a modification of a NOSC page terminal test of the same name.

TP01 executes test cases for various instantiations of the CAIS_PAGE_TERMINAL_IO package. The interfaces used in the test include the following: Open (5.3.9.2), Erase_in_Display (5.3.9.30), Page_Size (5.3.9.12), Set_Active_Position (5.3.9.10), Delete_Line

(5.3.9.28), Insert_Line (5.3.9.33), Erase_Line (5.3.9.31), Put (5.3.9.18), Get (5.3.9.19), Select_Graphic_Rendition (5.3.9.35), and Close (5.3.9.3).

The test strategy is to open a file node handle for the page terminal device, obtain the terminal characteristics, and exercise the terminal in a manner typically used for full screen editors or for forms entry. The following sequence is executed to test the paged terminal capabilities. Limited exception checking is performed during the test sequences.

- Label each line and place "#" in reverse video.
- Test the retrieval of Ada characters by requesting the input.
- Test the deleting of lines.
- Test for the insertion of lines.
- Move the cursor on the screen in random positions.
- Get a string of Ada characters.
- Get a single Ada character or a function key.

4.9 List Management Tests

These tests exercise the interfaces defined in the CAIS_LIST_MANAGEMENT package (5.4.1). CAIS_LIST_MANAGEMENT implements an abstract data type (i.e., list) that is constructed of items that may be sublists, strings, integers, floats, or identifiers. Items within lists may be either named or unnamed (i.e., positional). The operations provided in CAIS_LIST_MANAGEMENT include insertion and deletion of items, identification of sublists, replacement and extraction of items, conversions to and from text, and determination of the kind of an item or list.

4.9.1 Test Strategy

The list management tests were able to take advantage of several previously written tests. Tests originally developed by NOSC were used as the basis for all of the list management nominal tests. These tests were modified to reflect changes in the CAIS and to expand coverage. The modified NOSC tests were also used to develop the usability tests.

Both the nominal and the usability tests for list management follow the same basic format. Similar interfaces (e.g., Extract_List and Extract_Value) are grouped together within a single test; each test loops reading text input from an American standard code for information exchange (ASCII) file and after reading a set of data, a subtest is performed in which a list is created, the specified operation is performed, and actual results are compared against expected results. Because lists are read from the textual input file, it is easy to specify a variety of list forms so that the list management interfaces can be tested with respect to a

wide range of list structures and item values. It also allows new subtests to be run without modifying or recompiling the Ada source code for that particular interface.

Usability tests differ from nominal tests in that they exercise interfaces using complex lists, where a complex list is defined to be a list containing approximately 255 items and those items are a mixture of various item_kinds. The usability tests also check for specific limits on list sizes.

Because lists are abstract data types provided by CAIS and because these tests are designed to perform black-box testing, each test necessarily relies on list management interfaces other than the interfaces specifically being tested. The CAIS_LIST_MANAGEMENT interfaces used by other list management tests are Convert_Text_To_List, Set_To_Empty_List, Text_Form, Convert_Text_To_Token, and Insert. Proper implementation of these interfaces is essential to the proper testing of CAIS_LIST_MANAGEMENT using this test set.

4.9.2 Specific Support Packages

The list management tests make use of a generalized REPORT package. REPORT provides standardized interfaces for creating test reports. Procedures are provided for reporting test names, indicating pass/fail status for subtests, and reporting an overall test result (that indicates passage only if all subtests pass).

The usability tests for list management make use of additional support procedures:

- CREATE_COMPLEX_LIST--Reads the text form of items from an input file and inserts them one at a time into the complex list being created.
- PRINT_COMPLEX_LIST--Formats a complex list into 80-character lines and prints it to the test output file.

4.9.3 Organization of Node Model

There is no need to interact with the CAIS node management facilities in order to test the list management interfaces. These tests do not create or depend upon the existence of any nodes or particular instance of the node model.

4.9.4 Nominal Tests

4.9.4.1 TB01. Exercise the Copy_List interface (5.4.1.2). Obtain input from file TB01.IN. Simply loop, reading a text string from TBO1.IN on each pass. For each list copied, display both the original and the copied list, and report the result of the comparison as the test either passing or failing. After performing all subtests, issue a summary result indicating "pass" only if all lists were copied correctly.

4.9.4.2 TB02. Exercise the three interfaces defined for the Delete interface (5.4.1.7). Obtain input from file TB02.IN. Test all forms of delete (by position, by identifier, or by token). After calling Delete to remove an item from a list, compare the resulting list to an expected result. Display the original list, resulting list, expected result, kind of delete, and the deleted item's name or position, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only if all items were deleted correctly.

4.9.4.3 TB03. Exercise the Is_Equal interface (5.4.1.6). Obtain input from file TB03.IN. For each subtest compare the two input lists, and verify the result of this comparison against an expected result of either true or false. Report the two input lists and the expected result, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only if all comparisons were performed correctly.

4.9.4.4 TB05. Exercise the Extract_List interface (5.4.1.12), the interface for Get_Item_Name (5.4.1.19), and the three interfaces defined for Extract_Value (5.4.1.6). Obtain input from file TB05.IN. For each subtest, invoke the appropriate extraction interface, and then compare the result to the expected result. Report the kind of extraction, the original list, the specified name or position (or start position and count), the result, and the expected result, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only if all comparisons were performed correctly.

4.9.4.5 TB07. Exercise all of the interfaces for CAIS_Identifier_Item (5.4.1.22.2 - .8). This package provides interfaces for the manipulation of tokens within lists: Extract, Insert, Replace, and Position_by_Value. It also supports comparison of tokens and converting text to tokens. Test extraction, insertion, and replacement for all three forms of keys (i.e., by position, by identifier, and by token). Obtain input from file TB07.IN. For each subtest, invoke the appropriate CAIS_Identifier_Item interface, and print both the test inputs and the test results. The test result file must be read to verify the correctness of tests.

4.9.4.6 TB08. Exercise the three interfaces defined for Kind_of_Item (5.4.1.9). Test all three forms of keys (by identifier, by position, and by token) allowed for specifying the list item whose "kind" is being requested. Obtain input from file TB08.IN. For each subtest, compare an expected "kind" to the one actually returned. Report the test inputs, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only if all comparisons were performed correctly.

4.9.4.7 TB09. Exercise the three interfaces defined for Insert (5.4.1.21.3). Test all three methods of inserting a sublist into a list (into an unnamed list, into a named list using an identifier, and into a named list using a name token). Also check that exceptions are raised as expected. Obtain input from file TB09.IN. Perform the insertion and either compare the successful result to the expected result or, when an exception occurs, compare the exception to an expected error. Report the test inputs, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only when results occur as expected.

4.9.4.8 TB10. Exercise all of the interfaces for CAIS_Integer_Item (5.4.1.23.1 through 5.4.1.23.5). This provides for manipulation of integers within lists: Extracted_Value, Insert, Replace, and Position_by_Value. It also supports converting integer values to Text_Form. Test extraction, insertion, and replacement for all three forms of keys (i.e., by position, by identifier, and by token). Obtain input from file TB10.IN. For each subtest, invoke the appropriate CAIS_INTEGER_ITEM interface, and print both the test inputs and the test results, as well as a pass/fail indication.

4.9.4.9 TB11. Exercise the Number_of_Items interface (5.4.1.13) and the four interfaces defined for Text_Length (5.4.1.21.3). Test all three methods (by position, by identifier, and by token) of specifying a sublist whose length is requested. Also check that exceptions are raised as expected. Obtain input from file TB11.IN. Perform the length request and either compare the successful result to the expected result or, when an exception occurs, compare the exception to an expected error. Report the test inputs, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only when results occur as expected.

4.9.4.10 TB12. Exercise the Kind_of_List interface (5.4.1.6). Simply loop, reading two inputs from TB12.IN on each pass: a list to be checked, and an expected result of Unnamed, Named, or Empty. Perform the Kind_of_List operation on the input list, and compare the result to the expected result. Report the list, actual result, and expected result, as well as an indication of pass/fail in the test results file. After performing all subtests, report a summary result indicating "pass" only if all comparisons were performed correctly.

4.9.4.11 TB13. Exercise the Position_by_Value interface (5.4.1.21.4) and both interfaces (by identifier or by token) for the Position_by_Name operation (5.4.1.20). Also check that exceptions are raised as expected. Obtain input from file TB13.IN. Perform the position request, and either compare the successful result to the expected result or, when an exception occurs, compare the exception to the expected error. Report the test inputs, as well as an indication of pass/fail in the test results file.

4.9.4.12 TB14. Exercise the three interfaces defined for Replace (5.4.1.6). Obtain input from file TB14.IN. For each subtest, invoke the appropriate replacement interface, and then compare the result to the expected result. Report the kind of replacement, the original list, the specified identifier, token, or position being replaced, the result, and the expected result, as well as an indication of pass/fail in the test results file.

4.9.4.13 TB15. Exercise the Splice interface (5.4.1.10) and Concatenate_Lists interface (5.4.1.11). Also check for proper propagation of exceptions. Obtain input from file TB15.IN. Perform the appropriate merge and either compare the successful result to the expected result or, when an exception occurs, compare the exception to an expected error. Report the inputs, as well as an indication of pass/fail in the test results file.

4.9.4.14 TB16. Exercise all of the interfaces for CAIS_String_Item (5.4.1.25 .1 - .4). These interfaces support manipulations of strings within lists: Extracted_Value, Insert, Replace, and Position_by_Value. Test extraction, insertion, and replacement for all three forms of keys (i.e., by position, by identifier, and by token). Simply loop, reading a test indicator and the appropriate number of inputs from TB16.IN on each pass. For each subtest,

invoke the appropriate CAIS_STRING_ITEM interface, and print both the test inputs and the test results, as well as a pass/fail indication.

4.9.5 Usability Tests

4.9.5.1 TB25. Perform five different checks on the maximum sizes supported for simple lists. Create a list and add items (empty lists) to it until no more can be added. Determine that the list can be at least as long as CAIS_List_Length (255 items) and that Capacity_Error is raised when no more items can be inserted. Determine the condition of the node model at this point. Repeat the first test, creating the largest acceptable list. Determine whether other lists can grow to be as large as the first. Recursively create a list containing other lists until no more lists can be inserted. (Let each list contain nothing other than the next nested list.) Determine whether, at this point, the limit is on depth of nesting or on total number of contained items. Create, using calls to Insert and Set_To_Empty_List, the largest list of lists acceptable to the implementation. Compare its size with that from test 3 above. Create the largest list acceptable to the implementation; call Copy_List and determine that the second list is the same as the first.

4.9.5.2 TB26. Evaluate the use of Insert to create large, complex lists. Create a list of items of every type except floating point values. Repeat with each list having the deepest level of nesting accepted by the implementation. Create these complex lists by repeated calls to Insert (using all overloaded forms and using items of all types). Proper operation must be checked by printing and verifying the lists that are created.

4.9.5.3 TB27. Evaluate the Replace operation (5.4.1.21.2) on large, complex lists. Check all three forms (by position, by identifier, and by token) of specifying the replacement. Create a complex (e.g., many levels of nesting) list using items of all types. Call Replace (using all overloaded forms and using items of all types) some number of times to alter the list. Proper operation must be checked by printing and verifying the lists that are created.

4.9.5.4 TB28. Evaluate the Number_of_Items (5.4.1.13) and Text_Length (5.4.1.18) operations on large, complex lists. Call Text_Length on an empty list; determine that a positive value is returned. Create a complex named list and call Text_Length, using all interfaces. Determine that the results for each call are correct. Create an unnamed list and repeat the applicable tests. Also call Number_of_Items for these tests and determine that the results for each call are correct. Proper operation must be checked by printing and verifying the lists that are created.

4.9.5.5 TB29. Evaluate the Extract (5.4.1.12), Get_Item_Name (5.4.1.19) and Extract_Value (5.4.1.21.1) operations on large, complex lists. Create a complex list containing items of all types and with several levels of nesting. Call Extract_List on the resulting list to extract one of the original lists. Determine that the extracted list is the same as the original. Call Extract{ed}_Value for items of each type, using each overloaded form; determine that the proper value has been extracted in each case. Use the same complex lists to exercise the Get_Item_Name interface. Proper operation must be checked by printing and verifying the lists that are created.

4.9.5.6 TB30. Evaluate the Delete operation (5.4.1.7) on large, complex lists. All three forms (by position, by identifier, and by token) are evaluated. Create some complex lists (e.g., with deeply nested lists and containing items of all types). Delete items from the list, determining at each stage that the list is properly formed. Proper operation must be checked by printing and verifying the lists that are created.

4.9.5.7 TB31. Evaluate the Splice (5.4.1.10) and Concatenate_Lists (5.4.1.11) operations on large, complex lists. Create two complex named lists (e.g., with several levels of nesting). Concatenate and determine that the result list is properly formed. Repeat, with one of the lists named and the other unnamed. Concatenate and determine that List_Kind_of_Error is raised. Repeat, concatenating an empty list to a named list, then to an unnamed list. Determine in each case that the result list is the same as the original nonempty list. Splice one named list into another; determine that the result list is properly formed. Repeat with the two lists of different kinds; determine that List_Kind_Error is raised. Proper operation must be checked by printing and verifying the lists that are created.

4.9.5.8 TB32. Evaluate the Kind_of_List operation (5.4.1.21.2) on large, complex lists. Create complex lists from input, and compare the actual kind of list to the expected kind. Display the complex list, actual kind, expected kind, and a pass/fail indication for each sub-test.

4.9.5.9 TB33. Evaluate the Position_by_Name (5.4.1.20) and Position_by_Value (5.4.1.21.2) operations on large complex lists. Create complex lists from input, and perform requests for positions. Test both forms (by identifier and by token) of Position_by_Name. Also test for possible exceptions. For each test, the complex list, expected result, and actual result are printed, as well as a pass/fail indication.

4.9.5.10 TB34. Evaluate the Kind_of_Item (5.4.1.21.2) operation on large, complex lists. Test all three forms of Kind_of_Item (by position, by identifier, and by token). Report the complex list, item indicator, expected item kind, and actual item kind, as well as a pass/fail indication for each subtest.

APPENDIX

SWG CAIS Test Suite Traceability Matrix

CAIS_NODE_MANAGEMENT	
Test Name	Interface Tested
TC01	Open (5 1 2 1)
TC02	Close (5 1 2 2)
TC03	Kind_Of_Node (5 1 2 6)
TC04	Is_Open (5 1 2 4)
TC05	Open_Parent (5 1 2 19)
TC06	Delete_Node (5 1 2 23)
TC07	Create_Secondary_Relationship (5 1 2 25)
TC08	Delete_Secondary_Relationship (5 1 2 26)
TC09	Set_Current_Node (5 1 2 37)
TC10	Get_Current_Node (5 1 2 38)
TC11	Change_Intent (5 1 2 3)
TC12	Intent (5 1 2 5)
TC13	Open_File_Handle_Count (5 1 2 7)
TC14	Primary_Name (5 1 2 8)
TC15	Primary_Key (5 1 2 9)
TC16	Primary_Relation (5 1 2 10)
TC17	Path_Key (5 1 2 11)
TC101	Create_Node (5 1 5 1), Open (5 1 2 1), Close (5 1 2 2), Is_Open (5 1 2 4), and Delete (5 1 2 23)
TC102	17 interfaces from Open (5 1 2 1) through Is_Same (5 1 2 17)
TC103	Interfaces from Index (5 1 2 18) through Is_Inheritable (5 1 2 28) and Set_Current_Node (5 1 2 37) through Time_Attribute_Written (5 1 2 42)
TC104	Iterator interfaces Create_Iterator (5 1 2 30) through Delete_Iterator (5 1 2 36)

CAIS_ATTRIBUTE_MANAGEMENT	
Test Name	Interface Tested
TD01	Create_node_attribute (5.1.3.1)
TD02	Create_Path_Attribute (5.1.3.2)
TD03	Delete_Node_Attribute (5.1.3.3)
TD04	Delete_Path_Attribute (5.1.3.4)
TD05	Set_Node_Attribute (5.1.3.5)
TD06	Set_Path_Attribute (5.1.3.6)
TD07	Get_Node_Attribute (5.1.3.7)
TD08	Get_Path_Attribute (5.1.3.8)
TD101	Create_Node_Attribute (5.1.3.1)
TD102	Create_Path_Attribute (5.1.3.2)
TD103	Set_Node_Attribute (5.1.3.5) and Get_Node_Attribute (5.1.3.7)
TD104	Create_Path_Attribute (5.1.3.2), Create_Node_Attribute (5.1.3.1), and Create_Node (5.1.5.1)

CAIS_STRUCTURAL_NODE_MANAGEMENT	
Test Name	Interface Tested
TS101	Create_Node (5.1.5.1)
TS102	Create_Node (5.1.5.1)
TS103	Create_Node (5.1.5.1)
TS104	Create_Node (5.1.5.1)
TS105	Create_Node (5.1.5.1)
TS106	Create_Node (5.1.5.1) and Close (5.1.2.2)
TS107	Create_Node (5.1.5.1) and Delete_Node (5.1.2.23)
TS108	Create_Node (5.1.5.1), Create_Secondary_Relationship, (5.1.2.25), and Delete_Node (5.1.2.23)
TS109	Create_Node (5.1.5.1), Open (5.1.2.1), and Close (5.1.2.2)
TS110	Create_Node (5.1.5.1) and Copy (5.1.2.20)
TS111	Create_Node (5.1.5.1) and Create_Secondary_Relationship (5.1.2.25)
TS112	Create_Node (5.1.5.1) and Create_Secondary Relationship (5.1.2.25)
TS113	Create_Node (5.1.5.1), Create_Secondary_Relationship (5.1.2.25), and Delete_Secondary_Relationship (5.1.2.26)
TS114	Set_Current_Node (5.1.2.37)

CAIS_PROCESS_MANAGEMENT	
Test Name	Interface Tested
TG04	Spawn_Process (5.2.2.1)
TG05	Await_Process_Completion (5.2.2.2a)
TG06	Await_Process_Completion (5.2.2.2b)
TG07	Invoke_Process (5.2.2.3a)
TG08	Invoke_Process (5.2.2.3b)
TG09	Create_Job (5.2.2.4)
TG11	Delete_Job (5.2.2.5a)
TG12	Delete_Job (5.2.2.5b)
TG13	Append_Results (5.2.2.6)
TG14	Write_Results (5.2.2.7)
TG15	Get_Results (5.2.2.8a)
TG16	Get_Results (5.2.2.8b)
TG17	Get_Results (5.2.2.8c)
TG18	Get_Results (5.2.2.8d)
TG19	Current_Status (5.2.2.9a)
TG20	Current_Status (5.2.2.9b)
TG21	Get_Parameters (5.2.2.10)
TG22	Abort_Process (5.2.2.11a)
TG23	Abort_Process (5.2.2.11b)
TG24	Abort_Process (5.2.2.11c)
TG25	Abort_Process (5.2.2.11d)
TG26	Suspend_Process (5.2.2.12a)
TG27	Suspend_Process (5.2.2.12b)
TG28	Resume_Process (5.2.2.13a)
TG29	Resume_Process (5.2.2.13b)
TG30	Open_Node_Handle_Count (5.2.2.14a)
TG31	Open_Node_Handle_Count (5.2.2.14b)
TG32	Io_Unit_Count (5.2.2.15a)
TG33	Io_Unit_Count (5.2.2.15b)
TG34	Time_Started (5.2.2.16a)
TG35	Time_Started (5.2.2.16b)
TG36	Time_Finished (5.2.2.17a)
TG37	Time_Finished (5.2.2.17b)
TG38	Machine_Time (5.2.2.18a)
TG39	Machine_Time (5.2.2.18b)
TG40	Process_Size (5.2.2.19a)
TG41	Process_Size (5.2.2.19b)
TV01	Spawn_Process (5.2.2.1)
TV02	Spawn_Process (5.2.2.1)
TV03	Spawn_Process (5.2.2.1)
TV04	Invoke_Process (5.2.2.3)
TV08	Create_Job (5.2.2.4)
TV09	Create_Job (5.2.2.4)

CAIS_PROCESS_MANAGEMENT	
Test Name	Interface Tested
TV11	Abort_Process (5.2.2.11)
TV12	Abort_Process (5.2.2.11)
TV13	Abort_process (5.2.2.11)
TV14	Abort_Process (5.2.2.11)
TV15	Invoke_Process (5.2.2.3)
TV16	Abort_Process (5.2.2.11)
TV17	Delete_Job (5.2.2.5)
TV18	Delete_Job (5.2.2.5)
TV19	Await_Process_Completion (5.2.2.2)
TV20	Await_Process_Completion (5.2.2.2)
TV21	Suspend_Process (5.2.2.12) and Resume_Process (5.2.2.13)
TV22	Suspend_Process (5.2.2.12) and Resume_Process (5.2.2.13)
TV23	Suspend_Process (5.2.2.12) and Resume_Process (5.2.2.13)
TV24	Spawn_Process (5.2.2.1) and Get_Parameters (5.2.2.10)
TV32	Write_Results (5.2.2.7) and Get_Results (5.2.2.8)
TV33	Append_Results (5.2.2.6) and Get_Results (5.2.2.8)
TV34	Time_Started (5.2.2.16), Time_Finished (5.2.2.17), and Machine_Time (5.2.2.18)
TV35	Get_Parameters (5.2.2.10)
TV36	Current_Status (5.2.2.9)
TV37	Io_Unit_Count (5.2.2.15)
TV38	Process_Size (5.2.2.19)

CAIS_DIRECT_IO / CAIS_SEQUENTIAL_IO	
Test Name	Interface Tested
TI01/TJ01.	Create (5.3.4.2/5.3.5.2)
TI02/TJ02.	Is_Open (5.3.4/5.3.5)
TI03/TJ03.	Mode (5.3.4/5.3.5)
TI04/TJ04.	Close (5.3.4.4/5.3.5.4)
TI05/TJ05.	Open (5.3.4.3/5.3.5.3)
TI06/TJ06.	Write (5.3.4/5.3.5)
TI07/TJ07.	Reset (5.3.4.5/5.3.5.5)
TI08/TJ08.	End_Of_File (5.3.4/5.3.5)
TI09/TJ09.	Read (5.3.4/5.3.5)

CAIS_TEXT_IO	
Test Name	Interface Tested
TK01	Create, Open, Close, Is_Open, Mode, and End_Of_File for text files (5.3.6)
TK02	Get, Put, New_Line, Skip_Line, and End_Of_File for text files (5.3.6)
TK03	Set_Input and Set_Output interfaces for text files (5.3.6)
TK101	Put_Line (5.3.6)
TK102	Get_Line (5.3.6)
TK103	Synchronize (5.3.6.6)

CAIS_PAGE_TERMINAL_IO	
Test Name	Interface Tested
TP01	Page_Terminal interfaces (5.3.9)

CAIS_LIST_MANAGEMENT	
Test Name	Interface Tested
TB01	Copy_List (5.4.1.2)
TB02	Delete (5.4.1.7)
TB03	Is_Equal (5.4.1.6)
TB05	Extract_List (5.4.1.12), Get_Item_Name (5.4.1.19), Extract_Value (5.4.1.6)
TB07	CAIS_Identifier_Item (5.4.1.22.2 - 8)
TB08	Kind_Of_Item (5.4.1.9)
TB09	Insert (5.4.1.21.3)
TB10	CAIS_Integer_Item (5.4.1.23.1 - 5)
TB11	Number_Of_Items (5.4.1.13) and Text_Length (5.4.1.21.3)
TB12	Kind_Of_List (5.4.1.6)
TB13	Position_By_Value (5.4.1.21.4) and Position_By_Name (5.4.1.20)
TB14	Replace (5.4.1.6)
TB15	Splice (5.4.1.10) and Concatenate_Lists (5.4.1.11)
TB16	CAIS_String_Item (5.4.1.25.1 - 4)
TB25	Insert (5.4.1.3), Copy_List (5.4.1.2) and Set_To_Empty (5.4.1.1)
TB26	
TB27	Replace (5.4.1.21.2)
TB28	Number_Of_Items (5.4.1.13) and Text_Length (5.4.1.18)
TB29	Extract (5.4.1.12), Get_Item_Name (5.4.1.19) and Extract_Value (5.4.1.21.1)
TB30	Delete (5.4.1.7)
TB31	Splice (5.4.1.10) and Concatenate_Lists (5.4.1.11)
TB32	Kind_Of_List (5.4.1.21.2)
TB33	Position_By_Name (5.4.1.20) and Position_By_Value (5.4.1.21.2)
TB34	Kind_Of_Item (5.4.1.21.2)

REFERENCES

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, Honolulu, HI: Western Periodicals Co., , 1985.

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

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 B. Queyras, "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, Helen Gill, Charles Howell, Tana Reagan, and Thomas Smith, "Distributing the Common APSE (Ada Programming Support Environment) Interface Set (CAIS)," MTR-86W00181, McLean, VA: The MITRE Corporation, January 1987.

Bowerman, Rebecca E., *Study of the Common APSE Interface Set (CAIS)*, 1 October 1985.

Carney, David J., "On The CAIS Implementation," IDA Memorandum Report M-482, 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 Anthony F. Ferrara, "An Automated Pascal Multiple Condition Test Coverage Tool," in *Proceedings COMPSAC 84.*, pp. 20-26, IEEE Computer Society Press, 7-9 November 1984.

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

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

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

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

Henke, Friedrich W. von, David Luckham, Bernd Krieg-Brueckner, and Olaf Owe, "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., Jeff Facemire, and Dennis Kafura, "A Specification Technique for the Common APSE Interface Set," 84004-R, Computer Science Dept., VPI, April 1984.

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

Luckham, David and Friedrich W. von Henke, "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 Carl F. Schaefer, "DIANA Reference Manual," IR-MD-078, Intermetrics, Inc., 5 May 1985.

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

Nyberg, Karl A., Audrey A. Hook, and Jack F. Kramer, "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, Hans Schaller, and Anton Paul Spirk, "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 et al.,, "Validation, Verification, and Testing of Computer Software," *ACM Computing Surveys*, vol. 14, no. 2, pp. 159-192, ACM, June 1982.

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

"Using the ACVC Tests ," ACVC Version 1.9.

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

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

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

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

NATO, "NATO SWG APSE Requirements," , 25 August 1987.

NATO, *Requirements for Ada Programming Support Environments, STONEMAN*, , August 1980.

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

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

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.
CAIS-A	Common Ada Programming Support Environment (APSE) Interface Set Upgrade
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.
DEC	Digital Equipment Corporation
DIANA	Descriptive Intermediate Attributed Notation for Ada
DOD	Department of Defense.

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.
IBM	International Business Machines
I/O	Input/output
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.
NATO	North Atlantic Treaty Organization.
OS	Operating System
PCTE	Portable Common Tool Environment

PCTE+	Portable Common Tool Environment upgrade
SWG	Special Working Group
SWG CAIS	Title given to the document which provided the specifications specific CAIS implementation is 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, development 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.
UK	United Kingdom
UNIX	A widely-used operating system originally developed by Bell Telephone Laboratories.
VAX	Virtual Address eXtension. A trademark of Digital Equipment Corporation. The name of a widely-used computer system from Digital Equipment Corporation.
VMS	Virtual Memory System. A trademark of Digital Equipment Corporation. An 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.
Debugging	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 an item.
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. The state of a node model instance is the current status of that instance. Prior to test execution, an initial state of the node model instance should be defined.
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.