

AD-A215 838

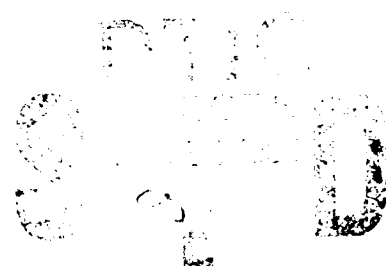
2

Technical Document 552
October 1989

KAPSE Interface Team (KIT) Public Report Volume VII

D. L. Hayward

Prepared for the Ada Joint Program Office



Approved for public release; distribution is unlimited

89 12 11 0

NAVAL OCEAN SYSTEMS CENTER

San Diego, California 92152-5000

J. D. FONTANA, CAPT, USN
Commander

R. M. HILLYER
Technical Director

ADMINISTRATIVE INFORMATION

This work was performed by the Computer Systems Software and Technology Branch, Code 411, of the Naval Ocean Systems Center, San Diego, CA, for the Ada Joint Program Office, Pentagon, 1211 S. Fern Street, Washington, DC 20301-3061 and represents evolving ideas and progress of the KAPSE Interface Team (KIT).

Released by
R. A. Wasilausky, Head
Computer Systems Software
and Technology Branch

Under authority of
A. G. Justice, Head
Information Processing and
Displaying Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

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

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE October 1989		3. REPORT TYPE AND DATES COVERED Final May 1985 to October 1985	
4. TITLE AND SUBTITLE KAPSE INTERFACE TEAM (KIT) PUBLIC REPORT Volume VII				5. FUNDING NUMBERS C: PE: 0603226F PR: WU: DN288 534	
6. AUTHOR(S) D. L. Hayward				8. PERFORMING ORGANIZATION REPORT NUMBER NOSC TD 552	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Ocean Systems Center San Diego, CA 92152-5000				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Ada Joint Program Office Pentagon 1211 S. Fern Street Washington, DC 20301-3381					
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report is the seventh in a series and represents evolving ideas and progress of the KAPSE Interface Team (KIT).					
14. SUBJECT TERMS software engineering CAIS Ada APSE interface standard programming languages				15. NUMBER OF PAGES 557	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT		

CONTENTS

Section

1. INTRODUCTION	1-1
Meetings	1-1
Common APSE Interface Set (CAIS)	1-1
Requirements and Design Criteria (RAC)	1-2
I&T Tools	1-2
Guidelines and Conventions Working Groups (GACWG)	1-2
Conclusion	1-3
2. TEAM PROCEEDINGS	2-1
3. KIT/KIT DOCUMENTATION	3-1
AIM Introduction of Players	3-1
Final Report on Interface Analysis and Software Engineering Techniques - Environment Interface Analysis Volume 1	3-105
Final Report on Interface Analysis and Software Engineering Techniques - Engineering Techniques Design and Implementation Experiences: The AIM Volume 2	3-185
Final Report on Interface Analysis and Software Engineering Techniques - Transporting an Ada Software Tool: A Case Study Volume 3	3-355
DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)	3-392
Rationale for the DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS)	3-421
DRAFT Guidelines and Conventions Working Group Ada Tool Transportability Study	3-482

Accession For	
NTIS GRA&I	☒
DTIC TAB	☒
Unannounced	☐
Justification	☐
By _____	
Distribution _____	
Availability _____	
Dist _____	
A-1	

Section 1

INTRODUCTION

INTRODUCTION

This report is the seventh in a series that is being published by the KAPSE Interface Team (KIT). The previous reports are as follows:

Vol. #	NOSC Report #	Date	NTIS Order #
I	TD-209	4/82	AD A115 590
II	TD-552	10/82	AD A123 136
III	TD-552	10/83	AD A141 576
IV	TD-552	4/84	AD A147 648
V	TD-552	8/85	AD A160 355
VI	TD-552	TBD	TBD

This series of reports serves to record the activities which have taken place to date and to submit for public review the products that have resulted. The reports are issued to cover approximate six-month periods. They should be viewed as snapshots of the progress of the KIT and its companion team, the KAPSE Interface Team from Industry and Academia (KITIA); everything that is ready for public review at a given time is included. These reports represent evolving ideas, so the contents should not be taken as fixed or final.

MEETINGS

During this reporting period (May 1985 through October 1985) the teams met in July 1985 in San Francisco, CA and in September 1985 in Saratoga Springs, NY. The approved minutes from these two meetings are included in this report. Also included are the minutes from the September COMPWG meeting.

COMMON APSE INTERFACE SET (CAIS)

Competitive procurement of a contractor for CAIS Version 2 is proceeding and is still scheduled to take place during 1985.

REQUIREMENTS AND DESIGN CRITERIA (RAC)

Perhaps the most significant products of this period were new drafts of the RAC and initial contributions to the RAC Rationale. New drafts of the RAC were produced in July, August and September, although only the September one is included here since the differences are small. Also included here is the first full draft of the RAC Rationale. It is built up of rationales written by independent groups for Sections 4, 5, and 6, with small parts of rationale for Section 2 provided by members of the RACWG. More work will be done on this in the future, but we are pleased to provide this much as an interim.

I&T TOOLS

The final report from the AIM implementation has been completed and briefed to the KIT/KITIA. The slides from the briefing and the report, entitled "APSE Interactive Monitor Final Report on Interface Analysis and Software Engineering Techniques", are included here. The report is in three volumes. The first, "Environment Interface Analysis," is an analysis of environment interface issues. The information in it is primarily a recap of data contained in previous AIM reports, with the addition of some information relevant to the Data General Ada Development environment (ADE). The second volume, "Design and Implementation Experiences: The AIM," covers design and implementation experience gained through work on the AIM. It includes information gathered by the AIM team during the implementation regarding both technical and managerial issues in using Ada. The final volume, "Transporting an Ada Software Tool: A Case Study," is a case study of the rehosting of the AIM from a DG Eclipse to a VAX 11/785. It contains information on the rehost effort, including transportability issues.

GUIDELINES AND CONVENTIONS WORKING GROUP (GACWG)

The GACWG has provided the first draft of the Ada Tool Transportability Guide for this volume of the Public Report. This guide is intended to address some of the I&T issues which transcend the use of a common interface set such as the CAIS. It draws on other published style guides, some of the AIM experiences and the knowledge of the GACWG members to provide various insights into improving tool transportability.

KIT Public Report

CONCLUSION

This Public Report is provided by the KIT and KITIA to solicit comments and feedback from those who do not regularly participate on either of the teams. Comments on this and all previous reports are encouraged. They should be addressed to:

Duston Hayward
Code 411
Naval Ocean Systems Center
San Diego, CA 92152-5000

or sent via ARPANET/MILNET to HAYWARD@NOSC-TECR.ARPA.

Section 2

TEAM PROCEEDINGS

KIT/KITIA MINUTES
MEETING OF 8-11 JULY 1985
SAN FRANCISCO, CALIFORNIA

ATTENDEES: SFF APPENDIX A

9 JULY 1985

1. OPENING REMARKS

- Introductions for new attendees Bob Ellison (SEI), Tanya Deriugin (Boeing), Lloyd Stiles (FCDSSA), Dave Pogge (NWC) and Bob Fainter (VPI) were made.

2. GENERAL BUSINESS

- No new contracts were awarded for KIT support or CAIS Version 2.0.
- Public Report V has not been turned in yet.
- The MIL-STD-CAIS, sent out at the end of April, is the only "accurate" copy (the green cover issue). Several members of the KIT/KITIA have not received their copy yet.
- The standardization process for the Common APSE Interface Set (CAIS) has started including a draft of a charter for a tri-service CAIS Control Board. This will not be a MIL-STD but DOD-STD (metric version).

More than 400 copies will be sent out to 5 industry associations, the SEI, and the Tri-Services. Each organization will consolidate its members' comments and will provide a consolidated position. The CAIS Control Board will receive 7 or 8 official responses and will have an independent contractor handle the responses with the CAISWG formulating the recommendations.

3. WORKING GROUP REPORTS

- KITIA REPORT - Herm Fischer reported that a document is being written to analyze the technological, economic and transitional issues involved in interface standardization. He also reported that a new KITIA working group is meeting this time to go over the document.
- E&V STATUS - Ray Szymanski (Wright-Patterson) was announced as the new chairman of the Evaluation and Validation Team. Two RFP's are coming, for the CAIS Validation suite and the Compiler Evaluation suite. The E&V team's CAISWG has changed their name to SEVWG

(Standards Evaluation & Validation Working Group). The next meeting will be held the first week of September in Dayton.

- DIANA - Rudy Krutar reported the copyright difficulties have been resolved with the draft of the "IDL" primer by Intermetrics.
- CAISWG - Jack Kramer reported the CAIS rationale (draft) document by Tim Harrison (TI) is under review. Tricia is shooting for January 1986 DOD-STD for CAIS. The proposed schedule is:

10-12 SEPT	CAISWG/CAIS Implementors Group joint meeting
23-27 SEPT	KIT/KITIA
1 OCT	General public review/comments//Cutoff of comments
21-25 OCT	CAISWG Comments Review
13-14 NOV	Public Review of Comments and CAISWG Responses
2-13 DEC	CAIS final draft

- RACWG - Hal Hart reported an editorial cleanup of the October RAC document. Sections 2,3 & 6 are pretty stable, but sections 4 and 5 are still under revision. Tricia wants to put the RAC out for public review; also SIGAda has shown interest. More than 400 will ultimately be sent out to the recipients of the proposed MIL-STD-CAIS.
- GACWG - Stewart French (TI) reported on the review of the draft Ada Tool Transportability Guide.
- COMPWG - Bernie Abrams reported ways of specifying semantics of CAIS to ensure compliance. They hope to have a document by the next KIT/KITIA meeting.
- STONEWG - Ann Reedy reported some writing assignments are to be handed in at this meeting.
- DEFWG - Hans Mumm has taken responsibility for the glossary effort. He is starting with good versions of CAIS and RAC glossaries.

4. ANNOUNCEMENTS

- ARTEWG is now a subcommittee in SIGAda. Dr. Lieblein says that funding for generation of a Ada run-time transportability guide to deliver to Congress is in the works through Wright-Patterson. The ARTEWG has a copy of our Ada Tool Transportability Guide as a foundation.
- Tim Lindquist has a PhD student looking at RISC architectures to support Ada.

Meetings Scheduled:

1985 9-12 SEP New York (GE)

1986 13-16 JAN San Diego
14-17 APR Atlanta (CDC)
7-10 JUL San Francisco (Ford Aerospace)
22-25 SEP Minneapolis, MN (Honeywell)

1987 19-22 JAN San Diego

5. PRESENTATIONS

- GSG Ada Environment work - Mike Vilot, a representative from the Center for Software Technology, a subsidiary of General System Group, presented information on a technology transfer environment developed by Europeans called the Integrated Development Environment for Ada. GSG is rehosting the Dansk Datamik compiler from VMS to UNIX based on PCTE (Portable Common Tool Environment), a multi-lingual environment.
- PLRS project - George Robertson (FCDSSA-SD) briefed the Position Locating Reporting System (PLRS), a project which provides position reporting, communication (voice & digital) and support navigation aid to the Army and Marine Corps users. The development and maintenance requirements for this project reiterated the need for a common set of interfaces like the CAIS. Asked if the CAIS was going to help he answered, "Once Ada is the programming language it will." (Currently CMS2 is the language.)

6. WORKING GROUP MEETINGS

WEDNESDAY JULY 10, 1985

7. AIM PRESENTATION

- Texas Instruments presented a briefing on the Ada Interactive Monitor (AIM) project. John Foreman, Stewart French and Tim Harrison spoke on the APSE Interactive Monitor, an Ada tool (developed for DG's ADE and subsequently rehosted to DEC's VAX/VMS) which acts as an interface between a user and APSE processes. Despite a high degree of complexity and a high degree of tasking, the rehost required only 2.4 man-months over 4 weeks. The compiler quality influenced the required time.
- The Model Comparisons produced some interesting results. The 40-20-40 mapping model (design, code, test) when compared with the actual AIM effort showed dramatic differences (See Table 1 for comparison results. This also proved to be true with the Brooks model (plan, code, test, system test) and the GTE mapping model (plan, specification, design, design, code). More time was spent in design and implementation than the models predicted. The AIM did not require as much testing as the models predicted. If more time is

spent in requirements definition and design, then less time will be spent in integration and testing. Using a source level debugger reduced testing and increased productivity.

	PLAN	SPEC	DESIGN	CODE	TEST	SYSTEST
40/20/40			40	20	20	
AIM			65.7	27.7	6.6	
BROOKS	33			17	25	25
AIM	65.7			27.7	2.6	4.1
GTE	2	8	40	15	25	10
AIM	11.9	3.9	49.9	27.7	2.6	4.1

Table 1. Models vs. AIM Results

- SoftCost, Price-S, and COCOMO costing models were also applied. The SoftCost and COCOMO models predicted higher costs than actually experienced by 30%.
- The team quality was pretty good considering the tool was pretty difficult.
- They used object oriented design (Booch), but had some problems and produced three different prototypes. One result of the environment comparison (DG ADE and VAX/VMS) is that it is essential to have debuggers on both systems. VMS is easier to use than the ADE. Problems were discussed concerning DG's ADE and AOS. The TI team's Ada tasking experiences produced a taxonomy of Ada Tasks: servers (agent, e.g., buffer tasks), actors (generate requests -e.g., producer or monitor tasks), and transducers (combination -e.g., msg router or secretary tasks). A majority of tasks come in actor/server pairs.
- VAX/VMS is more integrated than the DG ADE. Transporting the AIM from DG to DEC was done at the source code level. It took 2.4 man months in just under one calendar month using 240 separate text files. There was one transfer back to DG, then some changes made to both with no conditional compilation used in order to assure that both systems are running identical code (except for the small machine-dependent sections).
- System dependencies were categorized as terminal control and communications: open the computer terminal (no echo, no buffering), close the computer terminal (reset), write data to the computer terminal display; process control and communication: create/destroy son processes, read/write a line from/to a son process; environment variables: getting the file names and the terminal names.
- Rehost problems included module testing baggage, representation clauses and complexities related to system services.

- CAIS - related issues involved terminals, processes, interprocess communications and files. Terminal packages do not use TEXT IO; they are self-contained, independent packages. Scroll, page, and form terminal packages had exclusions and additions. The Process model inherits characteristics from a parent and communicates with a parent via standard input or output and the file names were system dependent.
- Conclusions: Buhr's book (System Design with Ada) is a good book. Object oriented design was used. Some other form of data flow design is necessary to complement object oriented design. Regarding environment comparisons, Ada promotes transportability problems with tasking, especially termination. Planning for transportability works. There are interesting problems with debugging a completed system that has been rehosted. There may be serious problems implementing the CAIS process control and communications (IPC) interfaces in modern OS's.

THURSDAY JULY 11, 1985

8. PCTE PRESENTATION

- Olivier Roubine briefed the PCTE (A Basis for a Portable Common Tool Environment) project of the ESPRIT (European Strategic Plan for Research in Information Technology) program. No requirement exists for PCTE to be portable. The requirement is that it support tools that are portable. PCTE was designed for acceptance, not to be forced on anyone. It is to support various application domains, development activities and methods, and project organizations, including various programming languages: Ada, C, Pascal, Lisp. There is a basic set of primitives (a la UNIX).
- There is the notion of providing a consistent user interface, and the system must run on equipment of various manufacturers. This is initially based on UNIX: for the short term main effort now it uses UNIX System V and virtual memory; for the long term it will be based on Ada. The target architecture is workstations rather than mainframes. Acceptance to the general community was a major factor, even more so than portability. A single user workstation, a complete environment connected by LAN, high resolution display and pointing devices, and substantial amounts of processing power and storage capacity are all planned. The basic mechanisms are similar to the CAIS (a set of interfaces).
- Communications mechanisms are based on UNIX. The interprocess communications mechanism is based on providing information between processes by message passing, pipes, signals, and shared memory. The object management system is close to the CAIS node model. The user interface uses windows. The distribution is based on a LAN model with transparent access to rare resources and other workstations. Long range tools and research plans include a knowledge based programmer's assistant (KBPA).

9. INTEROPERABILITY PRESENTATION

- The talk presented the GACWG's and Hans Mumm's thoughts on interoperability. It covered several definitions for interoperability, a suggested outline for a K/K interoperability guide, and topics that should and should not be emphasized in the guide. First, there were several definitions for interoperability. The KIT/KITIA in the 1982 Public Report defined interoperability as "The ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion." A broader definition is given by Trieber in The Journal of Systems and Software 2, 1981. This definition covers the interoperability problems that TI experienced when rehosting the AIM. Trieber's definition is in two parts. First, his definition of interoperability is "Systems are deemed interoperable when they are compatible and capable of mutually utilizing the information exchanged". His definition of compatibility is "Systems are deemed

compatible when they have the technical capability of exchanging information". So compatibility is the technical capability of exchanging information. Interoperability takes it one step further; you have to be able to use the information once it has been exchanged.

There are several categories of interoperability. They are:

- Inter-APSE - This refers to the transfer of data between APSEs. Here we are concerned with transmission media, data rates, and message structure.
- Inter-tool - This deals with communication between tools. It may be Ada programs that run in parallel or sequentially or programs that are implemented in various languages. An actual example of an inter-tool interoperability problem is writing a program that creates a data file using one Ada compiler but not being able to read the data file in with a program written using a different compiler that resides on the same computer.
- Intra-tool - This deals with the exchange of information between components within a tool. It may be communication between tasks, packages, procedures, and functions.

Here is a brief summary of the proposed outline.

Chapter 1 Introduction

This chapter covers the following topics:

- Purpose of guide
- Definitions of interoperability
- Scope of guide (Emphasis on inter-APSE interoperability using magnetic tape and data link transfers.)
- Specific uses
- Causes of interoperability problems
 - Differences in computers, operating systems, compilers (Word length, two's vs. ones's complement, representation of floating point, structure of files, etc.)
 - Differences related to FTPs (Handling of control characters, flow control problems, etc.)
 - Possible differences in DIANA files
 - Other

Chapter 2 Existing Tools and Techniques for Transferring Data Across Computers

This chapter will present a summary of the "good" existing magnetic tape and data link tools. It will also include other tools that are useful when doing data transfers, such as TI's tool PAGER, which is used to aggregate and disaggregate Ada programs. An example of a magnetic tape tool is TAPESEND. Examples of data link tools are KERMIT, TCP/FTP, and UUCP. Future data link tools include FTAM (OSI) and a WIS FTP.

Chapter 3 Development of CAIS Interoperability Tools

This chapter identifies specific interoperability tools that need to be developed. There is a need for both a magnetic tape and data link tools to allow the CAIS node model to be transferred across a number of APSEs. Such a tool requires export software that does a code conversion from the code of the first host computer to a canonical code (ASCII) and decomposes the node model into flat files. These files contain attributes, relationships, pointers, data, and everything that is needed to reconstruct the node model on the second host machine. Then import software will convert the canonical code to the code of the target computer and reconstruct the node model.

Chapter 4 Interoperability Problems Encountered By Users

This chapter reports Ada-related interoperability encountered by users. It may include problems encountered by TI in their AIM rehost effort, other Ada rehost efforts, future CAIS rehost efforts, and other miscellaneous problems reported by users.

Chapter 5 Solutions and Guidelines for Problems

Solutions and guidelines to the problems reported in Chapter 4 will be given.

APPENDIX A Related data transfer tools

APPENDIX B Standards related to data interchange

APPENDIX C References

APPENDIX D Glossary

Emphasis of the guide will be on inter-APSE interoperability. Topics not emphasized are language interoperability problems (implementations using multiple languages where interoperability problems are experienced such as transferring arrays between FORTRAN and Ada programs) and other non-CAIS but Ada interoperability problems. Several KIT/KITIA members thought that it was still important to include the unemphasized topics in the guide.

10. WIS PRESENTATION

- A short status report on WIS was presented. In the short term, WIS is looking at existing tools over the next four years. In the long term, WIS is looking at specific WIS nodes (Software Engineering Environment instances) and the underlying environment to support this SEE. There are three environment areas: databases, transaction processing and Ada. The RAC was adopted as a starting point, but another couple of months of WIS refinement of the RAC should be done.

11. FORMAL SEMANTICS PRESENTATION

- Roy Freedman reported on some efforts on Formal Ada Semantics by EEC contractors. DDC and CRAI's goals are to develop an Ada Formal Definition (FD), and Ada tools, with mappings to Ada/Ed, and to automate ACVC with the Ada FD. The objectives are: concise Ada references, support for proof systems, support for interpreters/compilers, support for generation and verification of test programs, support for derivation of informal but precise reference manuals, support for future standardization efforts, and unification of other approaches to specifying Ada. The approach uses static semantics and dynamic semantics descriptions, splitting the dynamic into sequential and parallel. They are using VDL for static semantics and augmented VDL (similar to algebraic approach - ASL) for dynamic sequential. Karlsruhe uses an attributed grammar. Ada FD tools will be deliverable tools usable in an APSE. They have a good model to apply to CAIS FD. The objectives for the CAIS FD are similar to those mentioned above for Ada FD.

12. WRAP UP

13. MEETING ADJOURNED

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
8-11 July 1985

KIT Attendees:

CHADWICK, Kevin	Canadian National Defense HQ
FITCH, Geoff	Intermetrics
FOREMAN, John	Texas Instruments
FRENCH, Stewart	Texas Instruments
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
HOUSE, Ron	NOSC
JOHNSTON, Larry	NADC
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
LAKE, Mike	IDA
MUMM, Hans	NOSC
MUNCK, Bob	NOSC
MYERS, Gil	NOSC
MYERS, Philip	NAVELEX
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA-DN
POGGE, Dave	NWS
ROBERTSON, George	FCDSSA-SD
ROBY, Clyde	IDA
STILES, Lloyd	FCDSSA-SD
TAYLOR, Guy	FCDSSA-VA

KITIA Attendees:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnell Douglas Astronautics
DERIUGIN, Tanya	Boeing Aerospace Co.
DRAKE, Dick	IBM
FAINTER, Bob	VPI
FISCHER, Herman	Litton Data Sytsems
FREEDMAN, Roy	Hazeltine Corp.
GARGARO, Anthony	CSC
GLASEMAN, Steve	Aerospace Corp.
HARNEY, Terry	Hughes Aircraft
HORTON, Michael	System Development Corp
LAHTINEN, Pekka	Oy Softplan AB
LAMB, J. Eli	Bell Labs
LeGRAND, Sue	Ford Aerospace
LINDQUIST, Tim	Virginia Institute of Technology
LYONS, Tim	Software Sciences Ltd.
McGONAGLE, Dave	GE
MORSE, H. R.	Frey Federal Systems
PEET, Dianna	CDC
PLOEDEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
ROUBINE, Olivier	Informatique Internationale
RUDMIK, Andres	GTE
RUDOLPH, Bruce	Norden Systems
STEIN, Larry	Aerospace Corp.
wILLMAN, Herb	Raytheon Company
WREGG, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communications Corp.

VISITORS

ELLISON, Bob	SEI
JONES, William	NASA - AMES
LAW, Don	Gould
McKEE, Gary	Martin Marietta
VILOT, Michael	General System GP

KIT/KITIA MINUTES
MEETING OF 10-12 SEPTEMBER 1985
SARATOGA SPRINGS, NEW YORK

ATTENDEES: SEE APPENDIX A

MEETING HANDOUTS: SEE APPENDIX B

10 SEPTEMBER 1985

1. OPENING REMARKS

- Tricia Oberndorf, KIT Chairperson, brought the meeting to order.
- New replacement members and visitors were introduced. Ray Szymanski, the Evaluation & Validation Team Leader, is replacing Jinny Castor from Wright-Patterson Air Force Base. Dr. Carl Schmiedekamp of Naval Air Development Center is performing as the STARS Software Engineering Environment team liaison contact. Matt Emerson is now the primary representative for Naval Avionics Center. Dave Pogge is the primary representative for the Naval Weapons Center. Dr. Roy Freedman has accepted a position at New York Polytechnic Institute. Chuck Weinstock is representing the Software Engineering Institute. Visiting members of the Ada Run-Time Environment Working Group in attendance included Chuck Arnold, Daryl Winters and Keith Pratt.

2. GENERAL BUSINESS

- LCDR Phillip Myers has been officially designated as the Navy Deputy for the Ada Joint Program office. Phillip has responsibility for the KIT and the E&V Team activities. Jinny Castor is now the acting Director of the Ada Joint Program Office.
- Hans Mumm will be acting for Tricia Oberndorf during her maternity absence starting in December 1 her return (IT'S A BOY!). Tricia will still periodically monitor ARPANET traffic, but Hans will be the KIT focal point during this period.
- Texas Instruments has completed their deliverables for the Ada Interactive Monitor (AIM) contract. The KIT support contract has not yet been awarded. The initial Delivery Order has been written including preparation of the Public Reports. This contract will also insure paid contractor support to all the working groups. The CAIS Version 2 contract is in technical review in response to the Best and Final, and the winner should be on board by the January meeting.
- The CAIS standardization process is continuing. Burt Newlin, an old hand at the standardization process, is on board at the AJPO to support this effort. It is hoped the charter for the CAIS Control

Board will accompany the package to identify this organization as the focal point for CAIS changes. The AJPO has 600 copies of CAIS Version 1 for distribution in support of this process. In addition to the three services, this review will solicit consolidated comments from professional activities such as the IEEE, NSIA, SIGAda, etc.. A Public Review will occur prior to a revision of the CAIS, which will be controlled by the CAIS Control Board. The two-page policy letter included in the initial distribution of CAIS Version 1 will not be included in the document during the standardization review to keep the technical discussions separate from policy discussions. There has been no activity to change the AJPO policy announced at Hyannis (formerly included in the CAIS document). DoD policy is promulgated via instructions such as 5000.29/5000.31 which are presently being considered for revision.

- Ray Szymanski and the E&V Team now has the Analytical Sciences Corp. on board working on a classification schema for APSE's. A draft is due to the E&V Team in October for review and subsequent distribution. The next E&V Team meeting is scheduled for Melbourne, Florida. The compiler evaluation procurement is in processing for release for bids.
- Rudy Krutar announced Intermetrics is performing as the DIANA maintenance support contractor. An IDL primer is now available.
- Mike Kamrad presented a status of the ARTEWG, which was formed to address run-time environment and performance issues. The ARTEWG has a plan of action and charter available. Their task is to collect run-time issues into two categories: requirements and implementations. The ARTEWG is organized into three sub-groups: implementation dependencies sub-group, applications sub-group and interface sub-group. They plan to have the next general membership meeting at the SIGAda in Boston. An ARTEWG-INFORMATION account will be coming on line on the ARPANET on ECLB. So far, the ARTEWG has identified 186 implementation dependencies through an analysis of the Ada Reference Manual and expects a possible total of 300 issues to be identified. The ARTEWG is working on a run-time dependencies handbook which may in fact become part of a transportability handbook. Inputs to the ARTEWG effort are welcomed and may be submitted to KAMRAD@HI-MULTICS.

3. NAMED WORKING GROUP REPORTS

- Jack Kramer reported progress on the CAIS Rationale document. Additional work was completed on the previously submitted CAIS comments. The answers developed were done so in the context of the current CAIS Version 1 and will soon be available on the ARPANET under the KIT-INFORMATION account.
- Hal Hart indicated the 12 July version of the Requirements and Criteria document is available in hard copy and on the ARPANET. This version includes a format for submission of comments. Draft rationales for RAC sections 4, 5, and 6 will be worked on at this meeting.

- Ron Johnson reported the GACWG is completing its work on the Transportability Guide.
- Ann Reedy reported the STONEWG is working on the integration of their recently produced material for presentation at the January meeting. A more descriptive presentation is scheduled for Thursday.
- Bernie Abrams reported the COMPWG is reviewing examples of where additional semantics may be required in the CAIS. Tim Lindquist has almost completed his operational analysis of the CAIS. A strong objection was raised in the apparent halting of the denotational and axiomatic analyses of the CAIS. The COMPWG will revisit this topic to examine what resources could be applied to this effort. The traceability of the RAC to the CAIS will continue with an examination of some automated support to avoid the giant matrix previously produced.

4. ANNOUNCEMENTS

- The RAC will be publically distributed at the November SIGAda meeting.
- MITRE has successfully installed software tools on their CAIS prototype.
- The Ada Language System/Navy development contract has been awarded to the CDC/TRW/SYSCON team.
- The meeting schedule is as follows:

13-16 January	1986	San Diego
14-17 April		Atlanta (CDC)
7-10 July		San Francisco (FACC)
22-26 September		Minneapolis (Honeywell)
19-22 January	1987	San Diego

5. MORNING BREAK

6. NAMED WORKING GROUP MEETINGS

7. LUNCH BREAK

8. NUMBERED WORKING GROUP MEETINGS

WEDNESDAY 11 SEPTEMBER 1985

9. PORTABLE COMMON TOOL ENVIRONMENT (PCTE)

- Herm Fischer gave a detailed presentation on the European PCTE project. The PCTE is both a set of interfaces written in C for UNIX and also a prototype implementation of these interfaces. The project

is sponsored by the European Economic Community ESPRIT program. The Portable Ada Prototype System (PAPS) was also presented with a contrast to PCTE. The PCTE is directed toward compatability with existing UNIX (binary compatability) while PAPS is targeting to Ada. The PCTE is strongly tied to UNIX whereas PAPS is not. PCTE is also multi-lingual whereas PAPS is Ada only.

10. MORNING BREAK

11. SEPARATE KIT AND KITIA MEETINGS

- Phillip Myers reported that Jinny Castor is working hard on the AJPO budget for 1986. AJPO is also planning a Tri-Service review for 4 October. Phillip is the point of contact for KIT, CAIS, and the Evaluation and Validation Team activities. Some of the additional new personnel at the AJPO include Burt Newlin, who will be supporting the CAIS Standardization process and Tom Quindry.
- The Army has committed to provide a person to support STARS and another to support the AJPO. They have not as yet been identified. The former NAVMAT OBY responsibilities for Navy software have basically transferred to OPNAV 945C. Stan Greenblatt will basically have the responsibility for Navy software policy.
- The DOD-STD-2167 will have a modification that will address Ada in 1987. The original "DeLauer Memo" will be re-issued by DoD.
- A discussion of the PCTE presentation/activities resulted in a desire for continuing liaison for future CAIS implications such as inclusion of a schema in future versions. Additional discussion included the possibility of a DoD Standard Operating System (Rich Thall), influence of Rational Machines hardware technology (Bob Munck), role and relationship of the CAISWG to the CAIS Version 2 design (Tricia), and the importance of prototyping for the Version 2 design phase (Tricia).

12. LUNCH BREAK

13. NAMED WORKING GROUP MEETINGS

THURSDAY, 12 SEPTEMBER

14. NAMED WORKING GROUP PRESENTATIONS

- RACWG - Hal Hart presented a list of 12 proposed changes to the current RAC. Discussion on the first item required two hours of discussion regarding security requirements. The remaining items were to be addressed via ARPANET discussion. Hal will present the RAC for review at the November SIGAda meeting in Boston.

15. MORNING BREAK

- STONEWG - Ann Reedy presented the current direction of the STONEWG as formulating a context for software engineering environments. This not only includes the process (methodology, etc.) but also the people and the products. It is from this framework that the STONEWG expects to derive both explicit and implicit requirements for interfaces. Phillip Myers raised the point that this activity seems more appropriate for STARS than the KIT/KITIA and suggested STONEWG prepare a position paper describing the relation of this concept definition to the future CAIS work.

16. LUNCH BREAK

17. KIT/KITIA WRAP-UP SESSION

- The CAISWG will continue working on responses to the CAIS comments and expects to have them available on the ARPANET by the next KIT/KITIA meeting.
- The STONEWG will prepare a position paper giving a synopsis of their rationale for requirements definition via articulation of the environment description. They are planning an interim meeting before the next KIT/KITIA meeting and expect to have a paper for the Public Report.
- The GACWG is completing their work on the Transportability Guide and will begin drafting an Interoperability Guide.
- The COMPWG will pursue formal semantic definition alternatives as well as the traceability analysis, quality assurance guidelines, and testing methodologies.
- Hans Mumm requested that any new definitions that are identified should be forwarded to DEFWG for inclusion in the KIT/KITIA Glossary.

18. NAMED AND NUMBERED WORKING GROUP MEETINGS

19. MEETING ADJOURNED

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
10-12 September 1985

KIT Attendees:

BELZ, Frank	TRW
EMERSON, Matt	NAC
FERGUSON, Jay	Department of Defense
FOIDL, Jack	TRW
FRENCH, Stewart	Texas Instruments
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
HOUSE, Ron	NOSC
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
MAGLIERI, Lucas	National Defense Hq., Canada
MUMM, Hans	NOSC
MUNCK, Bob	NOSC
MYERS, Gil	NOSC
MYERS, Philip	NAVELEX
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA-DN
POGGE, Dave	NWS
SCHMIEDEKAMP, Carl	NADC
STILES, Lloyd	FCDSSA-SD
SZYMANSKI, Raymond	AFWAL/AAAF-2
TAYLOR, Guy	FCDSSA-VA
THALL, Rich	SofTech

KITIA Attendees:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnell Douglas
DRAKE, Dick	IBM
FISCHER, Herman	Litton Data Sytsems
FREEDMAN, Roy	Hazeltine Corp.
GARGARO, Anthony	CSC
HARNEY, Terry	Hughes Aircraft
HORTON, Michael	System Development Corp
JOHNSON, Ron	Boeing Company
KAMRAD, Mike	Honeywell
LeGRAND, Sue	Ford Aerospace
LYONS, Tim	Software Sciences Ltd.
MARTIN, Ed	Lockheed Missles and Space
McGONAGLE, Dave	GE
PEET, Dianna	CDC
PLOEDEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
ROUBINE, Olivier	Informatique Internationale
RUDMIK Andres	GTE
RUDOLPH, Bruce	Norden Systems
STEIN, Larry	Aerospace Corp.
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.

VISITORS IN ATTENDANCE

ARNOLD, Charles	Texas Instruments
CHLUDZINSKI, John	Institute fo Defense Analyses
WEINSTOCK, Chuck	Software Engineering Institute

APPENDIX B
MEETING HANDOUTS
10-12 September 1985

1. DoD Requirements and Design Criteria for the Common APSE Interface Set (CAIS), KAPSE Interface Team (KIT) and the KIT-Industry-Academia (KITIA) for the Ada Joint Program Office, 13 September 1985.

MINUTES OF MEETING

COMPWG

KIT/KITIA

SEPTEMBER 9-12, 1985

SARATOGA SPRINGS, NY

ATTENDANCE

Bernie Abrams - Chairman
John Chludzinski
Dick Drake
Jack Foidl
Lloyd Styles
Ray Szymansky E & V
Guy Taylor

PRESENTATION

A presentation prepared by B. Abrams on the Specification of CAIS semantics was reviewed. The presentation showed specific examples of areas of the CAIS where some form of supplementary semantics is needed. Because of scheduling conflicts the presentation was given to KIT/KITIA in abbreviated form without a projector. A copy of the presentation is enclosed.

COORDINATION WITH E & V

The chairman of the E & V team, Ray Szymansky, attended our meeting. We plan to continue communication and cooperation between COMPWG and the E & V team. The E & V team is working on CAIS validation. They are not currently working on CAIS evaluation.

SEMANTICS

Operational semantics has been shown to be useful for CAIS by the work of T. Lindquist. We recommend that this work continue. None of the COMPWG members who had worked on denotational or axiomatic semantics were present. There was some doubt about the applicability of these methods to CAIS. J. Foidl will look into the possibility of getting support from a graduate student interested in denotational and axiomatic semantics.

PLANS FOR NEXT LINE

In the quarter between now and the January meeting members will work in the following activities. The results will be reported at the next meeting as an informal paper or presentation.

Traceability Matrix (RAC to CAIS)	J. Foidl,
G. Taylor	

Test Methods for Large Software Products like CAIS	B. Abrams

Testability of RAC	R. Drake

Quality Assurance Guidelines	L. Styles

Software Metrics	J. Chludzinski

In addition, the following activities were suggested for members who were not present or whose status is changing.

The application of Denotational and Axiomatic semantics to CAIS.

R. Freedman
L. Yelowitz

The application of Operational semantics to CAIS.

R. Fainter
T. Lindquist

RACWG

COMPWG met with RACWG to discuss plans for a matrix relating requirements to CAIS features.

Prepared by:

B. Abrams

①

CAIS

SEMANTICS

Why we need a method of
Specifying Semantics

Bernard Abrams
CompWg
Sept 5, 1985

(2)

The CAIS Semantics must be un-ambiguously specified in order to verify compliance.

COMPWG has looked at supplementary specification methods

- = Axiomatic
- = Denotational
- = Operational
- = Example

③

This presentation will show
where supplemantary specification
is needed.

The purpose is to give us a basis
for evaluation of specification
methods.

This is not a critique of MIL STD
CAIS

③

Excuse me for being
a nit picker - but
that is my job.



④

Reference:

Proposed MIL-STD-CAIS
31 January 1985

CAIS FUNCTION

procedure OPEN (Node :
Name :
Intent :
Time-Limit)

[5.1.2.1 Pg 51]

Supplementary Semantics Needed to Clarify:

- Are all combinations of INTENT possible?
- If Intent is (READ, EXCLUSIVE_READ_CONTENTS) is read of CONTENTS exclusive?

CAIS FUNCTION

procedure CLOSE (NODE: ...);

[5.1.2.2]

Pg 53

Supplementary Semantics needed:

NONE

CAIS FUNCTION

procedure CHANGE-INTENT (NODE
INTENT
TIME-LIMIT);

Supplementary Semantics Needed

Same as OPEN

⑧

CAIS FUNCTION

```
function Primary_Name (Node.....)  
    return Name-String;
```

[5.12.7 pg 57]

Supplementary Specification needed:

Is any legal format of the
Primary Name acceptable?

{ DOT (AB) }
{ . AB }

{ ' CHILD (ONLY ONE) }
{ ' CHILD }

Section 3

KIT/KITIA DOCUMENTATION

AIM

Introduction of Players

John Foreman

Program Manager

Stewart French

Lead Engineer
System Dependent Packages
Virtual Terminal
Rehost
Transportability Guide

Tim Harrison

CAIS Rationale

Texas Instruments

presents

The APSE Interactive Monitor
AIM

NOSC contract number N6601-82-C-0440

Presentation goal

This presentation is being made to inform the KIT/KITIA of the experiences gained while specifying, designing, implementing, and rehosting a tool written in Ada

Hopefully, it will also stimulate further discussions of:

transportability and interoperability

Lifecycle phases

CAIS interfaces

AIM

Introduction of Players (con't)

Jerry Baskette

Help Subsystem
Formal Testing
Rehost

Mark Borger

AIM Command Interpreter
Window Manager
Viewport Manager
Image Manager
Pad Manager
Primary Engineer on Integration
Rehost
Support Packages

Tom Duke

Program Manager
Info Subsystem
Error Handling
Rehost

Jim Rea

Document update

APSE Interactive Monitor

Agenda

8:00am Introduction John Forman

8:15am Lifecycle Analysis John Foreman

9:00am Experiences Stewart French

10:00am Rehost Stewart French

10:30am CAIS Related Issues Tim Harrison

11:00am Conclusions/Discussion Stewart French

APSE Interactive Monitor

Description

The AIM is a tool that acts as an interface between the user of an APSE (at their computer terminal) and APSE processes

The AIM

- Coordinates the input and outputs from APSE processes
- Provides a method for creating and manipulating new APSE processes
- Provides a device independent computer terminal interface

APSE Interactive Monitor

Project Goals/Objectives

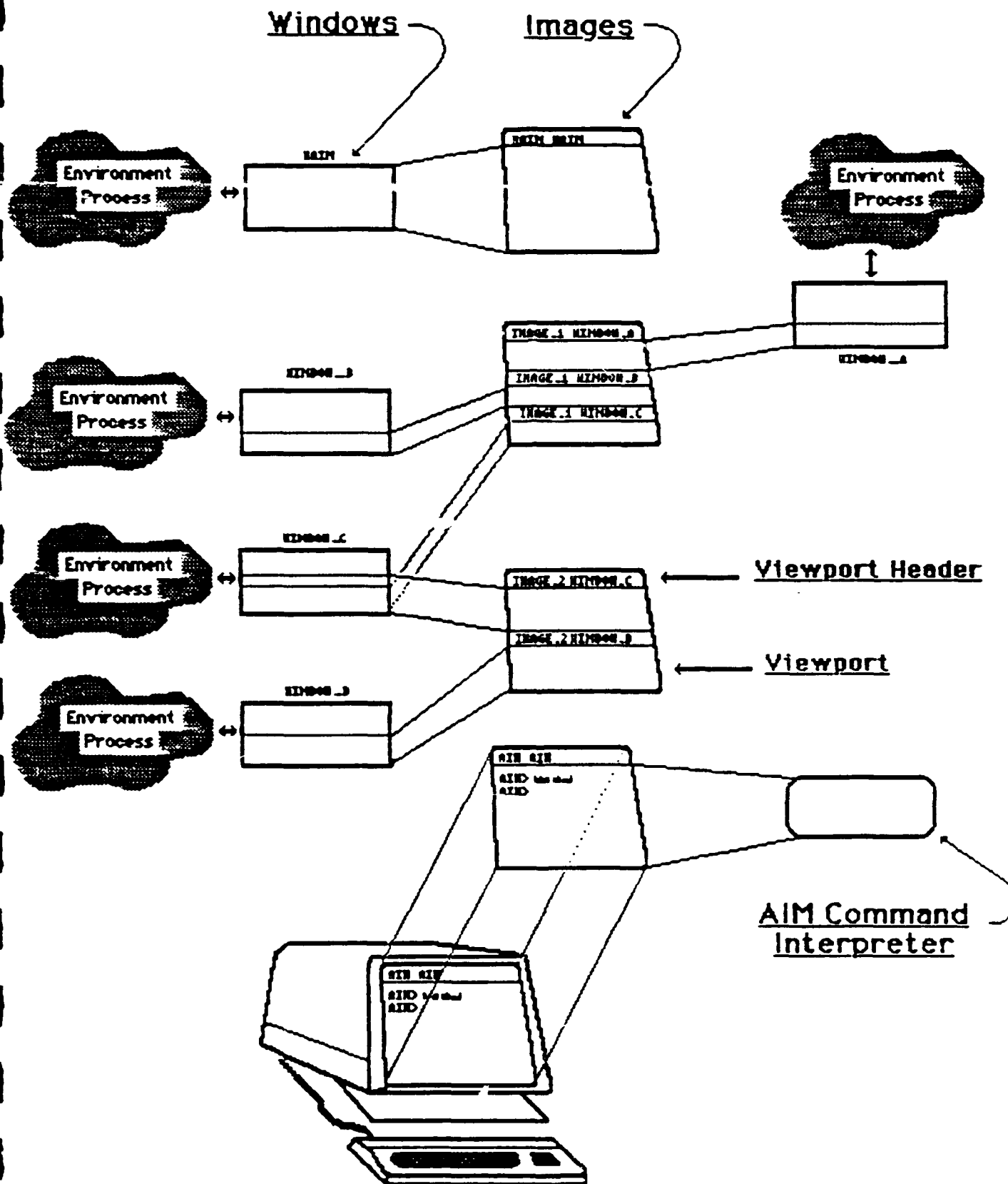
Assist the KIT in ferreting out APSE interface issues advise as to ALS/AIE discrepancies advise as to missing items what makes interfacing easy/hard

Produce a useful, operational, contributing tool to present APSEs

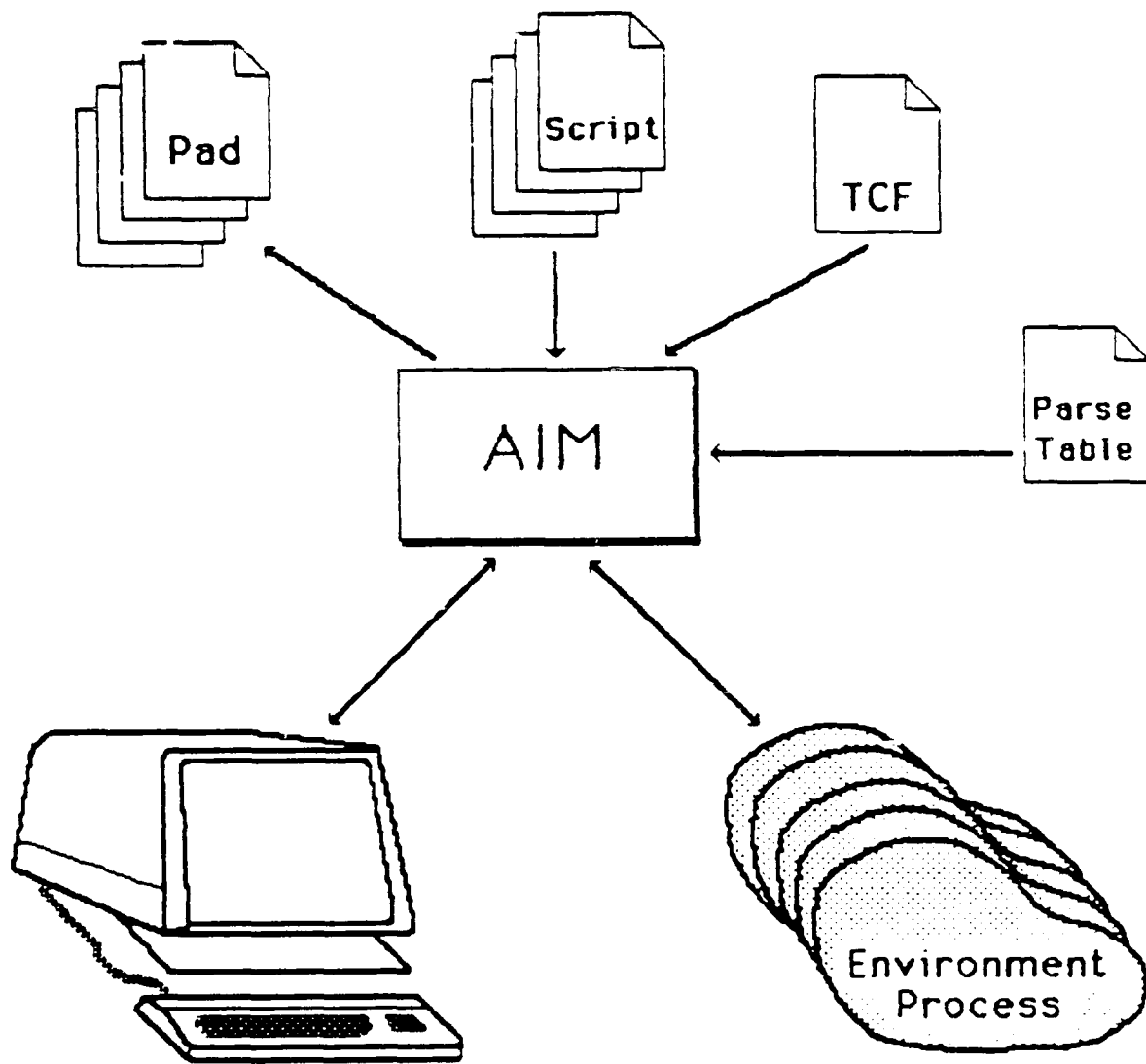
Portability/Extensibility of future APSEs

October-1982 through July-1985

AIM Definitions and Example



AIM Environment Interfaces



AIM Statistics

22,000 Lines of Code

Reusable Software Components

virtual Terminal
Help package
System dependencies
LALR Parser Support
General Support Packages (queue, stack)

Separately compiled modules

240 compilation units

Compiler/Runtime

Item	Data General AOS/VS	Dec VAX/VMS
Minimum number of tasks	14 Ada, 5 AOS/VS	16 Ada tasks
tasks per window	6 Ada, 1 AOS/VS	8 Ada Tasks
Max Number of Processes	27	small
Executable image size	625 KBytes	294 KBytes (Not Optimized)
Special Requirements	SuperUser to suspend/ resume process	expanded byte_count and subprocess_quota

AIM KAPSE Interface Requirements

Terminal Control and Communication

- o Open/Close Terminal (binary mode)
- o Echo Control
- o Read/Write character and/or strings

Process Control and Communication

- o Create/Delete process
- o Suspend/Resume Process Execution
- o Inter-process communication

Database

- o Open/Close Text File
- o Create/Delete Text File
- o Read/Write from/to Text File

Lifecycle Analysis

Introduction

Tasks

Deliverables

Testing Methodology

Lines-Of-Code

Model Comparisons

Conclusions

Lifecycle Analysis

Project Overview

Efforts Tracked for the AIM

Requirements Definition

Financial Accounting

System Specification

Preliminary Design

User Manual

Detailed Design

Hardware/Software Problems

Implementation

New Hire (Training)

Integration

Rehost

Quality Assurance

Testing

CAIS

GAC

Data Management

Interface Reports

Software Tools

Configuration Management

Program Management

Lifecycle Analysis

AIM Project Efforts

AIM Software Development Efforts

Typical

Atypical

Group 1

Requirements Definition
System Specification
Preliminary Design
Detailed Design
Test Plans And Procedures
Preliminary User's Manual
Implementation
Integration
Formal Testing
Documentation Updates

Interface Reports (3)
CAIS
GAC
Rehost

Group 2

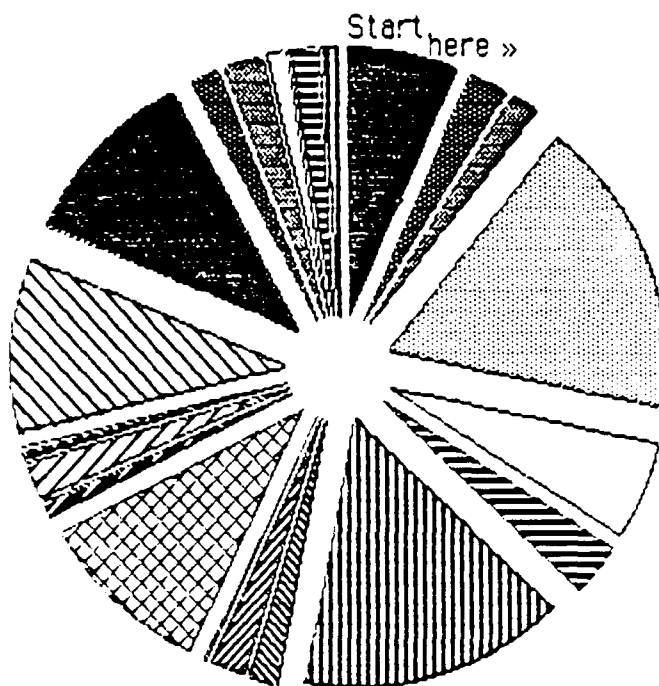
Configuration Management
Quality Assurance
Program Management
Data Management

Group 3

New Hire
Software Tools
Hardware/Software Problems
Financial Accounting

Lifecycle Analysis

AIM Project Total Efforts



- Requirements Definition (6.8%)
- System Specification (2.2%)
- Preliminary Design (1.5%)
- Detailed Design (17.9%)
- Test Plans/Procedures (5.7%)
- Preliminary User's Manual (3.3%)
- Implementation (15.8%)
- Integration (1.5%)
- Formal Testing (2.0%)
- Documentation Updates (.3%)
- Program Management (10.3%)
- Quality Assurance (.8%)
- Configuration Management (2.1%)
- Data Management (.5%)
- Interface Reports (11.0%)
- CAIS (10.7%)
- GAC (1.7%)
- Rehost (2.2%)
- New Hires (1%)
- Software Tools (1.3%)
- Hardware/Software Problems (1.3%)
- Accounting (.7%)

Lifecycle Analysis

Error Correction Bug Counts Types

AIM Bug Count And Type

	INFO	HELP	CLI

Format Error

1st Pass	14	3	25
2nd Pass	4	0	6
3rd Pass	0	0	0
4th Pass	0	0	0
Rehost	0	0	0
Re-Rehost	0	0	0

Logic Error

1st Pass	0	0	2
2nd Pass	0	0	2
3rd Pass	0	2	2
4th Pass	0	0	1
Rehost	0	0	2
Re-Rehost	0	0	0

Lifecycle Analysis

Testing Methodology

Documentation

Acceptance Test Plan
Acceptance Test Procedures
Computer Program Test Specification
System/Integration Test Plan
System/Integration Test Procedures

Testing

Unit Testing
Integration Testing
Rehost Testing

Acceptance Testing Will Be Performed At the end of July

Lifecycle Analysis

NOSC Tools LOC

	Virtual Terminal	Spell Checker	Style Checker	Batch/Forms Generator
Source	2421	2743	3189	2869
LOC/MD	11.4	14.5	17.3	16.7
LOC/MM	246.0	311.0	373.0	359.5
Source & Comnts	3011	4848	4681	4576
LOC/MM	14.2	25.6	25.5	26.7
LOC/MM	306.0	549.7	547.5	573.4
Total Lns	6300	7576	7880	8307
LOC/MD	29.8	40.0	42.9	48.4
LOC/MM	640.2	859.0	921.6	1041.0

Causes for the Higher Figures

Ada Experience Attributable to the AIM

Increased Awareness of Ada

Minimal Tasking

Small Scope

Little Documentation Required

The NOSC Tools Were Implemented On the DG System Three
Months

After the AIM Implementation Had Begun Thus Allowing the
NOSC

Engineers To Draw From the AIM Experience

Reusable Software Imported to Spell Checker & Style Checker
(25%)

Lifecycle Analysis

Lines Of Code

7384 Source		11190 Source & Comments		21059 Total Lines	
Day	Month	Day	Month	Day	Month
Grp 1 Only	55 120.0	8.4 181.8		15.8 342.2	
Grp 1 & 2	4.5 96.6	6.8 146.5		12.7 275.6	

Low Figure for the AIM

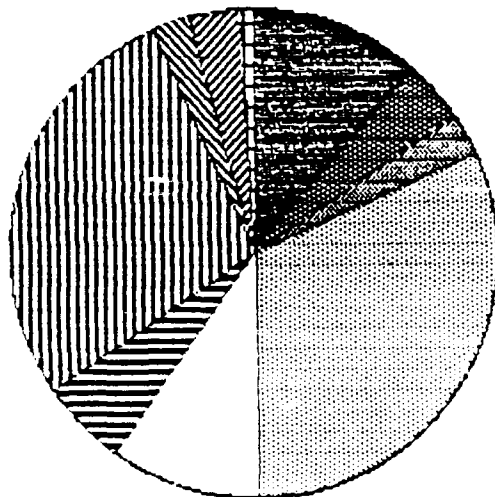
Lack of Experience by all Persons Participating on the AIM
Project

The Degree of Complexity of the AIM

Degree of Tasking

Lifecycle Analysis

Typical Project Effort



- Requirements Definition (11.9%)
- ▣ System Specification (3.9%)
- ▤ Preliminary Design (2.7%)
- ▥ Detailed Design (31.4%)
- Test Plans/Procedures (9.9%)
- ▧ Preliminary User's Manual (5.6%)
- ▨ Implementation (27.7%)
- ▩ Integration (2.6%)
- Formal Testing (3.5%)
- Documentation Updates (6%)

Lifecycle Analysis

Model Comparisons

Lifecycle Models

40-20-40 Model

Brooks Model

GTE Model

Costing Models

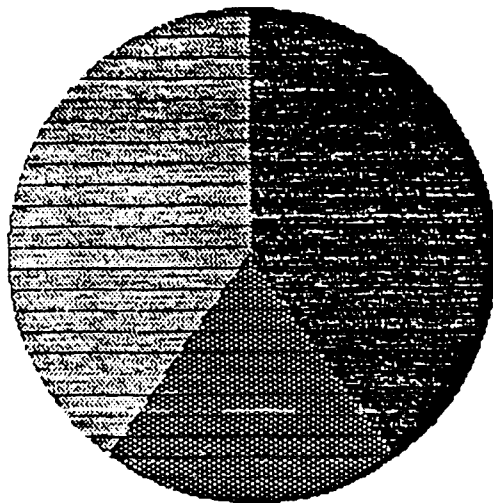
SoftCost Model

Price-S Model

COCOMO Model

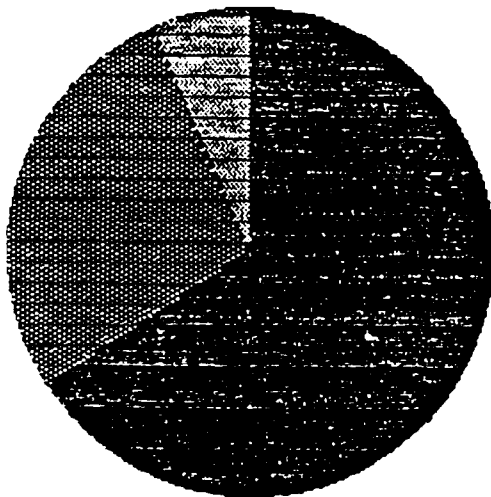
Lifecycle Analysis

40-20-40 Model



- Design Phase (40%)
- Coding Phase (20%)
- Testing Phase (40%)

AIM



- Design Phase (65.7%)
- Coding Phase (27.7%)
- Testing Phase (6.6%)

Lifecycle Analysis

40-20-40 Mapping

40-20-40

AIM

Design Efforts

Requirements Definition
System Specification
Preliminary Design
Detailed Design
Test Plans And Procedures
Preliminary User'S Manual

Coding Efforts

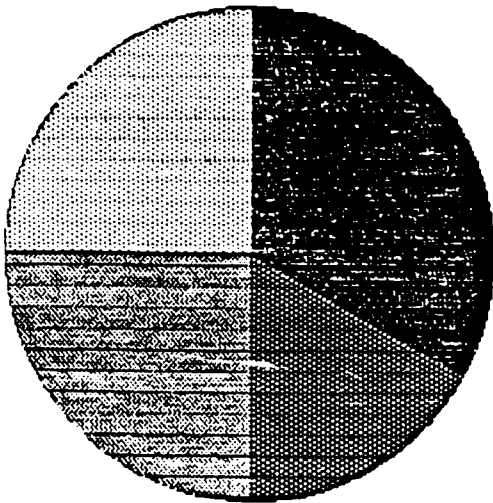
Implementation

Testing Efforts

Integration
Formal Testing
Documentation Updates

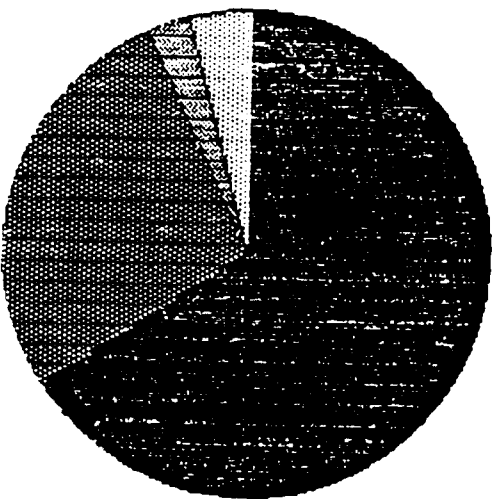
Lifecycle Analysis

Brooks Model



- Plan Phase (33%)
- ▨ Coding Phase (17%)
- ▤ Testing Phase (25%)
- ▥ Test System Phase (25%)

AIM



- Plan Phase (65.7%)
- ▨ Coding Phase (27.7%)
- ▤ Testing Phase (2.6%)
- ▥ Test System Phase (4.1%)

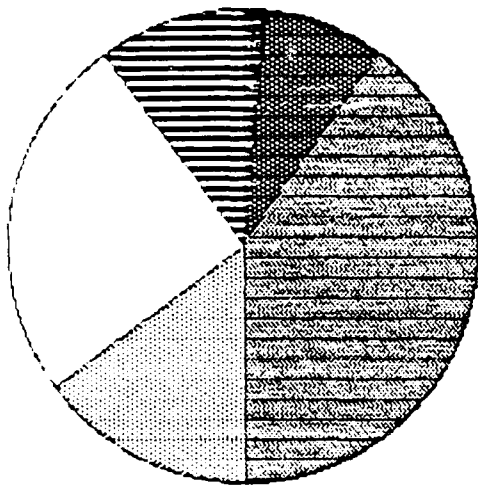
Lifecycle Analysis

Brooks Mapping

Brooks	AIM
Planning Efforts	Requirements Definition System Specification Preliminary Design Detailed Design Test Plans And Procedures Preliminary User'S Manual
Coding Efforts	Implementation
Testing Efforts	Integration
System Test Efforts	Formal Testing Documentation Updates

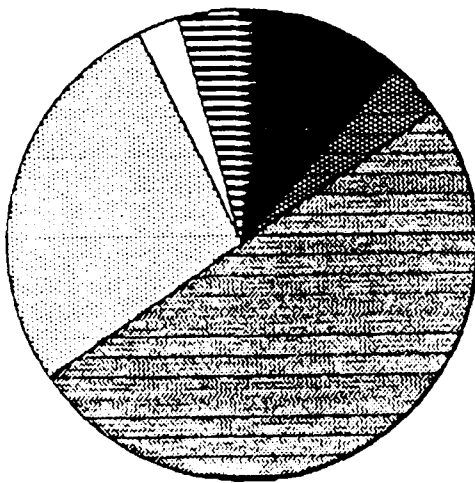
Lifecycle Analysis

GTE Model



- Plan Phase (2%)
- ▨ Specification Phase (3%)
- ▤ Design Phase (40%)
- ▧ Code Phase (15%)
- Test Phase (25%)
- ▥ System Test Phase (10%)

AIM



- Plan Phase (11.9%)
- ▨ Specification Phase (3.9%)
- ▤ Design Phase (49.9%)
- ▧ Code Phase (27.7%)
- Test Phase (2.6%)
- ▥ System Test Phase (4.1%)

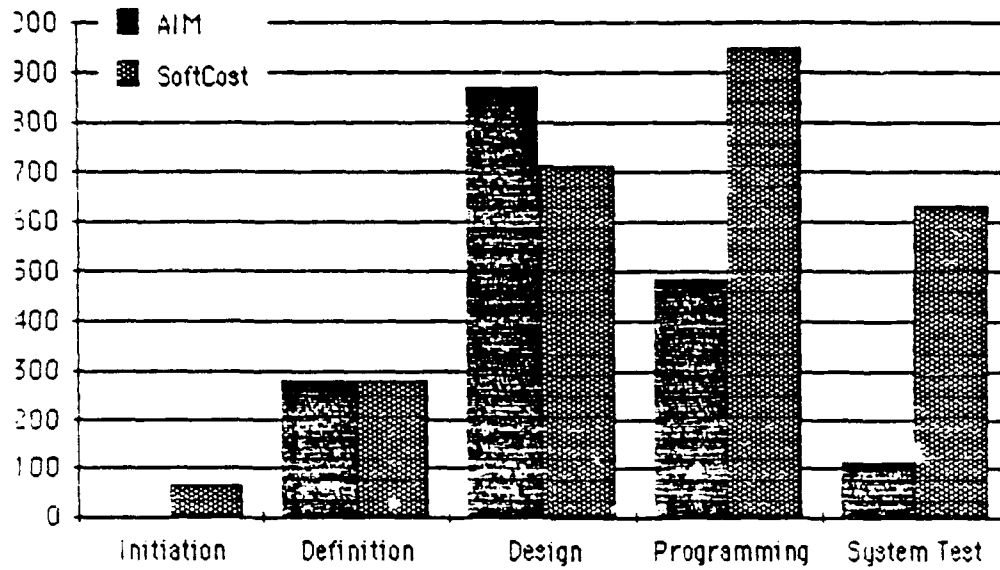
Lifecycle Analysis

GTE Mapping

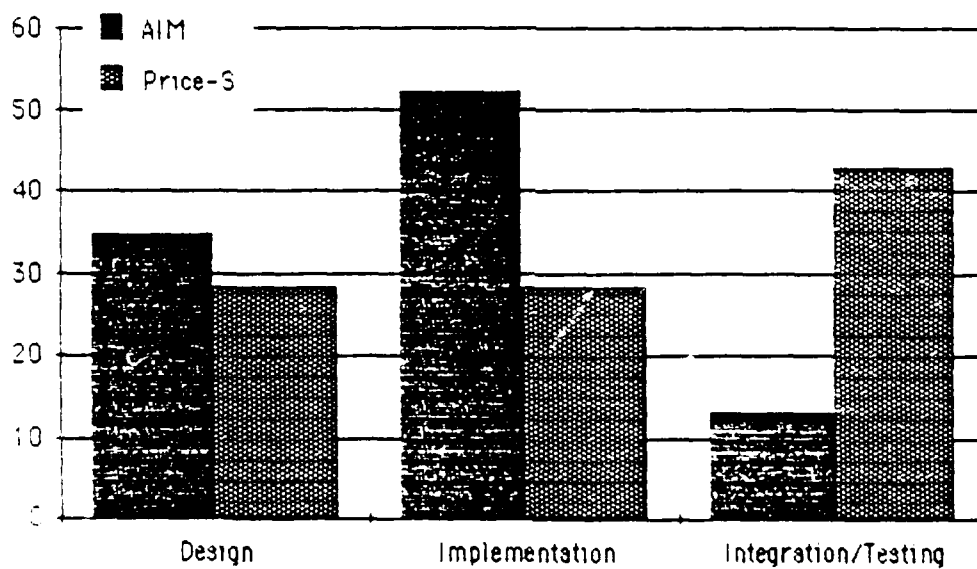
GTE	AIM
Planning Effort	Requirements Definition
Requirement Effort	System Specification
Design Effort	Preliminary Design Detailed Design Test Plans And Procedures Preliminary User'S Manual
Coding Efforts	Implementation
Testing Efforts	Integration
System Test Efforts	Formal Testing Documentation Updates

Lifecycle Analysis

Softcost



Price-S



Lifecycle Analysis

SoftCost Mapping

SoftCost	AIM
Initiation Phase	
Definition Phase	Requirements Definition System Specification
Design Phase	Preliminary Design Detailed Design Test Plans/Procedures User'S Manual Document Updates
Programming Phase	Implementation Integration
System Test Phase	Formal Testing

Price-S Mapping

Price-S	AIM
Design	Preliminary Design Detailed Design
Implementation	Implementation
Integration And Testing	Integration Formal Testing

COCOMO Mapping

COCOMO	AIM
Plans And Requirements	Requirements Definition System Specification
Design Phase	Preliminary Design Detailed Design
Programming Phase	Implementation
Testing Phase	Integration Formal Test

Lifecycle Analysis

Conclusions

Design Effort

Implementation Effort

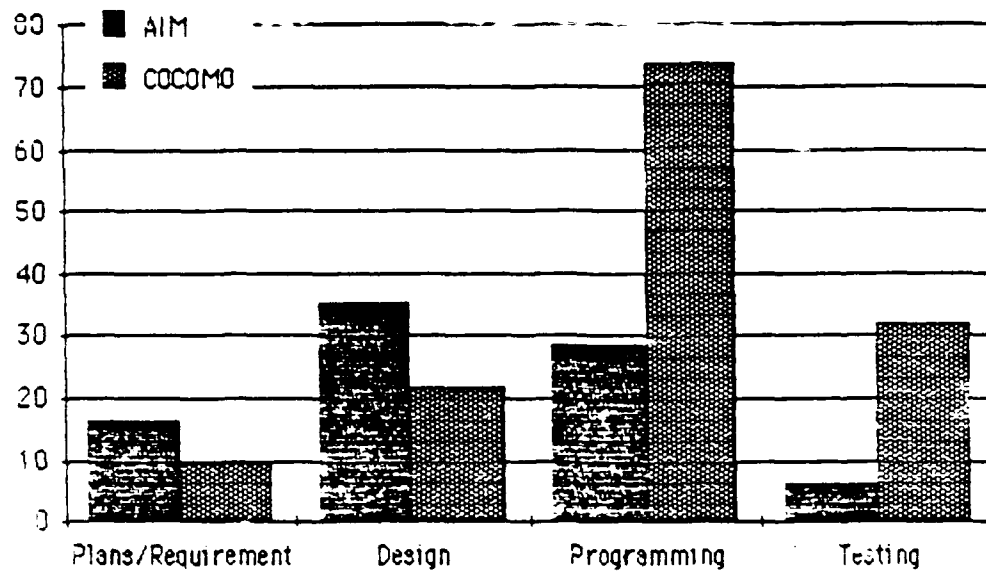
Testing Effort

LOC

Wrap-Up

Lifecycle Analysis

COCOMO



AIM Design Phase

Used Object-Oriented Design (OOD) Methodology

Easily identified major AIM objects and their associated operations (80%)

Attributes of objects created confusion
e.g.--window/pad relationship

AIM's underlying asynchronous data flow model not well defined using OOD (20%)

Augmented AIM design using conventional data flow techniques in conjunction with the concept of Ada task sets

Evolutionary process using prototypes to define final general purpose AIM data flow model

Experiences

Object-Oriented Design

Environment Comparison

DG ADE vs DEC VAX/VMS

ADE Compiler

Ada Language

Tasking
Exceptions
Strings

INFO.UTILITY

```

(INFO_STATES_ENUM)
(INFO_KEYWORD_TOKEN_ENUM)
(VALID_KEYWORD_RANGE)
(INFO_KEYWORD_RANGE)
(ALLOWABLE_TOKENS_TABLE_TYPE)
(INFO_KEYWORD_STRING_CONVERSION_ARRAY_TYPE)
(CURRENT_INFO_STATE)
(ALLOWABLE_TOKENS_TABLE)
(INFO_KEYWORD_TO_STRING)
(INFO_DISPLAY)
(NUMBER_OF_DISPLAY_LINES)
|
INITIALIZE
RETURN_TO_PREVIOUS_INFO_LEVEL
PRINT_CURRENT_PROMPT
APPEND_TO_DISPLAY
LIST_ALL_KEYSTROKE_COMMAND_NAMES
LIST_ALLOWABLE_INFO_TOKENS
COLLECT_AND_FORMAT_ALL_IMAGE_NAMES
COLLECT_AND_FORMAT_SPECIFIC_IMAGE_INFO
COLLECT_AND_FORMAT_ALL_IMAGE_INFO
COLLECT_AND_FORMAT_ALL_KEYSTROKE_NAMES
COLLECT_AND_FORMAT_SPECIFIC_KEYSTROKE_INFO
COLLECT_AND_FORMAT_ALL_KEYSTROKE_INFO
COLLECT_AND_FORMAT_ALL_TERMINAL_INFO
COLLECT_AND_FORMAT_ALL_WINDOW_NAMES
COLLECT_AND_FORMAT_SPECIFIC_WINDOW_INFO
COLLECT_AND_FORMAT_SPECIFIC_WINDOW_MODE_INFO
COLLECT_AND_FORMAT_SPECIFIC_WINDOW_PRO_INFO
COLLECT_AND_FORMAT_SPECIFIC_WINDOW_PROCESS_INFO
COLLECT_AND_FORMAT_SPECIFIC_WINDOW_ASSOCIATIONS_INFO
COLLECT_AND_FORMAT_ALL_WINDOW_INFO
COLLECT_AND_FORMAT_ALL_TOP_LEVEL_INFO
EMULATE_INFO_STATE
EMULATE_IMAGES_STATE
EMULATE_KEYSTROKE_STATE
EMULATE_TERMINAL_STATE
EMULATE_WINDOWS_STATE
EMULATE_SPECIFIC_WINDOW_STATE
IDENTIFY_INFO_KEYWORD
GET_NEXT_INFO_TOKEN
PARSE
INTERPRET
AIM_INFO
|
(INFO_KEYWORD_NOT_FOUND)
(INVALID_INFO_COMMAND)

```


INFO_UTILITY
INITIALIZE INFO_TERMINATE GET_TEXT_LINE AIRLINFO INFO_IS_TERMINATED <NOTHING_TO_OUTPUT>

Environment Comparison

The APSEs

DG ADE

The APSE is entered from the AOS/VS command language interpreter (CLI)

It extends the command set from AOS/VS

DEC ACS

The APSE is VMS

The ACS is entered from the VAX/VMS command language interpreter (DCL)

The ada command is available directly from DCL

All other ACS commands are executed from within the ACS

Environment Comparison

Data General AOS/VS
Ada Development Environment (ADE)

Digital Equipment Corporation VAX/VMS
Ada Compilation System (ACS)

Compiler
Linker
Debugger
Librarian
Configuration Management Tools
Text Editor
Electronic Mail

Example of Use (for compiler and linker)

Functional Capabilities

Integration Into the Environment

Environment Comparison

DG ADE Compilation example

> enter :user1:atb:aim

-> batch ada temp

Creating directory :UDD:BORGER:BATCH to hold batch job output files.
QUEUED, SEQ=18644, QPRI=127

-> baty

***** T1 Ada Work Center / BATCH OUTPUT FILE *****

AOS/VS 5.03 / EXEC 5.03 5-JUN-85 12:51:58

QPRI=127 SEQ=18644

INPUT FILE -- :USER1:ATB:AIM:FINAL_RELEASE:DEMO:2008.CLI.001.JOB

LIST FILE -- :QUEUE:BORGER.LIST.18644

LAST PREVIOUS LOGON 5-JUN-1985 12:41:20

AOS/VS CLI REV 05.01.00.00 5-JUN-85 12:51:59

> SEARCHLIST :USER1:ADE:MACROS,:MACROS,:UTIL,

:USER1:ATB:AIM:MACROS,:TCS

> DIRECTORY :USER1:ATB:AIM:FINAL_RELEASE:DEMO

> DEFACL SYS_MGR,OWARE,BORGER,OWARE,+,WRE

> enter/fbat/no_news/proj?=:USER1:ATB:AIM

ADE Revision 2.20.00.00 from directory :USER1:ADE

-> ada temp

Command line parsed.

Ada Compiler Rev. 02.20.00.00 6/5/85 at 12:52:11

Reading from :USER1:ATB:AIM:FINAL_RELEASE:DEMO:TEMP.ADA

5 syntax errors

Procedure body TEST_ERRORS has NOT been added to the library.

2 semantic errors

Code generation suppressed

Used 0:00:01 in 0:00:03

-> exit

Leaving the Data General/Rolm Ada Development Environment.

AOS/VS CLI TERMINATING 5-JUN-85 12:52:16

PROCESS 7 TERMINATED

ELAPSED TIME 0:00:16

(OTHER JOBS, SAME USERNAME)

USER 'BORGER' LOGGED OFF 5-JUN-85 12:52:16

* LIST FILE EMPTY, WILL NOT BE PRINTED

Environment Comparison

Compilers

ANSI standard Ada compilers

Parse the entire Ada source file;
if any syntax errors are encountered, the
compilation is terminated

Assuming no errors from step,
semantically check each compilation unit;
if any semantic errors are detected, compilation
terminates for that unit, but continues for the
remaining units

Generate machine code, in the form of relocatable
binary for each correct unit, and update the
program library accordingly

The Ada compiler can be invoked from the command
line or executed in a batch stream.

Environment Comparison

DEC ACS Compilation example

```
$ acs set librar [.lib]
```

```
$ ada temp/list temp
```

```
1 WITH nothing
```

```
.....1
%ADAC-E-INSSEMI, (1) Inserted ";" at end of line
```

```
3 TYPE mine IS RANGE ;
```

```
.....1
%ADAC-E-IGNOREUNEXP, (1) Unexpected ";" ignored
```

```
.....2
%ADAC-I-IGNOREDECL, (2) Declaration ignored due to syntactic
errors within it
```

```
6 hoo_doo();
```

```
.....1
%ADAC-E-IGNOREPARENS, (1) Empty parentheses ignored
```

```
7 END test_errors;
```

```
%ADAC-F-TERMSYNTAX, Terminating compilation due to syntax error(s)
```

```
%ADAC-F-ENDABORT, Ada compilation aborted
```

```
$ type temp.lis
```

```
5-Jun-1985 12:5
```

```
6:40 UAX Ada U1 0-7
```

```
Page 1
```

```
7:43 USER1: (FRENCH)TEMP.ADA;1
```

```
(1)
```

```
1 WITH nothing
```

```
.....1
%ADAC-E-INSSEMI, (1) Inserted ";" at end of line
```

```
2 PROCEDURE test_errors IS
```

```
3 TYPE mine IS RANGE ;
```

```
.....1.....2
%ADAC-E-IGNOREUNEXP, (2) Unexpected ";" ignored
```

```
%ADAC-I-IGNOREDECL, (1) Declaration ignored due to syntactic
errors within it
```

```
4 glitch;
```

```
5 BEGIN
```

```
6 hoo_doo();
```

```
.....1
%ADAC-E-IGNOREPARENS, (1) Empty parentheses ignored
```

```
7 END test_errors;
```

```
%ADAC-F-TERMSYNTAX, Terminating compilation due to syntax error(s)
```

-> type temp.lst

Ada 2.20.0.0 6/5/85 at 12:52:12

:USER1:ATB:AIM:FINAL_RELEASE:DEMO:TEMP ADA page 1

```
1 | WITH nothing
2 | PROCEDURE test_errors IS
```

```
***      Syntax error with input 'procedure' (Line 2, Column 1).
***      Inserting ';' immediately before 'procedure'
              (Line 2, Column 1)
```

```
3 | TYPE mine IS RANGE ;
```

```
***      Syntax error with input ';', (Line 3, Column 20)
***      Replacing ';' (Line 3, Column 20) with '+'.

```

```
4 | glitch;
```

```
***      Expression appears where range attribute is expected
```

```
5 | BEGIN
6 |   hoo_doo();
```

```
***      Empty parameter list, '()', in call
```

```
7 | END test_errors,
```

```
***      () not allowed in proc call.
==> 1 with NOTHING,
***      NOTHING denotes no unit in the library
==> 6 HOO_DOO;
***      HOO_DOO is undefined.
```

Environment Comparison

DG ADE Linker Example

-> batch adalink/debug test1
-> baty

**** T1 Ada Work Center / BATCH OUTPUT FILE ****

ROS/US 5 03 / EXEC 5.03 11-JUN-85 8:31:02
QPRI=127 SEQ=18737
INPUT FILE -- :USER1:ATB:AIM:2007.CLI.001.JOB (WILL BE DELETED AFTER
PROCESSING)LIST FILE -- :QUEUE:BORGER.LIST.18737

LAST PREVIOUS LOGON 11-JUN-1985 8:28:12

ROS/US CLI REV 05.01.00.00 11-JUN-85 8:31:04
> SEARCHLIST :USER1:ADE:MACROS, :USER1:BORGER:MACROS, :MACROS, UTIL,
SCRED, :USER1:ATB:AIM:MACROS, :TCS
> DIRECTORY :USER1:ATB:AIM
> DEFACL SYS_MGR,OWARE,BORGER,OWARE,+,WRE
>
> enter/flat/no_news/proj?=:USER1:ATB:AIM
ADE Revision 2.20.00.00 from directory :USER1:ADE
-> adalink/debug test1
Command line parsed.
Ada Loader Rev. 02.20.00.00 6/11/85 at 8:33:42
Creating PR file :USER1:ATB:AIM:TEST1

-> exit

Leaving the Data General/Rolm Ada Development Environment

ROS/US CLI TERMINATING 11-JUN-85 8 34 38

PROCESS 8 TERMINATED
ELAPSED TIME 0.03 34
(OTHER JOBS, SAME USERNAME)
USER :BORGER LOGGED OFF 11-JUN-85 8 34 38

* LIST FILE EMPTY WILL NOT BE PRINTED

- 1

Environment Comparison

Compiler Functional Capabilities

	URX/UMS	AOS/US/RDE
assembly language generation.	x	x
conditional compilation.	—	x
debug information generation.	x	x
enable and disable listing.	x	—
errors only listing.	—	x
set default directory for source.	—	x
set listing width and height.	—	x
specify different program library.	x	—
Specify main program.	—	x
disable use of SYSTEM library.	—	x
suppress all run-time checks.	x	x
Multiple files compiled at one time.	x	x
language sensitive editor support.	x	—
specifying an error limit.	x	—
enabling/disabling an error category.	x	—
enable/disable optimization.	x	—
syntax only checking.	x	—

Environment Comparison

Linker Functional Capabilities

		VAX/VMS	RIS/VS/RSX
non-Ada link capability	x		
deferred (after a specific time)	x		
enable/disable link map generation	x	x	
specify full/brief link map	x		
generate a link command file	x		
enable/disable symbol cross-reference	x		
generate debug information	x	x	
enable/disable executable file creation	x		
specify batch/nobatch operation	x		
specify map filename	x		
specify object filename	x		
specify diagnostic output file	x		
enable/disable system library search	x	x	
enable/disable traceback info	x		
library search capabilities	x	x	
extended options capabilities	x		
shareable image support	x		
specify maximum memory		x	
force load		x	
enable/disable library trace		x	
specify main program		x	
non-Ada main program	x		

Environment Comparison

DEC ACS Linker Example

```
$ acs link test1
$ACS-1-CL LINKING, Invoking the VAX/VMS Linker
$SET DEFAULT USER1:(FRENCH)
$LINK := ""
$LINK-
/NOMAP-
/EXE=()TEST1-
  SYS$INPUT:/OPTIONS
USER1:(FRENCH)TEST1.OBJ;1
SYS$COMMON:(SYSLIB.ADALIB)IO_EXCEPTIONS_.OBJ;1
SYS$COMMON:(SYSLIB.ADALIB)TEXT_IO_.OBJ;1
SYS$COMMON:(SYSLIB.ADALIB)TEXT_IO_.OBJ;1
USER1:(FRENCH.LIB)PACK1_.OBJ;4
USER1:(FRENCH.LIB)PACK1.OBJ;4
USER1:(FRENCH.LIB)TEST1.OBJ;4
$DELETE USER1:(FRENCH)TEST1.OBJ;1
$DELETE USER1:(FRENCH)TEST1.COM;1
$
```

Environment Comparison

Source Level Debugger Functional Capabilities

	UAX/UMS	AOS/US/ADE
Breakpoints (set/reset) on		
statements	X	X
program units		
subprograms	X	X
packages	X	X
tasks	X	X
generic units	X	X
exceptions	X	X
Tracepoints (set/reset) on		
statements	X	
program units		
subprograms	X	X
packages	X	
tasks	X	
generic units	X	
exceptions	X	X
rendezvous	X	X
Watchpoints for variables	X	
Display		
program source	X	X
history		X
stack	X	X
tasks	X	X
breaks	X	X
tracepoints	X	X
Evaluate Objects	X	X
Step		
single	X	X
by discrete amounts	X	X
into subprograms	X	X
over subprograms	X	X
to next rendezvous		X
to end of program unit		X
Miscellaneous		
symbol abbreviation		X
set context for program control	X	X
input debugger command files	X	X
modify variables' value	X	X
console interrupt	X	X
full screen mode	X	
keypad mode for entering commands	X	

Environment Comparison

Program Librarian Functional Capabilities

	UAX/UMS	RDS/US/RDE
Listing Information		
directory of unit names	x	x
associated file names for unit	x	x
units WITHing specified unit		x
units WITHed by specified unit		x
time-stamp information	x	x
kind of compilation unit	x	x
Completeness and Currency check	x	x
Automatic recompilation	x	
Spawn CLI subprocess	x	x
Remove compilation unit	x	x
Library Access Control		
Read Only	x	x
Exclusive	x	x

Environment Comparison

Generated Files

Compilers

DG ADE

DEC ACS

OBJ	OBJ	The object code
SR		Assembly language equivalent
LST	LIS	List file
TREE	ACU	Internal representation for PL
STR		String info for Dione tree
	ADC	A copied source file

Linkers

DG ADE

DEC ACS

MAP	MAP	A link map
PR	EXE	The executable program
ST		Symbol table info for debug
LOG	LIS	Link messages
	COM	A DCL command file

Environment Comparison

File Structure

DG ADE

Program library is a file in a given directory

These files are put in the director containing the program
library

The files that are required by the link/program librarian have a
unique name constructed from the original name and a number

DEC ACS

Program library is a directory

Generated files are placed in this directory
except for the list and map files

Environment Comparison

Configuration Management

Functional Capabilities

	UAX/UMS (CMS)	AOS/US (TCS)
Configuration Library	—	—
create	x	—
delete	x	—
verify	x	—
Library Elements	x	x
create	x	x
delete	x	x
fetch	x	x
reserve	x	x
unreserve	x	—
replace	x	x
differences	x	—
Element Classes	—	—
create	x	x
delete	x	—
insert element	x	x
remove element	x	—
Listings	—	—
elements	x	x
reservation	x	x
history	x	—
annotation	x	—

Environment Comparison

Configuration Management

DG ADE Text Control System TCS

Creates a control file for each file that is under CM
Like Unix SCCS

DEC VMS Code Management System CMS

places files in a controlled directory much like the Ada program
library

Environment Comparison

Electronic Mail Systems

Functional Capabilities

	UAX/UMS (MAIL)	ROS/US (MAIL.CLI)
Message related functions		
Send	x	x
Receive	x	x
Immediate forwarding	x	
Immediate reply	x	
Archive	x	x
Print	x	x
Search for string	x	x
Edit message to be sent	x	x
Read next message	x	
Read previous message	x	
Read first message	x	
Read last message	x	
Position to start of current message	x	
Miscellaneous		
Keypad support	x	
On-line help facility	x	x
Send to distribution lists	x	x
Send across DECnet	x	
Mail folders	x	

Environment Comparison

Text Editors

DG ADE SCRED

DEC YMS EMACS

Functional Capabilities

	VAX/VMS (EMACS)		ACS/US (SCRED)	
Cursor Movement				
Left, Right, Up, Down	X	X	X	X
Top, Bottom	X	X	X	X
Beginning/End of line	X	X	X	X
Next/Previous Word	X	X	X	X
Search/Replace				
Search Forward	X	X	X	X
Search Reverse	X	X	X	X
Regular Expression Search	X	X	X	X
Regular Expression Replace	X	X	X	X
Multiple Replace	X	X	X	X
Buffers				
Copy text to	X	X	X	X
Copy text from	X	X	X	X
Split Screen	X	X	X	X
Edit multiple files	X	X	X	X
Regions				
Set mark	X	X	X	X
Kill region	X	X	X	X
Copy region	X	X	X	X
Move region	X	X	X	X
File Manipulation				
Copy from file	X	X	X	X
Append to file	X	X	X	X
Macros				
Keyboard macros	X	X	X	X
Macro language	X	X	X	X
Ada Mode	X	X	X	X
Ada LRM automated access	X	X	X	X
Miscellaneous				
Terminal independent	X	X	X	X
On-line help facility	X	X	X	X
Minimal redisplay algorithm	X	X	X	X
Keypad, function key re-definition	X	X	X	X
Undo Capability	X	X	X	X
Spawn CLI	X	X	X	X
Command iteration	X	X	X	X
Command type-ahead	X	X	X	X

DG PROBLEMS/ISSUES

AOS/VS PROBLEMS/ISSUES

ADE PROBLEMS/ISSUES

Environment Comparison

Conclusions

The two environments contain very similar tools

The DEC ACS is more integrated into the environment than the
DG ADE

The DEC compiler/linker generates better error messages

The capabilities of the editors were similar except that EMACS
could

- edit multiple files at the same time
- window the display and allow editing in each window

The DG electronic mail system was insufficient

The DG ADE file structure was confusing and difficult to use

ADE PROBLEMS/ISSUES

VERSION 2.20

DG Ada COMPILER/RUNTIME

NO UNCHECKED DEALLOCATION

NO UNCHECKED CONVERSION

INEFFICIENT STORAGE ALLOCATION SCHEME

UNUSUAL RUN-TIME STORAGE MANAGEMENT SCHEME

DYNAMIC OBJECTS WITHIN TASKS ALLOCATED FROM TASK'S STACK

AGGREGATES

STRING SLICES

INADEQUATE DOCUMENTATION FOR DOING SYSTEM PROGRAMMING

PHANTOM BUGS

GENERIC

SYNTAX ERROR DETECTION/MARKING

LINKER/LIBRARY SEARCHLIST CLOSURE PROBLEM

PACKAGE BODY DEPENDENCY PROBLEM

DG BUILD UTILITY

SCRED

*COMBINATION COMPILE/LINK TIME IS TRUE
INDICATION OF A COMPILER'S PERFORMANCE*

AOS/VS PROBLEMS/ISSUES

VERSION 5.03

TERMINAL COMMUNICATION

RESTRICTED TO READING 1 CHARACTER AT A TIME
FROM THE TYPE-AHEAD BUFFER
CAN'T DEQUEUE I/O REQUESTS TO CONSOLE
CONSOLE I/O FROM Ada TASKS IMPLEMENTED VIA AOS/VS SERVER TASKS

PROCESS CONTROL

PROCESS CREATION IS OK
DELETE A FATHER PROCESS HANGS (DOES NOT KILL SON PROCESS)
RESUME/SUSPEND PROCESS EXECUTION REQUIRES SUPERUSER PRIVILEGE
MUST USE SUICIDE CALL TO TERMINATE AIM PROGRAM

PROCESS COMMUNICATION

IPC FILES WITH Ada TASKS HANDLING I/O SYNCHRONIZATION
CAN'T PASS IPC FILES TO SON PROCESS
AS @INPUT and @OUTPUT (AOS/VS DESIGN FLAW)
CLI UNDER AIM ACTS AS IF IN BATCH MODE (CONSOLE IS UNDEFINED)
SUSPENSION OF PROGRAM OUTPUT DONE VIA Ada TASK SUSPENSION
RATHER THAN SYSTEM SERVICES
PROGRAM LOGIC ASSUMES ASYNCHRONOUS PIPE FOR WRITING TO A PROCESS
(AFFECTED REHOST)

Ada Tasking Experiences

Taxonomy of Ada Tasks

Ada Task Sets (Building Blocks)

AIM Data Flow Model

Controlling Ada Task Start-up

Graceful Ada Task Termination

Conclusions

Taxonomy of Ada Tasks

Server Tasks

Actor Tasks

Transducer Tasks

Actor Tasks

Active constructs

Generate external requests

Infinite loop

Zero entries declared in specification
(except possibly a START entry)

Examples

Customer/Consumer Task

Producer Task

Monitor Task

Server Tasks

Passive constructs

React to external requests

Infinite loop encapsulating
selective wait statement

Entries correspond to exported services

Terminate alternative

Examples

Agent Task
Buffer Task
Synchronization Task

Ada Tasks Sets (Building Blocks)

Intrinsic Actor/Server relationship

Majority of Ada tasks cooperate in
Actor/Server pairs

Ada Task Sets

Assumptions

Task inter-relationships relative to Actor task
Actor/Server tasks occur in alternating patterns
Actor associated with at least one Server and vice versa

Canonical Task Sets

one-to-one
one-to-many
many-to-one
many-to-many
one-or-more-transducer

Transducer Tasks

Actor/Server Task Hybrid

Both Active and Passive

Body textually similar to Server Task

React to external requests

Infinite loop encapsulating
selective wait statement

Entries correspond to exported services

Terminate alternative

Generates external requests for services

Examples

Message Router Task

Secretary Task

Controlling Ada Task Start-up

Task Activation

Elaboration of task's declarative part

Automatically after elaboration of parent's declarative part

Occurs collectively for every task object
defined in the implied parent's declarative part

Parent's execution suspended

Task Execution

Execution of statements in task's body

Begins after the activation of every task
in its activation collection

Language provides no technique for controlling
the start of execution for tasks

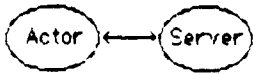
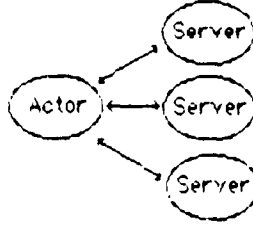
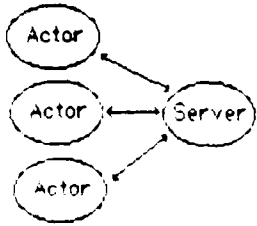
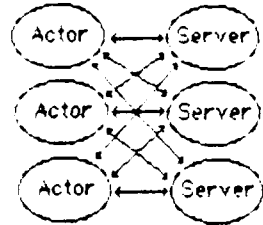
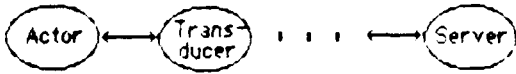
Programming Mechanism

```
select  
  accept START;  
or  
  terminate;  
end select;
```

Place initiation code after selective wait

Very effective when a tag parameter has to be passed
into a task before it can function correctly

Canonical Ada Task Sets

Set	Description	Notation	Example
1	one-to-one		Customer/Server
2	one-to-many		Customer/Multiple Servers
3	many-to-one		Producer/Buffer/Consumer
4	many-to-many		Multiple Customers/ Multiple Servers
5	one-or-more-transducer		Message Sender/ Message Dispatcher/ Message Receiver

Graceful Ada Task Termination

(con't)

Termination of Task Sets

Server Tasks present no problem

Actor tasks need programmed termination mechanism

Incumbent on Server task to trigger termination
for its dependent Actor tasks

Programming Mechanism

Export entry from Server task which will trigger the
termination for the task set

Use BOOLEAN entry parameter between Server and Actor(s)
to indicate when Actors should terminate

Graceful Ada Task Termination

Task Completion

Task has finished executing the statements in its body

Task Termination

(3 scenarios)

1. FOO has no dependent tasks

FOO terminates when it completes its execution

2. FOO has dependent tasks that have already terminated

FOO terminates when it completes its execution

3. FOO has dependent tasks that have not terminated

- a. FOO has completed

FOO terminates when all of its dependent task terminate

- b. FOO's execution is at open terminate alternative

FOO terminates if and only if the following conditions are satisfied:

- *FOO depends on some master whose execution is completed*
- *Each task that depends on this master is already terminated or similarly at an open terminate alternative*

Conclusions (con't)

Actor/Server task pairs occur
in alternating patterns

A Programming mechanism for
controlling the start of execution
of tasks is usually necessary

It is incumbent on Server tasks
to trigger the termination of their
respective dependent Actor tasks

Encapsulating tasks within a package
and then exporting a set of procedural
interfaces which parallels these tasks'
entries is quite effective

Complex concurrent systems can be
constructed using three simple steps:

1. Enumerate the fundamental task sets
2. Identify overlapping areas of these task sets
3. Dovetail task sets together

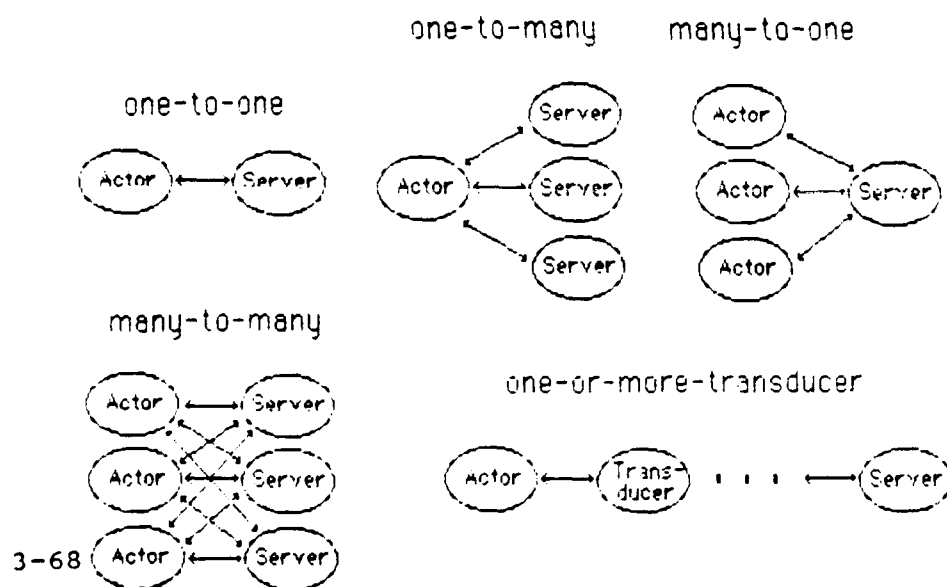
Conclusions

Most Ada Tasks are either
Servers or Actors

Ada Tasks cooperate and function in sets
rather than as stand-alone program units

The majority of concurrent systems can be
developed using a finite set of fundamental
(canonical) Ada tasking building blocks

Canonical Task Sets



Exceptions

No formal technique exists for incorporating and handling exceptions during the design phase of a system. This makes it difficult to take advantage of the exception capabilities of the Ada language

Exceptions are used to indicate exceptional conditions during program execution

Exceptional conditions are not restricted to errors. They can be used to indicate *any* condition (program state) that has a *lower* probability of occurrence

Exceptions can be propagated explicitly and implicitly. However, implicit propagation tends to hide the propagation path, making testing and integration more difficult

Exceptions can be a mechanism for controlling program flow. They initially provide an implicit *goto* to the end of a block

The use of exceptions can impact the testing and integration phases of system development

AIM

COMPONENTS

Ada supports and encourages reusable software components

This aspect of the language can be seen in the design and implementation of the AIM

Some AIM components:

Help	exported
Queue	could be exported - generic
Stack	exported - generic
String	could be exported
VT	imported

String Manipulation

There is a tendency to get *caught up* in the concept of typing and string lengths

Design procedures and functions with unconstrained string parameters to make it easier to use string slices as parameters

The procedures and functions become less susceptible to design and implementation changes

This can reduce string manipulation prior to making the call on the procedure or function

A string utility package can be implemented to perform many of the string manipulation functions required by the system

Exceptions

Typically, source level debuggers provide a program breakpoint when an exception is encountered during program execution

If some exception occurs too often, the constant breakpoints can be a source of irritation to the test engineer

While it is possible to override this feature of the debugger, this is normally not feasible due to testing requirements, particularly when the standard pre-defined exceptions are involved

Any exception which is raised relatively frequently is not a true exceptional condition and, therefore, should be handled in a different manner within the design and implementation of the system

Transporting the AIM

Procedures Followed

Physical transfer

Source code level transfer

240 separate text files

Accomplished using the text transport tool *Pager*
Pager was adapted from a tool in the SIMTEL-20
repository

ANSI magnetic tape support on both machines

It took one afternoon to accomplish the entire
physical transport

Transporting the AIM

The AIM was transported from the DG ADE to the VAX/VMS ACS at the source code level

The transport took 2.4 man-months

Procedures Followed

Rehost Problems

Conclusions

Transporting the AIM

A Feedback Loop

GOAL

Have the same source code on all systems except for the parts
explicitly system dependent

Once the rehosted AIM passed formal testing it was re-rehosted
into the ADE

How many times does one need to do this before the goal
is met?

Configuration management nightmare

Exactly the same code on both systems except for the system
dependent parts

21000 lines vs 1000 lines

Transporting the AIM

Debugging

How do you debug a completed system?

Break the system back down into modules and proceed with
module testing

Regenerate tests to test particularly offensive modules

Use a source level debugger

First Choice: *Use the debugger*

One Man-Month to completely debug and
pass the integration tests

AIM

Rehost Problems

Compiler/Debugger Problems

DEC BETA Compiler

returning aggregates and string constants
from a function

calling functions during variable initialization

get for integers does not scan
over blanks and newlines
different from the DG

Discriminant record with positive discriminant
raises numeric error

String problems in the debugger

Discriminant records and the debugger

Extremely complex interface to system services

Same aggregate problem with Parse tables
as on the DG

when trying to define a **very** large aggregate

Transporting the AIM

Rehost Problems

Code Work-Arounds

Program Termination

Transporting the AIM

System Dependencies

Computer Terminal Control and Communications

Open the Computer Terminal

- no echo
- no buffering

Close the Computer Terminal

- reset back to its original condition

Read Data From the Terminal Keyboard

- at least one character at a time
- no echo
- must not block the calling process
- must block the calling task until at least one character is available

Write Data to the Computer Terminal Display

- sends an ANSI string to computer terminal display
- no buffering
- no interpretation
- occurs immediately

Transporting the AiM

System Dependencies

The AiM has four areas with system dependencies

Computer Terminal Control and Communications

APSE Process Control and Communications

Environment Variables

Ada **length** Representation Clause

Transporting the AIM

System Dependencies

Environment Variables

Getting the Terminal Name

Names of Files

Terminal Capabilities Filename (TCF)

Initial Script Filename

Parse Table Filename

Help Filename

Ada *LENGTH* Representation Clause

for TASK_NAME'SORAGE_SIZE use TASK_SIZE;

Transporting the AIM

System Dependencies

APSE Process Control and Communications

Create a Son Process

IPC channels will be specified as the process' standard input and output
dependent on the AIM for its existence

Destroy a Son Process

pending I/O requests are dequeued
IPC channels are deleted
anything that is running in the son is stopped immediately

Read a Line From a Son

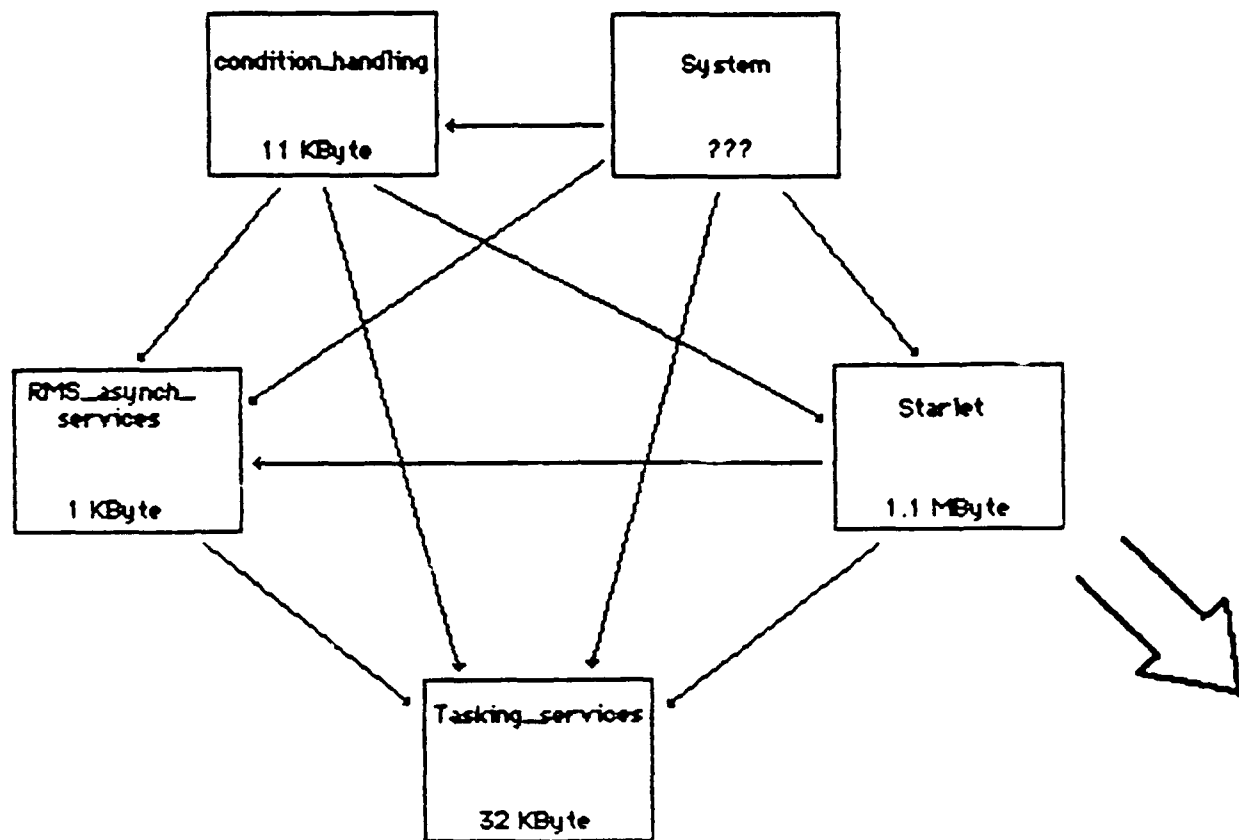
calling task is blocked until a line is available
calling process is NOT blocked
line is read from the IPC file associated with the son's standard output

Write a Line To a Son

calling task is not blocked, the data is queued to the IPC file associated
with the son's standard input

Various Status and Information Queries

System Dependencies Confusion



WITH/USE/RENAMES used everywhere

Source available for everything except
system

Sizes are specification sizes

Transporting the AIM

Rehost Problems

Module Testing Baggage

Representation Clauses

`for TASK'SORAGE_SIZE use TASK_SIZE,`

Complexities relating to System Services

PRIVATE

-- These things are system dependent and may(will) change when
-- transporting

max_filename_length : CONSTANT positive := 31;
-- AOS/US max filename len.

max_process_name_length : CONSTANT positive := 31;

TYPE channel_range IS RANGE 0..255; -- AOS/US max channel no.

TYPE process_id IS
RECORD

pid : integer; -- AOS/US Process ID
in_channel : channel_range; -- AOS/US input channel no.
out_channel : channel_range; -- AOS/US out channel no.
in_filename : string(1..max_filename_length);
-- input IPC filenames
in_filename_length : positive;

out_filename : string(1..max_filename_length);
-- output IPC filename
out_filename_length : positive;

process_name : string(1..max_process_name_length);
process_name_length : positive;

END RECORD;

END sysdep_process;

PACKAGE sysdep_process IS

TYPE io_status IS (io_OK, io_other);

TYPE process_id IS PRIVATE;

PROCEDURE create_process
 (process : IN OUT process_id);

PROCEDURE read_from_process
 (process : IN process_id;
 out_data : OUT string;
 length : OUT natural;
 status : OUT io_status);

PROCEDURE write_to_process
 (process : IN process_id;
 data : IN string;
 status : OUT io_status);

FUNCTION new_line RETURN string;

FUNCTION process_exists(process : IN process_id) RETURN boolean;

PROCEDURE process_name
 (process : IN process_id;
 process_name : OUT string;
 name_last : OUT natural);

FUNCTION sons_exist(process : IN process_id) RETURN boolean;

PROCEDURE destroy_process
 (process : IN OUT process_id);

PROCEDURE suicide;

PRIVATE

-- These things are system dependent and may(will) change when
-- transporting

max_filename_length : CONSTANT positive := 31;
-- UMS max filename len.

max_process_name_length : CONSTANT positive := 15;

TASK TYPE process_pusher IS
ENTRY go(process : IN process_id);
END process_pusher;

TASK TYPE process_buffer IS
ENTRY go;
ENTRY enqueue(data : IN string);
ENTRY get(data : OUT string;
last : OUT natural;
done : OUT boolean);
ENTRY quit;
END process_buffer;

TYPE process_buffer_pointer IS ACCESS process_buffer;
TYPE process_pusher_pointer IS ACCESS process_pusher;

TYPE process_id IS
RECORD

pid : starlet.process_id_type; -- VAX/UMS Process ID
in_channel : starlet.channel_type; -- VAX/UMS channel no.
out_channel : starlet.channel_type; -- VAX/UMS channel no.
in_filename : string(1..max_filename_length);
-- Logical name of input mailbox
in_filename_length : positive;
out_filename : string(1..max_filename_length);
-- Logical name of output mailbox
out_filename_length : positive;
process_name : string(1..max_process_name_length);
process_name_length : positive;
watcher : process_pusher_pointer;
buffer : process_buffer_pointer;
END RECORD;

END sysdep_process;

```

WITH starlet;
PACKAGE sysdep_process IS

    TYPE io_status IS (io_OK, io_other );

    TYPE process_id IS PRIVATE;

    PROCEDURE create_process
        ( process : IN OUT process_id );

    PROCEDURE read_from_process
        ( process : IN process_id;
          out_data : OUT string;
          length : OUT natural;
          status : OUT io_status );

    PROCEDURE write_to_process
        ( process : IN process_id;
          data : IN string;
          status : OUT io_status );

    FUNCTION new_line RETURN string;

    FUNCTION process_exists( process : IN process_id ) RETURN boolean;

    PROCEDURE process_name
        ( process : IN process_id;
          process_name : OUT string;
          name_last : OUT natural );

    FUNCTION sons_exist( process : IN process_id ) RETURN boolean;

    PROCEDURE destroy_process
        ( process : IN OUT process_id );

    PROCEDURE suicide;

    process_creation_error : EXCEPTION;

```

CAIS Related Issues

Terminals

Processes

Interprocess Communication

Files

Transporting the AIM

Conclusions

The AIM transported in 2.4 man-months

Most problems were due to

compiler bugs
inappropriate assumptions

- terminal control and communications
- process model, control, and communications

The transport was assisted by

using ACVC validated compilers
designing for transportability
using a source level debugger

Interesting issues were raised regarding

Debugging a completed system
Interfaces with host OS system services
Feedback loops

CAIS Related Issues

Terminals

Major Differences from CAIS

Single terminal only

Terminal packages are independent

CAIS Related Issues

Terminals

Design Approach

Target "asynchronous ASCII" terminals

User Level

SCROLL_TERMINAL
PAGE_TERMINAL
FORM_TERMINAL

Simulation Level

VIRTUAL_TERMINAL_CONTENTS
REDISPLAY

Translator Driver Level

DRIVER
TCF
SYSDEP
VIRTUAL_TERMINAL_INPUT

Terminal Capabilities File

CAIS Related Issues

Terminals

Page Terminal Differences from CAIS

Exclusions

Set_Echo/Echo

Vertical tabs

Some graphic renditions

Bold, Faint, Underscore, Slow_Blink, Rapid_Blink

Insert_Space

Graphic_Rendition_Support

Additions

Open/Close

Enter/Exit Insert_Mode

Update_Screen line range

Update_Line line number

Update_Cursor

Redraw_Screen

CAIS Related Issues

Terminals

Scroll Terminal Differences from CAIS

Exclusions

Set_Echo/Echo
Vertical tabs
Position/Size is a column number

Additions

Open/Close
Update_Line

CAIS Related Issues

Terminals

System Dependencies

Open the terminal for binary I/O, no echo

Close the terminal

Get a string from the keyboard with no echo and no translation

Get TCF Name

Get Terminal Name

Valid Character determination input and output

CAIS Related Issues

Terminals

Form Terminal Differences from CAIS

Exclusions

Function_Keys function

Area_Value type

No_Fill, Fill_With_Spaces, etc.

Form_Type type

Additions

Open/Close

Assumption that Area_Qualifier_Requires_Space

Renaming

Next_Qualified_Area is named Tab

Erase_Form is named Erase_Display

CAIS Related Issues

Processes

CAIS Interfaces

Starting a process

```
procedure SPAWN_PROCESS (...);  
procedure INVOKE_PROCESS (...);  
procedure CREATE_JOB (...);
```

Parameters to and from a process

```
procedure AWAIT_PROCESS_COMPLETION (...);  
procedure APPEND_RESULTS (...);  
procedure WRITE_RESULTS (...);  
procedure GET_RESULTS (...);  
procedure GET_PARAMETERS (...);
```

Process control

```
procedure ABORT_PROCESS (...);  
procedure SUSPEND_PROCESS (...);  
procedure RESUME_PROCESS (...);
```

Process *attributes*

```
function STATUS_OF_PROCESS (...) return PROCESS_STATUS;  
function HANDLES_OPEN (...) return NATURAL;  
function IO_UNITS (...) return NATURAL;  
function START_TIME (...) return TIME;  
function FINISH_TIME (...) return TIME;  
function MACHINE_TIME (...) return DURATION;
```


CAIS Related Issues

Processes

Model

Inherit environment from Parent

Communicate with Parent
via STANDARD_INPUT and STANDARD_OUTPUT

CAIS Related Issues

Files

AIM requires simple Ada TEXT_IO & a file name

CAIS Related Issues

Interprocess Communication

Data General IPC

Asynchronous with Files

VAX IPC

Synchronous with Mailboxes

CAIS IPC

Synchronous (and Asynchronous?) with Queues

Conclusions

Lifecycle

More time was spent in design and implementation than the models predicted

The AIM did not require as much testing as the models predicted

If more time is spent in requirements definition and design, then less time will be spent in integration and testing

Using a source level debugger reduced testing and increased productivity

Design

OOD works fairly well *except for data flow*

OOD supports software components

Some other form of data flow design is needed

A design technique should support:

- isolation and encapsulation (transportability)
- data and control flow specification
- error condition identification and handling
- data structure definition
- PDL

Conclusions

Environment

The DG toolset and DEC toolset are similar, however, certain features of specific tools increase designer/programmer productivity significantly

The DEC ACS is more integrated in the environment than the DG ADE

Ada Language

Ada promotes transportability

It is easy to get caught up in the idea of typing

There are some problems with tasking

Rehost

Planning for transportability works

Careful consideration must be given to the models of the underlying operating system interfaces and intertool communications

and even then there will be problems

There are interesting problems associated with debugging a completed system that has been rehosted

Conclusions

CAIS

The CAIS terminal interface will work
at least on systems with asynchronous ASCII terminals

There may be serious problems implementing the CAIS process
control and communications (IPC) interfaces in modern
operating systems

More generally:
The Least Common Denominator must be explored

UNIX
VAX/VMS
Some IBM operating system
LISP Machine
(micro ??)

APSE
INTERACTIVE MONITOR

Final Report on Interface
Analysis and Software
Engineering Techniques

VOLUME 1

Environment Interfaces
Analysis

Prepared for:

NAVAL OCEAN SYSTEMS CENTER (NOSC)
United States Navy
San Diego, CA 92152

Contract No. N66001-82-C-0440
CDRL No. A010

Equipment Group - ACSL
P.O. Box 801, M.S. 8007
McKinney, Texas 75069
15 July 1983

TEXAS INSTRUMENTS
INCORPORATED

Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO).

ADE is a trademark of ROLM Corporation.

DEC is a trademark of Digital Equipment Corporation.

ECLIPSE is a registered trademark of Data General Corporation.

ECLIPSE MV/10000 is a trademark of Data General Corporation.

ROLM is a registered trademark of ROLM Corporation.

VAX is a trademark of Digital Equipment Corporation.

VMS is a trademark of Digital Equipment Corporation.

CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE	1-1
1.2	BACKGROUND	1-2
CHAPTER 2	AIM INTERFACE REQUIREMENTS	
2.1	GENERAL	2-1
2.2	TERMINAL COMMUNICATION	2-1
2.2.1	Echo Control	2-2
2.2.2	Non-Filtered Keyboard Read	2-2
2.2.3	Non-Filtered Display Write	2-3
2.2.4	Screen-Oriented Facilities	2-4
2.2.5	Exclusive Access	2-4
2.2.6	Terminal Identification	2-4
2.3	PROCESS CONTROL AND COMMUNICATION	2-4
2.3.1	Process Initiation	2-5
2.3.2	Process Termination	2-5
2.3.3	Process Suspension/Resumption	2-5
2.3.4	Process Status Query	2-5
2.3.5	SubProcess Query	2-6
2.3.6	Transparent Interprocess Communication	2-6
2.4	DATABASE SERVICES	2-6
2.4.1	File Manipulation	2-6
2.5	GENERAL ENVIRONMENT ISSUES	2-7
CHAPTER 3	ADA LANGUAGE SYSTEM INTERFACES	
3.1	GENERAL	3-1
3.2	TERMINAL COMMUNICATION	3-1
3.2.1	Echo Control	3-2
3.2.2	Non-Filtered Keyboard Read	3-2
3.2.3	Non-Filtered Display Write	3-2
3.2.4	Screen-Oriented Facilities	3-3
3.2.5	Exclusive Access	3-3
3.2.6	Terminal Identification	3-3
3.3	PROCESS CONTROL AND COMMUNICATION	3-3
3.3.1	Process Initiation	3-4
3.3.2	Process Termination	3-4
3.3.3	Process Suspension/Resumption	3-4
3.3.4	Process Status Query	3-5
3.3.5	SubProcess Query	3-5
3.3.6	Transparent Interprocess Communication	3-6
3.4	DATABASE SERVICES	3-8
3.4.1	File Manipulation	3-8
3.5	ENVIRONMENT SPECIFIC ISSUES	3-10
3.5.1	ALS "Break-In" Facility	3-10
3.5.2	Broadcast Messages	3-11

3.5.3	Bypassing KAPSE Services For Terminal Control	3-11
3.5.4	Constraints On User Programs	3-11
3.5.4.1	APSE Program I/O	3-11
3.5.4.2	MASTER_IN, MASTER_OUT, And MESSAGE_OUT . . .	3-12

CHAPTER 4 ADA INTEGRATED ENVIRONMENT INTERFACES

4.1	GENERAL	4-1
4.2	TERMINAL COMMUNICATION	4-1
4.2.1	Echo Control	4-2
4.2.2	Non-Filtered Keyboard Read	4-2
4.2.3	Non-Filtered Display Write	4-2
4.2.4	Screen-Oriented Facilities	4-2
4.2.5	Exclusive Access	4-3
4.2.6	Terminal Identification.	4-3
4.3	PROCESS CONTROL AND COMMUNICATION	4-4
4.3.1	Process Initiation	4-4
4.3.2	Process Termination	4-5
4.3.3	Process Suspension/Resumption	4-5
4.3.4	Process Status Query	4-5
4.3.5	SubProcess Query	4-5
4.3.6	Transparent Interprocess Communication	4-6
4.4	DATABASE SERVICES	4-7
4.4.1	File Manipulation	4-7
4.5	ENVIRONMENT SPECIFIC ISSUES	4-9
4.5.1	Bypassing KAPSE Services For Program Control .	4-9
4.5.2	Broadcast Messages	4-9

CHAPTER 5 PROPOSED MIL-STD-CAIS

5.1	GENERAL	5-1
5.2	BACKGROUND	5-1
5.3	TERMINAL COMMUNICATION	5-2
5.3.1	Echo Control	5-2
5.3.2	Non-Filtered Keyboard Read	5-3
5.3.3	Non-Filtered Display Write	5-3
5.3.4	Screen-Oriented Facilities.	5-4
5.3.5	Exclusive Access	5-4
5.3.6	Terminal Identification	5-5
5.4	PROCESS CONTROL AND COMMUNICATION	5-6
5.4.1	Process Initiation	5-6
5.4.2	Process Termination	5-7
5.4.3	Process Suspension/Resumption	5-7
5.4.4	Process Status Query	5-8
5.4.5	SubProcess Query	5-8
5.4.6	Transparent Interprocess Communication	5-9
5.5	DATABASE SERVICES	5-9
5.5.1	File Manipulation	5-9
5.6	ENVIRONMENT SPECIFIC ISSUES	5-11

CHAPTER 6 ADA DEVELOPMENT ENVIRONMENT INTERFACES

6.1	GENERAL	6-1
6.2	TERMINAL COMMUNICATION	6-2
6.2.1	Echo Control	6-3
6.2.2	Non-Filtered Keyboard Read	6-3
6.2.3	Non-Filtered Display Write	6-5
6.2.4	Screen-Oriented Facilities	6-5
6.2.5	Exclusive Access	6-5
6.2.6	Terminal Identification	6-5
6.3	PROCESS CONTROL AND COMMUNICATION	6-6
6.3.1	Process Initiation	6-6
6.3.2	Process Termination	6-7
6.3.3	Process Suspension/Resumption	6-7
6.3.4	Process Status Query	6-7
6.3.5	SubProcess Query	6-8
6.3.6	Transparent Interprocess Communication	6-8
6.4	DATABASE SERVICES	6-9
6.4.1	File Manipulation	6-9

APPENDIX A AIM INTERFACES SUMMARY

APPENDIX B ARPANET COMMUNICATIONS

APPENDIX C GLOSSARY

APPENDIX D REFERENCES

D.1	GOVERNMENT STANDARDS	D-1
D.2	GOVERNMENT SPECIFICATIONS	D-1
D.3	OTHER GOVERNMENT DOCUMENTS	D-2
D.4	SPECIAL SOURCES	D-3
D.5	OTHER PUBLICATIONS	D-4

FOREWORD

This document is the final version of the Interface Report produced by Texas Instruments, developed under Navy contract number N66001-82-C-0440, CDRL number A010. This report, consisting of three volumes, contains several modifications to previous information as well as new information concerning experiences and implementation since the interim version of this report was produced.

- * Volume I, "Environment Interface Analysis", is an analysis of environment interfaces issues. The information in it is primarily a recap of data contained in the interim interface report. However, information covering the Data General Ada(tm) Development Environment (ADE)(tm) has been added.
- * Volume II, "Design and Implementation Experiences: The AIM", covers design and implementation experience gained through work on the AIM.
- * Volume III, "Transporting an Ada Software Tool: A Case Study", is a case study of the rehosting of the AIM from a Data General Eclipse MV/10000(tm) to a VAX(tm) 11/785. It is totally new and contains information on the rehost effort including transportability issues.

John Foreman is the project manager for this effort. The research for this report was performed by Jerry Baskette, Mark Borger, Thomas Duke, Stewart French, Tim Harrison, and Melody Moore.

CHAPTER 1

INTRODUCTION

1.1 PURPOSE

An Ada program requires clear and well-defined interfaces to interact with an Ada Programming Support Environment (APSE). To meet the goals of interoperability and transportability, an Ada program intended to execute under more than one APSE should interface equally well with each system. To accomplish this requires a detailed study of existing and planned APSE features mapped against the requirements of the Ada program. Through analysis, APSE interface strengths and deficiencies are revealed.

This report is an analysis of the APSE Interactive Monitor (AIM) developed by Texas Instruments under NOSC contract N66001-82-C-0440. AIM requirements are mapped against the designed features of four systems:

- * U.S. Army Ada Language System (ALS),
- * U.S. Air Force Ada Integrated Environment (AIE),
- * the MIL-STD Common APSE Interface Set (CAIS) currently under development by the KAPSE Interface Team (KIT),
- * the Data General AOS/VS operating system running the Ada Development Environment (ADE).

The analyses of the ALS and the AIE were performed with design information prior to completed implementations of those systems. Similarly, the CAIS interfaces compose a draft standard subject to change.

INTRODUCTION BACKGROUND

1.2 BACKGROUND

The APSE Interactive Monitor (AIM) is a tool designed to act as an interface between the user of the APSE and the programs the user executes in the APSE. It enables a user to execute multiple APSE programs from a single terminal while keeping their interactive inputs and outputs separate both logically and physically. For a complete description of AIM functionality, consult [TI83A].

The primary objective of the AIM project is to assist the KAPSE Interface Team (KIT) in studying KAPSE interface issues while secondarily producing a useful tool for APSEs. The reader should be familiar with the AIM Program Performance Specification [TI83A], the ALS System Specification [SOF83], the AIE System Specification [INT82], the ADE [DAT84], and the draft MIL-STD CAIS [KIT85]. Additional AIM references include: [TI83B], [TI83C], [TI83D], [TI83E], [TI83F], [TI83G], [TI83H], [TI85A], [TI85B], [TI85C], [TI85D], [TI85E], [TI85F], [TI85G], and [TI85H].

CHAPTER 2

AIM INTERFACE REQUIREMENTS

2.1 GENERAL

This chapter outlines the KAPSE interfaces the AIM requires and the rationale behind them. Described below are general issues and ideal solutions to AIM implementation problems. In the next four chapters of this report, AIM requirements are mapped against ALS, AIE, CAIS, and ADE facilities.

The AIM requires well-defined interfaces in the following areas:

1. Terminal Communication
2. APSE Process Control and Communication
3. Database Services

2.2 TERMINAL COMMUNICATION

The AIM interacts with page mode physical terminals; these screen-oriented terminals transmit and receive single characters at a time and possess extended two-dimensional functional capabilities. The capabilities of a page mode terminal intended for use with the AIM must be a functionally compatible subset of the standard capabilities described in [ANSI79].

The AIM requires the following support from the KAPSE terminal communication services:

1. Echo Control
2. Non-Filtered Keyboard Read

AIM INTERFACE REQUIREMENTS

TERMINAL COMMUNICATION

3. Non-Filtered Display Write
4. Screen-Oriented Facilities
5. Exclusive Access
6. Terminal Identification

2.2.1 Echo Control

Description: The ability to enable and disable character echo on the display as characters are input from the terminal keyboard.

Rationale: Screen echo may be controlled from a variety of sources:

- * Directly in the terminal
- * From a communication line (i.e., modem)
- * From the KAPSE
- * From the host operating system
- * From an APSE program

To permit asynchronous update and to control the screen display, the AIM requires the ability to disable echo in order to place characters in the correct screen location. For example, if the cursor is involved in a write operation at the top of the screen, and a character intended to follow a command prompt at the bottom of the screen is typed from the keyboard, the newly-input character could be echoed in the middle of the write transmission unless the AIM can disable character echo. The AIM itself should receive the character and echo it in the appropriate place. The KAPSE terminal communication services should permit echo disabling when possible in order to facilitate this diverse control.

It should also be noted that some interactive editing tools (such as EMACS) cannot be implemented unless echo disabling and non-buffered I/O are provided. These tools must also be able to intercept data without echo in order to control the display of data on the screen.

2.2.2 Non-Filtered Keyboard Read

Description: The ability to read characters from the keyboard of a terminal with no interpretation or translation by the environment.

Rationale: The AIM belongs to a group of highly interactive programs

AIM INTERFACE REQUIREMENTS TERMINAL COMMUNICATION

which ideally should have access to characters immediately as they are generated by the keyboard. In order to afford maximum control of terminal I/O to these programs, there should be no interpretation of key sequences entered at the host OS or KAPSE level; interactive programs provide their own interpretation. Buffering usually implies an interpretation of at least one character (the "end of buffer" mark, usually carriage return). This rationale does not preclude buffering which supports the desirable type-ahead capability. Characters may be piped into a channel to wait for reading and still be accessible singly.

In addition to the AIM, programs such as screen editors and powerful interactive command language interpreters require character-by-character input for implementation. This need is further supported in [COX83]:

"Since it would be unreasonable to make it impossible to implement screen oriented text editors or advanced command line interpreters in the APSE, immediate acquisition of input characters must be provided. The Ada language definition avoids this issue and leaves the Ada programmer at the mercy of side effects arising from system buffering of I/O. This I/O facility should therefore be provided to Ada programmers through the KAPSE interface." [COX83]

I/O buffering also affects the AIM updating mechanism. The cursor is always positioned at the current read or write location on the screen. The AIM updates several viewports asynchronously while allowing the user to enter keystrokes destined for a particular APSE program's input. Consequently, the AIM requires the freedom to move the cursor immediately to any location where an I/O transaction is destined to occur. If characters are buffered by the terminal (i.e., transmitted only on depression of carriage return), the AIM waits for the input of an entire character string before releasing the cursor for I/O elsewhere on the screen. This buffering would limit the AIM asynchronous screen updating mechanism, because the AIM would not receive each character as it was generated.

2.2.3 Non-Filtered Display Write

Description: The ability to write characters to the terminal display device exactly as they are represented in the generating program.

Rationale: The AIM must be able to write one or more characters to the screen with no extraneous characters (such as line feed) automatically appended, and no character translation operations occurring. Terminals are controlled by specific protocols transmitted by the AIM and interpreted by the display device.

AIM INTERFACE REQUIREMENTS

TERMINAL COMMUNICATION

Terminal communication protocols consist of character sequences in which the relationships between specific characters are given meaning. Adding or removing any character in a sequence may alter its meaning.

2.2.4 Screen-Oriented Facilities

Description: The ability to perform operations such as delete-line, delete-character, insert-line, insert-character, clear-screen.

Rationale: The AIM requires the control to position or move the cursor and perform simple editing functions on its two dimensional display. With these minimum capabilities, the AIM can simulate other more complicated editing functions such as insert or delete line.

2.2.5 Exclusive Access

Description: The ability to obtain exclusive access to the user's terminal.

Rationale: The AIM needs to be able to manage all data destined for, or received from, the terminal. If the AIM cannot protect the screen from other programs' I/O, any APSE program could write to the screen, perhaps causing undesirable data transmissions to collide with or overwrite AIM transmissions.

2.2.6 Terminal Identification

Description: The ability to obtain specific information concerning the terminal's capabilities and features.

Rationale: Some standard method of naming terminal types must exist to enable terminal capabilities to be mapped into a database file. The AIM must be able to retrieve this information and identify the correct physical terminal in order to initialize the correct logical terminal.

2.3 PROCESS CONTROL AND COMMUNICATION

The AIM provides the user with the capability of starting, stopping, resuming, and otherwise controlling processes and subprocesses. Processes must be able to send and receive information to and from other processes. Several requirements are made of the environment to accomplish this communication:

1. Process Initiation

AIM INTERFACE REQUIREMENTS
PROCESS CONTROL AND COMMUNICATION

2. Process Termination
3. Process Suspension/Resumption
4. Process Status Query
5. SubProcess Query
6. Transparent Interprocess Communication

2.3.1 Process Initiation

Description: The ability to initiate the execution of a program and execute in parallel with the resulting invoked process.

Rationale: The AIM controls APSE programs and subordinate programs spawned from these programs through this interface. The AIM user may invoke APSE programs from the AIM and control their execution. The above program control functions are defined as basic APSE requirements in "STONEMAN" [DOD80] and therefore should be common to all APSEs.

2.3.2 Process Termination

Description: The ability to terminate the execution of a subordinate process (i.e.--one that has been invoked by the AIM).

Rationale: The AIM requires the ability to terminate any process that it invokes and any subprocesses spawned by that process.

2.3.3 Process Suspension/Resumption

Description: The ability to suspend the execution of a subordinate process and later resume the suspended process.

Rationale: The AIM requires the capability to control the execution activity of any process that is subordinate to it, in order to provide users with as much control as possible over their programming environment and the activities taking place in it.

2.3.4 Process Status Query

Description: The ability to determine the status of a subordinate process, typically, information such as Awaiting-IO, Suspended, or Running.

Rationale: The AIM provides process status information, via a viewport header, for any APSE program whose corresponding AIM window

AIM INTERFACE REQUIREMENTS

PROCESS CONTROL AND COMMUNICATION

is being displayed on the physical screen; furthermore, the AIM's Info Utility displays process status information for any of the currently active APSE programs subordinate to the AIM.

2.3.5 SubProcess Query

Description: The ability to determine whether a process (subordinate to the AIM) has any processes that are subordinate to it.

Rationale: The AIM needs to determine all activity in the system that is a result of its own execution. This activity necessarily includes any processes that have come into existence because of processes invoked by the AIM itself. The AIM queries the program subprocess structure to determine subordinate program information, and examines this structure to manage programs executing under its control. For example, the user may not exit the AIM unless all subordinate programs have terminated. The AIM determines which programs and subprograms are currently executing by examining its subprocesses.

2.3.6 Transparent Interprocess Communication

Description: Interprocess Communication is an important AIM interface. The AIM "captures" terminal-directed output from APSE programs and provides program-directed input from the terminal.

Rationale: The AIM requires that all data be received in the same order as it was transmitted, in essence, a data "pipe". This pipe scheme should be totally transparent to the APSE program running under the AIM; the AIM should not in any way affect the implementation of APSE program I/O.

Since an APSE program is defined as one which only uses KAPSE services, the AIM requires APSE programs to use only Standard_In and Standard_Out for terminal directed I/O. An APSE program which bypasses the KAPSE to use underlying host services is considered erroneous, and may produce undesirable results when executed under the AIM. Only one terminal may be associated with an APSE program, since there is no method of identifying multiple terminals.

2.4 DATABASE SERVICES

2.4.1 File Manipulation

The AIM manipulates database files during initialization and execution. The packages TEXT_IO, SEQUENTIAL_IO, and DIRECT_IO defined in [DOD83] provide the AIM with clear interfaces for file manipulation. Most APSE's augment the I/O capabilities defined in the Ada language with extended features. The following is a list of

AIM INTERFACE REQUIREMENTS DATABASE SERVICES

AIM functional requirements which the KAPSE database services should fulfill:

1. Open/Close a file for reading/writing
2. Read/Write a file
3. Create/delete a file

Rationale: During its initialization the AIM reads information from specific predefined KAPSE database files: the LALR parse table file, the AIM initialization command script file, and the Terminal Capabilities File; furthermore, the AIM must open and read the Help file when on-line help is initially requested. The KAPSE database services must support opening, reading, and closing these database files.

The AIM also requires additional database services during execution. The AIM supports the notion of input and output pads on a per window (program) basis; these pads are merely APSE database files that respectively capture all input and output transferred between an KAPSE program executing in the environment and its associated AIM window. The AIM requires a method of creating, deleting, and controlling pad files from the KAPSE database services.

2.5 GENERAL ENVIRONMENT ISSUES

Although the AIM is intended to be transparent to APSE programs, it is possible that APSE programs may be required to follow some guidelines in order to interface with the AIM, especially in the area of terminal I/O. This section describes some restrictions that must be imposed upon programs which are intended to execute under the AIM.

If the appropriate KAPSE services are provided by some environment, the AIM mechanism required to intercept program I/O will not impact APSE program design. APSE programs should need no special I/O file parameters to function in the AIM. APSE program terminal-destined I/O must be accomplished through the STANDARD_IN and STANDARD_OUT files; an APSE program which accesses the terminal in any other way might not be AIM-compatible. APSE programs should generate only the standard printable ASCII characters. If an APSE program generates characters outside the ASCII printable range, the AIM will remove those characters.

In order for the AIM to obtain exclusive access to an APSE program's terminal input and output, the user program may use only STANDARD_IN and STANDARD_OUT for terminal I/O. There may be no other files associated with the terminal. Terminal I/O is accomplished through these standard files which are connected to the terminal. No logical

AIM INTERFACE REQUIREMENTS
GENERAL ENVIRONMENT ISSUES

name translation is provided to access a unique device.

An APSE program is by definition a program which uses only KAPSE services. Therefore, a program which bypasses KAPSE services for any reason is not an APSE program and therefore might not be AIM compatible.

CHAPTER 3

ADA LANGUAGE SYSTEM INTERFACES

3.1 GENERAL

The Ada Language System (ALS) has been recently released by SofTech, Inc. under U.S. Army contract number DAAK80-80-C-0507. This analysis was performed using detailed design documentation made available prior to the release of the ALS [SOF83]. This section evaluates the ALS against the requirements of the AIM.

3.2 TERMINAL COMMUNICATION

The terminal communications facilities of the ALS are sorely deficient for purposes of the AIM. None of the AIM required capabilities exist in the ALS. Below, the interfaces are listed with explanations of ALS insufficiencies.

Insufficient Interfaces

1. Echo Control
2. Non-filtered keyboard read
3. Non-filtered display write
4. Screen-oriented facilities
5. Exclusive access
6. Terminal identification

ADA LANGUAGE SYSTEM INTERFACES

TERMINAL COMMUNICATION

3.2.1 Echo Control

The ALS defines only two packages to supplement Ada file I/O: packages `BASIC_IO` and `AUX_IO`; there is no `TERMINAL_IO` package. Neither of the defined packages support echo enable/disable.

3.2.2 Non-Filtered Keyboard Read

According to ARPANET response from SofTech, the ALS KAPSE does not perform any I/O buffering. However, the underlying host (VMS) buffers keyboard input except for special control character sequences which cause interruption (such as `<CNTRL>-Y`). The ALS provides no mechanism for bypassing this VMS buffering. (See Appendix B question 3.)

The VMS terminal driver waits for a carriage return to transmit characters. The AIM can circumvent buffering of character I/O by using the VAX QIO macros [DEC82]. This is a significant interface issue because it requires bypassing the ALS KAPSE. Consequently, the portability of the AIM is reduced.

3.2.3 Non-Filtered Display Write

There are two mechanisms for doing writes: `BASIC_IO` and `TEXT_IO`.

The ALS KAPSE does no character translations of its own. Package `BASIC_IO` provides byte string read and write from the KAPSE to the VMS terminal driver. Again, however, the ALS does not allow the user to control the VMS device driver through the KAPSE. (See Appendix B, question 1.)

The VMS terminal driver has the potential to perform character string translations. For example, if a terminal uses an 8-bit ASCII character code and the `TTSM_EIGHTBIT` mode is not set, the device driver assumes a seven-bit code, masking out the eighth bit (dropping a bit from the received byte). Clearly, this behavior could alter an AIM transmission. VMS also allows syntax validation of escape sequences if `TTSM_ESCAPE` mode is set, which forces certain interpretations of AIM control sequences. The ALS KAPSE provides no services for setting these VMS terminal characteristic modes; the user must perform an `ESCAPE` to the underlying VMS operating system.

To write strings exactly as represented, the ALS KAPSE may be bypassed to access the VMS device driver and set the terminal characteristic `TTSM_PASSALL`. This mode ensures that all input and output is binary and that no interpretation whatsoever occurs in the device driver. Again, AIM transportability is significantly reduced by the VMS dependent services required. ([DEC82] p 9-19)

3.2.4 Screen-Oriented Facilities

As described above, the ALS defines only two auxiliary I/O packages to supplement Ada I/O. Neither of these packages mentions any screen manipulation procedures, and ARPANET communications confirmed that explicit x-y cursor positioning is not supported by the ALS. (Appendix B, question 2.)

3.2.5 Exclusive Access

The ALS documents were not clear on the subject of user terminal access. Verbal response from SofTech [RT83] indicated that exclusive access to the user terminal is not provided by the ALS.

The ALS defers to VMS which allows concurrent read and write access to multiple internal files which are associated with the terminal. The KAPSE provides no mechanism for obtaining exclusive access to the APSE program's terminal I/O (Appendix B, question 4). VMS, however, provides a device allocation service (\$ALLOC) which reserves the device for the exclusive use of the requesting process and its subprocesses. Again, this requires bypassing the ALS KAPSE.

3.2.6 Terminal Identification

The ALS documents do not describe a method for obtaining terminal identification, and verbal communication confirmed that this AIM requirement is not supported. [RT83]

3.3 PROCESS CONTROL AND COMMUNICATION

Sufficient Interfaces

1. Process initiation
2. Process termination
3. Process suspension/resumption
4. Process status query
5. SubProcess query

Insufficient Interfaces

1. Transparent interprocess communication

The AIM program control requirements are supported by the ALS as follows:

ADA LANGUAGE SYSTEM INTERFACES
PROCESS CONTROL AND COMMUNICATION

3.3.1 Process Initiation

Procedure CALL_WAIT is provided in package PROG_CALL to invoke a program and wait until the program completes its execution. Procedure CALL_NO_WAIT invokes a new APSE program allowing the caller to continue execution without waiting for the invoked program to complete: ([SOF83] p 90-154)

```
procedure CALL_WAIT
  (PROGRAM_NAME : in      KAPSE_DEFS.SHORT_ID_STRING;
   PROGRAM_FILE : in      KAPSE_DEFS.NODE_NAME;
   PARAMETER_LIST : in    PROG_DEFS.PARM_LIST_REC;
   STDIN_FILE : in      KAPSE_DEFS.NODE_NAME;
   STDOUT_FILE : in      KAPSE_DEFS.NODE_NAME;
   MSGOUT_FILE : in      KAPSE_DEFS.NODE_NAME;
   PROGRAM_STATUS : in out PROG_DEFS.CALL_STATUS_REC);
```

```
procedure CALL_NO_WAIT
  (PROGRAM_NAME : in      KAPSE_DEFS.SHORT_ID_STRING;
   PROGRAM_FILE : in      KAPSE_DEFS.NODE_NAME;
   PARAMETER_LIST : in    PROG_DEFS.PARM_LIST_REC;
   STDIN_FILE : in      KAPSE_DEFS.NODE_NAME;
   STDOUT_FILE : in      KAPSE_DEFS.NODE_NAME;
   MSGOUT_FILE : in      KAPSE_DEFS.NODE_NAME;
   PROGRAM_STATUS : in out PROG_DEFS.CALL_STATUS_REC);
```

3.3.2 Process Termination

A process and all of its subordinate processes may be terminated by use of the procedure REQ_ABORTION ([SOF83] p 90-177) in the package PROG_CONTROL.

```
procedure REQ_ABORTION
  (PROG_NAME : in      STRING_UTIL.VAR_STRING_REC;
   REQUEST_STATUS : out PROG_DEFS.PROCAL_STATUS_ENU);
```

3.3.3 Process Suspension/Resumption

A process and all of its subordinate processes may be suspended/resumed by use of the procedures REQ_SUSPENSION ([SOF83] p 90-172) and REQ_RESUMPTION ([SOF83] p 90-178) in the package PROG_CONTROL:

```
procedure REQ_SUSPENSION
  (PROG_NAME : in      STRING_UTIL.VAR_STRING_REC;
   REQUEST_STATUS : out PROG_DEFS.PROCAL_STATUS_ENU);
```

```
procedure REQ_RESUMPTION
  (PROG_NAME      : in      STRING_UTIL.VAR_STRING_REC;
   REQUEST_STATUS : out PROG_DEFS.PROCAL_STATUS_ENU);
```

3.3.4 Process Status Query

The status of a process can be determined by use of the procedure REQ_STATUS ([SOF83] p 90-173) in the package PROG_CONTROL.

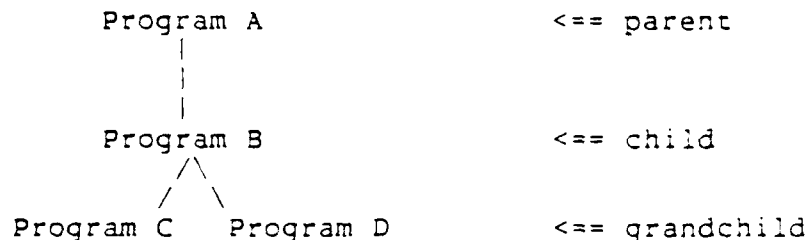
```
procedure REQ_STATUS
  (PROG_NAME      : in      STRING_UTIL.VAR_STRING_REC;
   PROGRAM_STATUS : out PROGRAM_INFO_REC);
```

3.3.5 SubProcess Query

Procedure REQ_STATUS (above) also provides the capability to query the call tree ([SOF83] p 90-173). The PROGRAM_INFO_REC contains descendant and sibling fields which indicate the presence of subprocesses, by listing subprocess names.

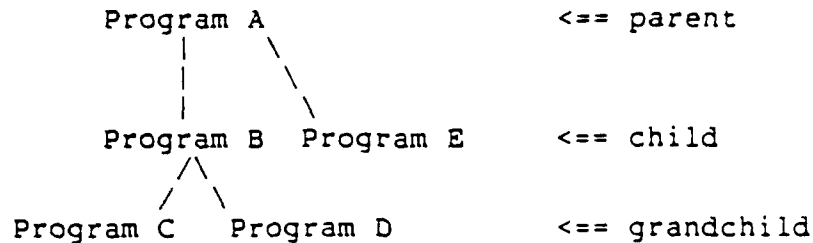
The user may repeatedly call REQ_STATUS to retrieve names one by one, in effect traversing the call-tree for information.

This approach to information retrieval has some interesting implications. The call-tree is a dynamic structure. It is not clear if the REQ_STATUS procedure takes a "snapshot" of the call-tree at a given instant, or (more likely) the call-tree continues changing as the status of various programs also change. A user that repeatedly calls REQ_STATUS to traverse the call-tree has no way of assuring that the information retrieved in one instant will be valid the next instant. For example, if the user wishes to count the total number of subordinate programs of a running program "A", a call to REQ_STATUS would indicate one child program, "B", directly under "A":



ADA LANGUAGE SYSTEM INTERFACES
PROCESS CONTROL AND COMMUNICATION

The user would then call REQ_STATUS again to determine if program "B" has any subordinate programs. While the user is querying program "B", program "A" could start another program, "E":



The user will never know about program "E" unless he or she performs another REQ_STATUS on program "A". Therefore, the program count will probably be invalid.

Fortunately, the AIM requires call-tree information to determine simply whether or not a program has subordinate programs, so repeated calls to REQ_STATUS and the possible problems described above are not anticipated.

3.3.6 Transparent Interprocess Communication

There are no provisions in the ALS for intercepting an APSE program's terminal I/O. However, the ALS provides program communication in the form of package PROG_COM. Package KAPSE_COM within package PROG_COM contains routines to create and utilize Interprocess channels:

1. Establish a logical connection between the calling program and another program ([SOF83] p 90-208).

```
procedure CONNECT
(CHANNEL      : out COM_DEFS.COM_CHANNEL_PRV;
 PATH_NAME    : in  STRING_UTIL.VAR_STRING_REC;
 PORT_NAME    : in  KAPSE_DEFS.SHORT_ID_STR;
 OPTIONS      : in  COM_DEFS.COM_OPTIONS_REC;
 STATUS       : out COM_DEFS.COM_STATUS_ENU;
 STATUS_STRING : in out STRING_UTIL.VAR_STRING_REC);
```

ADA LANGUAGE SYSTEM INTERFACES
PROCESS CONTROL AND COMMUNICATION

2. Accept a request to establish a communications channel
([SOF83] p 90-210).

```
procedure ACCEPT_CONNECTION
  (CHANNEL      : out COM_DEFS.COM_CHANNEL_Prv;
   PATH_NAME    : in  STRING_UTIL.VAR_STRING_REC;
   PORT_NAME    : in  KAPSE_DEFS.SHORT_ID_STR;
   OPTIONS      : in  COM_DEFS.COM_OPTIONS_REC;
   STATUS       : out COM_DEFS.COM_STATUS_ENU;
   STATUS_STRING : in out STRING_UTIL.VAR_STRING_REC);
```

3. Send a var_string across the specified channel
([SOF83] p 90-212).

```
procedure SEND
  (CHANNEL      : out COM_DEFS.COM_CHANNEL_Prv;
   BUFFER       : in  STRING_UTIL.VAR_STRING_REC;
   PORT_NAME    : in  KAPSE_DEFS.SHORT_ID_STR;
   OPTIONS      : in  COM_DEFS.COM_OPTIONS_REC;
   STATUS       : out COM_DEFS.COM_STATUS_ENU;
   STATUS_STRING : in out STRING_UTIL.VAR_STRING_REC);
```

Unfortunately, the routines outlined above require that both the calling and called processes must have provisions for interprocess communication. The called routine must accept data from the calling program, which is not acceptable for the AIM requirement of transparent communication.

To support this requirement, the AIM may have to bypass the ALS KAPSE to access underlying VMS mailboxes. The VAX Create Mailbox and Assign Channel (\$CREMBX) system allows processes to create channels for terminal I/O, which may allow the AIM to intercept an APSE program's I/O destined for the terminal. ([DEC82] p 8-2) Naturally, transportability is impaired whenever the KAPSE is bypassed.

The ALS makes no provisions for this capability. There is also no way to open an I/O file in SHARED_STREAM mode, as the AIE describes. As described above, the ALS KAPSE can be bypassed to access VMS mailboxes to accomplish this communication, which hinders transportability.

ADA LANGUAGE SYSTEM INTERFACES

DATABASE SERVICES

3.4 DATABASE SERVICES

3.4.1 File Manipulation

The ALS KAPSE Database Services are complete and sufficient for AIM implementation.

Sufficient Interfaces

1. Open/Close a database file
2. Read/Write to a database file
3. Create/Delete a database file

The ALS replaces and augments the file manipulation capabilities defined in TEXT_IO [DOD83] with the package BASIC_IO, which contains procedures to control the standard ALS I/O streams. Although the TEXT_IO provisions are sufficient for AIM needs, BASIC_IO enhances I/O for the ALS interface. The AIM database interface requirements are fulfilled as follows:

1. Open/Close a database file

The ALS provides open and close procedures in package BASIC_IO:
([SOF83] p 90-52,55)

```
procedure OPEN_FILE
  (STREAM      : out IO_DEFS.STREAM_ID_PRIV;
   NAME        : in  STRING_UTIL.VAR_STRING_REC;
   MODE        : in  IO_DEFS.IO_MODE_ENU;
   FILE_CLASS  : out IO_DEFS.FILE_CLASS_ENU;
   RECORD_FORMAT : out IO_DEFS.RECORD_FORMAT_ENU;
   RECORD_LENGTH : out IO_DEFS.DATA_INDEX_INT;
   RESULT      : out IO_DEFS.IO_RESULT_ENU;
   RESULT_STRING : in out STRING_UTIL.VAR_STRING_REC);
```

```
procedure CLOSE_FILE
  (STREAM      : in  IO_DEFS.STREAM_ID_PRIV;
   RESULT      : out IO_DEFS.IO_RESULT_ENU;
   RESULT_STRING : in out STRING_UTIL.VAR_STRING_REC);
```

2. Read/write to a database file

Package BASIC_IO in the ALS supports procedure READ_FILE which reads data from an open input file. Similarly, procedure WRITE_FILE writes data to an open output file:
([SOF83] p 90-57,59)

```

procedure READ_FILE
  (STREAM      : in      IO_DEFS.STREAM_ID_PRIV;
   BUFFER      : in      KAPSE_DEFS.ADDRESS_INT;
   LENGTH      : in      IO_DEFS.DATA_INDEX_INT;
   COUNT       :         out IO_DEFS.DATA_LENGTH_INT;
   RESULT      :         out IO_DEFS.IO_RESULT_ENU;
   RESULT_STRING : in out STRING_UTIL.VAR_STRING_REC);

```

```

procedure WRITE_FILE
  (STREAM      : in      IO_DEFS.STREAM_ID_PRIV;
   BUFFER      : in      KAPSE_DEFS.ADDRESS_INT;
   COUNT       : in      IO_DEFS.DATA_INDEX_INT;
   RESULT      :         out IO_DEFS.IO_RESULT_ENU;
   RESULT_STRING : in out STRING_UTIL.VAR_STRING_REC);

```

3. Create database files

File creation is also handled in package BASIC_I/O by procedure MAKE_FILE ([SOF83] p 90-48):

```

procedure MAKE_FILE
  (STREAM      : out IO_DEFS.STREAM_ID_PRIV;
   NAME        : in   STRING_UTIL.VAR_STRING_REC;
   MODE        : in   IO_DEFS.IO_MODE_ENU;
   FILE_CLASS  : in   IO_DEFS.FILE_CLASS_ENU;
   RECORD_LENGTH : in   IO_DEFS.DATA_INDEX_INT;
   RESULT      :         out IO_DEFS.IO_RESULT_ENU;
   RESULT_STRING : in out STRING_UTIL.VAR_STRING_REC);

```

This procedure associates an I/O stream with the new file.

4. Delete database files

Package BASIC_IO also supports file deletion by procedure DELETE_FILE: ([SOF83] p 90-51)

```

procedure DELETE_FILE
  (STREAM      : in      IO_DEFS.STREAM_ID_PRIV;
   RESULT      :         out IO_DEFS.IO_RESULT_ENU;
   RESULT_STRING : in out STRING_UTIL.VAR_STRING_REC);

```

The database file associated with the STREAM parameter is deleted, after all streams with which the file is associated are closed.

ADA LANGUAGE SYSTEM INTERFACES ENVIRONMENT SPECIFIC ISSUES

3.5 ENVIRONMENT SPECIFIC ISSUES

This section describes facilities provided in the ALS KAPSE which may affect the operation of the AIM program. The ALS provides the user with the capability to suspend program and terminal I/O from the terminal with the "break-in" facility. This allows the user to suspend the AIM itself. The user has control of the terminal and may randomly change screen data without AIM supervision. When AIM execution is resumed, the user may become confused because the AIM assumes that mappings between AIM images and the actual display are intact, when in reality they have been changed.

3.5.1 ALS "Break-In" Facility

The ALS allows a user to type <ctrl>-C to receive control of any currently running job [SOF83]. This gives the user control of the screen and terminal-directed I/O. Since the AIM is an APSE program, it may be suspended by the "break-in" facility, which returns control of the user's terminal I/O to the KAPSE. If the user changes the information on the screen and then resumes AIM execution, the results are unpredictable. The AIM makes certain assumptions about the screen and mappings among images, windows, and viewports, and if the user moves or deletes screen information, the display will not correctly reflect AIM mappings. Therefore, the use of the ALS "break-in" key is potentially hazardous to the AIM user.

If the "break-in" facility is suspendable, however, these problems could be alleviated because the AIM would then control the break-in. The break-in facility would even become useful for the AIM itself to control programs and terminal I/O. Conversely, the facility is designed as an "emergency" mechanism to provide the user absolute control, and implementing it as a program may defeat its purpose.

Another possibility is that the KAPSE might reinstate the screen status automatically upon program resumption. This would ensure that the AIM screen mappings would remain valid even if the AIM is suspended.

The AIM cannot automatically refresh the screen upon resumption because it has no knowledge of having been suspended. The user must choose to restore the screen display arbitrarily. The AIM itself will provide a screen refresh function invoked by a special key sequence [TI85A]. If the AIM is suspended for some reason (intentional or unintentional) and the screen data is modified, this function will reinstate the correct mappings between the internal AIM images and the screen.

3.5.2 Broadcast Messages

The ALS does not provide a mechanism for granting the AIM exclusive access to the user terminal. The absence of exclusive access enables the host operating system to generate system "broadcast" messages which may overwrite portions of the screen. The user must reinstate the screen with the AIM refresh function.

3.5.3 Bypassing KAPSE Services For Terminal Control

The ALS permits the user to perform an implicit escape to the underlying host operating system for some terminal control functions (such as opening the terminal file). The use of this feature may alter the appearance of the screen to incorrectly reflect AIM mappings. A program which bypasses KAPSE services in this manner is not a true APSE program and is considered erroneous.

3.5.4 Constraints On User Programs

Although the AIM is intended to be transparent to APSE programs, it is possible that APSE programs may be required to follow some guidelines in order to interface with the AIM, especially in the area of terminal I/O. This section describes some restrictions that might be imposed upon programs which are intended to execute under the AIM.

3.5.4.1 APSE Program I/O

If the appropriate KAPSE services are provided by the ALS, the AIM mechanism required to intercept program I/O will not impact APSE program design. APSE programs should need no special I/O file parameters to function in the AIM. APSE program terminal-destined I/O must be accomplished through the STANDARD_IN and STANDARD_OUT files; an APSE program which accesses the terminal in any other way might not be AIM-compatible. No logical name translation is provided to access a unique device. The ALS, however, allows the user to bypass the KAPSE to use VMS services to perform logical name translation.

An APSE program is by definition a program which uses only KAPSE services. Therefore, a program which bypasses KAPSE services for any reason is not an APSE program and therefore might not be AIM compatible. For example, the ALS permits the following:

```
Open ("<<VMS>>TT:")
```

This statement causes an implicit ALS ESCAPE to the underlying host operating system (VMS, in this case). Host services are used to open

ADA LANGUAGE SYSTEM INTERFACES
ENVIRONMENT SPECIFIC ISSUES

the terminal file.

3.5.4.2 MASTER_IN, MASTER_OUT, And MESSAGE_OUT

The ALS defines two extraneous files for I/O besides STANDARD_IN and STANDARD_OUT, called MSTR_IN and MSTR_OUT. These two extra files are always associated with the terminal. The purpose of these files was not clear from the documents. Verbal communications [RT83A] indicated that MSTR_IN and MSTR_OUT are provided to allow batch streams to send status messages to the terminal, such as "please mount tape". There is no way to disassociate MSTR_IN and MSTR_OUT from the terminal; these messages will always be directed to the screen. This could disrupt the user of an interactive program (such as the AIM or a text editor) if a batch stream sends a message which demands a response.

The AIM restricts APSE programs to use only a single pair of files for terminal I/O, specifically STANDARD_IN and STANDARD_OUT. Therefore, using MSTR_IN and MSTR_OUT by definition creates an APSE program which might not run correctly under the AIM. The ALS additionally defines MSG_OUT, a file which is always associated with the terminal (presumably intended for system messages, [RT83A]). Use of this file will also create problems when executing under the AIM.

CHAPTER 4

ADA INTEGRATED ENVIRONMENT INTERFACES

4.1 GENERAL

Intermetrics, Inc. is performing the design and implementation of the Ada Integrated Environment (AIE) under U.S. Air Force contract number F30602-80-C-0291. Neither the design nor the implementation of the AIE are complete at this time.

4.2 TERMINAL COMMUNICATION

The terminal communication interfaces are fairly well-defined and generally sufficient for the requirements of the AIM. Only the keyboard read is insufficient:

Sufficient Interfaces

1. Echo control
2. Non-filtered display write
3. Screen-oriented facilities
4. Exclusive access
5. Terminal identification

Insufficient Interfaces

1. Non-Filtered keyboard read

ADA INTEGRATED ENVIRONMENT INTERFACES
TERMINAL COMMUNICATION

4.2.1 Echo Control

The AIE provides package INTERACTIVE_IO as an extension to TEXT_IO. The following procedures are defined in this package for echo control:

```
procedure SET_ECHO
  (INPUT : in FILE_TYPE;
   OUTPUT : in FILE_TYPE);
```

```
procedure NO_ECHO
  (INPUT : in FILE_TYPE);
```

```
procedure NO_ECHO
  (OUTPUT: in FILE_TYPE);
```

SET_ECHO enables echo at the current line and column of input. NO_ECHO breaks echo associations for either input or output. ([INT82] p 73)

4.2.2 Non-Filtered Keyboard Read

ARPANET response from Intermetrics indicated that I/O operations to interactive devices are buffered to permit local line-editing before the characters are received as part of the text input file. (Appendix B, question 3) Buffers will be delimited by ENTER/Carriage Return characters. Projections indicate that the SET_INPUT_INFO procedure of package INTERACTIVE_IO will provide control of this buffering. ([INT82] p 73)

4.2.3 Non-Filtered Display Write

This functionality is defined in the package TEXT_IO by the following procedure: ([DOD83] p 14-19)

```
procedure PUT
  (FILE : in FILE_TYPE;
   ITEM : in STRING);
```

4.2.4 Screen-Oriented Facilities

Since the AIE treats the terminal display as a random text file, the package TEXT_ACCESS defines several procedures applicable to two-dimensional display manipulation. The AIM can simulate all the screen "editing" functions it requires with the primitives in this package. For example, a "clear to end of line" command can be implemented by positioning the cursor to the appropriate line and column, and writing blanks across the line. Included in package TEXT_ACCESS are: ([INT82] p 70)

ADA INTEGRATED ENVIRONMENT INTERFACES
TERMINAL COMMUNICATION

1. Procedure SET_OFFSET - selects the next read/write character.

```
procedure SET_OFFSET
(FILE : in FILE_TYPE;
 TO   : in COUNT);
```

2. Procedure SET_LINE - positions the cursor vertically at the beginning of a selected line.

```
procedure SET_LINE
(FILE : in FILE_TYPE;
 TO   : in COUNT);
```

3. Procedure SET_COL - positions the cursor horizontally at a selected column.

```
procedure SET_COL
(FILE : in FILE_TYPE;
 TO   : in COUNT);
```

4.2.5 Exclusive Access

The AIE documents presently contain minimal information about terminal I/O. The TEXT_IO package [DOD83] defines facilities for many I/O needs, but does not define a method for connecting the Standard_In and Standard_Out files to the terminal, as required by the AIM. Therefore, it is difficult to determine whether the AIM may be granted exclusive access to an APSE program's terminal I/O.

Although this requirement is not discussed in the documents, verbal communication with Intermetrics indicated that there will exist a method for obtaining exclusive access to the user terminal. [TT83]

4.2.6 Terminal Identification.

Package TERMINAL_IO is defined in the AIE for terminal interaction. The following primitives could potentially retrieve a terminal identification string: ([INT82] p 58)

```
procedure GET_TERMINAL_INFO
(TERM : in INTEGER;
 INFO : out TERMINAL_INFO_BLOCK);
```

If TERMINAL_INFO_BLOCK (yet undefined) contains a component such as TERMINAL_ID_STRING, these procedures would suffice for this interface requirement. Verbal communication [TT83] confirmed this assumption.

ADA INTEGRATED ENVIRONMENT INTERFACES PROCESS CONTROL AND COMMUNICATION

4.3 PROCESS CONTROL AND COMMUNICATION

The AIE defines package PROGRAM_INVOCATION to support program control requirements: ([INT82] pp 105-6).

Sufficient Interfaces

1. Process initiation
2. Process termination
3. Process suspension/resumption

Insufficient Interfaces

1. Process status query
2. SubProcess query
3. Transparent interprocess communication

4.3.1 Process Initiation

Function CALL_PROGRAM invokes an APSE program as a subprogram of the caller. The calling program is suspended until completion of the subprogram. Procedure INITIATE_PROGRAM invokes an APSE program in a manner similar to CALL_PROGRAM, but the calling program is not suspended:

```
function CALL_PROGRAM
  (PROGRAM_PATH : in STRING;
   PARAMETERS   : in PARAMS_STRING;
   CONTEXT_NAME : in STRING := ".SUE CONTEXT";
   STD_IN       : in TEXT_IO.FILE_TYPE := CURRENT_INPUT;
   STD_OUT      : in TEXT_IO.FILE_TYPE := CURRENT_OUTPUT)
  return RESULTS_STRING;
```

```
procedure INITIATE_PROGRAM
  (PROGRAM_PATH : in STRING;
   PARAMETERS   : in PARAMS_STRING;
   CONTEXT_NAME : in STRING;
   STD_IN       : in TEXT_IO.FILE_TYPE;
   STD_OUT      : in TEXT_IO.FILE_TYPE);
```

4.3.2 Process Termination

Procedure EXIT_PROGRAM stops program execution. A boolean parameter indicates whether to wait for subprocesses to complete, or to abort all subprocesses.

```
procedure EXIT_PROGRAM  
  (RESULTS      : in RESULTS_STRING;  
   ABORT_SUB_CONTEXTS : in BOOLEAN := FALSE);
```

4.3.3 Process Suspension/Resumption

Procedure SUSPEND_PROGRAM allows execution to be temporarily stopped.

```
procedure SUSPEND_PROGRAM  
  (CONTEXT_NAME : in STRING);
```

Procedure RESUME_PROGRAM restarts a suspended program.

```
procedure RESUME_PROGRAM  
  (CONTEXT_NAME : in STRING);
```

4.3.4 Process Status Query

There is no explicit provision in [INT82] for querying the status of a program. Package PROGRAM_INVOCATION contains procedure SUSPEND_PROGRAM, which stops the execution of the named program, "allowing the state of the execution to be examined" ([INT82] p 106). This procedure has no OUT parameters, so it is not clear how the status query is accomplished. It is also not clear if the state of execution is the current status, or the state of the program before suspension. Since it would be useless to query the status of a process that is known to be suspended, one would assume that the execution state consists of program information (such as register contents). Under this assumption, program status information is not available to the user.

4.3.5 SubProcess Query

Call-tree information services could not be found in the AIE documents. The ability to access call-tree information is implied by procedures such as EXIT_PROGRAM, which can wait for sub-contexts to complete. However, a procedure which explicitly allows the user to query the call-tree is not provided. ([INT82] p 106)

4.3.6 Transparent Interprocess Communication

The AIM desires channels for pipe I/O that are transparent to an APSE program. According to ARPANET response from Intermetrics, pipe I/O can be accomplished through package TEXT_IO, opening the file in SHARED_STREAM mode. The OPEN procedure allows a FORM string parameter which can be specified as SHARED_STREAM through a label=>value list: ([INT82] p 72) ([DOD83] 14.3.10) (Appendix B, question 4)

```
OPEN (FILENAME, IN_FILE, "RESERVE_MODE => SHARED_STREAM");
```

SHARED_STREAM mode allows synchronization of database object access so that WRITE ORIGINAL access is "reserved" (granted) only at the time of the READ or WRITE. ([INT82] p 94)

This approach to pipe I/O may not be acceptable for the AIM. The AIM requires that APSE programs use only the STD_IN and STD_OUT files for terminal I/O. These files are immediately opened upon program invocation, so normally the APSE program never calls the OPEN procedure. The AIE mechanism requires that the APSE user program explicitly open STD_IN and STD_OUT in SHARED_STREAM mode. This implies that the AIM pipe mechanism would not be transparent to user programs.

A more attractive alternative is for the AIM itself to open the file in SHARED_STREAM mode, and then pass the file to the APSE program as part of the INITIATE_PROGRAM call. The AIM pipe mechanism would then be transparent to the APSE program. Verbal communications indicate that this is a viable solution. [TT83A]

AIM interprocess communication consists of intercepting input and output from the APSE program executing in the environment. The AIM may be able to accomplish this with the provision for opening a file in SHARED_STREAM mode (see item 1 above).

It is not clear that the AIM will need further interprogram communications facilities, but for the sake of completeness, the existing AIE IPC interfaces are analyzed below.

The package INTER_PROGRAM_COMMUNICATION defines several functions and procedures for manipulating and controlling program I/O channels. The communicating programs must agree on the format and interpretation of PARAMS_STRING and RESULTS_STRING for interprogram communication. ([INT82] p 110)

1. Accept next waiting entry call:

ADA INTEGRATED ENVIRONMENT INTERFACES
PROCESS CONTROL AND COMMUNICATION

```
function IPC_ACCEPT  
  (CHANNEL_NAME : in STRING;  
   TIME_LIMIT   : in DURATION := DURATION'LAST)  
  return PARAMS_STRING;
```

2. Resume IPC_ENTRY_CALL after an IPC_ACCEPT:

```
procedure IPC_END_RENDEZVOUS  
  (CHANNEL_NAME : in STRING;  
   RESULTS      : in RESULTS_STRING);
```

3. Send data through channel:

```
function IPC_ENTRY_CALL  
  (CONTEXT_NAME : in STRING;  
   CHANNEL_NAME : in STRING;  
   TIME_LIMIT   : in DURATION := DURATION'LAST;  
   PARAMS       : in PARAMS_STRING)  
  return RESULTS_STRING;
```

4. Select channel for IPC:

procedure IPC_SELECT is named but not yet defined in this package.

Facilities to create channels are not provided in this package; however, Intermetrics has indicated verbally that channel creation will be provided. [TT83]

4.4 DATABASE SERVICES

4.4.1 File Manipulation

The KAPSE Database Services interfaces are sufficiently defined for the AIM implementation in the package TEXT_IO: ([DOD83] 14.3.10)

1. Open/Close a database file

Package TEXT_IO contains Open and Close procedures:
([DOD83] p 14-2,3)

ADA INTEGRATED ENVIRONMENT INTERFACES
DATABASE SERVICES

```
procedure OPEN
  (FILE : in out FILE_TYPE;
   MODE : in      FILE_MODE := OUT_FILE;
   NAME  : in      STRING;
   FORM  : in      STRING);
```

```
procedure CLOSE
  (FILE : in out FILE_TYPE);
```

2. Read/write data to a database file

The TEXT_IO package defined in [DOD83] contains PUT and GET procedures for file I/O which support variable length strings: ([DOD83] p 14-19) String I/O is accomplished by calls to PUT and GET single characters for the length of the string.

```
procedure PUT
  (FILE : in FILE_TYPE;
   ITEM : in STRING);

procedure GET
  (FILE : in      FILE_TYPE;
   ITEM : out     STRING);
```

3. Create database files

Procedure CREATE in TEXT_IO allows the AIM to create database file objects. ([DOD83] p 14-3)

```
procedure CREATE
  (FILE : in out FILE_TYPE;
   MODE : in      FILE_MODE := DEFAULT_MODE;
   NAME  : in      STRING := "";
   FORM  : in      STRING := "");
```

4. Delete database files

Procedure DELETE in package TEXT_IO is sufficient for file deletion. ([DOD83] p 14-4)

```
procedure DELETE
  (FILE : in out FILE_TYPE);
```

4.5 ENVIRONMENT SPECIFIC ISSUES

This section describes features provided in the AIE KAPSE which already have, or possibly may, (adversely) affect the operation of the AIM program.

4.5.1 Bypassing KAPSE Services For Program Control

The AIE provides the user with the capability to suspend program and terminal I/O from the terminal with "scroll mode control". This allows the user to suspend the AIM itself. The user has control of the terminal and may randomly change screen data without AIM supervision. When AIM execution is resumed, the user may become confused because the AIM assumes that mappings between AIM images and the actual display are intact, when in reality they have been changed.

The AIE extends the ALS "break-in" facility to include terminal I/O functions ([INT82] p 114). The user types a <CNTRL>-S to stop terminal output and enter scroll control mode. This mode is intended to provide a "cache" of output which the user may have lost from scrolling or printer malfunction. Once in scroll control mode, the user has control of terminal I/O and may scroll the screen or perform simple editing functions through a "terminal handler". The user may also interrupt program execution. All terminal input and output is stored in temporary files for historical purposes.

If a user invokes scroll control mode while under the AIM, the consequences are rather unpredictable. It is not clear if all running programs are automatically suspended, or if execution continues. Either result could adversely affect the function of the AIM.

4.5.2 Broadcast Messages

Intermetrics has indicated that the AIE will enable the AIM to obtain exclusive access of the user terminal I/O. The ALS and ADE do not provide a mechanism for granting the AIM exclusive access to the user terminal. The absence of exclusive access enables the host operating system to generate system "broadcast" messages which may overwrite portions of the screen. The AIM should allow system messages to be generated, but these messages may disturb the progress of an APSE programming session. The user's recourse is to reinstate the screen with the AIM refresh function described above.

CHAPTER 5

PROPOSED MIL-STD-CAIS

5.1 GENERAL

This interface report evaluates the specifications defined in the Proposed Military Standard Common APSE Interface Set (CAIS) "Proposed MIL-STD-CAIS" [KIT85] as developed by the KIT/KITIA CAIS Working Group. This section examines the CAIS interfaces with respect to the AIM requirements, identifying those which are insufficient or missing.

5.2 BACKGROUND

The Common APSE Interface Set (CAIS) [KIT85], is a result of an effort by "technical representatives from the two DoD APSE contractors, representatives from the DoD's Kernel Ada Programming Support Environment (KAPSE) Interface Team (KIT), and volunteer representatives from the KAPSE Interface Team from Industry and Academia (KITIA)" [KIT85]. The CAIS defines a set of interfaces that are designed to promote the source-level portability of Ada programs. The scope of the interfaces provided by the CAIS are "those services, traditionally provided by an operating system, that affect tool transportability." [KIT85] The CAIS addresses the areas of:

1. Name space management,
2. Process management,
3. File input/output,
4. Interactive terminal control, and
5. Magnetic tape control.

The AIM requires facilities in all of these areas except for magnetic tape control.

5.3 TERMINAL COMMUNICATION

Sufficient Interfaces

1. Echo control
2. Non-filtered keyboard read
3. Non-filtered display write
4. Screen-oriented facilities
5. Exclusive access
6. Terminal Identification

The CAIS defines packages that provide input/output services that are specific to terminal communications. These services are sufficient to supply the needs of the AIM. The requirements of the AIM with respect to interactive terminal services are enumerated in the following sections with an indication of the CAIS interfaces that support the desired functionality.

The preceding enumeration of terminal communication interfaces assumed that abstractions of terminals were not provided in the environment (as they were not provided in the AIE, ALS, or CAIS at the beginning of the AIM project). However, the CAIS provides abstractions for three types of terminals (Scroll, Page, and Form) that are almost identical to the terminal abstractions defined in the AIM. The abstractions provided by the CAIS in many cases remove the need for some of the previously enumerated interfaces. In particular, the Form terminal of the AIM can be implemented using the Form terminal of the CAIS, removing the need for the previously enumerated interfaces for the implementation of the Form terminal. Therefore, the Form terminal interfaces require no discussion and are excluded from this section.

5.3.1 Echo Control

The procedure SET_ECHO ([KIT85] pp 137,152) in the packages SCROLL_TERMINAL and PAGE_TERMINAL provide the capability to enable/disable echoing of characters for a specific terminal. The function ECHO ([KIT85] pp 138,153) in the packages SCROLL_TERMINAL and PAGE_TERMINAL provide the capability to determine whether the echoing of characters is enabled. The specifications for SET_ECHO and ECHO in both SCROLL_TERMINAL and PAGE_TERMINAL are syntactically identical.

```
procedure SET_ECHO
  (TERMINAL : in out FILE_TYPE;
   TO       : in      BOOLEAN := TRUE);
```

```
function ECHO
  (TERMINAL : in out FILE_TYPE)
  return BOOLEAN;
```

5.3.2 Non-Filtered Keyboard Read

The GET procedures ([KIT85] pp 139-141, 154-156) in the packages SCROLL_TERMINAL and PAGE_TERMINAL provide the capability of obtaining either (1) a single character or function key or (2) all available characters or functions keys. The GET procedures in SCROLL_TERMINAL and PAGE_TERMINAL are syntactically identical.

```
procedure GET -- a single character or function key
  (TERMINAL : in out FILE_TYPE;
   ITEM     : out CHARACTER;
   KEYS     : out FUNCTION_KEY_DESCRIPTOR);
```

```
procedure GET -- all available characters or function keys
  (TERMINAL : in out FILE_TYPE;
   ITEM     : out STRING;
   LAST     : out NATURAL;
   KEYS     : out FUNCTION_KEY_DESCRIPTOR);
```

Although the GET procedures do not guarantee that the data received is non-filtered, the function INTERCEPTED_CHARACTERS ([KIT85] pp 125) in the package IO_CONTROL permits the user to determine which characters will be filtered out. Use of the INTERCEPTED_CHARACTERS would permit the AIM to adjust itself to different environments.

```
function INTERCEPTED_CHARACTERS
  (TERMINAL : in out FILE_TYPE)
  return CHARACTER_ARRAY;
```

5.3.3 Non-Filtered Display Write

The overloaded PUT procedures in the packages SCROLL_TERMINAL and PAGE_TERMINAL ([KIT85] pp 136-137, 151-152) provide the capability for writing single characters and variable length strings to a terminal. The specifications for the PUT procedures are syntactically identical in both the SCROLL_TERMINAL and PAGE_TERMINAL packages.

```
procedure PUT
  (TERMINAL : in out FILE_TYPE;
   ITEM     : in      CHARACTER);
```


PROPOSED MIL-STD-CAIS
TERMINAL COMMUNICATION

```
procedure PUT
  (TERMINAL : in out FILE_TYPE;
   ITEM     : in     STRING);
```

5.3.4 Screen-Oriented Facilities.

The packages `SCROLL_TERMINAL`, `PAGE_TERMINAL`, and `FORM_TERMINAL` provide screen-oriented capabilities. The capabilities provided by these packages are sufficient to replace the virtual terminal portion of the AIM design.

The facilities provided include the operations provided by most currently manufactured character-imaging interactive terminals as well as the ability to query the size of the terminal display and query the position of the cursor. The minimal set of interfaces that would enable the AIM to be implemented on the CAIS are:

```
type CURSOR_POSITION is
  record
    LINE      : POSITIVE;
    COLUMN    : POSITIVE;
  end record;

function SIZE
  (TERMINAL : in FILE_TYPE)
  return CURSOR_POSITION;

function CURSOR
  (TERMINAL : in FILE_TYPE)
  return CURSOR_POSITION;

procedure SET_CURSOR
  (TERMINAL : in out FILE_TYPE;
   POSITION  : in     CURSOR_POSITION);
```

5.3.5 Exclusive Access

The `OPEN` procedures in the package `NODE_MANAGEMENT` ([KIT85] pp 38-39) in combination with the `OPEN` procedures in the package `CAIS.TEXT_IO` ([KIT85] pp 115-116) provide the capability of obtaining exclusive access to a terminal. The procedure `CHANGE_INTENTION` ([KIT85] pp 40-41) in the package `NODE_MANAGEMENT` provides the capability to obtain exclusive access to a node (file) that has already been opened.

```

procedure OPEN -- in package NODE_MANAGEMENT
  (NODE      : in out NODE_TYPE;
   NAME      : in      NAME_STRING;
   INTENT    : in      INTENTION := (1 => READ);
   TIME_LIMIT : in      DURATION := NO_DELAY);

procedure OPEN -- in package NODE_MANAGEMENT
  (NODE      : in out NODE_TYPE;
   BASE      : in      NODE_TYPE;
   KEY       : in      RELATIONSHIP_KEY;
   RELATION  : in      RELATION_NAME := DEFAULT_RELATION;
   INTENT    : in      INTENTION := (1 => READ);
   TIME_LIMIT : in      DURATION := NO_DELAY);

procedure OPEN -- in package CAIS.TEXT_IO
  (FILE : in out FILE_TYPE;
   NODE : in      NODE_TYPE;
   MODE : in      FILE_MODE);

procedure OPEN -- in package CAIS.TEXT_IO
  (FILE : in out FILE_TYPE;
   NAME : in      NAME_STRING;
   MODE : in      FILE_MODE);

procedure CHANGE_INTENTION
  (NODE      : in out NODE_TYPE;
   INTENT    : in      INTENTION;
   TIME_LIMIT : in      DURATION := NO_DELAY);

```

5.3.6 Terminal Identification

The CAIS provides facilities for the identification of a terminal's capabilities via the attributes of a node. The predefined attribute `TERMINAL_KIND` may be read to determine whether a terminal is a `SCROLL`, `PAGE`, or `FORM` terminal ([KIT85] p 102). The procedure `GET_NODE_ATTRIBUTE` ([KIT85] pp 69) in the package `ATTRIBUTES` provides the capability to read the `TERMINAL_KIND` attribute.

In addition, it would be possible to define a user attribute that identifies the protocol of the terminal. The procedure `SET_NODE_ATTRIBUTE` ([KIT85] pp 66-67) in the package `ATTRIBUTES` provide the capability to set the value of a node attribute.

```

procedure GET_NODE_ATTRIBUTE
  (NODE      : in      NODE_TYPE;
   ATTRIBUTE : in      ATTRIBUTE_NAME;
   VALUE     : in out LIST_TYPE);

```

PROPOSED MIL-STD-CAIS
TERMINAL COMMUNICATION

```
procedure SET_NODE ATTRIBUTE
  (NODE      : in NODE_TYPE;
   ATTRIBUTE : in ATTRIBUTE_NAME;
   VALUE     : in LIST_TYPE);
```

5.4 PROCESS CONTROL AND COMMUNICATION

Sufficient Interfaces

1. Process initiation
2. Process termination
3. Process suspension/resumption
4. Process status query
5. SubProcess query
6. Transparent interprocess communication

The package PROCESS_CONTROL provides the capabilities of process creation, termination, query, suspension, and resumption. The procedures which satisfy the AIM APSE program interfaces are as follows:

5.4.1 Process Initiation

The procedure SPAWN_PROCESS ([KIT85] pp 83-84) in the package PROCESS_CONTROL creates an APSE process as a "child" of the currently executing process. The procedure INVOKE_PROCESS ([KIT85] pp 85-87) in the same package also creates an APSE process as a "child" of the currently executing process, but suspends the task making the call until the process completes execution.

```
procedure SPAWN_PROCESS
  (NODE      : in out NODE_TYPE;
   FILE_NODE : in      NODE_TYPE;
   INPUT_PARAMETERS : in  PARAMS_STRING;
   KEY       : in      RELATIONSHIP_KEY := LATEST_KEY;
   RELATION  : in      RELATION_NAME := DEFAULT_RELATION;
   ACCESS_CONTROL : in  LIST_TYPE := EMPTY_LIST;
   LEVEL     : in      LIST_TYPE := EMPTY_LIST;
   ATTRIBUTES : in      LIST_TYPE := EMPTY_LIST;
   INPUT_FILE : in      NAME_STRING := CURRENT_INPUT;
   OUTPUT_FILE : in     NAME_STRING := CURRENT_OUTPUT;
   ERROR_FILE : in      NAME_STRING := CURRENT_ERROR;
   ENVIRONMENT_NODE : in  NAME_STRING := CURRENT_NODE);
```

```
procedure INVOKE_PROCESS
(NODE           : in out NODE_TYPE;
 FILE_NODE      : in   NODE_TYPE;
 RESULTS_RETURNED : in out RESULTS_LIST;
 STATUS         :      out PROCESS_STATUS;
 INPUT_PARAMETERS : in   PARAMS_STRING;
 KEY            : in   RELATIONSHIP_KEY := LATEST_KEY;
 RELATION        : in   RELATION_NAME := DEFAULT_RELATION;
 ACCESS_CONTROL  : in   LIST_TYPE := EMPTY_LIST;
 LEVEL          : in   LIST_TYPE := EMPTY_LIST;
 ATTRIBUTES      : in   LIST_TYPE := EMPTY_LIST;
 INPUT_FILE      : in   NAME_STRING := CURRENT_INPUT;
 OUTPUT_FILE     : in   NAME_STRING := CURRENT_OUTPUT;
 ERROR_FILE      : in   NAME_STRING := CURRENT_ERROR;
 ENVIRONMENT_NODE : in   NAME_STRING := CURRENT_NODE;
 TIME_LIMIT      : in   DURATION := DURATION'LAST);
```

5.4.2 Process Termination

The procedure ABORT_PROCESS ([KIT85] pp 93-94) in the package PROCESS_CONTROL provides the capability to abort (terminate) a process.

```
procedure ABORT_PROCESS
(NODE : in NODE_TYPE;
 RESULTS : in RESULTS_TYPE);
```

```
procedure ABORT_PROCESS
(NAME : in NAME_STRING;
 RESULTS : in RESULTS_TYPE);
```

```
procedure ABORT_PROCESS
(NODE : in NODE_TYPE);
```

```
procedure ABORT_PROCESS
(NAME : in NAME_STRING);
```

These procedures abort the process specified by NAME or NODE as well as all processes that are rooted at the specified process. In addition, the string RESULT is appended to the RESULTS attribute of the node represented by NODE or NAME.

5.4.3 Process Suspension/Resumption

The procedure SUSPEND_PROCESS ([KIT85] pp 94-95) in the package PROCESS_CONTROL provides the capability to suspend a process.

```
procedure SUSPEND_PROCESS
(NODE : in NODE_TYPE);
```

PROPOSED MIL-STD-CAIS
PROCESS CONTROL AND COMMUNICATION

```
procedure SUSPEND_PROCESS  
  (NAME : in NAME_STRING);
```

If the process identified by NODE (or NAME) is the parent of other process nodes, the other processes are likewise suspended.

The procedure RESUME_PROCESS ([KIT85] pp 95-96) in the same package provides the capability to resume a process.

```
procedure RESUME_PROCESS  
  (NODE : in NODE_TYPE);
```

```
procedure RESUME_PROCESS  
  (NAME : in NAME_STRING);
```

If the process identified by NODE (or NAME) is the parent of other process nodes, the other processes are likewise resumed.

5.4.4 Process Status Query

The function STATUS_OF_PROCESS ([KIT85] p 91) in the package PROCESS_CONTROL provides the capability to determine the status of a process.

```
function STATUS_OF_PROCESS  
  (NODE : in NODE_TYPE)  
  return PROCESS_STATUS;
```

```
function STATUS_OF_PROCESS  
  (PROCESS : in NAME_STRING)  
  return PROCESS_STATUS;
```

These procedures would enable the AIM to determine the status of the processes that it has initiated.

5.4.5 SubProcess Query

Call-tree information can be obtained by traversal of the nodes emanating from any process. These facilities are made available by the procedures and functions in the package NODE_MANAGEMENT. The procedure ITERATE ([KIT85] pp 58-59) creates an iterator that can be used with the procedure MORE ([KIT85] p 59) to determine if there are any more relationships emanating from the specified process, with GET_NEXT ([KIT85] pp 59-60) to obtain the next node in the list of nodes emanating from the specified process.

```

procedure ITERATE
  (ITERATOR      : out NODE_ITERATOR;
   NODE          : in  NODE_TYPE;
   KIND          : in  NODE_KIND;
   KEY           : in  RELATIONSHIP KEY_PATTERN := "";
   RELATION      : in  RELATION_NAME_PATTERN := DEFAULT_RELATION;
   PRIMARY_ONLY  : in  BOOLEAN := TRUE);

function MORE
  (ITERATOR : in NODE_ITERATOR)
  return BOOLEAN;

procedure GET_NEXT
  (ITERATOR : in out NODE_ITERATOR;
   NEXT_NODE : in out NODE_TYPE;
   INTENT    : in  INTENTION := (1=>EXISTENCE);
   TIME_LIMIT : in  DURATION := NO_DELAY);

```

5.4.6 Transparent Interprocess Communication

A facility for transparent interprocess communication is provided in the CAIS by queue nodes. The procedure CREATE ([KIT85] pp 113-115) in the package CAIS.TEXT_IO provides the ability to create a solo queue that may be used for interprocess communication. A solo queue may be "passed" to an initiated process as its standard input or standard output file such that the initiated process is not aware that the referenced node is a queue.

```

procedure CREATE
  (FILE      : in out FILE_TYPE;
   NAME      : in  STRING;
   MODE      : in  FILE_MODE := INOUT_FILE;
   FORM      : in  LIST_TYPE := EMPTY_LIST;
   ATTRIBUTES : in  LIST_TYPE := EMPTY_LIST;
   ACCESS_CONTROL : in  LIST_TYPE := EMPTY_LIST;
   LEVEL     : in  LIST_TYPE := EMPTY_LIST);

```

A solo queue node when opened as the standard input or standard output file is indistinguishable from that of a text file.

5.5 DATABASE SERVICES

5.5.1 File Manipulation

The name space of the CAIS is a multi-way tree of primary pathnames threaded by secondary pathnames. Each node in the tree may be any of the following: a file node, a structural node, or a process node. Each type of node has different "contents", but the operations on all types of nodes are similar. Nodes may be created and deleted and may

PROPOSED MIL-STD-CAIS
DATABASE SERVICES

contain files (as the content of a particular node). The following procedures fulfill the database service requirements of the AIM:

1. Open/Close a database file.

The procedures OPEN ([KIT85] pp 115-116) and CLOSE ([KIT85] p 113) in the package CAIS.TEXT_IO provide the capabilities for opening and closing a database file, respectively.

```
procedure OPEN
  (FILE : in out FILE_TYPE;
   NODE : in      NODE_TYPE;
   MODE : in      FILE_MODE);
```

```
procedure OPEN
  (FILE : in out FILE_TYPE;
   NAME : in      NAME_STRING;
   MODE : in      FILE_MODE);
```

```
procedure CLOSE
  (FILE : in out FILE_TYPE);
```

2. Read from/Write to database files.

The procedures GET ([KIT85] p 118) and PUT ([KIT85] p 118) in the package CAIS.TEXT_IO provide the capability to read and write database files. Following are two of the procedures representative of all those provided:

```
procedure GET
  (FILE : in      FILE_TYPE;
   ITEM : out STRING);
```

```
procedure PUT
  (FILE : in FILE_TYPE;
   ITEM : in STRING);
```

3. Create database files.

The procedure CREATE ([KIT85] pp 113-115) in the package CAIS.TEXT_IO provides for the creation of database files.

```
procedure CREATE
  (FILE      : in out FILE_TYPE;
   NAME      : in      STRING;
   MODE      : in      FILE_MODE := INOUT_FILE;
   FORM      : in      LIST_TYPE := EMPTY_LIST;
   ATTRIBUTES : in      LIST_TYPE := EMPTY_LIST);
```

```
ACCESS_CONTROL : in    LIST_TYPE := EMPTY_LIST;  
LEVEL          : in    LIST_TYPE := EMPTY_LIST);
```

4. Delete database files.

The procedure DELETE ([KIT85] pp 116-117) in the package CAIS.TEXT_IO provides for the deletion of database files.

```
procedure DELETE  
  (FILE : in out FILE_TYPE);
```

5.6 ENVIRONMENT SPECIFIC ISSUES

Implementors of the CAIS are given a great deal of freedom in the definition of the interfaces it defines. In many cases the freedom was intentionally granted, but there may be cases that were overlooked by the designers of the CAIS. This freedom means that there may be features of the environment which will not be anticipated in the writing of the AIM. However, until such environments are actually implemented nothing can be said about the "CAIS environment."

CHAPTER 6

ADA DEVELOPMENT ENVIRONMENT INTERFACES

6.1 GENERAL

The Data General Ada Development Environment (ADE) is an integrated set of tools, conventions, and underlying operating system features which collectively form a minimal Ada Programming Support Environment. References for the ADE include the Ada Development Environment (ADE)(AOS/VS) User's Manual [DAT84] and the Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual - Volume 1 [DAT83A] and Volume 2 [DAT83B]. The ADE is built on top of the Data General (DG) AOS/VS operating system and thus, provides its users with some of AOS/VS's more attractive features, including its:

1. hierarchical file system,
2. file security mechanism via the access control list (ACL) which is associated with every directory and file within the system,
3. directory searchlist mechanism, and
4. powerful filename template matching capability.

This section evaluates the ADE KAPSE interfaces with respect to the AIM's environmental requirements. Note that this evaluation is in actuality an evaluation of the underlying operating system's services since a true set of (Ada) KAPSE interfaces does not exist in the ADE.

This evaluation is also much more detailed than the others. This is due to the fact that the AIM implementation proceeded in this environment due to the apparent support of most of the needed functions. The evaluation was updated after the implementation was completed and, therefore, the information is completely up to date and reflects EXACTLY what was used to implement the AIM.

ADA DEVELOPMENT ENVIRONMENT INTERFACES TERMINAL COMMUNICATION

6.2 TERMINAL COMMUNICATION

Sufficient Interfaces

1. Echo control
2. Non-filtered keyboard read
3. Non-filtered display write
4. Screen-oriented facilities
5. Exclusive access
6. Terminal Identification

Insufficient Interfaces

- * None

The ADE defines several packages which support primitive functions required by the AIM terminal interface. These packages are:

- * SYS_CALLS - provides a callable set of Ada interfaces to the operating systems functions (including but not limited to I/O). This includes such things as: process control and communications; information services; memory management; file control (creation, deletion, renaming, etc); etc.
- * FILE_IO - a package that supplies interfaces to all file input/output operations in the AOS/VS operating system. This is the generalized case, from which other I/O packages can be (and probably were) implemented (It is presumed that this package was itself built on top of SYS_CALLS).
- * FILE_DEFINITIONS - the definitions required to support the FILE_IO package.
- * TTY_IO - a generalized package supporting console communications and control (probably built on top of FILE_IO).
- * BIT_OPS - supplies operations such as left/right shifting of words and bytes. This one is required to obtain "byte pointers" to strings for calls to FILE_IO, TTY_IO, and SYS_CALLS.

6.2.1 Echo Control

The capability to enable/disable echoing of characters for a specific terminal is provided for in the ADE. A procedure is called to retrieve the characteristics of the console. A field of the retrieved record is set to disable the automatic echo of characters. Then the characteristics are reset appropriately.

```
FILE_IO.GET_CHARACTERISTICS
  ( TERMINAL, CONSOLE_CHARACTERISTICS, ERROR_CODE );

CONSOLE_CHARACTERISTICS.ECHO := FILE_IO.NO_ECHO;

FILE_IO.SET_CHARACTERISTICS
  ( TERMINAL, CONSOLE_CHARACTERISTICS, ERROR_CODE );
```

This works similarly for turning the echo back on.

6.2.2 Non-Filtered Keyboard Read

Facilities exist to read a single character from the type-ahead buffer with no filtering (but still supporting XON/XOFF flow control). The procedure called to perform the read is:

```
FILE_IO.READ( TERMINAL,
               ERROR_CODE,
               BYTES_READ,
               BUFFER_BYTE_POINTER,
               FILE_DEFINITIONS.BINARY_IO,
               1 );
```

Where-

1. TERMINAL - the private type FILE_DEFINITIONS.CHANNEL_NUMBER which identifies to the operating system the device of interest.
2. ERROR_CODE - returned from the call.
3. BYTES_READ - a count of the number of bytes read in.
4. BUFFER_BYTE_POINTER - a pointer (address) of the starting byte position to which the characters will be read into. It is constructed by the Ada statement:

ADA DEVELOPMENT ENVIRONMENT INTERFACES
TERMINAL COMMUNICATION

```
BUFFER := BIT_OPS.LOGICAL_RIGHT_SHIFT( BUFFER, 24 );  
where -
```

BUFFER is an integer.

We store the character into an integer variable then extract it later to return it to the user. More on this later.

5. FILE_DEFINITIONS.BINARY_IO - this specifies the binary I/O mode to the operating system read operations. Again, XON/XOFF flow control remains in effect, all other characters typed at the console will come through to the program.
6. the "1" specifies the number of characters to read.

To support the task only blocking (that is, when a console GET is called, the calling task is blocked but not other tasks within the same process), the FILE_IO interfaces must be called by a task (not a block statement). To support such, an intermediate task needs to be created.

```
task TTY_SERVER is  
  entry GO;  
  entry START_GET;  
  entry GET( DATA : out STRING;  
            LAST : out NATURAL );  
end TTY_SERVER;
```

This server is called from the package level interface in the following manner.

```
procedure GET( DATA : out STRING;  
              LAST : out NATURAL ) is  
begin  
  TTY_SERVER.START_GET;  
  TTY_SERVER.GET( DATA, LAST );  
end GET;
```

The rendezvous TTY_SERVER.START_GET causes the I/O request to get started. It does this OUTSIDE of the rendezvous after the call on START_GET and before accepting the entry call on TTY_SERVER.GET. When TTY_SERVER.GET is called the calling task is blocked until the called task is ready to accept the rendezvous. This is due to the nature of Ada tasking, and by definition, will not block the calling task.

6.2.3 Non-Filtered Display Write

The procedure

```
TTY_IO.PUT( TERMINAL, DATA );
```

will put data to the console (or any channel specified) untranslated.

6.2.4 Screen-Oriented Facilities

The ADE does not provide a high level Ada package for supporting screen-oriented facilities. However, they do provide a SCREEN_IO package which supports simple screen-oriented facilities for Data General terminals.

An Ada package was imported into the AIM to provide the terminal-independent screen oriented facilities required by the AIM. Therefore, this interface can be considered sufficient.

6.2.5 Exclusive Access

It appears from the ADE manuals that a program (the AIM) can have exclusive access to a user's terminal. The interface

```
procedure FILE_IO.OPEN  
  ( FILE_NAME : in STRING;  
    CHANNEL   : out FD.CHANNEL_NUMBER;  
    ERROR     : out INTEGER;  
    OPTIONS   : in  INTEGER;  
    ...  
  );
```

where the OPTIONS field is specified with FILE_DEFINITIONS.EXCLUSIVE_OPEN and the FILE_NAME is "3CONSOLE" can open the terminal with exclusive access.

The AIM did not use this feature, a repaint entire screen was added to allow recovery from when the screen was trashed due to an unexpected message appearing on the console. In this way the user can still get messages (such as "system going down in 5 minutes") and recover the screen to the correct state with modest effort.

6.2.6 Terminal Identification

Through the use of TEXT_IO to a file, a means exists of determining the name of the terminal that the user is communicating with. It is the responsibility of the system manager or the user themselves to actually place a meaningful string into this terminal identification file.

ADA DEVELOPMENT ENVIRONMENT INTERFACES
PROCESS CONTROL AND COMMUNICATION

6.3 PROCESS CONTROL AND COMMUNICATION

Sufficient Interfaces

- * Process initiation
- * Process status query
- * SubProcess query

Insufficient Interfaces

- * Process termination
- * Transparent interprocess communication
- * Process suspension/resumption

6.3.1 Process Initiation

An AOS/VS system service ?PROC supports the creation of subprocesses. This system service can be called from Ada with a call on:

```
SYSTEM_OPERATION := SYS_CALLS.PROC;  
AC0 := 0;  
AC1 := 0;  
AC2 := INTEGER( PROCESS_RECORD'address );  
SYS_CALLS.SYS( SYSTEM_OPERATION, AC0, AC1, AC2, SYS_ERROR );
```

where -

- * SYSTEM_OPERATION - identifies which operating system service call is to be performed.
- * AC0 through AC2 - are registers to be set appropriately for the system service call.
- * SYS_ERROR - returns any error status.

The AIM uses the ?PROC service call to asynchronously spawn off CLI's and pass IPC files to them as "%INPUT" and "%OUTPUT".

The order of doing these operations is:

1. Create the input and output IPC files. These are real files that exist in the callers default directory.

2. Spawn the subprocess.
3. Open the input and output files (created in step 1 above).

6.3.2 Process Termination

In Superprocess mode, the ?TERM call can be used to terminate any AOS/VS process; otherwise, ?TERM can be used to terminate the execution of the calling process or any of its subordinates.

This system service can be called from Ada with a call on:

```
SYSTEM_OPERATION := SYS_CALLS.TERM;  
AC0 := PROCESS.PID;  
AC1 := 0;  
AC2 := 0;  
SYS_CALLS.SYS( SYSTEM_OPERATION, AC0, AC1, AC2, SYS_ERROR );
```

Where the definitions are as above, except AC0 contains the operating system process id number.

This system service has a significant bug in it. If the specified process has son processes at the time of the call, then control never returns to the calling process.

6.3.3 Process Suspension/Resumption

The AOS/VS system services ?BLKPR and ?UBLPR respectively suspend and resume the execution of an AOS/VS process.

This system service has a problem though. It requires that the calling process have Superprocess privilege to be able to suspend or resume a given process. This severely limits the usefulness of this system service for the AIM.

6.3.4 Process Status Query

The AOS/VS system service ?PSTAT returns execution status information for a given AOS/VS process.

This system service can be called from Ada with a call on:

ADA DEVELOPMENT ENVIRONMENT INTERFACES
PROCESS CONTROL AND COMMUNICATION

```
SYSTEM_OPERATION := SYS_CALLS.PSTAT;  
AC0 := _PROCESS.FID;  
AC1 := 0;  
AC2 := STATUS.address;  
SYS_CALLS.SYS( SYSTEM_OPERATION, AC0, AC1, AC2, SYS_ERROR );
```

The STATUS parameter, AC2, returns a record of information.

6.3.5 SubProcess Query

The AOS/VS system service ?PSTAT returns subordinate process status information via its ?PSSN bit array for a given AOS/VS process.

As described above, a call on ?PSTAT will return a record of information. There are (among other things) 30 consecutive bytes in this record that are used to store the subprocess number for processes subordinate to the calling process (sons). Each bit position in these 30 bytes corresponds to a unique process id number. If the bit is set, then the process with the corresponding bit position number as its process id, is a son of the calling process.

This system service has a significant problem. The nature of AOS/VS is to spawn new sons from a given process to perform almost every operation (especially when the parent process is the AOS/VS Command Language Interpreter (CLI)). Because of this, it is not possible to guarantee after a given call on the system service, and before the caller acts on the returned information, that the number of son processes and their associated ids have not changed.

6.3.6 Transparent Interprocess Communication

AOS/VS supports a limited form of Interprocess Communication via IPC files. It is limited due to the fact that an IPC file passed into the standard input and output of a spawned son has the ability to communicate through these "pipes" with the parent. However, if this son himself spawns a son (a "grandson" of the original calling process) and passes the same IPC files for standard input and output to this son, the parent does NOT have the capability to communicate with the grandson. This is a fundamental operating system design decision made by the AOS/VS design engineers when the operating system was being developed.

In order to communicate transparently with a son process, one must:

1. Create the IPC file. To perform this, a call on a system service must be made.

ADA DEVELOPMENT ENVIRONMENT INTERFACES
PROCESS CONTROL AND COMMUNICATION

```
SYSTEM_OPERATION := SYS_CALLS.CREATE;  
AC0 := INFILE_PTR;  
AC1 := 0;  
AC2 := TEMP_IPC_PACKET'address;  
SYS_CALLS.SYS( SYSTEM_OPERATION, AC0, AC1, AC2, SYS_ERROR );
```

Where -

- * AC0 - contains a byte pointer to the name of the input file (to be created). This can be any valid filename not already in use.
- * TEMP_IPC_PACKET - specifies the default characteristics of the IPC file.

2. Create the process - as described in the Process Initiation section.
3. Open the IPC files. Do this once for each of the input and output IPC files. The Ada call is:

```
FILE_IO.OPEN  
( FILE_NAME => IPC_INFILE,  
  CHANNEL => TEMP_IN_CHANNEL,  
  ERROR => ERROR,  
  OPTIONS => ( FILE_DEFINITIONS.OPEN_FOR_OUTPUT +  
               FILE_DEFINITIONS.FORCE_OUTPUT +  
               FILE_DEFINITIONS.IPC_NO_WAIT +  
               FILE_DEFINITIONS.DATA_SENSITIVE ),  
  SHARED_OPEN => TRUE,  
  FILE_TYPE => FILE_DEFINITIONS.IPC_FILE );
```

The call is similar for opening the output IPC file.

6.4 DATABASE SERVICES

6.4.1 File Manipulation

The KAPSE Database Services interfaces are sufficiently defined for the AIM implementation in the package TEXT_IO: ([DOD83] 14.3.10)

1. Open/Close a database file

Package TEXT_IO contains Open and Close procedures:
([DOD83] p 14-2,3)

ADA DEVELOPMENT ENVIRONMENT INTERFACES
DATABASE SERVICES

```
procedure OPEN
  (FILE : in out FILE_TYPE;
   MODE : in      FILE_MODE := OUT_FILE;
   NAME : in      STRING;
   FORM : in      STRING);
```

```
procedure CLOSE
  (FILE : in out FILE_TYPE);
```

2. Read/write data to a database file

The TEXT_IO package defined in [DOD83] contains PUT and GET procedures for file I/O which support variable length strings: ([DOD83] p 14-19) String I/O is accomplished by calls to PUT and GET single characters for the length of the string.

```
procedure PUT
  (FILE : in FILE_TYPE;
   ITEM : in STRING);
```

```
procedure GET
  (FILE : in      FILE_TYPE;
   ITEM : out     STRING);
```

3. Create database files

Procedure CREATE in TEXT_IO allows the AIM to create database file objects. ([DOD83] p 14-3)

```
procedure CREATE
  (FILE : in out FILE_TYPE;
   MODE : in      FILE_MODE := DEFAULT_MODE;
   NAME : in      STRING := "";
   FORM : in      STRING := "");
```

4. Delete database files

Procedure DELETE in package TEXT_IO is sufficient for file deletion. ([DOD83] p 14-4)

```
procedure DELETE
  (FILE : in out FILE_TYPE);
```

APPENDIX A
AIM INTERFACES SUMMARY

INTERFACE COMPARISON

TERMINAL COMMUNICATION

AIM Interface Requirements	ALS	AIE	CAIS	ADE
Echo control	No	Yes	Yes	Yes
Nonfiltered keyboard read	No	No	Yes?	Yes
Nonfiltered display write	No	Yes	Yes?	Yes
Screen-oriented facilities	No	Yes	Yes	Yes
Exclusive access	No	Yes	No	Yes
Terminal identification	No	Yes	Yes	Yes

PROCESS CONTROL AND COMMUNICATION

AIM Interface Requirements	ALS	AIE	CAIS	ADE
Process initiation	Yes	Yes	Yes	Yes
Process termination	Yes	Yes	Yes	Yes*
Process suspension/resumption	Yes	Yes	Yes	Yes*
Process status query	Yes	No	Yes	Yes
Subprocess query	Yes	No?	Yes	Yes*
Transparent interprocess communication	No	Yes	Yes	Yes*

Yes => Support provided
 Yes* => Limited support provided
 No => No support provided
 ? => Could not determine if support is provided

AIM INTERFACES SUMMARY

DATABASE SERVICES

AIM Interface Requirements	ALS	AIE	CAIS	ADE
Open/close a file	Yes	Yes	Yes	Yes
Read from/write to a file	Yes	Yes	Yes	Yes
Create/delete a file	Yes	Yes	Yes	Yes

Yes => Support provided

No => No support provided

? => Could not determine if support is provided

APPENDIX B
ARPANET COMMUNICATIONS

Confusion about document content spawned the following question and answer exchange between Texas Instruments and the APSE contractors Intermetrics (AIE) and SofTech (ALS). Most of this information pertains directly to KAPSE interfaces, so it is included in transcribed form. (Answers are dated 8 Nov 82 for AIE, 12 Nov 82 for ALS.)

1. Question:

Within the KAPSE is there a facility for directly referencing an interactive device? (ie. can character sequences be sent to and received from the device without any translation?)

AIE: The initial KAPSE/Tool interfaces include no mechanism for direct reference to an interactive device. Instead, full-screen terminals are made to look like a text file with random access to line and column (see below). -

ALS: In the ALS, interactive devices can be referenced in two ways:

- a. Explicitly open the "file" named "<<VMS>>TT:", where TT: is the name that VMS assigns to the device, in this case the terminal.
- b. If you want the terminal that the user is connected to, use the predefined and preopened "files" named .MSTRIN (keyboard) and .MSTROUT (terminal display device).

Once open, you will be able to use `basic.io.read_file` and `basic.io.write_file` to pass byte strings to and from the DEVICE DRIVER. The ALS KAPSE will not do any translation of the bytestrings. However, you will have to get by the VMS device driver. This could be the subject of a VMS experiment. The ALS KAPSE does not support any official way of opening a device in

"raw" mode. If you can do it by passing bytestrings to the opened devices, then it can be done, otherwise not. I do not know the nature of the character translation performed by the VMS device driver.

2. Question:

Does the KAPSE support any functionality for interactive devices other than teleprinters? Are there any multidimensional capabilities, for example, cursor positioning?

AIE: The KAPSE supports x-y cursor positioning using the primitives of the package SIMPLE_OBJECTS.TEXT_ACCESS (see KAPSE B5 IR-678-1, p. 24), SET_LINE, SET_COL.

ALS: Unless it can be done by passing a byte string, there is no explicit x-y cursor positioning supported by the ALS KAPSE. The notion of a two dimensional display is not supported by the KAPSE. However, the Ada TEXT_IO package should work for CRT based terminals.

3. Question:

Are I/O operations to interactive devices buffered? Must NEW_LINE or PUT_LINE be called before the text is actually sent to the device?

AIE: It is our intention that I/O may be buffered. Probably an end-of-line will cause flushing, but in any case, requesting input from a file which is echoing on the output file (see B5 p. 65, package INTERACTIVE_IO), will cause a flush. Input is buffered up so that local_line-editing may be performed before the characters are received as part of the text input file. The initial KAPSE will probably always buffer up input until an ENTER/Carriage Return key is depressed. Eventually, using the SET_INPUT_INFO call of Package INTERACTIVE_IO, more control will be available.

ALS: Keyboard input is buffered by VMS which does the line editing. In general, the KAPSE sees no keyboard input until the line is sent by use of the return key. The exceptions to this are some of the special control operations like control-C and control-Y used for interruption; but these are very special-purpose operations. For most of the standard DEC terminals, the CPU sees each keystroke. I believe that the device driver performs the buffering, not the hardware. The ALS KAPSE does no input buffering itself.

For output, buffering is performed when using Ada TEXT_IO. A new_line or put_line is necessary to obtain the transmission of the characters buffered. Characters are also transmitted when

the line length is exceeded. Presumably, the length could be set to 0 or 1, but this would cause the insertion of the line mark after each character. Basic io.write_file performs no buffering. Every call to this service will result in transmission to the device driver.

4. Question:

Can two or more logical devices have concurrent access rights to an interactive device? (Two "internal files" referencing the user's terminal.)

AIE: It will probably be undefined what happens when two programs/tasks try to read from the same terminal input stream (and hence "erroneous" if not an explicit exception). To accomplish your task, I would recommend that your virtual terminal manager be the only process with the terminal input/terminal output open, and that all other processes do interactive I/O by using pipes to the virtual terminal manager. The KAPSE allows multiple (Ada) tasks within the same program to each be doing synchronous pipe I/O, with only the particular task suspended which is actually waiting for input.

Pipe I/O is accomplished using normal TEXT_IO, but with the pipe/file opened in "SHARED_STREAM" mode (see B5 pp. 40, 41 for explanation of Shared Stream read/write, and p. 25 for explanation of use of FORM string for RESERVE_MODE specification).

ALS: If the device is a terminal, VMS will allow concurrent read and write access by multiple "internal files".

5. Question:

Does the AIE work with 3270-compatible devices?

AIE: The AIE will support 3270 compatible terminals, but will not initially support the field protect/field read features in a way that is useful to the application programmer. Instead, the terminals will be made to look as much like a full-screen ASCII terminal with cursor addressing.

ALS: N/A

APPENDIX C

GLOSSARY

ADE

Ada Development Environment

AIE

Ada Integrated Environment

AIM

APSE Interactive Monitor

ALS

Ada Language System

AOS/VS

Advanced Operating System/Virtual Storage, a Data General Corporation operating system for the ECLIPSE MV/Family of machines.

APSE

Ada Programming Support Environment

APSE program

A program that can be executed in the hosting APSE and uses only KAPSE supplied services to perform its function.

CAIS

Common APSE Interface Set, the KIT/KITIA effort to standardize certain KAPSE interfaces.

CAISWG

Common APSE Interface Set Working Group, a working group within the KIT/KITIA effort.

GLOSSARY

character

A member of a set of elements that is used for the organization, control, or representation of data.

character echo

The act of re-transmitting a character immediately upon receipt of it back to the entity that originally transmitted it.

character imaging device

A device that gives a visual representation of data in the form of graphic symbols using any technology, such as cathode ray tube or printer.

character stream

An unbounded sequence of ASCII characters.

character string

A bounded sequence of ASCII characters.

command script

A database file containing commands to the AIM command interpreter. The command interpreter reads commands from the command script rather than prompting the user interactively.

database file

A standard file in the APSE database.

DG

Data General

display

The area for visual presentation of data on a character imaging device.

display terminal

A data communications device composed of a keyboard and a display screen (usually a cathode ray tube).

EDT

An interactive full-screen editor supported by DEC on the VAX machine.

environment-dependent

Using features which are unique to a specific Ada Program Support Environment (such as ALS or AIE).

erroneous

An Ada program which does not conform to the requirements of an APSE program. The program might execute correctly within an APSE in a given situation, but the program may not be considered entirely reliable. An APSE program must use only KAPSE services; any other services (such as host services) result in an erroneous program.

exclusive access

Control of a file (or, the terminal, in this case) which prohibits any other program besides the AIM from writing to the terminal screen.

host services

Facilities provided by the operating system of the host machine underlying the KAPSE.

image

An analog of the physical display device. The image is the entity that is mapped onto the display. Given a number of user defined images, only one at a time can be mapped onto the display. The rest exist and are updated asynchronously but are not mapped onto the display until the user requests it.

interface

The place at which independent systems meet and act on or communicate with each other.

IPC

Interprocess communication.

KAPSE

Kernel Ada Programing Support Environment.

keyboard

The physical input device.

KIT

KAPSE Interface Team.

KITIA

KAPSE Interface Team from Industry and Academia.

LALR

Lookahead Left to Right; a method for parsing grammars.

GLOSSARY

line

A set of adjacent character positions in a visual display that have the same vertical position.

mappings

The relationships managed by the AIM connecting logical representations of windows, images, and viewports to physical representations on a display device.

MIL-STD

Military Standard.

node

Pertaining to the KAPSE database, either a file or a directory in the tree-structured database.

NOSC

Naval Ocean Systems Center

pad

Two files which contain a complete history of window activity that transpires from the beginning of pad mode until it is terminated by the user or the window is destroyed. One pad, the INPUT pad, includes the input to the APSE program from the user through the keyboard. The other pad, the OUTPUT pad, logs the output to the display from the AIM and any program initiated by the AIM.

Page mode terminal

A screen-oriented display device which possesses extended two-dimensional functional capabilities. Characters are transmitted and received one at a time.

pipe

A logical connection between an output file of one program and an input file of another program.

screen

The area for visual presentation of data on any type of character imaging device, including printer and cathode ray tube device.

STANDARD IN and STANDARD OUT

Input and output files defined in the package TEXT_IO. For AIM purposes, these must be the only files used for terminal I/O.

task

An Ada program unit that operates in parallel with other program units.

terminal

A data communications device consisting of a keyboard and a character imaging device.

Terminal Capabilities File

A file which describes common terminal functions in terms of device-specific control sequences, for many different terminals.

terminal communication protocols

Sequences of characters in which the relationships between specific characters are given meanings for different types of terminals.

transmit

To send data as a data stream for purposes of information interchange.

user terminal

The terminal with which a user interacts in order to communicate with an APSE program.

VMS

Virtual Memory System, the DEC operating system for the VAX 11-780.

viewport

The portion of the window displayed in the image.

viewport header

A single highlighted line located at the top of a viewport.

window

An analog of the APSE program's view of the terminal.

APPENDIX D

REFERENCES

D.1 GOVERNMENT STANDARDS

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [DOD80] United States Department of Defense, "Requirements for Ada Programming Support Environments" ("STONEMAN"), February 1980.
- [DOD83] United States Department of Defense, "Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983," February 17, 1983.
- [DID73] Data Item Description, "Informal Technical Information, DI-S-30593," March 73.

D.2 GOVERNMENT SPECIFICATIONS

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the document referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [INT82] Intermetrics Inc., "IR-678-1 Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type B5," Wakefield, MA, November 1982.
- [KIT83] KAPSE Interface Team (Ada Joint Program Office), "Common APSE Interface Set", Version 1.1, September 1983.

REFERENCES

GOVERNMENT SPECIFICATIONS

- [KIT85] KAPSE Interface Team (Ada Joint Program Office), "Proposed Military Standard Common APSE Interface Set (CAIS)", January, 1985.
- [SOF82] SofTech Inc., Ada Problem Report 602, Waltham, MA, November 1982.
- [SOF83] SofTech Inc., "Draft Ada Language System Specification," Waltham, MA, November 28, 1983

D.3 OTHER GOVERNMENT DOCUMENTS

The following documents of the latest issue per date of this report form a part of this specification.

- [TI82] Texas Instruments, Advanced Computer Systems Laboratory, "Proposal for Development of Ada Software Tools and Interface Standards," Lewisville, TX, February 1982.
- [TI83A] Texas Instruments, "APSE Interactive Monitor (AIM) Program Performance Specification (PPS)," Contract N66001-82-C-0440, 19 September 1983.
- [TI83B] Texas Instruments, "APSE Interactive Monitor (AIM) Software Development Plan (SDP)," Contract N66001-82-C-0440, 10 July 1983.
- [TI83C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Plan (SITP)," Contract N66001-82-C-0440, 23 December 1983.
- [TI83D] Texas Instruments, "APSE Interactive Monitor (AIM) Software Quality Assurance Plan (QA)," Contract N66001-82-C-0440, 28 March 1983.
- [TI83E] Texas Instruments, "APSE Interactive Monitor (AIM) Computer Program Test Specification (CPTS)," Contract N66001-82-C-0440, 15 September 1983.
- [TI83F] Texas Instruments, "APSE Interactive Monitor (AIM) Configuration Management Plan (CM)," Contract N66001-82-C-0440, 28 March 1983.
- [TI83G] Texas Instruments, "Interim Report on Interface Analysis and Software Engineering Techniques," Contract N66001-82-C-0440, May 1983.

REFERENCES
OTHER GOVERNMENT DOCUMENTS

- [TI83H] Texas Instruments, "Interim Report on Interface Analysis and Software Engineering Techniques," Contract N66001-82-C-0440, December 1983.
- [TI85A] Texas Instruments, "APSE Interactive Monitor (AIM) User's Manual (UM)," Contract N66001-82-C-0440, July 1985.
- [TI85B] Texas Instruments, "APSE Interactive Monitor (AIM) Program Design Specification (PDS)," Contract N66001-82-C-0440, July 1985.
- [TI85C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Procedures (SITPRO)," Contract N66001-82-C-0440, July 1985.
- [TI85D] Texas Instruments, "CAIS Rationale," Contract N66001-82-C-0440, July 1985.
- [TI85E] Texas Instruments, "Transportability Guide," Contract N66001-82-C-0440, July 1985.
- [TI85F] Texas Instruments, "Installation and Maintenance Guide for the APSE Interactive Monitor (AIM)," Contract N66001-82-C-0440, July 1985.
- [TI85G] Texas Instruments, "APSE Interactive Monitor (AIM) Acceptance Test Plan (ATP)," Contract N66001-82-C-0440, 15 July 1985.
- [TI85H] Texas Instruments, "APSE Interactive Monitor (AIM) Acceptance Test Procedures (ATPRO)," Contract N66001-82-C-0440, 15 July 1985.

D.4 SPECIAL SOURCES

- [RT83] Verbal communications with Rich Thall of SofTech, Inc., Jan 26, 1983 at the San Diego KIT meeting.
- [RT83A] Verbal communications with Rich Thall of SofTech, Inc., April 20, 1983 at the Willow Grove, PA KIT meeting.
- [TT83] Verbal communications with Tucker Taft of Intermetrics, Inc., Jan 26, 1983 at the San Diego KIT meeting.

REFERENCES
SPECIAL SOURCES

- [TT83A] Verbal communications with Tucker Taft of Intermetrics, Inc.. April 21, 1983 at the Willow Grove, PA KIT meeting.

D.5 OTHER PUBLICATIONS

- [ABB82] Abbott, Russell J., "Program Design by Informal English Descriptions," Unpublished.
- [AKIN81] Akin, T. Allen, "Virtual Terminal Handler Preliminary Quick Reference," School of Information and Computer Science, Georgia Institute of Technology, April 1981.
- [ANSI73] American National Standards Institute, "American National Standard Graphic Representation of the Control Characters of American National Standard Code for Information Interchange (ANSI Standard X3.32-1973)," July 1973.
- [ANSI77] American National Standards Institute, "American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)," June 1977.
- [ANSI79] American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)," July 1979.
- [APSE82] "Working Paper: Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for Standard Interface Specifications," Not Approved, October 1982.
- [BOO83] Booch, Grady., Software Engineering with Ada. Benjamin Cummings Publishing Company, Menlo Park, CA. Copyright 1983.
- [BOR85] Borger, Mark W., "Software Design Issues in Ada," Journal of Pascal, Ada, and Modula2, Volume 3, Number 3, March-April 1985.
- [BUH84] Buhr, R. J. A., System Design with Ada, Prentice-Hall, Inc., 1984.
- [COX83] Cox, Fred, "KAPSE Support for Program/Terminal Interaction", Working paper for KITIA/ Working Group 1, February 1983.

REFERENCES
OTHER PUBLICATIONS

- [CSC82A] Computer Sciences Corporation, "Configuration Management System Program Performance Specification (Draft)," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract N00123-80-D-0364.
- [CSC82B] Computer Sciences Corporation, "Configuration Management System Interim Report on Interface Analysis," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract N00123-80-D-0364.
- [DAT83A] Data General Corporation, "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual Volume 1 System Concepts", Westborough, Massachusetts, March 1983.
- [DAT83B] Data General Corporation, "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual Volume 2 System Calls", Westborough, Massachusetts, March 1983.
- [DAT84] Data General Corporation, "Ada Development Environment (ADE) (AOS/VS) User's Manual", Westborough, Massachusetts, March 1983.
- [DEC82] Digital Equipment Corporation, "VAX/VMS I/O User's Guide (Volume 1)", Maynard, Massachusetts, May 1982.
- [DP82] Datapro Reports on Data Communications, vol 2., Sept 1982, "Display Terminals", p C25-10-101
- [ELS73] Elson, Mark., Concepts of Programming Languages, Science Research Associates, Inc. Paris, France 1973.
- [FH83] French, Stewart and Harrison, Tim, "The APSE Interactive Monitor," Texas Instruments, Inc., March 1983.
- [FOR83] Foreman, John, "Experiences With Object-Oriented Design," AdaTEC, Cherry Hill, NJ, June 1983.
- [FRA] Franck, R., "Design and Implementation of a Virtual Terminal for a Real-time Application System"
- [FRE83] French, Stewart L., "A Virtual Terminal Specification and Rationale," IEEE Proceedings, 7th International Computer Software and Applications Conference, COMPSAC 83, November 7-11, 1983.
- [GOL83] Goldberg, A. and Robson, D., SMALLTALK-80 The Language and its Implementation, Addison-Wesley Publishing Company, Reading, MA, 1983.

REFERENCES
OTHER PUBLICATIONS

- [GOO75] Goodenough, John B., "Exception Handling Design Issues," ACM SIGPLAN Notices, July 1975, pp 41-45. Association for Computing Machinery, Inc.
- [GREN80] Greninger, Lars and Roberts, Roger, "Considerations for a Local Virtual Terminal Interface," Presented at IEEE Conference, September 1980.
- [GRR80] Groves, L.J. and Rogers, W.J., "The Design of a Virtual Machine for Ada," Communications of the ACM, 1980.
- [HAP83] Habermann, A.N., and Perry, D.E., Ada For Experienced Programmers, Addison-Wesley Publishing Company, 1983.
- [HOA81] Hoare, C.A.R., "The Emperor's Old Clothes," 1980 ACM Turing Award Lecture, Communications of the ACM, Vol 24 No 2, Feb 1981.
- [ISO642] International Standards Organization. Standard number: ISO DP 6429, "Additional Control Functions for Character Imaging Devices (Draft)," Not approved, April 1982.
- [JOY81] Joy, W. and Horton, M., "TERMCAP," UNIX Programmer's Manual, Seventh Edition, Berkeley release 4.1, June 1981.
- [LAN79A] Lantz, Keith A., et.al., "RIG: An Overview, Working Paper," University of Rochester, Rochester, NY, 1979.
- [LAN79B] Lantz, Keith and Rashid, Richard, "Virtual Terminal Management in a Multiple Process Environment," Proceedings of the Seventh Symposium on Operating Systems Principles, (December 10-12, 1979).
- [LAW78] Lawson, James T. and Mariani, Michael P., "Distributed Data Processing System Design - A Look at the Partitioning Problem," IEEE Press, 1978.
- [LOV81] Loveman, David, "Ada Resolves the Unusual with 'Exceptional' Handling," Electronic Design, January 22, 1981.
- [MAC81] MacEwen, Glen H. and Martin, T. Patrick, "Abstraction Hierarchies in Top-Down Design," The Journal of Systems and Software 2, 213-224(1981), Elsevier Science Publishing Co.
- [MAG79] Magnee, F., Endrizzi, A., and Day, J., "A Survey of Terminal Protocols," Computer Networks, 1979, pp 299-314.

REFERENCES
OTHER PUBLICATIONS

- [MEY81] Meyrowitz, Norman and Moser, Margaret, "BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems," Department of Computer Science, Brown University, December 1981.
- [OLS83] Olsen, Eric W. and Whitehall, Stephen B., Ada for Programmers, Reston Publishing, Inc., 1983.
- [PAR72] Parnas, D.L., "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, Volume 15 Number 12, December 1972.
- [PER83] Perry, John W., "Are We Wearing the Emperor's Old Clothes?", INFO-ADA ARPAnet message, 4 Nov 1983.
- [SCH78] Schicker, P. and Duenki, A., "The Virtual Terminal Definition," Computer Networks, 1978, pp 429-441.
- [SIM76] DEC-System 10 Simula Language Handbook: Part 1: The Programming Language Simula. Report no. C8398. Part 2: DEC-System 10 Dependent Information, Debugging. Report no. C8399. Part 3: Utility Library. Report no. C10045. Rapportcentralen, FOA 1, S-104 50 Stockholm 80 Sweden.
- [SPE81] Spencer, P.D. and Gordon, D., "Software Development Methods For Use With the IAPX432 Microprocessor," EUROMICRO, North-Holland Publishing Co., 1981.
- [STE81] Stenning, Vic, Et Al., "The Ada Environment: A Perspective," Computer. Volume 14, number 6, June 1981, pp 26-34, 36.
- [SUK81] Sukamar, Srinivas and Wiese, John D, "Hardware and Firmware Support for Four Virtual Terminals in One Display Station," Hewlett-Packard Journal, March 1981.
- [TAF82] Taft, S. Tucker, "Portability and Extensibility in the Kernel and Database of a Programming Support Environment," Intermetrics, March 1982.
- [TAJ79] Tajima, Takashi and Katsuyama, Yoshiki, "Layered and Parametric Approach to Terminal Virtualization," Presented at International Conference on Communications, Boston, MA, June 1979.
- [TI81A] Texas Instruments, "Ada Integrated Environment," Lewisville, TX, March 1981. Prepared for Rome Air Development Center (RADC) under DoD Contract F30602-80-C-0293.

REFERENCES
OTHER PUBLICATIONS

- [THA82] Thall, Richard, "The KAPSE for the Ada Language System, SofTech Inc, Proceedings of the AdaTEC conference on Ada, October 1982.
- [WEB80] Websters New Collegiate Dictionary, G. and C. Merriam Company, Springfield, MA, 1980.
- [WOL81] Wolfe, Martin I., et al., "The Ada Language System," Computer, Volume 14, number 6, June 1981, pp 37-45.

APSE
INTERACTIVE MONITOR

Final Report on Interface
Analysis and Software
Engineering Techniques

VOLUME 2

Design and Implementation
Experiences : The AIM

Prepared for:

NAVAL OCEAN SYSTEMS CENTER (NOSC)
United States Navy
San Diego, CA 92152

Contract No. N66001-82-C-0440
CDRL No. A010

Equipment Group - ACSL
P.O. Box 301, M.S. 3007
McKinney, Texas 75069
15 July 1985

TEXAS INSTRUMENTS
INCORPORATED

Ada is a registered trademark of the U.S. Government, Ada Joint Program Office (AJPO).

ADE is a trademark of ROLM Corporation.

DEC is a trademark of Digital Equipment Corporation.

ECLIPSE is a registered trademark of Data General Corporation.

ECLIPSE MV/10000 is a trademark of Data General Corporation.

ROLM is a registered trademark of ROLM Corporation.

VAX is a trademark of Digital Equipment Corporation.

VMS is a trademark of Digital Equipment Corporation.

CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

2.1	OBJECT-ORIENTED DESIGN REFRESHER	2-1
2.1.1	Defining The Problem	2-1
2.1.2	Developing An Informal Strategy	2-1
2.1.3	Formalizing The Strategy	2-2
2.2	HOW WE APPROACHED THE PROBLEM	2-4
2.2.1	Developing The Informal Strategy	2-4
2.2.2	The Two-pronged Analysis	2-5
2.3	AREAS OF CONFUSION	2-7
2.3.1	Objects	2-7
2.3.2	Terminology	2-8
2.3.3	Non-applicable Information	2-8
2.3.4	Attributes	2-9
2.3.5	Asynchronous Data And Tasks	2-9
2.4	IMPLEMENTATION EXPERIENCES BASED ON THE AIM'S OBJECT-ORIENTED DESIGN	2-10
2.4.1	AIM Window/Pad Relationship	2-10
2.4.2	AIM Image/Viewport Relationship	2-10
2.4.3	AIM Data Flow Model	2-11
2.5	OBJECT-ORIENTED DESIGN AND MIL-STD-1679	2-12
2.6	CONCLUSIONS	2-13

CHAPTER 3 EXPERIENCES USING ADA LANGUAGE FEATURES

3.1	INTRODUCTION	3-1
3.2	PACKAGES	3-1
3.2.1	Exporting Entities	3-1
3.2.2	Importing Entities	3-3
3.2.2.1	Context Clauses	3-3
3.2.2.2	Cyclic Dependencies	3-4
3.2.3	Recompilation	3-5
3.3	GENERICIS	3-5
3.3.1	Instantiating Generics	3-6
3.3.2	Summary	3-8
3.4	VARIANT RECORDS	3-9
3.4.1	Assignment To Variant Records	3-9
3.4.2	Summary	3-10
3.5	STRONG TYPING	3-10
3.5.1	Type Mismatches	3-11
3.5.2	Derived Types	3-11
3.5.3	Subtypes	3-12
3.5.4	Range Constraints	3-13
3.5.5	Operations On Imported Types	3-14
3.6	STRING MANIPULATION	3-16
3.6.1	Formatting Strings	3-16

3.6.2	String Parameters	3-17
3.6.3	A String Utility Package	3-17

CHAPTER 4 ADA TASKING

4.1	INTRODUCTION	4-1
4.2	A TAXONOMY OF ADA TASKS	4-1
4.2.1	Server Tasks	4-2
4.2.2	Actor Tasks	4-3
4.2.3	Transducer Tasks	4-4
4.3	ADA TASKING BUILDING BLOCKS	4-5
4.3.1	Canonical Ada Task Sets	4-6
4.3.2	A Simple Example	4-7
4.4	ADA TASK START-UP	4-8
4.4.1	Control Of A Starting Task's Execution	4-8
4.5	ADA TASK TERMINATION	4-9
4.5.1	Graceful Termination Of Canonical Task Sets	4-10
4.5.2	Ada Task Termination Concerns	4-13
4.6	ADA TASKING AND AIM DATA FLOW	4-15
4.6.1	AIM Data Flow Model Skeleton	4-15
4.7	CONCLUSION	4-24

CHAPTER 5 EXCEPTIONS AND ERROR PROCESSING

5.1	INTRODUCTION	5-1
5.2	ERROR HANDLING TECHNIQUES	5-1
5.2.1	In-line Error Handling	5-2
5.2.2	Centralized Error Handling	5-4
5.3	ERROR HANDLING USING EXCEPTIONS	5-6
5.3.1	In-line Error Handling With Exceptions	5-6
5.3.2	Centralized Error Handling Using Exceptions	5-7
5.3.3	Handling The Standard Exceptions	5-10
5.4	DESIGNING EXCEPTIONS INTO ADA PROGRAMS	5-10
5.4.1	Propagating Exceptions	5-10
5.4.2	Exception Visibility	5-11
5.4.3	The Cross-Reference Matrix	5-13
5.5	USING EXCEPTIONS FOR PROGRAM CONTROL	5-15
5.5.1	Detecting Expected Exceptional Conditions	5-15
5.5.2	Side Effects Of Using Exceptions	5-17
5.5.2.1	Intermediate Values Of Variables	5-17
5.5.2.2	Experiences While Testing	5-18
5.5.2.3	Suppressing Runtime Checks For Errors	5-18
5.6	GUIDELINES FOR USING EXCEPTIONS	5-19

CHAPTER 6 ENVIRONMENT EXPERIENCES

6.1	COMPILER EXPERIENCES	6-1
6.1.1	Storage Allocation Scheme	6-1
6.1.2	Storage Management Scheme	6-2
6.1.3	Implementation Of Tasking	6-3
6.1.4	Implementation Dependent Features	6-3

6.2	ENVIRONMENT COMPARISON: AOS/VS/ADE VS VMS	6-4
6.2.1	Compiler	6-4
6.2.1.1	Method Of Compilation	6-4
6.2.1.1.1	AOS/VS	6-4
6.2.1.1.2	VAX/VMS	6-5
6.2.1.2	Error Messages	6-6
6.2.1.2.1	AOS/VS	6-6
6.2.1.2.2	VAX/VMS	6-7
6.2.1.3	Resultant Files	6-8
6.2.1.3.1	AOS/VS	6-8
6.2.1.3.2	VAX/VMS	6-8
6.2.1.4	Integration Into Environment	6-9
6.2.1.4.1	AOS/VS	6-9
6.2.1.4.2	VAX/VMS	6-11
6.2.1.5	Functional Capabilities	6-12
6.2.2	Linker	6-13
6.2.2.1	Method Of Linking	6-13
6.2.2.1.1	AOS/VS	6-13
6.2.2.1.2	VAX/VMS	6-15
6.2.2.2	Error Messages	6-16
6.2.2.2.1	AOS/VS	6-16
6.2.2.2.2	VAX/VMS	6-17
6.2.2.3	Resultant Files	6-17
6.2.2.3.1	AOS/VS	6-17
6.2.2.3.2	VAX/VMS	6-17
6.2.2.4	Integration Into Environment	6-17
6.2.2.5	Functional Capabilities	6-18
6.2.3	Ada Source Code Debugger	6-18
6.2.3.1	Compiler/Linker Requirements	6-19
6.2.3.2	Functional Capabilities	6-19
6.2.4	Program Librarian And Library Structure	6-21
6.2.4.1	ADE Program Library	6-21
6.2.4.2	ACS Program Library	6-22
6.2.4.3	Functional Capabilities	6-22
6.2.5	Configuration Management And File Structure	6-24
6.2.5.1	ADE Configuration Management	6-24
6.2.5.2	VAX/VMS Configuration Management	6-24
6.2.5.3	Functional Capabilities	6-26
6.2.6	Text Editor	6-27
6.2.6.1	ADE Text Editor	6-27
6.2.6.2	VAX/VMS Text Editor	6-27
6.2.6.3	Functional Capabilities	6-28
6.2.7	Electronic Mailer	6-30
6.2.7.1	ADE Electronic Mailer	6-30
6.2.7.2	VAX/VMS Electronic Mailer	6-30
6.2.7.3	Functional Capabilities	6-30
6.2.8	Conclusions	6-31

CHAPTER 7 LIFECYCLE ANALYSIS

7.1	INTRODUCTION	7-1
7.2	PROJECT OVERVIEW	7-1
7.2.1	Systems	7-4

7.2.2	Personnel	7-4
7.3	AIM PROJECT EFFORTS	7-6
7.4	TESTING METHODOLOGY	7-10
7.4.1	Error Correction	7-10
7.5	LINES OF CODE	7-11
7.6	MODEL COMPARISONS	7-14
7.6.1	Lifecycle Models	7-14
7.6.1.1	40-20-40 Model	7-15
7.6.1.2	Brooks Model	7-18
7.6.1.3	GTE Model	7-20
7.6.2	Costing Models	7-22
7.6.2.1	SoftCost	7-22
7.6.2.2	Price-S	7-24
7.6.2.3	COCOMO	7-25
7.7	CONCLUSIONS	7-27
7.7.1	Design Effort	7-27
7.7.2	Implementation Effort	7-29
7.7.3	Testing Effort	7-29
7.7.4	LOC	7-30
7.7.5	Wrap-Up	7-30

CHAPTER 8 DIDS

8.1	PURPOSE	8-1
8.2	OVERVIEW	8-1
8.3	PROGRAM PERFORMANCE SPECIFICATION (PPS)	8-2
8.3.1	Requirements	8-2
8.3.2	Testing	8-2
8.4	ACCEPTANCE TEST PLAN (ATP)	8-3
8.5	COMPUTER PROGRAM TEST SPECIFICATION (CPTS)	8-4
8.6	ACCEPTANCE TEST PROCEDURES (ATPRO)	8-4
8.7	PROGRAM DESIGN SPECIFICATION (PDS)	8-4
8.8	SYSTEM/INTEGRATION TEST PLAN (SITP) AND PROCEDURES (SITPRO)	8-5
8.9	SUMMARY	8-5

APPENDIX A GLOSSARY

APPENDIX B REFERENCES

B.1	GOVERNMENT STANDARDS	B-1
B.2	GOVERNMENT SPECIFICATIONS	B-1
B.3	OTHER GOVERNMENT DOCUMENTS	B-2
B.4	SPECIAL SOURCES	B-3
B.5	OTHER PUBLICATIONS	B-4

CHAPTER 1

INTRODUCTION

This volume is the second of three which comprise the Final Report on Interface Analysis and Software Engineering Techniques, as part of NOSC contract N66001-82-C-0440. Presented within are the knowledge and experience gained by the project team during the design and implementation of the APSE Interactive Monitor (AIM).

The AIM is a tool designed to act as an interface between the user of the APSE and the programs the user executes in the APSE. It is designed to enable a user to execute multiple APSE programs from a single terminal while keeping their interactive inputs and outputs separate both logically and physically. The primary objective of the AIM project is to assist the KAPSE Interface Team (KIT) in studying interface issues while secondarily producing a useful tool for APSEs. For a complete description of AIM functionality, consult [TI83A].

The AIM was designed using an object-oriented methodology with Ada as the design language. Object-oriented design together with Ada as the Program Design Language (PDL) is attractive to modern software developers for many reasons [BOO83]:

1. object-oriented design directly supports software engineering principles such as abstraction, information hiding, modularity, and localization,
2. the Ada language embodies these concepts of modern software methodologies,
3. Ada is more than just another programming language; it is a language suitable for expressing solutions to problems throughout the life cycle of a software project, and
4. object-oriented design exploits the expressive power of the Ada language.

INTRODUCTION

Within this volume, specific areas of experience are discussed.

- * Chapter 2 discusses object-oriented design and its use in the design of the AIM. An evaluation of our use of the design methodology, composed after the implementation was complete, is also included.
- * Chapter 3 presents general knowledge, related to the Ada language, gained during the life of the AIM project.
- * Chapter 4 provides a detailed discussion of tasking within the Ada language, including: classification of tasks, Ada tasking building blocks, task start-up, and task termination.
- * Chapter 5 discusses exceptions and error processing. Side effects experienced while using exceptions and some guidelines for the use of exceptions are also presented.
- * Chapter 6 describes the experiences of the AIM project team while using the tools provided in two Ada programming environments. A comparison of the two Ada programming environments is provided.
- * Chapter 7 is a detailed presentation and evaluation of the life cycle of the AIM project. The results are evaluated using a number of models and an analysis is presented.
- * Chapter 8 discusses the contract specified Data Item Descriptions (DIDs) and the problems encountered while using them.

CHAPTER 2

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

2.1 OBJECT-ORIENTED DESIGN REFRESHER

The AIM was designed using the Object-Oriented Design (OOD) technique presented by Grady Booch in his book "Software Engineering with Ada" [BOO83]. An extremely brief condensation of this technique follows.

The steps involved in informal Object-Oriented Design are:

1. Define the problem
2. Develop an informal strategy
3. Formalize the strategy

2.1.1 Defining The Problem

Defining the problem is done in Object-Oriented Design using the same techniques that are used in analyzing any problem: functional description, data flow analysis, etc. It is important to note that understanding a problem is typically a process that continues throughout the life of a project. As understanding deepens, the problem definition is refined and changed to match this newfound understanding. This is reflected in changes to the informal strategy.

2.1.2 Developing An Informal Strategy

An informal strategy is an attempt to partition the problem space into functional areas that parallel the conceptual view of the problem. Here, the problem and its logical and physical partitions are expressed in natural English descriptions using the terminology that exists in the problem space, where no restrictions are placed on the form of the text. At this time no attempt is made to force a

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

Developing An Informal Strategy

design structure onto the resultant partitions; however, later these descriptions will be used to develop a more formal representation of the problem space.

2.1.3 Formalizing The Strategy

The most difficult step in Object-Oriented Design is the formalization of the strategy. In this phase, the objects, its attributes, and its associated operations are identified. The actual process is as follows:

1. Extract the nouns and qualifying adjectives from the informal strategy. A subset of these will become the objects.
2. Extract the verb phrases from the informal strategy. A subset of these will become the operations.
3. Establish the relationships among the objects.
4. Implement.
5. Perform object-oriented analysis on the objects as they in turn are decomposed into simpler objects.

Identifying the nouns, adjectives, and verb phrases involves simply going into the informal descriptions and underlining these words. According to Booch, this should be a completely mechanical process. For later reference and discussion, a definition of these terms is important. A definition of noun, verb, adjective, and attribute is taken from Webster's New Collegiate Dictionary [WEB80].

1. Noun - 1. A word that is the name of a subject of discourse (as a person, animal, plant, place, thing, substance, quality, idea, action, or state) and that in languages with grammatical number, case, and gender is inflected for number and case but has inherent gender. 2. a word except a pronoun used in a sentence as subject or object of a verb, as object of a preposition, as the predicate after a copula, or as the name in an absolute construction.
 - a. copula - something that connects: as a: the connecting link between subject and predicate of a preposition, b: a word or expression that links a subject with its predicate (as a form of be, become, feel, or seem).

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN
Formalizing The Strategy

2. Verb - a word that characteristically is the grammatical center of a predicate and expresses an act, occurrence, or mode of being, that in various languages is inflected for agreement with the subject, for tense, for voice, for mood, or for aspect, and typically has rather full descriptive meaning and characterizing quality but is sometimes nearly devoid of these especially when used as an auxiliary or copula.
3. Adjective - a word belonging to one of the major form classes in any of numerous languages and typically serving as a modifier of a noun to denote a quality of the thing named, to indicate its quantity or extent, or to specify a thing as distinct from something else.
4. Attribute - 1: an inherent characteristic; also: an accidental quality. 2: an object closely associated with or belonging to a specific person, thing, or office. 3: a word ascribing a quality.
 - a. Characteristic - a distinguishing trait, quality, or property.

According to Booch, nouns fall into three categories:

1. Common nouns - name of a class of entities (e.g., table, terminal, sensor, switch).
2. Mass nouns and units of measure - name of a quality, activity, or substance, or a quantity of the same (e.g., water, matter, fuel).
3. Proper nouns and nouns of direct reference - name of a specific being or entity (e.g., nozzle-pressure sensor, my table, abort switch).

The first two categories identify abstract data types, while the proper nouns identify objects.

The adjectives identify attributes and qualities of the objects. These adjectives can identify value ranges, parallelism, and other characteristics of objects.

Verb-phrases translate into the operations performed upon the objects. Adverb-phrases are also gathered and associated with the operations. These are used to identify limits on the operations.

The next step is the establishment of the relationships. Here we

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

Formalizing The Strategy

formally define the visible interfaces using Ada as the design language. This is accomplished using the objects and their operations that were identified earlier. The scope of visibility is defined. This identifies which object is visible to which other object(s). The scope and visibility is best presented graphically.

The implementation of the operations defined in the interfaces is performed in Ada. As the implementation proceeds and the nature of the objects themselves are uncovered, the Object-Oriented Design techniques are applied iteratively to the objects. This continues until the entire system is designed.

2.2 HOW WE APPROACHED THE PROBLEM

2.2.1 Developing The Informal Strategy

We had the task of specifying and designing a tool that could identify interface issues within an APSE. Due to the nature of the contract, essentially a research and development effort, we had a great deal of latitude in the functionality of the AIM. Our definition of the problem took the following form.

An exhaustive literature search was followed by trips to Carnegie-Mellon University in Pittsburgh and University of Rochester in New York. The Gandalf system was examined at CMU for ideas concerning virtual terminals and Ada support environments. The Rochester Intelligent Gateway [LAN79A] [LAN79B] was examined at University of Rochester for applicability to this project. These systems were examined for ideas and terminology that were subsequently folded into the AIM. Data flow diagrams were constructed in an attempt to identify the data producers that were being serviced by the AIM. Similarly, an itemized list of the functions that the AIM would support was developed, reviewed, and modified. The parallels that were found between the systems researched and the AIM were exhaustively analyzed.

A problem area became apparent immediately. We started defining the AIM by not only researching the existing similar systems, but also gathering and examining the specifications for the two APSEs of greatest interest: the Ada Language System (ALS) being developed by the Army and the Ada Integrated Environment (AIE) being developed by the Air Force. The examination of these specifications was a mistake. There were easily identifiable areas within the ALS and AIE where the AIM (as it stood then) could not operate. The natural tendency was to define the AIM in terms of what the APSEs could support, thus negating the research goals of the project. As we realized that this was happening, we took the following steps:

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

Developing The Informal Strategy

1. We shelved the specifications of the APSEs.
2. We concentrated only on what we wanted the tool to do.
3. We developed a plan of attack as to when we would again refer to the specifications.

By doing the definition in this manner we were able to specify a tool that truly identified interface issues and problems.

All of this was refined as necessary to provide the material that went into the Program Performance Specification (PPS) [TI83A]. The PPS was the informal strategy.

2.2.2 The Two-pronged Analysis

The informal strategy became a document in the form of MIL-STD-1679 Program Performance Specification (PPS). We proceeded to analyze it using the Object-Oriented Design techniques. However, it became apparent that something was not functioning correctly. The identification of nouns (underlining them in the document) was turning up far too many to deal with properly.

Our process was then divided into two approaches.

1. Identify the nouns, verbs, etc exactly as described in the Object-Oriented Design techniques. We would use them for comparison later.
2. Temper the identification of nouns with reason, attempting to locate nouns and phrases that meant the same thing or said little of actual interest for a design. Again, list them for comparison later.

One design engineer took one approach; another engineer took the other approach. We found the second approach much more effective. The reasons were as follows:

1. the non-applicable information could be ignored.
2. the nouns (both implied and actual) could be identified correctly.
3. the true attributes of the objects could be identified.

These will be covered in greater detail in the next section.

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

The Two-pronged Analysis

As the Object-Oriented Design process continued we identified the objects, their attributes, and the operations that were to be performed on them. The operations were separated into two areas: selectors and actors. A selector is a function that returns a value and does nothing to change the characteristics of an object. An actor is a procedure or function that changes some characteristic(s) of an object.

The major objects identified for the AIM's Structure Manager include: windows, images, viewports, AFSE programs, and the user's terminal. As an example of the actor/selector interface classification, the interfaces identified for window objects is listed below.

Window

attributes

- name
- is full
- suspends output on full
- input pad
- output pad

actors

- create
- delete
- switch to named window
- next page of window
- clear window
- set suspend output on full attribute
- reset suspend output on full attribute
- suspend output
- resume output
- create/delete/close input/output pad
- write to input/output pad

selectors

- get first window
- get next window
- get previous window
- get name
- is on the screen
- is full
- is last window
- get input/output pad name

From lists like this, all the AIM's interfaces were developed and coded into Ada. These Ada interfaces (and many other things) were incorporated into the Program Design Specification (PDS) [TI85B].

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN AREAS OF CONFUSION

2.3 AREAS OF CONFUSION

As the objects, their attributes, and the operations on them were identified, problems arose. For many of them there were obvious answers; for others, the answers were not so obvious. These areas will be discussed here:

1. objects - what exactly is an object?
2. terminology - inconsistencies are easily identified,
3. non-applicable information - other information is discussed in a PPS besides the information necessary for object identification,
4. attributes - what are these and how are they identified?
5. asynchronous data and tasks - difficult to identify and quantify.

2.3.1 Objects

An object in the real world is something physical or mental of which a subject is cognitively aware [WEB80]. More simply, it is something that can be seen, touched or otherwise sensed. In the SmallTalk Environment, an object is a component of the system represented by some private memory and a set of operations [GOL83]. Similarly, in Ada we are going to represent our objects as packages; therefore an object is a collection of data structures and the operations that are performed on those data structures. This is called data abstraction.

We are trying to represent the objects in the problem space with objects in our abstraction. Given the definition of an object in the real world, an abstract object can take many forms. Consider the window as an example.

A window in the AIM system is defined to be an analog of the APSE program's view of the terminal. The AIM will contain as many windows as there are APSE programs running under the control of the AIM. The question we faced was: Should the object (and thus the package) be an individual window or the collection of all windows?

If the object was one window then the collection of these windows would have to be made in the object that was using windows. Creating or deleting windows would translate into simple initialization or deletion of the window's internal data structures. It would be the responsibility of the object using windows to maintain the list of all windows and perform the operations to traverse this list. In the implementation of the window package, whenever a function or

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

Objects

procedure was called the window upon which the operation was to be performed would have to be passed in as a parameter.

If the object was the collection of all windows, then the object making use of windows would not have to maintain the list. Creating or deleting windows would involve both initializing or deleting the internal data structures and removing the window from the list of all windows. This removes some of the burden of maintenance from the object using windows. However, to perform an operation (other than create) another call must be made to retrieve the window of interest given the name of the window. Although this is another level of overhead, it maps closely to the functionality of the AIM command interpreter.

We chose the latter approach for Structure Manager's objects.

2.3.2 Terminology

A very great side benefit of formalizing the strategy is the identification of synonyms. We found that many synonyms existed for exactly the same concept, object, operation, and/or attribute. A complete list of these was developed, then the most appropriate word (or phrase) was chosen to represent the multitude. We edited the PPS to reflect the single word (or phrase) for each concept, object, etc. The document was then much more readable, less confusing, and more consistent.

2.3.3 Non-applicable Information

To perform the formalization of the strategy, the nouns, verbs, adjectives, etc. are underlined in the informal strategy specification (the PPS). We found that the document contained much information that was not a part of the AIM itself. This information took the form of:

1. Requirements on the APSE. Statements such as: "The AIM must have the ability to create, open, and write to an indefinite number of files using the existing APSE database services."
2. The required paragraphs in the PPS that have nothing to do with the AIM in a technical sense. For example: 1.2 Mission; Section 2: Applicable Documents. Actually, only Section 3 was used to create the informal strategy.
3. Transition material such as introductions.
4. The terminal capabilities configurator. This is another complete tool that was requested by the customer. However, it was requested that the description of the tool be provided with the

PPS of the AIM system.

This material had to be carefully sifted out.

2.3.4 Attributes

Given the above definition of "attribute" and "characteristic", it is easy to conclude that Booch's definition of attribute is rather limited. Attributes need not be limited to just the adjectives, they can also be nouns. Using the window example again, a window has the following attributes:

1. its name,
2. a quality of being full,
3. a pad files,
4. a quality of being suspended.

A name is definitely a noun. A pad is not only a noun, it is an identified object. Similar attributes exist for images, viewports, and APSE programs.

It actually became quite confusing. Consider the viewport. Is a window an "attribute" of a viewport? Taking the second part of the definition of attribute, it seems that it is. If it is not an attribute, then what is this association? These questions were difficult to answer. We eventually assumed that they were attributes, and therefore structured our system accordingly.

2.3.5 Asynchronous Data And Tasks

The AIM has three sources of asynchronous data:

1. The user through his terminal,
2. The command interpreter through the script mechanism,
3. The APSE programs.

We had an extremely difficult time attempting to identify how the asynchronous nature of the data flow would map into Ada. Object-Oriented Design did not seem to provide support for asynchronous situations. The Object-Oriented Design techniques were supplemented with data-flow analysis. Even with this supplement, it was difficult and confusing to identify appropriate objects and how they mapped into Ada tasks.

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

Asynchronous Data And Tasks

We eventually decided to use data queues to handle the asynchronous nature of the flow. These queues became objects and translated into generic packages. The problem typically associated with asynchronous data flow, namely, critical regions, was also handled in this way.

Our team had very little experience with asynchronous data flow within systems. This may have caused us some problems. However, it is evident that Object-Oriented Design may have some deficiencies when applied to asynchronous data flow.

2.4 IMPLEMENTATION EXPERIENCES BASED ON THE AIM'S OBJECT-ORIENTED DESIGN

Given the aforementioned design issues, it was to be expected that the actual implementation of the AIM program would necessitate some modifications in the original design. The major design changes were limited to the following areas:

- * the relationship between windows and pads,
- * the relationship between images and viewports, and
- * the underlying data flow model.

A discussion of the details surrounding each of these areas follows.

2.4.1 AIM Window/Pad Relationship

In the original AIM design, windows and pads were considered individual objects that were to be maintained in separate global linked lists. Furthermore, there was confusion surrounding the exact relationship between these objects: is a pad an attribute of a window, or is a window an attribute of a pad, or both?

An implementation decision was made to encapsulate the pad related interfaces within the Window Manager and consider a pad an object attribute of a window. This decision eliminated the need for a global linked list of pads, and thus, simplified the AIM design.

2.4.2 AIM Image/Viewport Relationship

In the original AIM design, a viewport was an object that contained a window/image pair and was to be inserted into a global linked list within the AIM system. Furthermore, the original definition of an image object included a viewport component, and therefore, the image/viewport inter-relationship had redundancies.

For simplicity, the notion of a global linked list of viewports was eliminated in the AIM implementation. A more efficient technique of

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

AIM Image/Viewport Relationship

maintaining viewport linked lists on an image-by-image basis was instrumented. Although this approach eliminated the redundancies of the original design, there was another inherent problem caused by the inter-relationship of the images and viewports: since IMAGE and VIEWPORT are private data types in different packages, an additional set of interfaces was needed by the Image Manager to maintain its images' linked lists of viewports as it could not do so directly.

For instance, to create a viewport (i.e.--window/image association pair) the AIM Command Interpreter must call the Viewport Manager's ASSOCIATE procedure which in turn calls the Image Manager's INSERT_VIEWPORT procedure which in turn must call the Viewport Manager's INSERT_VIEWPORT_IN_LIST procedure. The total effect of this calling sequence is that a new viewport (object) is created and it is inserted into a linked list belonging to its implied image.

2.4.3 AIM Data Flow Model

The original AIM data flow model was designed in a somewhat ad-hoc manner. The Object-Oriented Design applied well for the major objects of the AIM system: windows, image, viewports; however, it left a large void in the heart of the system, namely in the areas of data flow and program control. There are various rationalizations that might argue for why this situation came about:

- * Inexperience in using Object-Oriented Design,
- * Naivety towards Ada language (specifically Ada tasking),
- * Inexperience in designing/developing multi-tasking systems, and
- * Deficiency of Object-Oriented Design in the areas of concurrency and data flow.

To solve this problem, our design team attempted to fill these gaps by using conventional data flow design techniques. This application of a traditional design methodology lead to a data flow model which consisted of one global asynchronous data queue and various Ada tasks to service this queue (the design goal was to minimize the number of Ada tasks wherever possible due to our tasking inexperience). Needless to say, due to the AIM's highly asynchronous nature, this single queue data model was inadequate.

During the implementation, a general purpose data flow model was conceived using the notion of Ada task sets (see Chapter 4) in combination with a traditional data flow analysis approach. Although this technique was quite effective, the final data flow model still evolved through two generations of modifications before it was

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN

AIM Data Flow Model

generalized enough to be used for the AIM.

In retrospect, there seemed to be one major drawback to using Object-Oriented Design for the definition of the original and subsequent AIM data flow models: Object-Oriented Design does not identify the program control operations and inter-relationships that typically are associated with a system of Ada tasks. In this vein, it is our contention that Object-Oriented Design alone is not conceptually powerful enough to support the design of complex concurrent systems, and therefore must be used in conjunction with a data flow analysis approach based on the fundamental notion of Ada Tasking building blocks.

2.5 OBJECT-ORIENTED DESIGN AND MIL-STD-1679

The informal strategy was documented as MIL-STD-1679 Program Performance Specification. The most applicable sections were "3.3 Functional Description" and "3.4 Detailed Functional Requirements." Originally each functional unit was described using the following format:

1. Functional description
 - a. narrative description,
 - b. inputs,
 - c. outputs,
 - d. processing requirements,
 - e. miscellaneous.
2. Interfaces
 - a. narrative description,
 - b. description of the data that this interface handles,
 - c. miscellaneous.
3. Miscellaneous information and notes.

The format described above was eventually condensed and converted into the required format for the PPS. Due to the research nature of the AIM contract, our format was adapted slightly from the actual PPS

EXPERIENCES WITH INFORMAL OBJECT-ORIENTED DESIGN
OBJECT-ORIENTED DESIGN AND MIL-STD-1679

format as defined in the Military standard. This adaptation added more subdivisions to increase clarity. The final format used was:

1. introduction,
2. interface summary,
3. inputs,
4. processing description,
5. errors,
6. outputs.

This was a perfectly acceptable format for developing the formal strategy.

2.6 CONCLUSIONS

The informal Object-Oriented Design methodology worked quite well after overcoming the problems. As more projects use this technique, it should be refined to reflect this experience. Refinement can take many forms. Choices include:

1. Limit the form of the Object-Oriented Design informal strategy. It is obvious that the informal strategy can be more effective if Object-Oriented Design techniques are in mind when writing the specification of the informal strategy.
2. Formalize the technique. Object-Oriented Design using a more formal approach could provide benefits in software productivity, maintainability, and reliability.
3. Mate the methodology with another. Data flow and/or data structure analysis can be used to supplement Object-Oriented Design and provide more insight into the problem space.
4. More rigorously define the Object-Oriented Design steps mapping the informal strategy into the formal one.

The Military's Data Item Descriptions (DID's) are oriented toward an assembly language implementation. This is because most projects are confined to assembly language due to performance constraints. The DID's were designed to support this type of project well. Now, however, the DOD will start requiring Ada on all embedded computer projects. As such, the DID's should be restructured to more accurately reflect the design techniques that Ada supports the best.

CHAPTER 3

EXPERIENCES USING ADA LANGUAGE FEATURES

3.1 INTRODUCTION

This chapter is the first of three chapters which presents a discussion of our experiences using the Ada language throughout the design and implementation phases of the AIM. This chapter addresses a variety of Ada language features: packages, generics, variant records, strong typing, and string manipulation. Detailed discussions on Ada tasking and exceptions are presented in the two chapters following this one.

3.2 PACKAGES

An Ada package is a fundamental program unit which allows the user to encapsulate a group of logically related entities. As such, packages directly support the software principles of data abstraction and information hiding.

Within the AIM, each major object identified via the object-oriented design process, along with its attributes and operations, was directly translated into Ada package specifications and bodies. Collectively, all the package specifications represent the top-level view of the AIM program.

The Ada package is an important language construct in terms of designing a software tool using Ada. This section addresses some of the issues one faces when using packages in the design of a large software tool.

3.2.1 Exporting Entities

Any object, type, or subtype declared in the visible part of a package specification is said to be exported by that package since it is accessible from outside the package itself. Additionally, any operations to be performed on the exported private types must be provided by that package. This will require that functions and/or

EXPERIENCES USING ADA LANGUAGE FEATURES

Exporting Entities

procedures be defined to provide the necessary operations.

When numerous program units share exported objects an interesting design decision must be made. Consider the following scenario: several, perhaps physically separate, yet logically inter-related, program units need to access the same data structures (objects). What is the best design scheme to use in this situation? A list of viable alternatives follows:

1. pass the necessary data structures to the program units via parameters,
2. nest the program units within a higher level package which encapsulates the data structures, or
3. globally export the data structures from a high level (support) package.

The first alternative would be less efficient than the other two as it expends additional overhead in processing the parameters. Also, it does not completely solve the problem since some program units must still export a global object to be used as the actual parameter.

The second alternative has a couple of drawbacks. First, nesting program units within a higher level package makes the private part of the encapsulating package visible; therefore, a level of information hiding is lost. Secondly, the nesting structure does not necessarily reflect the original design. If the program units are indeed tightly inter-related then the original design should have reflected this nesting structure; otherwise, another design scheme is needed.

The last alternative is a simple, general purpose scheme. Its only drawback lies in the fact that all the data objects are globally accessible.

All three design schemes have their drawbacks, but depending on the inter-relations of the program units sharing the data objects, the second alternative is probably the most attractive to the software design engineer. Even though the private part of the encapsulating package becomes visible to the nested program units, the shared data objects themselves can be hidden from outside the package by elaborating them within the body of the encapsulating package. The third alternative is simple and general purpose, but it does not directly support information hiding and thus is less attractive. The parameter passing approach is more cumbersome for the implementor and less efficient due to the overhead costs of processing the additional parameters.

3.2.2 Importing Entities

Any object, type, or subtype required by a compilation unit that is not defined within that compilation unit must be imported. Importation means to make the objects, types, or subtypes visible to the compilation unit to allow use of them by the compilation unit. The imported entities are made visible (if only the WITH clause is included) or directly visible (if the WITH and USE clauses are included). If two or more compilation units are mutually dependent, cyclic dependencies can occur.

3.2.2.1 Context Clauses

A context clause is used to specify the library units whose names are needed within a compilation unit. Dependencies among Ada compilation units are defined by WITH clauses; that is, a compilation unit that mentions other library units in its context clause depends on those library units. A library unit mentioned in a WITH clause becomes visible to the compilation unit. If a USE clause is also included, the library unit becomes directly visible, eliminating the need to use the library unit name to completely qualify any of the library unit's entities. To avoid additional dependencies among compilation units and thus unnecessary compilations, the use of context clauses should be deferred as long as possible. Consider the following code:

```
-- #1 --
package A is

    procedure FOO;      -- depends on package B

end A;

-- #2 --
package body A is

    procedure FOO is separate;

end A;

-- #3 --
separate (A)
procedure FOO is
begin
    B.SUBTLE;
end FOO;
```

Example 3.1--Placement of a context clause.

EXPERIENCES USING ADA LANGUAGE FEATURES

Context Clauses

The procedure A.FOO depends on the package B; therefore, there are logically three places that the necessary context clause ("WITH B;") could be placed in this code:

- * before the specification of package A,
- * before the body of package A, or
- * before the body of the procedure FOO.

The placement of the context clause has some subtle ramifications which argue for deferring it for as long as possible:

- * if it is placed before package A's specification, the specification and body of package A are both potentially affected by a change in package B,
- * if it is placed before the body of package A, the entire body of package A is potentially affected by a change in package B,
- * if it is placed before procedure FOO's body, the body of FOO is the only compilation unit potentially affected by a change in package B.

Given the above rationale, the WITH clause should be placed before the body of the procedure FOO.

Another reason for deferring the use of context clauses is to avoid cyclic dependencies among compilation units.

3.2.2.2 Cyclic Dependencies

A cyclic dependency occurs when compilation units are mutually dependent. The simplest case of this is when the specification of unit A depends on unit B and vice versa. This type of problem is generally caused by the misuse of context clauses; therefore, in most cases, it can be avoided by merely using them correctly. The Ada Language Reference Manual (LRM) [DOD83] states that "a compilation unit must be compiled after all library units named by its context clause". This implies that the specification of package A must be compiled after the specification of package B, but since B also names A in its context clause, the specification of package A must be compiled before the specification of package B. This mutual dependency results in a classic forward referencing problem.

This problem can be simply avoided by preceding the bodies, rather than the specifications, with the necessary context clauses. This concept can be carried even farther by deferring the use of context clauses to separate compilation units at the package body level. In

EXPERIENCES USING ADA LANGUAGE FEATURES

Cyclic Dependencies

general, unless data types or objects are being imported, any compilation unit, excluding the main program, that depends on another package(s) should be compiled as a separate subunit of its parent; the separate compilation unit can then be preceded by the necessary context clause. This convention not only supports the deferral of using context clauses, but also localizes their effect.

3.2.3 Recompilation

The adverse effect of loss of time due to recompilation can be reduced if properly addressed during the design phase. Without careful planning, even minor changes to the design during the implementation phase can require a massive recompilation of the system. Critical design reviews are one mechanism for insuring that the functionality of each package is clearly defined, thereby reducing the potential need for changes to packages. The key to critical design reviews is the group involvement they encourage. Ideas and concerns expressed from a number of sources help to insure that the functionality of each package is clearly defined.

During the implementation phase, the guidelines described in the cyclic dependencies section go a long way toward reducing the amount of recompilation required by most changes to a system design. For now, ignore the obvious recompilation issues related to package dependencies and concentrate on the package itself. Obviously, a change to the package specification requires that the complete package body must be recompiled. Similarly, a change to the package body proper requires that all procedures, task bodies, and functions within that package be recompiled, even if they are separately compilable units. The point to be made is: with proper design techniques, most changes that occur during the implementation phase only involve specific procedures or functions. If they are separately compilable, the time involved in recompilation is minimized since other subprograms within the package need not be recompiled.

3.3 GENERICS

A generic unit is a template, which can be parameterized and from which corresponding nongeneric subprograms or packages can be obtained. The resulting program units are said to be instances of the original generic unit; an instantiation is defined as the process of creating an instance of a generic unit.

EXPERIENCES USING ADA LANGUAGE FEATURES

Instantiating Generics

3.3.1 Instantiating Generics

A generic unit is a very powerful construct of the language. It defines a program unit template, along with generic parameters which support the tailoring of that template to particular needs. The true power of a generic is evident when it is instantiated. Consider the following Ada code:

```
generic

  type ELEMENTS is private;
  SIZE : POSITIVE;

package STACK_PACKAGE is

  type THE_STACK is private;

  function TOP_ELEMENT( STACK : in THE_STACK )
    return ELEMENTS;

  function STACK_IS_EMPTY( STACK : in THE_STACK )
    return BOOLEAN;

  procedure CLEAR_STACK( STACK : in out THE_STACK );

  procedure PUSH          ( FRAME : in ELEMENTS;
                           STACK : in out THE_STACK );

  procedure POP           ( FRAME : out ELEMENTS;
                           STACK : in out THE_STACK );

  NULL_STACK      : exception;
  STACK_OVERFLOW  : exception;
  STACK_UNDERFLOW : exception;

private

  type STACK_LIST is array ( 1 .. SIZE ) of ELEMENTS;

  type THE_STACK is
    record
      CONTENTS : STACK_LIST;
      TOP      : NATURAL range NATURAL'FIRST .. SIZE := NATURAL'FIRST;
    end record;

end STACK_PACKAGE;
```

Example 3.2--Generic Stack Package.

EXPERIENCES USING ADA LANGUAGE FEATURES

Instantiating Generics

The instantiation of this generic package requires a type parameter for both `ELEMENTS` and `SIZE`. This allows the user of the generic to specify the type of elements that will be on the stack and also the logical limitation of the stack's size. Assuming the generic package is called `STACK_PACKAGE`, a typical instantiation would look like:

```
MAX_LEVEL : constant := 10;
MAX_STRING_LENGTH : constant := 30;
subtype CURRENT_LEVEL_RANGE is POSITIVE range 1..MAX_LEVEL;

type STACK_FRAME is
  record
    PROMPT_STRING : STRING(1..MAX_STRING_LENGTH);
    CURRENT_LEVEL : CURRENT_LEVEL_RANGE;
  end record;

with STACK_PACKAGE;
package PROMPT_STACK is new STACK_PACKAGE (
  ELEMENTS => STACK_FRAME,
  SIZE => MAX_LEVEL );

MY_STACK : PROMPT_STACK.THE_STACK;
```

Example 3.3--Instantiation of the stack package.

This instantiation has the following effect:

1. the stack elements are of type `STACK_FRAME`,
2. the logical size of the stack is `MAX_LEVEL` elements,
3. a stack type, `THE_STACK`, is now exported by `PROMPT_STACK`,
4. the user of the `PROMPT_STACK` package can now elaborate a family of identically configured stacks using the exported stack type,
5. the following operations are now available for objects of type `PROMPT_STACK.THE_STACK`:
 - a. `CLEAR_STACK`
 - b. `TOP_ELEMENT`
 - c. `PUSH`

EXPERIENCES USING ADA LANGUAGE FEATURES

Instantiating Generics

- d. POP
6. the following exceptions can be raised by using the stack's operations:
 - a. NULL_STACK
 - b. STACK_OVERFLOW
 - c. STACK_UNDERFLOW
 7. the implementation details of the stack are not visible outside the generic stack package.

Referring to points 4 and 5, the example STACK PACKAGE provides the capability to create and manipulate numerous Identical stacks of the type STACK PACKAGE.THE_STACK with one instantiation of the generic package; that is, N stacks per instantiation. This is possible since every function and procedure uses a parameter of type STACK PACKAGE.THE_STACK. In this case, the various stacks will truly be identical. The elements of each stack will be of the same type and the size of each stack will be identical. If a system needs to use more than one stack, and each is structured the same, then this implementation would be beneficial since it can reduce the total amount of code in the system. However, there are a few points to consider.

Objects of type STACK PACKAGE.THE_STACK must be declared and the instantiation must precede the declaration. This may prevent some desired level of information hiding. Usually, numerous identical stacks are not required. If the STACK PACKAGE is implemented as one stack/instantiation, then the type declaration for THE_STACK can be removed from the visible section of the specification. Naturally, all of the parameters of type THE_STACK can then be removed from the procedures and functions.

3.3.2 Summary

The full power of generic program units cannot be completely appreciated until they are instantiated and used. A prime example of this is a generic package. Often, packages (including generics) export one or more types. Generic packages with parameters provide the facility for tailoring a set of logically related entities (types, objects, procedures, functions, exceptions) to a particular type parameter. In essence, the generic package 'becomes' the type.

For example, a generic package can encapsulate its own type declarations; therefore, if these declarations depend on the generic's type parameter, the instantiating program unit inherits 'new' data types which are derived from its own specification of the actual generic parameter. In the above example, PROMPT_STACK is an instantiation of STACK_PACKAGE with STACK_FRAME and MAX_LEVEL as the actual generic parameters. PROMPT_STACK has thus inherited the new type, THE_STACK, which is defined in terms of the actual generic parameter STACK_FRAME. This technique provides the capability of N identical stacks/instantiation. In this way, one can essentially define generic types for data structures. If numerous identical stacks would not be required, the type THE_STACK could be removed from the specification, thereby eliminating the need for most of the parameters to the procedures and functions.

3.4 VARIANT RECORDS

A variant record in Ada is a record type with a discriminant and a variant part. The variant part specifies alternate lists of components for the corresponding value or values of the discriminant.

3.4.1 Assignment To Variant Records

In Ada one can declare a variable of a variant record type either as an object that can represent any one of the variants or as an object that has a specific variant. The distinction is made in the object declaration by specifying the discriminant value or by omitting it. For example,

```

type TOKEN_ENUM is (PROCSYM, RETURN_SYM, ENDSYM);
subtype TOKEN_STRING_TYPE is STRING(1..5);

type BOOLEAN_RECORD ( UNIQUE : boolean := FALSE)
  is record
    case UNIQUE is
      when TRUE =>
        TOKEN_TYPE : TOKEN_ENUM;
        TOKEN_STRING : TOKEN_STRING_TYPE;
      when FALSE =>
        null;
    end case;
  end record;

REC_1, REC_2 : BOOLEAN_RECORD;
REC_TRUE : BOOLEAN_RECORD(TRUE);
REC_FALSE : BOOLEAN_RECORD(FALSE);

```

Example 3.4--Variant Records.

EXPERIENCES USING ADA LANGUAGE FEATURES

Assignment To Variant Records

In this example, REC_1 and REC_2 are elaborated as objects that can represent both variant cases of the variant record type; whereas, REC_TRUE and REC_FALSE are objects that specifically represent the TRUE and FALSE variants of BOOLEAN_RECORD respectively. As a side note, unconstrained elaborations, such as REC_1 and REC_2, are allowed only if the discriminant in the type declaration has a default value.

Direct assignment to a discriminant of an object is not allowed by the language rules; therefore, the only way to change the value of a discriminant of a variable is to assign a (complete) value to the variable itself. Thus, the only way to assign a new value to a variant record variable is via an aggregate assignment, provided the first value of the aggregate corresponds to the discriminant. Examples:

```
REC_1 := BOOLEAN_RECORD'(TRUE,PROCSYM,"ABCDE");
REC_FALSE := BOOLEAN_RECORD'(UNIQUE => FALSE);
-- Note: named notation is required for single valued aggregates
```

Example 3.5--Aggregate Assignment Statements

3.4.2 Summary

In order to have variant records in Ada one must elaborate record objects without specifying a value for the discriminant(s). Such object declarations are allowed only if the record's discriminant has a default value. Since direct assignments to discriminants are prohibited by the language rules, only an aggregate record assignment can change the value of the discriminant, and thus, change the value of the entire record.

3.5 STRONG TYPING

Ada is a strongly typed language. This means that objects of a given type may take on only those values that are appropriate to that type, and furthermore, the only operations supported for an object are those defined by its type. This section discusses some issues concerning Ada's typing mechanism, including:

1. parameter type mismatches,
2. derived types,
3. subtypes,

4. range constraints, and
5. operations on imported types.

3.5.1 Type Mismatches

Type mismatches between the formal and actual parameters of a program unit are a common mistake made when developing a large software tool. This kind of problem can be addressed in the design phase of the tool. For example, in the design of the AIM, a general pool of data types and objects was encapsulated in the AIM_SUPPORT package. The contents of this support package were then used by other program units to ensure consistency between parameter types, data types, and actual data objects. As further testimony to this design approach, other support packages evolved for the AIM, including HELP_INFO_SUPPORT, COMMAND_INTERPRETER_SUPPORT, STRING_UTILITIES, STACK_PACKAGE, and QUEUE_PACKAGE.

3.5.2 Derived Types

A derived type definition defines a new (base) type whose characteristics are derived from those of the parent type. Declaring a derived type can have interesting side effects. Consider the following:

```
with TEXT_IO;
procedure DERIVED_STRING is

    LENGTH : constant := 20;
    type MY_STRING_TYPE is new STRING(1..LENGTH);
    MY_STRING : MY_STRING_TYPE;

begin
    MY_STRING := "This should not work";
    TEXT_IO.PUT(MY_STRING);
end DERIVED_STRING;
```

Example 3.7--Derived Types.

In this example, MY_STRING_TYPE is a derived type of STRING with a range constraint. By declaring this 'new' string type, NONE of the TEXT_IO procedures with a parameter of type STRING can be used directly with the 'new' string type. To use the TEXT_IO procedures an explicit conversion, from MY_STRING_TYPE to STRING, is necessary; therefore, the code shown in example 3.6 is incorrect; the Data General/Rolm Ada compilation listing for this code is presented in example 3.7.

EXPERIENCES USING ADA LANGUAGE FEATURES

Derived Types

Ada 2.20.0.0 5/17/85 at 8:35:15

:USER1:TESTADA:TEST.ADA

```
-----+-----
1 |      with text_io;
2 |      procedure DERIVED_STRING is
3 |
4 |          LENGTH : constant := 20;
5 |          type MY_STRING_TYPE is new STRING(1..LENGTH);
6 |          MY_STRING : MY_STRING_TYPE;
7 |
8 |      begin
9 |
10 |          MY_STRING := "This should not work";
11 |          TEXT_IO.PUT(MY_STRING);
-----+-----
==>      TEXT_IO.PUT (MY_STRING);
***      Expanded name TEXT_IO.PUT has no definition that matches
          parameter list (MY_STRING).
-----+-----
12 |
13 |      end DERIVED_STRING;
=====+=====
```

Example 3.7--Incorrect Ada Program

3.5.3 Subtypes

A subtype, unlike a derived type, does not define a new type; rather, it provides a new name for another (potentially) constrained data type. As is implied here, a constraint on the base type of the subtype is optional; therefore, a subtype name can be a synonym of its base type's name.

This feature has an interesting, but potentially troublesome, application. It can be used within a program unit to locally define data types that are imported by that program unit. Consider the following:

```
package SYSTEM_SUPPORT is
```

```
    . . .
```

```
    MAX_NAME_LENGTH : constant := 20;    -- characters in a name
```

```
    type INTEGER_RANGE is range 1..MAX_INTEGER;  
    type NAME_INDEX_RANGE is range 1..MAX_NAME_LENGTH;
```

```
    subtype NAME is STRING( 1..MAX_NAME_LENGTH );
```

```
    . . .
```

```
end SYSTEM_SUPPORT;
```

```
with SYSTEM_SUPPORT;
```

```
package SOME_PACKAGE is
```

```
    subtype INTEGER_RANGE is SYSTEM_SUPPORT.INTEGER_RANGE;  
    subtype NAME_INDEX_RANGE is SYSTEM_SUPPORT.NAME_INDEX_RANGE;  
    subtype NAME is SYSTEM_SUPPORT.NAME;
```

```
end SOME_PACKAGE;
```

Example 3.8--Local synonyms for imported data types

Some of the types (and subtypes) of AIM_SUPPORT have been given local names in SOME_PACKAGE. This 'aliasing' localizes the scope of those imported types' names. This technique might be considered for heavily used imported types.

The disadvantage of aliasing types imported from another package involves traceability. Much like the context clause USE, aliasing types tends to make it difficult to determine the origin of the type(s). In essence, aliasing removes the need to completely qualify a type. Often during the maintenance phase, someone other than the original designer or implementor is involved with the code. It should be obvious that this added level of indirection has the potential for hindering the maintenance of a package.

3.5.4 Range Constraints

The ability to specify a range constraint for a given data type is a valuable feature of the Ada language and should be utilized whenever possible. By using range constraints, a programmer is insuring that the source code will be more portable than code that does not use them. Consider the following:

EXPERIENCES USING ADA LANGUAGE FEATURES

Range Constraints

```
subtype INT_1_TYPE is INTEGER;  
subtype INT_2_TYPE is INTEGER range 1..100;
```

The range of values for objects of subtype INT_2_TYPE is explicitly known to be 1..100, whereas, the range of values for objects of subtype INT_1_TYPE is implementation dependent. Thus, INT_2_TYPE is more portable than INT_1_TYPE since the latter's range of values could vary among compiler implementations. For example, a single precision INTEGER value on an 8-bit machine will have quite a different set of values from that of a 32-bit machine.

The use of previously declared range constraints in specifying the index range for an array is also a recommended practice. Meaningful names associated with index constraints of arrays can better represent real world abstractions.

3.5.5 Operations On Imported Types

Operations on types in Ada are restricted by the strong typing built into the language. For types that are imported from another package, even the operations on these types must be imported. This requires that operators, like other imported entities, must be completely qualified to be used. Noting that WITH clauses provide visibility and USE clauses provide direct visibility, consider the following example:

EXPERIENCES USING ADA LANGUAGE FEATURES
Operations On Imported Types

```
package MY_PACKAGE is
    type INTGR is new integer;
    YOURS : INTGR := 2;
end MY_PACKAGE;

with MY_PACKAGE;
use MY_PACKAGE;
procedure TEST is

    MINE : MY_PACKAGE.INTGR := 1;
begin
    MINE := MY_PACKAGE."+"(MINE,MY_PACKAGE.YOURS);
    YOURS := YOURS + MINE;
    MINE := MINE + MY_PACKAGE.YOURS;
end TEST;
```

Example 3.9--Operator notation for imported types

The prefix operator notation used in the first assignment statement is required when the package MY_PACKAGE is only visible to the procedure TEST. This is due to the Requirement to completely qualify all imported entities. The infix operator notation used in the second assignment statement is allowed when the imported package is directly visible. In the third assignment statement, objects are completely qualified as an aid in traceability of the object YOURS.

Admittedly, the infix notation is easier to read because of its widespread use. However, while significantly more cumbersome, the prefix notation does NOT hide the fact that the "+" operator is an imported operation on an imported type. As mentioned before, traceability can be a significant factor once a system is in the maintenance phase. With this in mind, a guideline should be adopted that recommends the avoidance of the USE clause. By normally only allowing visibility, and not direct visibility, the origin of imported types and their operations will be easily recognized due to the need to completely qualify all imported entities.

If a project determines that the readability of the infix notation

EXPERIENCES USING ADA LANGUAGE FEATURES

Operations On Imported Types

warrants the USE clause, all imported entities should still be completely qualified as part of a continued effort to aid traceability. Additionally, the effect of the USE clause can be localized by including it within individual procedures and functions rather at the package level.

3.6 STRING MANIPULATION

During the design phase of the AIM, there was a tendency to get "caught up" in the concept of typing and string lengths. Initially, the design used numerous string subtypes. During implementation, it was found that it is generally unnecessary to put extensive restrictions on strings. A few general guidelines were identified and are listed below. Note that each point is a natural progression from the previous point.

1. The formatting of strings to specific lengths should be deferred as long as possible.
2. String parameters to procedures and functions should be unconstrained whenever possible.
3. A string utility package should be implemented to perform many of the string manipulation functions required by the system.

3.6.1 Formatting Strings

The formatting of strings within string variables involves a number of possible operations including left or right justification and blank filling unused character positions. Since strings most often are left justified within a string variable, either automatically or by the programmer's choice, it can be advantageous to left justify all strings and maintain a length variable for each string variable. This minimal "pre-formatting" clearly defines the string and can reduce the logic required to copy or move a string (or string slice) from one string variable to another. In particular, the length variable eliminates a need to prematurely blank fill trailing character positions in a string variable. Ultimately, many intermediate operations can be reduced by following this plan. In turn, the total program size and the execution time can be reduced.

3.6.2 String Parameters

Procedures and functions should be designed to use unconstrained string parameters whenever possible. The benefits of designing in this manner include:

- * It is easier to use string slices as parameters,
- * string manipulations prior to making a call to the procedure or function can be reduced, and
- * the procedures and functions become less susceptible to design and implementation changes.

Unconstrained string parameters actually encourage the use of string slices and go a long way toward reducing the need for intermediate string manipulations. If a procedure is designed to accept unconstrained string parameters, the procedure is generalized and is, in fact, more transportable than a procedure with constrained string parameters. Fewer assumptions are made about the strings being used, thus providing a more stable procedure.

3.6.3 A String Utility Package

During the design and implementation of the AIM, it was found that a string manipulation package could be used to centralize many of the string functions required by the AIM. Much like the concept of centralized error handling (described in chapter 5 of this volume), a string utilities package can reduce the potential for unnecessary duplication of code. This can make the overall system easier to test and more dependable.

CHAPTER 4

ADA TASKING

4.1 INTRODUCTION

A task is one of Ada's four primary program units, the others being subprograms, packages, and generic units. An Ada task is very similar in form to a package as both are comprised of two textual parts: a specification that describes its external appearance and an executable body which defines its internal behavior. The major difference between the two program units is that a package is a passive construct providing visibility control whereas a task is active and provides the capability of parallelism.

It is the intent of this chapter to present a detailed discussion of some concurrent Ada programming issues that were uncovered during the implementation of the AIM program. A taxonomy of Ada tasks is first presented to establish the Ada tasking model's intrinsic Actor/Server relationship and to introduce the fundamental notion of Ada Tasking building blocks. Relative to these concepts, the balance of the chapter presents a discussion regarding: controlling Ada task start-up, graceful Ada task termination, and some specific details about our implementation of the AIM program.

4.2 A TAXONOMY OF ADA TASKS

Most collections of interacting tasks are organized in such a way that the individual tasks involved can be classified into functional groups, where these classifications are based on the functions the tasks perform relative to the overall system. In particular, from a given point of reference most Ada tasks can be broadly categorized as either Servers or Actors [BOO83, OLS83].

Usually, Actor tasks are active constructs which utilize Servers to accomplish their function, whereas Server tasks are passive constructs that react to the external requests generated by Actors. Of course there is nothing prohibiting a Server task from requesting service(s) from another task and thus creating an Actor/Server task.

ADA TASKING

A TAXONOMY OF ADA TASKS

This type of hybrid task is termed a Transducer task [BOO83].

Note: in the context of this chapter the terms Actor, Server, and Transducer will be used synonymously with Actor task, Server Task, and Transducer task respectively.

4.2.1 Server Tasks

There are obviously numerous services that Ada tasks could provide: queue/buffer management, shared resource control, message routing, asynchronous communications control, etc. Typically, a Server task providing services of this nature is implemented using an infinite loop whose body contains a selective wait statement with a terminate alternative, where the task's entries correspond to the various services that it provides. A short description of some of the more common types of Server tasks follows:

Server Task Type	Description/Characteristics
Agent	performs a short-lived job in behalf of another Ada task; dynamically created; access object pointing to it is often passed as an entry parameter between a Server/Actor task pair.
Slave	body is an infinite loop; continually performs a well-defined function in behalf of its master task.
Buffer	encapsulates a data structure(s); exports entries (OPEN, PUT, GET, CLOSE) corresponding to applicable data structure operations; infinite loop encapsulating selective wait with a terminate alternative; typically links Producer(Actor)/Consumer(Actor) task pair.
Scheduler	delays the acceptance of calls on particular entries, subject to prevailing conditions; infinite loop encapsulating selective wait with a terminate alternative; could be used to schedule access to a shared resource.
Synch	synchronization task; exports entries (LOCK, UNLOCK) to support mutual exclusion; infinite loop encapsulating selective wait with a terminate alternative.
Interrupt Handler	traps and handles hardware/software interrupts; stand alone task; its single entry is associated with handling a hardware/software interrupt; typically, simple sequential execution within task body.
Device Driver	combination of an Interrupt Handler and Buffer task; handles device interrupts; internally buffers device's

data (both input and output); infinite loop encapsulating selective wait with a terminate alternative.

4.2.2 Actor Tasks

Obviously, as a complement to the various types of Servers, there can be numerous variations of Actor tasks. The general characteristics of Actors include:

- * zero entries declared in its specification; however, sometimes a start entry is declared,
- * an infinitely executing body, and
- * entry calls to Servers from its body.

A short description of some of the more common types of Actors follows:

Actor Task Type	Description/Characteristics
Customer/ Consumer/ Requestor	makes entry calls to other (Server) tasks; requests services from Server tasks; consumes data produced by other (Actor) tasks; infinite loop body; does not export any entries except for perhaps START.
Producer	produces data to be processed by other Ada tasks; infinite loop producing data packets.
Monitor	infinite loop body; no entries other than a possible initiating one; performs a "watchdog" function.

ADA TASKING

Transducer Tasks

4.2.3 Transducer Tasks

Occasionally the complexity of real world problems is such that the simple Actor/Server tasking paradigm must be enhanced. Typically in such cases, the function of the Server task requires the services of other tasks, and thus, a hybrid Actor/Server or Transducer task is required. A task of this nature not only accepts entry calls from Actors, but also invokes the entries of other Servers. Usually, a Transducer task's body is textually similar to that of a Server in that it is composed of an infinite loop that encapsulates a selective wait statement with a terminate alternative. A short description of some of the more common types of Transducer tasks follows:

Transducer Task Type	Description/Characteristics
Message Manager	accepts requests to route messages to other tasks; calls entries of other tasks to pass along data; infinite loop encapsulating selective wait with a terminate alternative.
Secretary	not only provides services and schedules activities, but also calls other Servers to report the results [BUH84].

4.3 ADA TASKING BUILDING BLOCKS

Given the above taxonomy of Ada tasks, it is obvious that the Ada tasking model has an intrinsic Actor/Server task relationship. In particular, the majority of Ada tasks operate in conjunction with other tasks, as Actor/Server task pairs, rather than as stand-alone program units. Relative to the above Ada task taxonomy, consider the following typical Ada tasking scenarios:

Notational summary:

Symbol	Purpose/Use
--	Precedes comment text.
{ }	Indicates zero or more occurrences of the enclosed object.
()	Enclosed noun classifies a specific task according to the Actor/Server/Transducer paradigm.
<---->	Indicates tasking interaction.

Ada Tasking Set Scenarios

Monitor	-- Stand-alone	
Interrupt Handler	-- Stand-alone	
Any Master	<----> Slave (Server)	
Any Server	<----> { Agent (Server) }	<----> Customer (Actor)
Producer (Actor)	<----> { Buffer (Server) }	<----> Consumer (Actor)
Hardware Device	<----> Device Driver (Server)	<----> Any Actor
Actor		
.		
.	<----> Synch or Scheduler (Server)	
Actor		
Actor		Server
.		.
.		.
Actor	Secretary or <----> Message Dispatcher (Transducer)	<----> Server

ADA TASKING

ADA TASKING BUILDING BLOCKS

This Actor/Server task phenomenon leads to the assertion that most Ada tasks cooperate and function as a set rather than as stand-alone units and that, furthermore, there exist fundamental "building block" task sets from which more complicated concurrent systems may be built.

4.3.1 Canonical Ada Task Sets

The majority of concurrent systems can be developed using a finite set of fundamental (canonical) Ada task building blocks consisting of various interactions of Server, Actor, and Transducer tasks. This finite set of Ada task building blocks consists of the following canonical forms:

Set	Description	Notation	Example
1.	one-to-one:	A <----> S	Customer/Server
2.	one-to-many:	A <----> S S	Customer/Multiple Servers
3.	many-to-one:	A A <----> S A	Producer/Buffer/Consumer
4.	many-to-many:	A S A <----> S A S	
5.	one-or-more- Transducers:	A <----> T{T} <----> S	Message Sender/ Message Dispatcher/ Message Receiver

Note: there are some underlying assumptions that have been observed for the construction of this canonical task set classification: the tasks' inter-relationships (descriptions) are relative to the Actor task, an Actor is always associated with a least one Server and vice versa, Actor/Server tasks occur in alternating pattern pairs, and a Transducer is textually similar to a Server (has a Server-like body which makes entry calls to other tasks). Of these canonical tasking sets, the most widely used include types 1, 3, and 5, from which most concurrent systems can be built.

4.3.2 A Simple Example

This section presents the technique for constructing a simple concurrent system using the aforementioned canonical task sets.

Consider the situation where it is necessary to asynchronously buffer data that is being entered via a CRT; furthermore, assume that numerous terminal data consumers may exist. A small concurrent system of this nature could be constructed as follows:

Step	Canonical Task Set	Description
1	A1 <----> S1 <----> CRT	Classic Customer/Server.
Explanation: The task S1 acts as a filter between the Actor task A1 and the terminal. S1 loops, first reading data from the terminal and then accepting a GET entry call from A1 (or any other Actor task) to pass the data back.		
2	A2 <----> S2 <----> A3	Classic Producer/Buffer/Consumer.
Explanation: This task set is the classic A-S-A configuration where the Actors are a Producer and Consumer respectively and the Server is a Buffer task. This task set will asynchronously buffer the data that is being produced by task A2, and subsequently consumed by task A3.		
3	A12 <----> S1 <----> CRT ^ +-----> S2 <----> A3	Merge results of steps 1 and 2.

Explanation:

As the final step in our solution, the A-S task set from step 1 is merged with the A-S-A task set from step 2. Thus, via merging these task sets together, we have a new task, namely A12, which incorporates the functions of tasks A1 and A2. In this final step, task A12 is a Consumer/Producer which first consumes data from task S1 and then, secondly, produces data for buffering by task S2.

This technique will be examined further in the context of creating a fully asynchronous data pipeline as the AIM's underlying data flow model, but first a discussion of task initiation and termination is in order.

4.4 ADA TASK START-UP

Ada's tasking model defines the execution of an Ada task as a two-phase process: task activation and the normal execution of the task body's statements. Task activation, which consists of elaborating the declarative part, if any, of the task body, occurs automatically after the elaboration of the declarative part of the task's parent unit. This activation occurs collectively for every task object defined within the same declarative part; note that the implied parent unit's execution is suspended at its initial BEGIN until all of the tasks in the activation collection have been completely activated. For each task in an activation collection, the first statement after its declarative part is executed only after the conclusion of the activation of every task of the implied collection of task objects.

4.4.1 Control Of A Starting Task's Execution

Since the start of a task body's execution is, by default of the language definition, a well-defined, yet spontaneous event, a programming mechanism is often needed for controlling the execution start of all the tasks defined in a system. This task synchronization mechanism can be implemented by placing the following simple selective wait statement at the beginning of each task's body:

```
select
  accept START;
or
  terminate;
end select;
```

Of course, a sequence of statements representing the task's initialization can follow either the "accept START;" statement or the "end select;" line. Furthermore, depending on the nature of a task's inter-dependence with others in its tasking set, the START entry may have a tag parameter that is needed for the proper functioning of the task. For instance, in the following example, the CONSUMER_PRODUCER task body is that of a task type which must interact with two sibling tasks that, in addition to itself, will be dynamically created as part of an object of the record data type WINDOW. In this case, the CONSUMER_PRODUCER has no notion of the name of its parent WINDOW record object and thus of its sibling tasks with which it must interact, and therefore, the creator of a WINDOW object must pass the appropriate WINDOW tag into the CONSUMER_PRODUCER task via its START entry (lines 8--10). As a final note, the terminate alternative (line 12) of this initial selective wait statement is a programming safeguard which prevents a deadlock situation during the start-up of a system's tasks.

```
1  separate (WINDOW_MANAGER)
2  task body CONSUMER_PRODUCER is
3      MY_WINDOW           : WINDOW;
4      MY_PACKET           : WINDOW_DATA_RECORD;
5      WINDOW_QUEUE_IS_OPEN : BOOLEAN := TRUE;
6  begin
7      select
8          accept START (WINDOW_ID : WINDOW) do
9              MY_WINDOW := WINDOW_ID;
10         end START;
11      or
12         terminate;
13      end select;
14
15      loop
16          MY_WINDOW.QUEUE_MANAGER.READ_DATA_FROM_QUEUE
17              (MY_PACKET, WINDOW_QUEUE_IS_OPEN);
18
19          exit when not WINDOW_QUEUE_IS_OPEN;
20
21          MY_WINDOW.WINDOW_BUFFER.PUT_DATA (MY_PACKET);
22      end loop;
23  end CONSUMER_PRODUCER;
```

Example 1--Task start-up via an initial selective wait statement.

4.5 ADA TASK TERMINATION

Like task execution, task termination occurs in a two-stage process: first, a task must complete its execution, and then, secondly, it can be terminated. A task is said to have completed its execution when "it has finished the execution of the sequence of statements that appears after the reserved word BEGIN in the corresponding body" [DOD83]. There are really three basic scenarios that are possible when terminating a task named FOO:

1. FOO has no dependent tasks -- FOO terminates when it has completed its execution,
2. FOO has dependent tasks all of which have already terminated -- FOO terminates when it has completed its execution, and
3. FOO has dependent tasks that have not terminated -- there are two subcases underneath this scenario:

ADA TASKING
ADA TASK TERMINATION

- * FOO has completed its execution -- FOO terminates when all of its dependent task are terminated.
- * FOO's execution has reached an open terminate alternative in a select statement -- FOO can terminate, if and only if, the following conditions are satisfied:
 - The task (FOO) depends on some master whose execution is completed.
 - Each task that depends on the master considered is either already terminated or similarly waiting on an open terminate alternative of a select statement." [DOD83]

When both of these conditions are satisfied FOO becomes terminated together with all the other tasks which also depend on the implied master.

4.5.1 Graceful Termination Of Canonical Task Sets

Due to the nature of an Actor/Server task pair, task termination in this context warrants careful consideration. In particular, a Server presents no termination problem as it typically has a selective wait statement with a terminate alternative, and thus, terminates on its own; however, due to the infinite loop nature of an Actor task, some type of programmed termination mechanism is required to assure a graceful shutdown. Consider the following canonical task set:

```
Consumer  Buffer
A <----> S
```

The Actor (Consumer task) loops infinitely performing its designated function, whereas the Server (Buffer task) selectively services the Actor's requests. In this situation it is incumbent on the Server task to somehow indicate to the Actor when the services will be discontinued (i.e.--when the Server is ready to terminate). The easiest technique for accomplishing this shutdown mechanism is to:

1. export an entry from the Server task to allow an external unit to trigger its shutdown, and
2. indicate to the Actor task(s), via a BOOLEAN entry call parameter, that the services are being discontinued,

as is demonstrated in the following Ada code from the AIM program

ADA TASKING
Graceful Termination Of Canonical Task Sets

(Examples 2(a) and 2(b)):

```
1  separate (WINDOW_MANAGER)
2
3  -----
4  --
5  -- Abstract      : Infinitely loop getting data packets from my window's
6  -----: queue and then forwarding that packet to my window.
7  --
8  -----
9  task body WINDOW_QUEUE_CONSUMER is
10     MY_WINDOW      : WINDOW;
11     MY_PACKET      : WINDOW DATA RECORD;
12     WINDOW_QUEUE_IS_OPEN : BOOLEAN := TRUE;
13 begin
14     select
15         accept START (WINDOW_ID : WINDOW) do
16             MY_WINDOW := WINDOW_ID;
17         end START;
18     or
19         terminate;
20     end select;
21
22     loop
23         MY_WINDOW.QUEUE_MANAGER.READ_DATA_FROM_QUEUE
24             (MY_PACKET, WINDOW_QUEUE_IS_OPEN);
25
26         exit when not WINDOW_QUEUE_IS_OPEN;
27
28         MY_WINDOW.WINDOW_BUFFER.PUT_DATA (MY_PACKET);
29     end loop;
30 end WINDOW_QUEUE_CONSUMER;
```

Example 2(a)--A Simple Consumer Task.

ADA TASKING

Graceful Termination Of Canonical Task Sets

```

1  with QUEUE;
2  separate (WINDOW_MANAGER)
3  task body WINDOW_QUEUE_MANAGER is
4    package WQ is new QUEUE(WINDOW_DATA_RECORD);
5    AIM_NAME      : constant NAME := "AIM" & (4..NAME'LAST => ' ');
6    MY_PACKET     : WINDOW_DATA_RECORD;
7    NULL_PACKET   : WINDOW_DATA_RECORD;
8    WINDOW_QUEUE_IS_OPEN : BOOLEAN := FALSE;
9  begin
10   select
11     accept START;
12       NULL_PACKET := (AIM_SUPPORT.FROM_TERMINAL,
13                       WINDOW_MANAGER.NULL_WINDOW, null);
14       WINDOW_QUEUE_IS_OPEN := TRUE;
15   or
16     terminate;
17   end select;
18   loop
19     select
20       accept ABORT_QUEUE_PROCESSING;
21         WINDOW_QUEUE_IS_OPEN := FALSE;
22     or
23       accept QUEUE_DATA_TO_WINDOW( DATA_PACKET : WINDOW_DATA_RECORD;
24                                     QUEUE_IS_OPEN : out BOOLEAN ) do
25         MY_PACKET := DATA_PACKET;
26         QUEUE_IS_OPEN := WINDOW_QUEUE_IS_OPEN;
27       end QUEUE_DATA_TO_WINDOW;
28       if WINDOW_QUEUE_IS_OPEN then
29         WQ.INSERT_ELEMENT(MY_PACKET);
30       end if;
31     or
32       when not WQ.QUEUE_IS_EMPTY or else
33         not WINDOW_QUEUE_IS_OPEN =>
34         accept READ_DATA_FROM_QUEUE( DATA_PACKET : out WINDOW_DATA_RECORD;
35                                       QUEUE_IS_OPEN : out BOOLEAN ) do
36           if WINDOW_QUEUE_IS_OPEN then
37             WQ.GET_ELEMENT(DATA_PACKET);
38           else
39             DATA_PACKET := NULL_PACKET;
40           end if;
41           QUEUE_IS_OPEN := WINDOW_QUEUE_IS_OPEN;
42         end READ_DATA_FROM_QUEUE;
43     or
44       terminate;
45   end select;
46   end loop;
47 end WINDOW_QUEUE_MANAGER;

```

Example 2(b)--An Intermediate Buffering Task.

A few notes are in order for this code:

- * the Server has an ABORT_QUEUE_PROCESSING entry which allows an external unit to shutdown the WINDOW_QUEUE processing (2(b), line 20),
- * the semantics of this ABORT_QUEUE_PROCESSING entry is to merely set an internal flag, WINDOW_QUEUE_IS_OPEN, to FALSE (2(b), line 21),
- * the WINDOW_QUEUE_IS_OPEN flag's value is passed back to the calling tasks via the Server's read and write entries (2(b), lines 26,41)
- * the Actor is an infinite loop which terminates when the Server returns the fact that the QUEUE is closed (2(a), line 26),
- * the Server terminates on its own via the terminate alternative (2(b), lines 30, or 44).

4.5.2 Ada Task Termination Concerns

Each Ada task in a system depends on at least one master, where "a master is a construct that is either a task, a currently executing block statement or subprogram, or a library package (a package declared within another program unit is not a master)." [DOD83] Furthermore, this dependence on a master is "a direct dependence in the following two cases:

1. The task designated by a task object that is the object, or a subcomponent of the object, created by the evaluation of an allocator depends on the master that elaborates the corresponding access type definition.
2. The task designated by any other task object depends on the master whose execution creates the task object." [DOD83]

As a consequence of dependency case 1, all accessed tasks are dependent on the program unit which contains their access type definition rather than the one containing their allocator. In this vein, it is very easy to develop code containing tasks which are directly dependent on library packages. Consider the following example:

ADA TASKING

Ada Task Termination Concerns

```

1  package WINDOW_MANAGER is
2      type WINDOW is private;
    . . .
22 private
23     task type WINDOW_QUEUE_CONSUMER is
24         entry START (WINDOW_ID : WINDOW);
25     end WINDOW_QUEUE_CONSUMER;
26
27     task type WINDOW_QUEUE_BUFFER is
28         entry START;
29         entry RESUME_WINDOW_OUTPUT;
30         entry PUT_DATA(DATA_PACKET : WINDOW_DATA_RECORD);
31     end WINDOW_QUEUE_BUFFER;
32
33     task type WINDOW_QUEUE_MANAGER is
34         entry START;
35         entry ABORT_QUEUE_PROCESSING;
36         entry QUEUE_DATA_TO_WINDOW(DATA_PACKET : WINDOW_DATA_RECORD;
37                                     QUEUE_IS_OPEN : out BOOLEAN);
38         entry READ_DATA_FROM_QUEUE(DATA_PACKET : out WINDOW_DATA_RECORD;
39                                    QUEUE_IS_OPEN : out BOOLEAN);
40     end WINDOW_QUEUE_MANAGER;
41
42     type WINDOW_BUFFER_PTR is access WINDOW_QUEUE_BUFFER;
43     type WINDOW_CONSUMER_PTR is access WINDOW_QUEUE_CONSUMER;
44     type WINDOW_MANAGER_PTR is access WINDOW_QUEUE_MANAGER;
    . . .
59     type WINDOW_RECORD;
60     type WINDOW is access WINDOW_RECORD;
61
62     type WINDOW_RECORD is
63         record
64             WINDOW_NAME           : NAME;
65             CONTENT                : WINDOW_ACCESS;
66             NEXT                   : WINDOW;
67             PREVIOUS               : WINDOW;
68             SUSPENDS_OUTPUT_ON_FULL : BOOLEAN;
69             IS_FULL                : BOOLEAN;
70             CURRENT_LINE_POSITION : NATURAL;
71             VIRTUAL_CURRENT_LINE   : NATURAL;
72             INPUT_PAD              : PAD_RECORD;
73             OUTPUT_PAD             : PAD_RECORD;
74             BUFFER                 : WINDOW_BUFFER_PTR;
75             QUEUE_MANAGER          : WINDOW_MANAGER_PTR;
76             QUEUE_CONSUMER         : WINDOW_CONSUMER_PTR;
77         end record;
78 end WINDOW_MANAGER;

```

Example 3--Library Package Task Dependency

Since the access types: WINDOW_BUFFER_PTR, WINDOW_CONSUMER_PTR, and WINDOW_MANAGER_PTR are declared within the specification of the WINDOW_MANAGER package, any accessed tasks created by the allocation of a WINDOW object will be directly dependent on the WINDOW_MANAGER package.

Although this is in accord with the normal scoping rules for objects created by allocators, an interesting transportability issue is left unresolved by the Ada language definition: are tasks dependent on library packages required to terminate? Fortunately, every Ada compiler that the AIM project has used to date chose to terminate such tasks.

4.6 ADA TASKING AND AIM DATA FLOW

It is the purpose of this section to use the already established task building blocks to incrementally reconstruct a skeleton of the AIM's underlying asynchronous data flow model as a proof of concept.

4.6.1 AIM Data Flow Model Skeleton

The AIM's underlying data flow model is essentially an asynchronous data pipeline. As the term pipeline implies, this model is truly built upon small individual pipe segments that were ultimately dovetailed together. The key segment of the pipeline was the A-S-A task building block as it is present in no fewer than four places. During the implementation of the AIM program, the following ground-level task building blocks were identified and used to construct the underlying data flow model:

Component	Type	Description

1.	A-S-CRT	Keyboard Watcher/TTY Server

Explanation:

The TTY Server task acts as a filter between the Keyboard Watcher task and the user's terminal. It first reads data from the terminal and then accepts an entry call from the Keyboard Watcher to pass the data back. This is identical to step 1 of the previous asynchronous buffer example.

ADA TASKING
AIM Data Flow Model Skeleton

2. A-S-A Keyboard Watcher/Image Buffer/
Image Queue Consumer

Explanation:

This task set is the classic A-S-A configuration where the Actors are a Producer and Consumer respectively and the Server is a Buffer task. The Image Buffer task asynchronously buffers the data that is being produced by the Keyboard Watcher, and subsequently consumed by the Image Queue Consumer task. This is identical to step 2 of the previous asynchronous buffer example. This component is truly the first segment of the AIM's data pipeline. Note that the Keyboard Watcher task links components 1 and 2 as it functions in a Consumer/Producer capacity. See Figure 4.1.

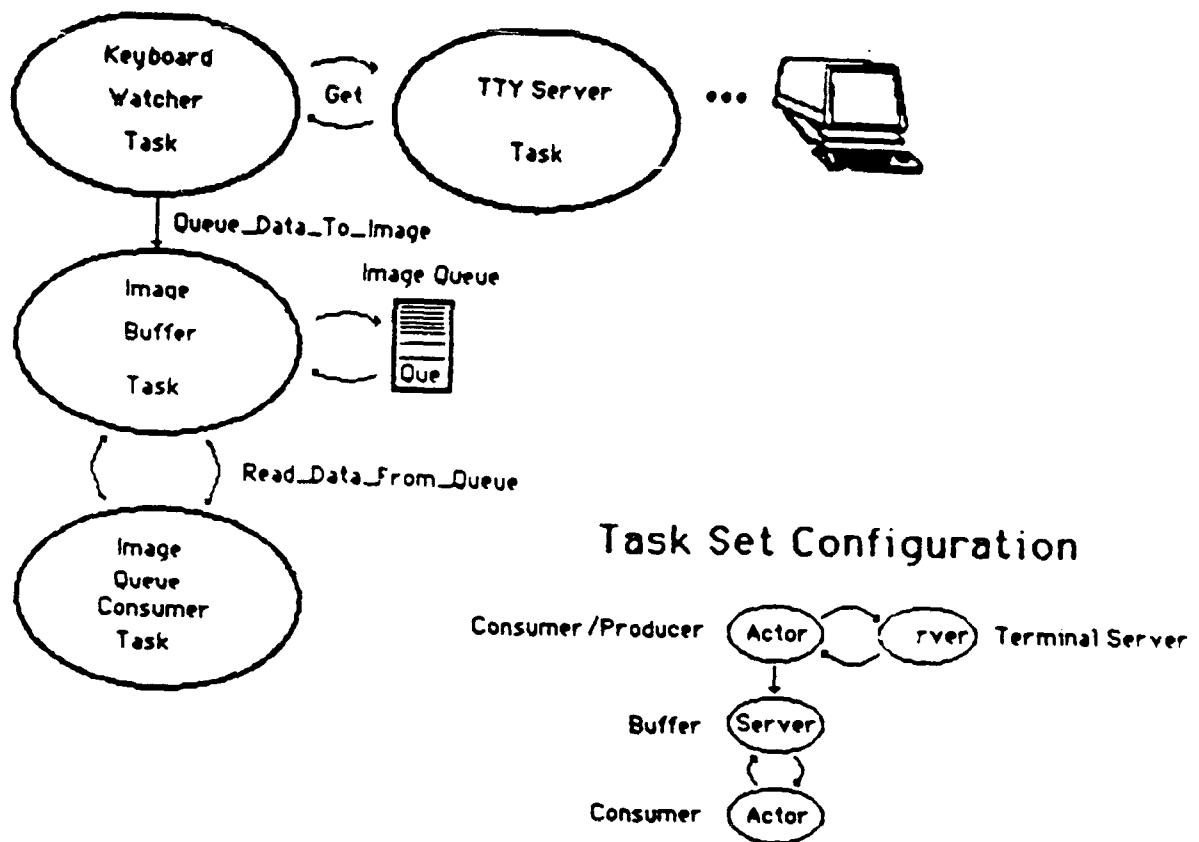


Figure 4.1--Segment 1 of AIM Data Flow Pipeline.

3. A-S-A Image Queue Producer/Window Queue Manager/
Window Queue Consumer

Explanation:

This task set is also a classic A-S-A configuration where the Actors are a Producer and Consumer respectively and the Server is a Buffer task. It is the second segment in the data pipeline. The Window Queue Manager task will asynchronously buffer the data that is being produced by the Image Queue Producer, and subsequently consumed by the Window Queue Consumer task. Note that in the final construction of the entire data flow pipeline, the Image Queue Consumer (of component 2) and the Image Queue Producer tasks will be merged into one Actor task functioning in a Consumer/Producer capacity. See Figure 4.2.

Task Set Configuration

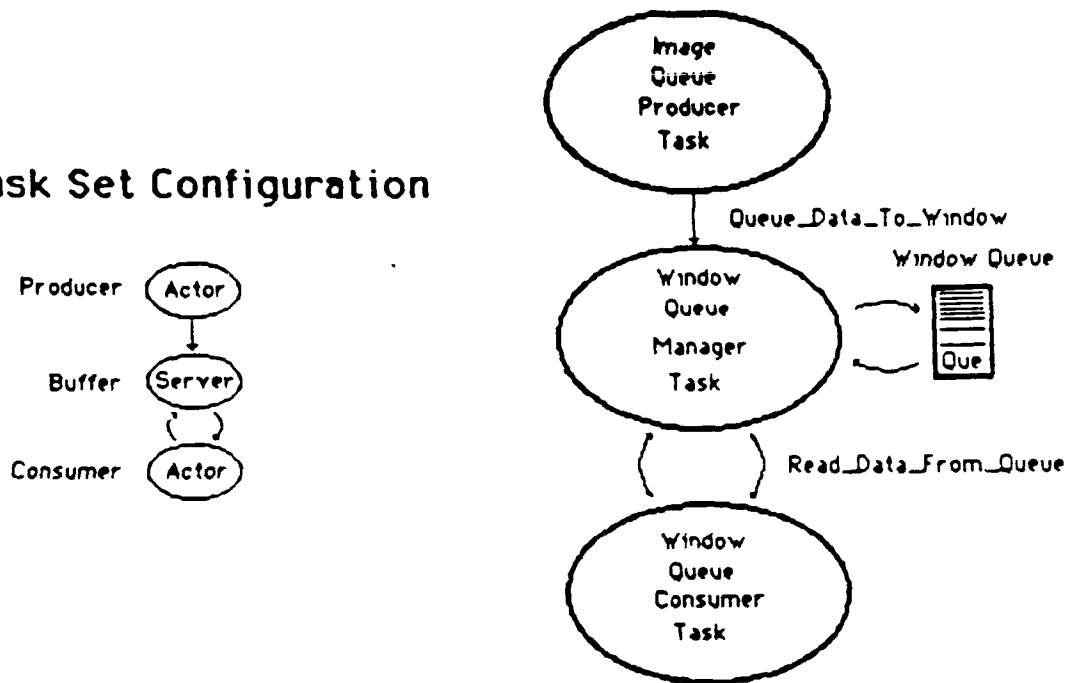


Figure 4.2--Segment 2 of AIM Data Flow Pipeline.

4. A-S Window Queue Producer/Window Buffer

Explanation:

This is the third component of the data pipeline. It is a Actor/Server task pair. The Window Queue Producer task pushes data packets at the Window Buffer task; subsequently, the Window Buffer task will accept these data packets assuming the internal AIM window with which it is associated is not full. Note that in the final construction of the entire data flow pipeline, the Window Queue Consumer (of component 3) and the Window Queue Producer task will be merged into one Actor task functioning in a Consumer/Producer capacity. See Figure 4.3.

Task Set Configuration

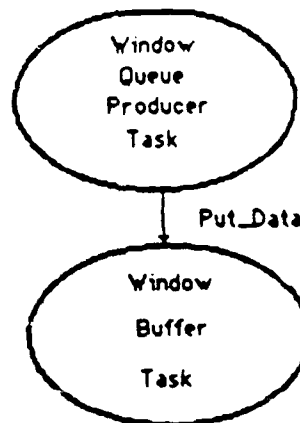
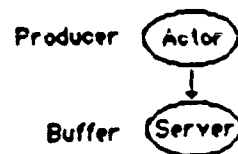


Figure 4.3--Segment 3 of AIM Data Flow Pipeline.

5. A-S Window Buffer/Synchronization

Explanation:

This Actor/Server task pair implements the AIM's system synchronization technique. It assures mutually exclusive access to the AIM's internal data structures. See Figure 4.4.

Task Set Configuration

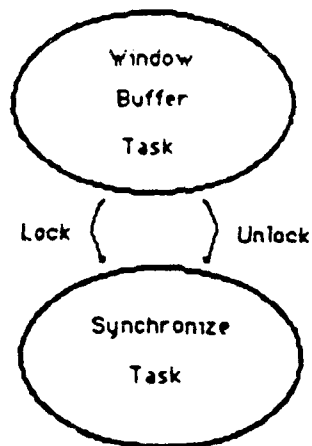
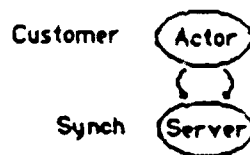


Figure 4.4--AIM System Synchronization Technique.

6. A-S Window Buffer/Script Handler

Explanation:

This Actor/Server task pair implements the AIM's Script file processing. The Script Handler task has entries for starting and stopping the processing of an AIM command file. This STOP entry allows for an immediate break-in and abort of the script file processing by the user. See Figure 4.5.

Task Set Configuration

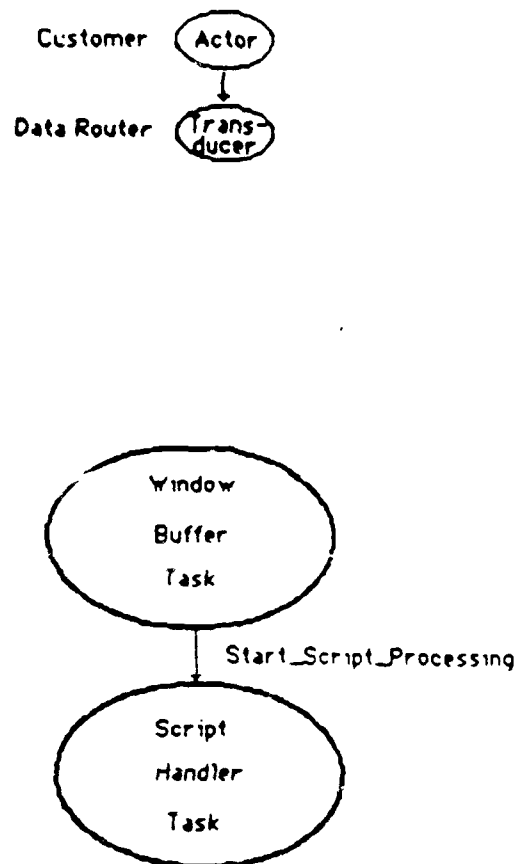


Figure 4.5--AIM Script File Handler.

7. A-S-A Write_To_Window/Buffer/Read_From_Process

Explanation:

This task set is also a classic A-S-A configuration where the Actors are a Producer and Consumer respectively and the Server is a Buffer task. It is the fourth segment in the data pipeline. The Buffer task will asynchronously buffer the data that is being produced by the Read_From_Process task. The Write_To_Window task consumes the process output from the buffer and pushes it at the Window Queue Manager task of component 3. See Figure 4.6.

Task Set Configuration

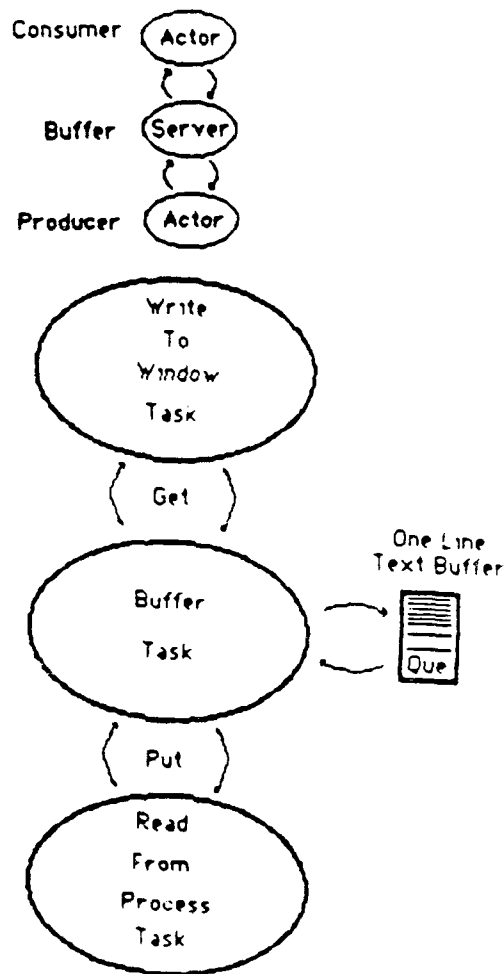


Figure 4.6--Segment 4 of AIM Data Flow Pipeline.

ADA TASKING
AIM Data Flow Model Skeleton

8. A-S-A Read_From_Process/Process Buffer/Process Watcher

Explanation:

This task set is also a classic A-S-A configuration where the Actors are a Consumer and Producer respectively and the Server is a Buffer task. It is the fifth segment in the data pipeline. The Process Buffer task will asynchronously buffer the data that is being produced by an APSE process, and thus the Process Watcher task. The Read_From_Process task consumes the data produced by an APSE process as needed and pushes it at the Buffer task of component 7. Note that the Read_From_Process task of this component is the exact same one as in component 7. See Figure 4.7.

Task Set Configuration

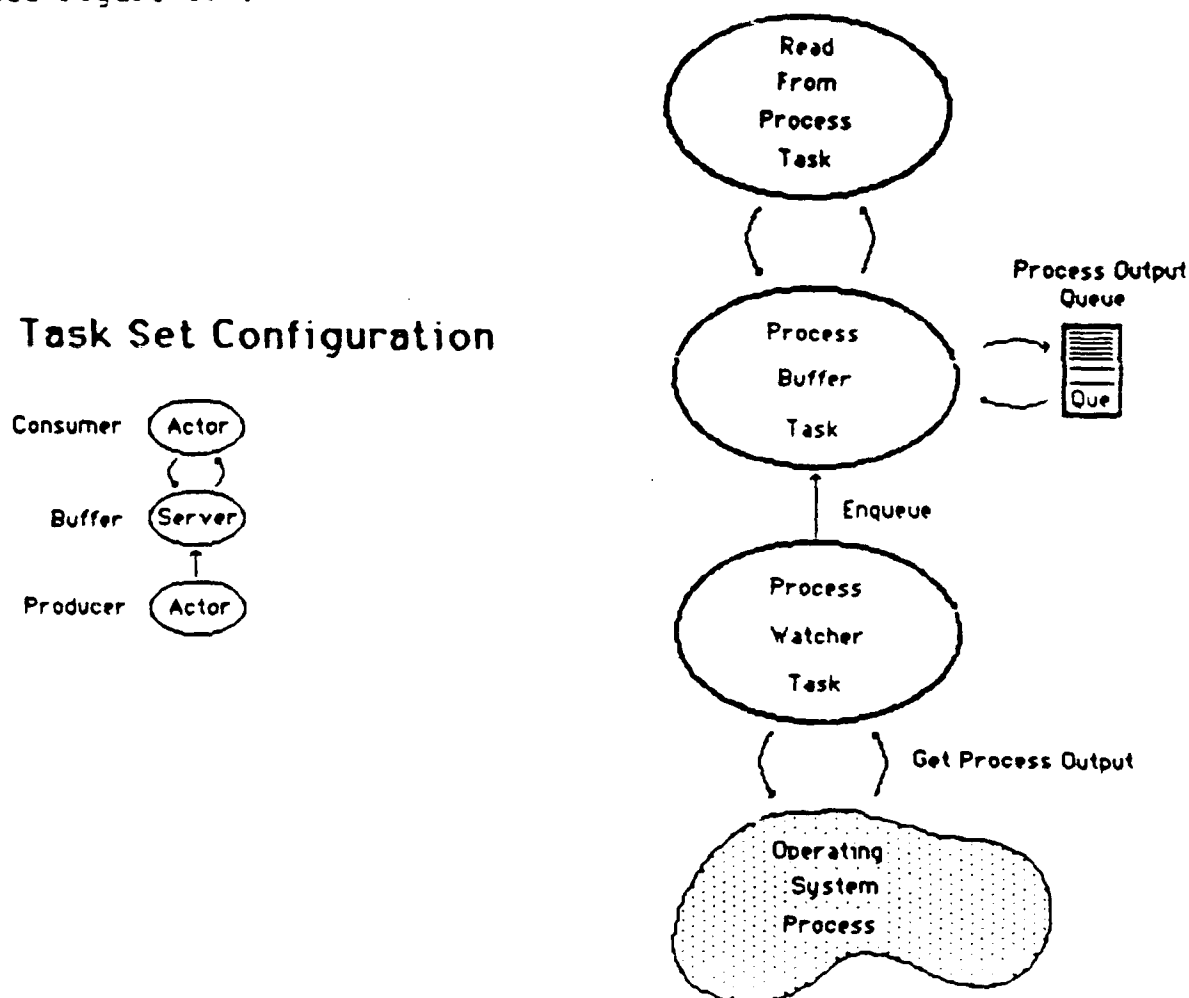


Figure 4.7--Segment 5 of AIM Data Flow Pipeline.

A skeleton of the entire AIM data flow model appears in Figure 4.8.

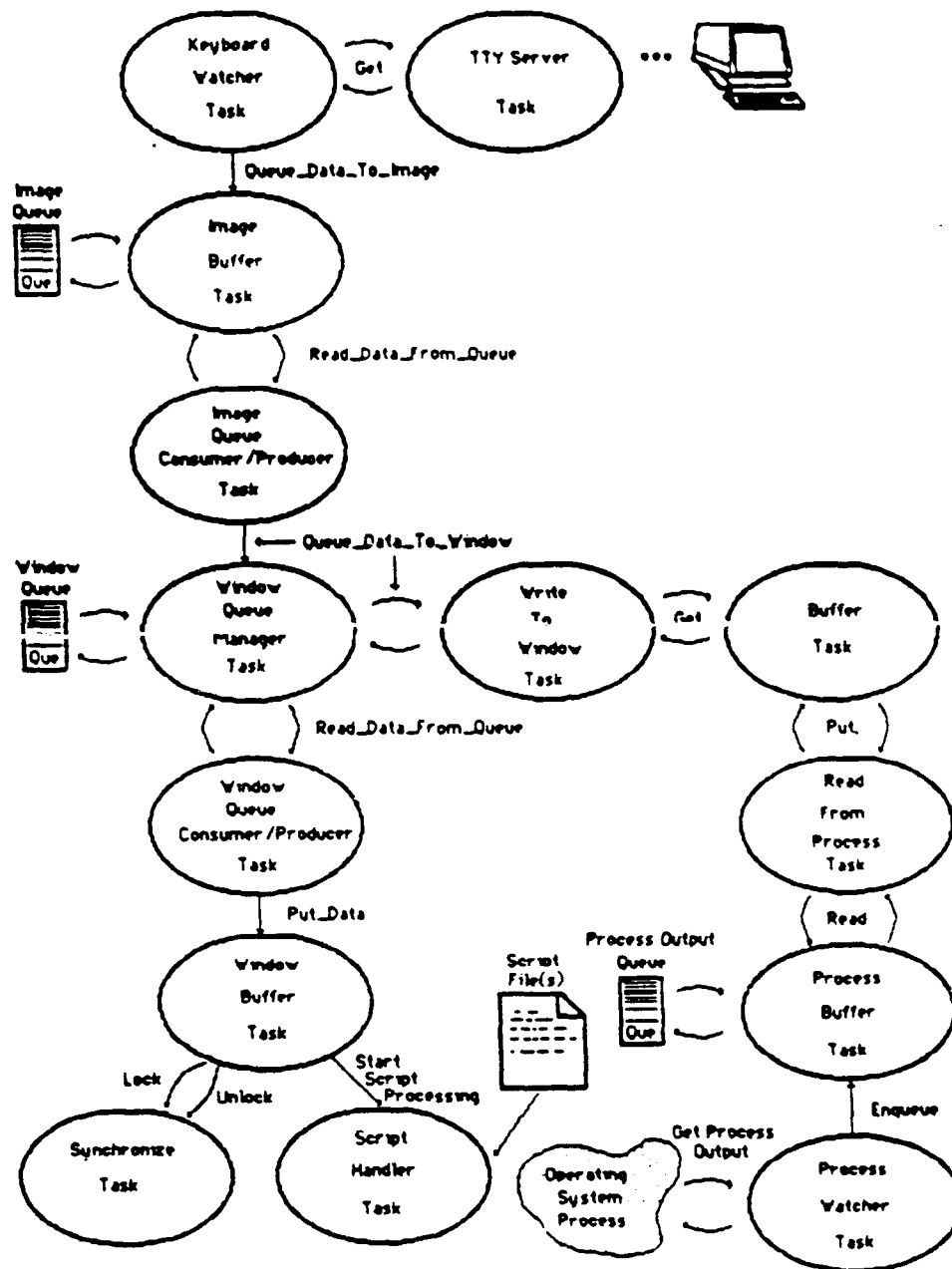


Figure 4.8--AIM Data Flow Model

ADA TASKING
AIM Data Flow Model Skeleton

A few general notes about the AIM implementation are in order:

- * In the actual implementation, various subprograms were used as intermediate steps between the tasks as presented in Figure 8.
- * The task set comprised of the Window Queue Manager, Window Queue Consumer/Producer, and the Window Buffer tasks, as well as that comprised of the Write To Window, Buffer, and Read From Process tasks, are dynamically created for every window in the AIM system.
- * The interface to a window's Window Queue Manager task is via a subprogram defined in the WINDOW_MANAGER package; furthermore, this interface uses the (passed in) WINDOW tag to call the appropriate window's Queue Manager Task.

4.7 CONCLUSION

A taxonomy of Ada tasks was presented in order to establish the notion of canonical task sets and to expose the intrinsic Actor/Server task relationship of the Ada tasking model. Following this common thread, the issues of task initiation, interaction, and termination were presented. Lastly, as a proof of the Ada tasking building block concept, a skeleton of the AIM's underlying data flow model was incrementally re-constructed using application specific clones of the previously established tasking building blocks.

There are numerous conclusions that may be drawn from these experiences:

- * Most Ada tasks can be broadly categorized as either Servers or Actors.
- * Most Ada tasks cooperate and function as a set rather than as stand-alone program units.
- * The majority of concurrent systems can be developed using a finite set of fundamental (canonical) Ada task building blocks consisting of various interactions of Server, Actor, and Transducer tasks.
- * Actor/Server task pairs occur in alternating patterns (A-S-A-S...)
- * A programming mechanism is often needed for controlling the execution start of all the tasks defined in a task set; furthermore, depending on the nature of a task's inter-dependence with others in its tasking set, this START entry may require a

tag parameter for the proper functioning of the task.

- * Due to the infinite loop nature of an Actor task, some type of programmed termination mechanism is required to assure a graceful shutdown; therefore, it is incumbent on a Server to trigger the termination of its associated Actors.
- * A complex concurrent system can be easily built by:
 - a. enumerating the fundamental task sets involved,
 - b. identifying the overlapping components of the fundamental task sets involved, and lastly,
 - c. dovetailing all the task sets together.
- * The programming technique of encapsulating a task(s) within a package and then exporting a set of procedural interfaces which parallels its entries is quite effective; furthermore, as an extension to this programming concept, the body of Server task can be encapsulated within the context of a generic package providing a parameterized Server task interface.

CHAPTER 5

EXCEPTIONS AND ERROR PROCESSING

5.1 INTRODUCTION

Ada's exception handling capabilities have been both lauded and condemned. Raising an exception has been called everything from "a disciplined way of coordinating actions on different levels" [GOC75] to "dangerous" [HOA81]. Others regard exceptions as a necessary evil; a "user beware" feature of the language. Many Ada proponents feel that the exception feature, when used carefully, can be a valuable mechanism for graceful error handling and program control.

Exceptions are important in the development of large Ada programs, not only at the implementation level, but also at the design level. Exceptions can serve as communication tools between procedures and packages; therefore, they become an interface issue.

The AIM project illuminated some of the problems associated with exceptions in the design stages of a project. Unlike data structures and procedures, exceptions do not easily "fall out" of Object-Oriented Design. In fact, significant effort was required to keep track of exceptions, where they could be raised, and where they were handled. The fact that exceptions may be used for purposes other than error handling added to the confusion. Exception propagation was found to require extreme care to avoid transferring control to the wrong handler. We experimented with both local and central error handling, finding merits and faults in both approaches.

5.2 ERROR HANDLING TECHNIQUES

Error detection and recovery for interactive systems is rather nebulous at the design methodology level. Judging from recent literature on software engineering and design, there are numerous techniques for identifying data structures and refining algorithms, while errors are left to be detected and handled almost at the whim of the software designer.

PTIONS AND ERROR PROCESSING R HANDLING TECHNIQUES

Object-Oriented Design, the methodology employed by the AIM design team, places heavy emphasis on discerning real-world objects which are later abstracted into data structures and encapsulated into packages. The actions performed upon these objects become the procedures and functions which define how the data structures are related and how the system works. The methodology, however, does not address how to design what happens when the system does NOT work. Perhaps this deficiency is the fault of the problem definition; however, the methodology does not encourage thought along these lines. (A thorough study of the Object-Oriented Design methodology can be found in chapter 2 of this volume.)

The apparent neglect of error handling in design methodologies is ironic, in light of its importance to the actual performance of the system. According to [PER83], error checking code comprises greater than half of all code in an interactive system, since "... there are usually many ways that things can go wrong and only one way in which they can go right". It is incongruous to design half of a system very methodically and formally, only to "ad-hoc" the other half. This approach is made likely, however, by the lack of support in current design methodologies for error handling. Error detection is often relegated to the ranks of 'implementation detail' rather than 'design issue'. Error recovery then becomes more of an afterthought than a design criterion.

In spite of the lack of formal definition and design techniques, error handling can be divided into two approaches: in-line and centralized error handling.

5.2.1 In-line Error Handling

Error handling is most easily performed at the place of the error detection. This approach, however, encourages inconsistent error handling and possibly duplication of code. Each error is handled separately, making error message standardization difficult and system responses variable. Similar errors may not be handled similarly, especially in large systems which require a group of software engineers to design. Engineers will usually develop error handling in their own style, and standardizing this effort requires a phenomenal feat of coordination. Following is a simple example of in-line error handling:

```
procedure DO_STUFF is
    NO_DATA      : boolean;
    INPUT_ERROR  : boolean;
begin
    ...
    if INPUT_ERROR then
        text_io.put (SOME_DEVICE, "You goofed, try again!");
    elsif NO_DATA then
        text_io.put ("Enter some data!");
    else
        DO MORE_STUFF;
    end if;
    ...
end DO_STUFF;
```

Example 5.1

While this approach is simple to code, the decentralized error handling can lead to programming bugs. In the above example, our programmer has left out the "SOME_DEVICE" parameter in the second TEXT_IO.PUT, which means that the second message will probably be written to the wrong file or device. Even trivial errors such as this one take valuable time to debug.

In-line error handling also makes it difficult to vector or defer the handling of an error to another scope level. For example, a low-level subprogram could detect a serious error which must be handled by a driver subprogram. If the low-level subprogram was not called directly by the driver subprogram, the flow of program control may need to "rise through" (return through the calling chain) several scope levels until the appropriate level is reached. This means that error checks must be at each scope level to allow control to return to the higher level routine. Each routine along the path of the error detection must contain code to handle or vector the error.

In general, the maintainability of code with in-line error handling is also reduced, since changes in the system are likely to cause changes in the error handling and recovery. If the error code is widely dispersed throughout the system, changes could lead to confusion, bugs, and fractured code.

EXCEPTIONS AND ERROR PROCESSING

Centralized Error Handling

5.2.2 Centralized Error Handling

An improvement over the in-line 'ad-hoc' approach is to centralize error handling to one routine or a series of routines. This approach serves to consolidate code and make error handling more consistent. Diagnostics are made more difficult, however, because the error routine must have a parameter for every data type needed for error handling. For example, a low-level parsing routine which detects an error in an identifier could call the error routine to handle "invalid identifier", but in order for the error routine to notify the user which identifier is in error, the error routine must know the representation of an identifier. This means additionally that the rest of the code must 'know' a lot about the error routine and what parameters it expects. Vectoring to another level for error handling is still difficult, even with a centralized routine. The centralized routine will always return to its point of call, so once an error is handled in a low-level routine, returning to a higher scope level must be accomplished in the same manner described above for in-line error handling. Example 5.2 details an example of centralized error handling:

EXCEPTIONS AND ERROR PROCESSING
Centralized Error Handling

```
package ERROR_PACKAGE is
    type ERROR_ENUM is (INVALID_INPUT, NO_DATA, OTHER_ERRORS);
    procedure ERROR (ERROR_NAME : in ERROR_ENUM);
    ...
end ERROR_PACKAGE;

with text_io;
package body ERROR_PACKAGE is
    MESSAGE : string(1..24);

    procedure ERROR (ERROR_NAME : ERROR_ENUM) is
    begin
        ...
        case ERROR_NAME is
            when INVALID_INPUT => MESSAGE := "You goofed! Try again. ";
            when NO_DATA       => MESSAGE := "Enter some data, please.";
            ...
        end case;

        text_io.put (MESSAGE);
    end ERROR;
end ERROR_PACKAGE;

with ERROR_PACKAGE;
procedure DO_STUFF is

    NO_DATA      : boolean;
    INPUT_ERROR  : boolean;

begin
    ...
    if INPUT_ERROR then
        ERROR_PACKAGE.ERROR(ERROR_PACKAGE.INVALID_INPUT);
    elsif NO_DATA then
        ERROR_PACKAGE.ERROR(ERROR_PACKAGE.NO_DATA);
    else
        DO_MORE_STUFF;
    end if;
    ...
end DO_STUFF;
```

Example 5.2

EXCEPTIONS AND ERROR PROCESSING

ERROR HANDLING USING EXCEPTIONS

5.3 ERROR HANDLING USING EXCEPTIONS

The exception handling capabilities of the Ada language simplify both in-line and central error handling. The exception handler part of a block is clearly identifiable and obvious. Each subprogram must declare or have visibility to the exceptions that it is allowed to raise or propagate. This forces a software designer to explicitly define the paths of error handling, rather than rely on estimation to determine error paths. When properly handled, exception propagation allows an error to be vectored to the appropriate level. Exception handling can make error detection and recovery more visible, organized, and more consistent. The concept can be used to more easily design error handling into interactive systems.

5.3.1 In-line Error Handling With Exceptions

Errors can be handled locally quite easily using exceptions. Rather than hiding the error handling code in-line, it can be separated into an exception handler at the end of the code block. Exceptions can be used to correct errors and retry operations or to pass control back up to a higher scope level. Similar errors can simply raise the same exception and propagate to the same handler. This allows uniform, consistent error handling, even if errors are not vectored to a central location. Care must still be taken to ensure uniformity of error messages and diagnostics. However, the exception feature makes this code much more visible and easy to locate, simplifying system changes. Compare the following example to Example 5.1 above:

EXCEPTIONS AND ERROR PROCESSING

In-line Error Handling With Exceptions

```
with TEXT_IO;
procedure DO_STUFF is
  NO_DATA      : boolean;
  INPUT_ERROR  : boolean;
  INVALID_INPUT : exception;
  NO_DATA_ERROR : exception;

begin
  -- ...
  if INPUT_ERROR then
    raise INVALID_INPUT;

  elsif NO_DATA then
    raise NO_DATA_ERROR;

  else DO_MORE_STUFF;
  end if;
  -- ...
exception
  when INVALID_INPUT =>
    TEXT_IO.PUT ("You goofed, try again! ");
    DO_STUFF;
  when NO_DATA_ERROR =>
    TEXT_IO.PUT ("Enter some data, please");
    DO_STUFF;
end;
```

Example 5.3

In the above example, the procedure can generate an error message and retry the operation (by the call to DO_STUFF in the exception handler). Error messages are in the same proximity and are therefore easier to standardize and maintain.

5.3.2 Centralized Error Handling Using Exceptions

Centralized error handling was the error handling method chosen for the AIM design. It was found that exceptions can encourage centralized error handling due to their propagation capability. Even so, the centralized technique basically remains the same: when a subprogram detects an error, an exception is raised. The handler for this exception contains a call to a central error-handling routine. Exceptions can be used throughout the code to generate calls to the error package. However, an exception cannot propagate directly to the error routines unless the error routine is part of the main program (or at least is part of the calling sequence for each routine). Resuming an abandoned procedure is difficult from a

EXCEPTIONS AND ERROR PROCESSING

Centralized Error Handling Using Exceptions

centralized error routine; however, using exceptions, the subprogram can be re-called from the exception handler after the call to the error routine. Following is the same problem as Example 5.2 above, using exceptions:

```
with TEXT_IO;
package ERROR_PACKAGE is

    type ERROR_ENUM is (INVALID_INPUT, NO_DATA, OTHER_ERRORS);

    procedure ERROR (ERROR_NAME : in ERROR_ENUM);
    -- ...
end ERROR_PACKAGE;

package body ERROR_PACKAGE is

    MESSAGE : string(1..24);

    procedure ERROR (ERROR_NAME : in ERROR_ENUM) is
    begin
        ...
        case ERROR_NAME is
            when INVALID_INPUT => MESSAGE := "You goofed! Try again. ";
            when NO_DATA       => MESSAGE := "Enter some data, please.";
            when OTHERS        => MESSAGE := "See operator. ";

        end case;
        -- ...
        TEXT_IO.PUT (Message);

    end ERROR;
end ERROR_PACKAGE;
```


EXCEPTIONS AND ERROR PROCESSING
Centralized Error Handling Using Exceptions

```
with ERROR_PACKAGE;
procedure DO_STUFF is

    NO_DATA      : boolean;
    INPUT_ERROR  : boolean;
    INVALID_DATA : exception;
    NO_DATA_ERROR: exception;

begin
    -- ...
    if INPUT_ERROR then
        raise INVALID_DATA;

    elsif NO_DATA then
        raise NO_DATA_ERROR;

    else
        DO_MORE_STUFF;
    end if;

    exception
        when INVALID_DATA => ERROR_PACKAGE.ERROR
                                (ERROR_PACKAGE.INVALID_INPUT);
        when NO_DATA_ERROR => ERROR_PACKAGE.ERROR (ERROR_PACKAGE.NO_DATA);
end DO_STUFF;
```

Example 5.4

In the example above, all the error handling data structures and types can be encapsulated into the package ERROR_PACKAGE. This scheme is amenable to changes, since all the interacting structures are in the same place. The design of the centralized error handling still looks similar to Example 5.2, however, in that the actual resolution of the error is accomplished by a call to procedure ERROR.

During the AIM design, an interesting problem occurred with Object-Oriented Design and exception handling. Since the methodology keys on grouping data structures with the procedures and functions which act upon them, the procedures all belonged at the same hierarchy level. The resulting design tended to produce a "flat" structured system; most packages and procedures were declared at the same scope level. Therefore, when routines from one package called routines from another package, exception handling became a quandary: where did the handler belong? Exceptions declared in package A could easily be propagated through routines from package B, which meant that package B routines had visibility to package A's exceptions. Ultimately, this potential problem was solved by more clearly defining the exceptions and identifying their causes.

EXCEPTIONS AND ERROR PROCESSING

Handling The Standard Exceptions

5.3.3 Handling The Standard Exceptions

Ada provides several standard exceptions which can serve as run-time checks in an interactive system. These exceptions, such as `CONSTRAINT_ERROR`, are raised by the system, but may be handled in the Ada program. Therefore, instead of the program failing when an invalid data item is entered, the program can trap this error and handle it. If the exception is not handled locally, it will propagate through the calling sequence. The exception must be handled at some point in the calling sequence, or the program will fail. We took advantage of the standard exceptions in the AIM design, treating them like any other exceptions. However, we found that extra care must be taken when using these exceptions, because they were not explicitly declared in our packages. Since these exceptions can be raised from almost anywhere (such as when an array index goes out of range), the handler for a standard exception could be invoked for the wrong reason, creating debugging confusion.

5.4 DESIGNING EXCEPTIONS INTO ADA PROGRAMS

Determining that a procedure needs to be able to raise an exception is not difficult. Designing the exception handling so that correct propagation occurs IS difficult. Currently, there is no formal technique for developing exceptions and associating them with the correct exception handlers. The AIM design demonstrated the confusion that can result from attempting to organize exception handling; to alleviate the chaos, we developed a cross-reference mechanism for exception propagation tracking.

5.4.1 Propagating Exceptions

Exceptions can be propagated through a system both explicitly and implicitly. Explicit propagation occurs when exception handlers are provided throughout the propagation path of the exception and each handler re-raises the exception as part of its processing. Implicit propagation occurs when an exception is raised and there is no exception handler within that local block or frame that explicitly traps the exception. The exception is propagated through its propagation path to the innermost frame which does have an exception handler that explicitly traps the exception.

Although providing explicit propagation may create an excessive amount of code, it can provide a visible indication of the exceptions expected to be raised within or propagated through each block within the system. Note that the `WHEN OTHERS` clause of an exception handler must be classified as providing explicit propagation even though the exception is not uniquely identified in the exception handler. This is because the propagation of the exception is handled explicitly by the program and not implicitly by the runtime package.

The disadvantage of implicit exception propagation is that, without a source code level debugger, it is very difficult to trace the origin of an exception. This makes testing and maintenance very difficult and time-consuming. The advantage of implicit exception propagation is two-fold. The size of the executable image is reduced and the overall speed of execution may be increased since the runtime package handles exception propagation automatically. Of course, this assumes that many of the exception handlers would do nothing more than re-raise exceptions.

5.4.2 Exception Visibility

A prime consideration when designing with exceptions is the sensitive rule of exception definition and scope. Once an exception is raised, control is transferred to the innermost enclosing frame that has an exception handler. If at any level along a propagation path the exception is not declared or the exception name is not visible, the exception name becomes "anonymous" at that point: the program knows an exception has been raised, but does not know the name of the exception. The exception can then be handled only by an OTHERS clause in an exception handler. However, if there is no OTHERS clause in the main level exception handler, the program will fail. Consider the following example:

EXCEPTIONS AND ERROR PROCESSING

Exception Visibility

```
with A, B, ERROR;
procedure MAIN is
begin
  -- ...
  A.PROC_A;
  -- ...
exception
  when ERROR.OOPS => A.START_OVER;
  when others => A.ABORT_PROGRAM;
end MAIN;

package A is
  -- ...

  procedure PROC_A is
  begin -- Procedure A.PROC_A
    -- ...
    B.PROC_B;
    -- ...
  exception
    when others => raise;
  end PROC_A;
end A;

with ERROR;
package B is
  -- ...

  procedure PROC_B is
  begin
    -- ...
    raise ERROR.OOPS;
    -- ...
  end PROC_B;
end B;
```

Example 5.5

Notice the procedure MAIN and package B have made the package ERROR visible. If procedure PROC_B in package B raises the exception ERROR.OPS, the exception will propagate back up to the procedure MAIN. However, since the package ERROR is not visible to package A, the exception ERROR.OPS will become "anonymous" as it propagates through procedure PROC_A in package A. This will force the exception to be handled by the OTHERS clause of procedure MAIN instead of the explicit handler for the exception ERROR.OPS. It is now obvious that extreme care must be taken to determine any and all paths for exception propagation.

5.4.3 The Cross-Reference Matrix

The AIM design concentrated mainly on developing the packages and procedures specified by the Object-Oriented Design. Exceptions were a design consideration at the procedure level from the very beginning; exceptions were dutifully outlined in each subprogram specification and declared in the corresponding packages. However, we found that keeping track of which routine could raise which exception was impossible at the design level, because declaring exceptions at the procedure level requires partial implementation of the routine. Therefore, we had to rely on human memory in many cases to detail the exception in the abstract for the routine. The design code for each subprogram, however, did not reflect the exceptions. If an exception was declared at the package level, it was impossible to tell from the package specification which routines in the package could raise the exception without reading through every subprogram specification in the package.

After some consternation and frustration trying to keep up with exception declaration and handler location, we experimented with an exception "cross-reference" matrix for the AIM. Each exception was associated with the routines that could raise it and the places where it could be handled. For example:

Exception	Raised	Handled
NO_SUCH_IMAGE	IM.GET_NAME	IM.GET_IMAGE
	IM.GET_IMAGE	CI.PERFORM_COMMAND
	VM.ASSOCIATE	CI.PERFORM_COMMAND
NO_SUCH_PROGRAM	PM.GET_PROGRAM	PM.NEXT_PROGRAM
	PM.SUSPEND_APSE_PROGRAM	CI.PERFORM_COMMAND
...		
	KEY	
	IM - IMAGE MANAGER	
	VM - VIEWPORT MANAGER	
	PM - PROGRAM MANAGER	

EXCEPTIONS AND ERROR PROCESSING

The Cross-Reference Matrix

The development of this matrix led to several valuable discoveries about our design:

1. There were several routines which raised the same exception but required different handling.

In several cases, procedures from different packages used the same exception name for different purposes. Especially confusing were the instances of routines needing local handlers with the same name as a "centralized" handler. Propagation sequences were hard to define when the origin of the exception was not clear.

2. There were several exceptions declared in packages but never raised in any subprograms.

The size of the AIM system design was a factor in this oversight. The structure of the design document did not readily lend itself to checking the completion of exception handling. In some cases, the nature of the package warranted the exception declaration, but it was not clear which subprogram would raise or handle it.

3. The propagation sequences of some exceptions were not easily definable.

When one package WITHs several others, it is very difficult to predict ALL of the calling sequences possible for ALL of the routines. However, to attain correct exception propagation, the exceptions must at least be visible at each level. Otherwise, the aforementioned "anonymous" exception problem occurs.

4. Our central error handling scheme required every other package to WITH the error package, creating some potential cyclic dependency problems.

The error package, in some instances, needed to call procedures in other packages to obtain diagnostic information. The error package is WITHed by all the other packages to provide centralized error handling. This created some cyclic dependency problems (e.g. the error package WITHed a package which in turn WITHed the error package, causing the AIM compilation to fail under the NYU/AdaEd compiler. This was a compiler problem, but we could not compile our code without remedying the situation.) Our solution was to precede the package bodies by "WITH ERROR_PACKAGE" and to compile separately the routines of the ERROR_PACKAGE that needed to WITH other packages. Eventually (because of our access to other Ada compilers), only the package bodies WITHed the ERROR package. Additionally, only the ERROR package body WITHed other packages. This technique, although somewhat confusing to novice Ada developers, eliminated the

cyclic dependency problem.

5. Exceptions were not used in a consistent manner across the design.

The cross-reference matrix began as an attempt to map all exceptions to the errors that were found. The AIM design, however, uses exceptions for program control also, not just for error handling. These exceptions were generally handled locally and needed to be separated from the error flag exceptions. We classified the exceptions into two categories: "local" and "global" exceptions. Only the global exceptions were part of the cross reference matrix, since the local exceptions were assumed to be handled locally.

Developing the exception cross-reference matrix was an illustrative exercise. Maintaining the matrix was important as small design changes were made. The interconnections and relationships among the packages often affected exception propagation sequences, and any changes could impact the matrix.

5.5 USING EXCEPTIONS FOR PROGRAM CONTROL

Exceptions and exception handlers can provide a mechanism for controlling program flow. In general, exceptions affect program flow by providing an implicit GOTO to the end of the current block. This action is INESCAPABLE and forms the basis for using exceptions to control program flow. While deciding whether or not to incorporate exceptions as part of the program control methodology for a system, the designer must consider how and when exceptional conditions will be detected. Additionally, the side effects of using exceptions must be considered.

5.5.1 Detecting Expected Exceptional Conditions

In [BOO83], Booch states, "... we should not use exceptions to provide some sort of implicit GOTO facility. Rather, when modeling solutions, we should try to recognize the possible error states of our objects and algorithms and explicitly use exceptions only to plan for their resolution".

Booch limits his use of exceptions only to error handling. However, he neglects one of the other purposes of exception handling: detecting expected exceptional conditions. In [GOO75], John Goodenough maintains that exceptions are needed for three basic purposes:

EXCEPTIONS AND ERROR PROCESSING

Detecting Expected Exceptional Conditions

- a. To signify operation failure,
- b. To classify a valid result of an operation so the result is used appropriately,
- c. To monitor a computation's intermediate results or to request additional information.

Exceptions can be useful not only for error handling and recovery, but also for simple program control in exceptional situations. Instead of setting and checking a variable to exit a loop, the procedure can raise an exception which terminates the loop and allows the programmer to easily do post-processing. Almost all of these exceptions are handled strictly locally. For example, a generic stack package could use the exception `CONSTRAINT_ERROR` to recognize stack overflow and underflow. Referring to the specification for a generic `STACK_PACKAGE` in example 3.2 (where `STACK.TOP` can range from 0 to the maximum stack size), the implementation of the `PUSH` procedure could look like the following:

package body `STACK_PACKAGE` is

. . .

```
-----
-- procedure PUSH  -- This procedure attempts to push another element
--                  -- onto the stack. If the stack is full, a
-- constraint error occurs. The exception handler will then raise the
-- STACK_OVERFLOW exception and pass it to the calling procedure.
-----
```

```
procedure PUSH      ( FRAME : in ELEMENTS;
                     STACK : in out HELP_INFO_STACK ) is
begin
  STACK.TOP := STACK.TOP + 1;  -- constraint_error can be raised here
  STACK.CONTENTS(STACK.TOP) := FRAME;
exception
  when CONSTRAINT_ERROR =>  -- STACK.TOP = SIZE
    raise STACK_OVERFLOW;
  when others =>
    raise;
end PUSH;
```

. . .

end `STACK_PACKAGE`;

Example 5.6

EXCEPTIONS AND ERROR PROCESSING

Detecting Expected Exceptional Conditions

This example shows how exceptions can be used for program control by detecting expected exceptional conditions and acting upon them. Be aware that `CONSTRAINT_ERROR` would be raised after the evaluation of the expression `"STACK.TOP + 1"`, but before `STACK.TOP` is actually updated. Depending upon the designer's point of view, the exception `CONSTRAINT_ERROR` would signify operation failure or classify a valid result so that the result could be used appropriately. Note, however, that it is not always immediately obvious as to which statements in the code actually raise the exception `CONSTRAINT_ERROR`. This implicit GOTO capability should be clearly documented to improve the maintainability of the code.

5.5.2 Side Effects Of Using Exceptions

Effectively incorporating the use of exceptions into the design of a system is clearly a difficult task. Once the design is being implemented, some additional concerns, or side effects, must be addressed. These side effects include:

- * the potential for variables having intermediate values when exceptions are detected,
- * the effect that using exceptions has on the testing phase, and
- * determining how to use standard exceptions when certain runtime checks must be suppressed.

5.5.2.1 Intermediate Values Of Variables

Care must be taken if values of variables are to be used in an exception handler. Chapter 11, section 6, of the Ada language reference manual specifies conditions under which actions may be performed earlier or later than specified by other rules in the language.

As prescribed by the manual, optimized code MAY NOT create extraneous exceptions; however, exceptions that are created may be raised at some point during execution different from that indicated by the given source code. This may cause some variables to have only intermediate values or even to be undefined instead of having some values as implied by the flow of the given source code. In general, variables which are updated within a block of code where an exception is raised are the variables which may have intermediate or undefined values. If such variables are evaluated for use in an exception handler, unexpected results may occur. This flexibility provided to compiler implementors is another reason why code such as the example stack package should be used with care and tested thoroughly before release. Without a clearer definition than that provided by the

EXCEPTIONS AND ERROR PROCESSING

Intermediate Values Of Variables

current version of chapter 11 of the Ada language reference manual, code such as found in the above stack package may even be system (compiler implementation) dependent.

5.5.2.2 Experiences While Testing

During the development of the AIM using the Data General ADE and during rehost onto a DEC VAX 11/785, the source code level debuggers supplied with each system were used extensively. As exceptions are frequently used in the AIM to indicate exceptional conditions, the development team experienced an interesting side effect during the debugging and testing phases of development.

The debug systems create a breakpoint whenever an exception is raised. This includes the standard pre-defined exceptions as well as AIM system defined exceptions. Some heavily used procedures routinely raise an exception as a method for returning information to the calling procedure. At times, development team members became quite agitated at the constant creation of breakpoints and warning messages during testing. Although the debuggers provide a mechanism for overriding the setting of breakpoints when exceptions are raised, in general this capability must be ignored in order to allow proper testing and debugging. Otherwise, it is often difficult, if not impossible, to trace the source of an unexpected exception.

The general consensus of the AIM design and implementation team is that the effects of using exceptions which have a high probability of occurrence must be examined carefully. One might conclude that an exception which is raised relatively frequently is not a true exceptional condition and, therefore, should be handled in a different manner within the design and implementation of the system.

5.5.2.3 Suppressing Runtime Checks For Errors

A problem associated with using standard pre-defined exceptions for program control is that one cannot suppress runtime checks related to those exceptions within the region of code where the exceptions are used. Consider the previous example of a generic stack package. The example package will not work if the runtime checks `INDEX_CHECK` and `RANGE_CHECK` are suppressed.

Now consider an example where, due to execution speed requirements, it is necessary to suppress the runtime checks which can raise the exception `CONSTRAINT_ERROR`. In order to use a package such as the example stack package, that package must be isolated from the other compilation units in such a way as to remove it from the scope of any pragma `SUPPRESS` statements. To implement this example on a VAX using the DEC Ada compiler, it was necessary NOT to instantiate the stack package in the package that contained the pragma `SUPPRESS_ALL`. This

EXCEPTIONS AND ERROR PROCESSING

Suppressing Runtime Checks For Errors

level of indirection was necessary to remove the stack package from the scope of the pragma statement. If the generic stack package had been instantiated in the package containing the pragma `SUPPRESS_ALL`, the stack procedures would have fallen within the scope of the pragma statement. One implication of this solution is that the benefits of using a generic package are reduced when it must be instantiated in a package separate from the one where it is actually used. The separate package adds a logical level of indirection, whether real or imagined.

5.6 GUIDELINES FOR USING EXCEPTIONS

A natural outgrowth of our experiences is the following list of guidelines for using exceptions. While some points are inter-related, and may even overlap in some areas, it is best that they be listed so as to serve as visible reminders to the design team. Note that any questions listed should be answered during the design phase of system development, due to their long range impact on the system.

1. Identify System requirements
 - a. Is overall code size a factor?
 - b. Is execution speed a factor?
 - c. Will the pragma `SUPPRESS` be used?
 - d. Is maintainability the primary concern?
2. Identify exception usage methodology
 - a. Will centralized or local error handling be predominant?
 - b. Will exceptions be used to control program flow?
3. Keep in mind the side effects of using exceptions
 - a. Traceability of the origin of each exception
 - b. Scope of visibility of exception names
 - c. Possible dependency upon compiler implementations for reliable values of some variables

EXCEPTIONS AND ERROR PROCESSING
GUIDELINES FOR USING EXCEPTIONS

d. Impact of frequent exception occurrences during testing

In studying the AIM design, we found exceptions to be a powerful tool for error handling and for program control in exceptional situations. Exceptions enhanced local error handling and made error code more maintainable. We were able to design a centralized error handling package for the AIM which could be invoked from exception handlers throughout the system. Since there are no formal techniques for designing exceptions into Ada programs, the cross-reference matrix was found to be a useful tool. Finally, as with all the other new programming features that Ada provides, exceptions must be used with care to avoid any of their undesirable side effects.

CHAPTER 6

ENVIRONMENT EXPERIENCES

6.1 COMPILER EXPERIENCES

This section summarizes some fundamental Ada compiler design and/or implementation decisions that can directly effect Ada software development. Categorically, these decisions include the compiler's:

- * Storage Allocation Scheme (Data Representation),
- * Underlying Storage Management Scheme,
- * Implementation of Tasking, and
- * Implementation Dependent Features.

6.1.1 Storage Allocation Scheme

The storage allocation scheme instrumented by a code generator for an Ada compiler is quite important as it has a direct correlation to the efficiency of the executable code produced. For example, the DG ADE 2.20 Ada compiler allocates one 32-bit word for every scalar object: BOOLEAN, CHARACTER, INTEGER, etc. When this simplistic storage allocation paradigm is extended from scalar objects to composite data structures (arrays and records), the inefficient storage utilization is greatly magnified. This inefficiency also carries over into slower execution time when these (internally larger) composite data structures have to be frequently copied due to: simple assignments, aggregate assignments, parameter passing, and array slices.

It is important to consider the following data representation issues when evaluating an Ada compiler:

ENVIRONMENT EXPERIENCES

Storage Allocation Scheme

- * Mapping of scalar data types,
- * Mapping of composite data types (records, arrays),
- * Representation of discriminant records, and
- * Implementation of unconstrained array types (does the compiler allocate the maximum space ever needed).

6.1.2 Storage Management Scheme

The Ada language seems to have two different storage management schemes in mind, either:

1. an automatic dynamic storage reclamation mechanism, or
2. a user controlled storage management capability directly supported by the generic Ada procedure, `UNCHECKED_DEALLOCATION`.

Most Ada compiler run-time implementations will support the latter storage management scheme; however, the Ada run-time models for use in embedded systems will more than likely support both of these scenarios.

No matter which scenario is supported by the Ada run-time model, certain issues must be considered:

- * is there any "hidden" dynamic storage allocation (aggregates, slices),
- * what type of stack management scheme is instrumented (primary stack, secondary stack)
- * how is a task's stack-heap allocated, managed, and subsequently deallocated,
- * from where is dynamic storage allocated when a procedure, on the dynamic call chain of an Ada task, creates a local object via the evaluation of an allocator, and
- * what method of storage management is used for handling access type collections.

6.1.3 Implementation Of Tasking

There are numerous implementation issues that must be addressed in the realm of Ada tasking:

- * Storage management scheme for task's stack-heap space - An Ada run-time model must support a storage management scheme powerful enough to support Ada tasking. Specifically, Ada tasks must have their own run-time stack for local variables as well as frames corresponding to subprograms invocation. An Ada run-time model must support the allocation, access management, and deallocation of these task stacks.
- * Handling of Asynchronous Input/Output - An Ada compiler's implementation of asynchronous input/output from within a multi-tasking application can be an important concern. For instance, the DG Ada compiler utilizes operating system (server) tasks to support asynchronous input and the DEC Ada compiler uses Asynchronous Traps (AST's). In this comparison, the DG Ada compiler implementation is restricted to having 32 simultaneous Ada tasks performing input operations whereas the DEC Ada compiler has no such restriction.
- * Accuracy of the real-time clock and the DELAY statement - The granularity of the real-time clock and the open-ended nature of the semantics of Ada's DELAY statement are major implementation concerns of the real-time programming community.
- * Method of implementing rendezvous - The underlying implementation and the relative speed of the Ada rendezvous are also vital real-time programming concerns.

6.1.4 Implementation Dependent Features

Chapter 13 of the Ada LRM [DOD83] enumerates some compiler dependent features that are suggested, but not required, in the implementation of a validated Ada compiler. As an Ada compiler matures and its users' system programming requirements increase, all of these features will probably have to be implemented. Minimally, every Ada compiler should support:

- * unchecked storage deallocation,
- * unchecked type conversion,
- * length clauses, and

ENVIRONMENT EXPERIENCES

Implementation Dependent Features

- * programming interfaces to other non-Ada languages.

6.2 ENVIRONMENT COMPARISON: AOS/VS/ADE VS VMS

A comparison of the two development environments is given here. The purpose is to provide both objective and subjective information concerning the most used tools in a software development environment. The tools examined are:

- * Compiler
- * Linker
- * Debugger
- * Program Librarian
- * Configuration manager
- * Editor
- * Electronic mail system

6.2.1 Compiler

Both systems have an ANSI standard Ada compiler. This means that they both pass the ACVC tests and are considered production quality. In this section the method of compilation, error messages, and resultant files will be examined. Also, the degree to which the compiler is integrated into the environment will be explored.

Finally, a table is given showing the objective capabilities of the two compilers.

6.2.1.1 Method Of Compilation

6.2.1.1.1 AOS/VS - The compilation process involves three steps:

1. Parse the entire Ada source file; if any syntax errors are encountered, the compilation is terminated,
2. assuming no errors from step 1, semantically check each compilation unit. If any semantic errors are detected, compilation terminates for that unit, but continues for the remaining units, and

3. generate machine code, in the form of relocatable binary, for each correct unit, and update the program library accordingly.

The Ada compiler can be invoked from the command line or executed in a batch stream. We limited the execution of the compiler to only run in a batch stream at reduced priority due to the heavy resource demand the compiler makes on the system. Within the batch stream we limited the number of concurrent compiles to 5 (on an MV 10000 with 12MByte of memory). This number was reached by trial and error.

The compiler is invoked via the command

-) ADA filename [filename]*

where "filename" names the source file(s) that you want to compile. If no extension is given, the extension ".ADA" is assumed.

Command switches of the form:

-) ADA [/switch [=option]]*

allow the programmer to enable specific features (as described in the capabilities section below). The switches can be abbreviated to a certain degree.

6.2.1.1.2 VAX/VMS - The Ada compiler can be invoked from the command line or executed in a batch stream. We did not limit the execution to a batch stream since the compiler does not seem to burden the system quite as much as the DG compiler.

The compiler can be invoked in three ways:

- * from the command line to compile (an) Ada program(s),
- * from the Ada Compilation System (ACS) with the COMPILE command, and,
- * from the ACS with the RECOMPILE command.

To compile from the command line the form is:

\$ ADA filename [, filename]*

where "filename" names the source file(s) that you want to compile. If no extension is given, the extension ".ADA" is assumed.

Command switches of the form:

ENVIRONMENT EXPERIENCES

Method Of Compilation

\$ ADA [/switch [=option]]*

allow the programmer to enable and disable specific features (as described in the capabilities section below). The switches can be uniquely abbreviated. These switches only apply to the source files that they appear with.

The ACS COMPILE command forms the closure of the set of units specified; compiles, from current source files, any unit in the closure that was revised since last compiled into the current program library; recompiles any unit in the closure that needs to be made current. This is a form of minimal recompilation and will be discussed more later. It is enough to say that this command invokes the compiler to compile all the source necessary to update the specified unit. The form of the command is:

ACS> COMPILE unitname [, unitname]*

Where the unitname is the Ada program unit (subprogram, package, task). The librarian makes the association between the unitname and the file that contains it.

The ACS RECOMPILE command recompiles (makes current) any obsolete unit that is part of the closure of the set of units specified. The form of the command is:

ACS> RECOMPILE unitname [, unitname]*

Other than this, the RECOMPILE is similar to the COMPILE command.

6.2.1.2 Error Messages

Each system has a general format for messages. A sample program is used to demonstrate the error message formats and place of occurrence.

6.2.1.2.1 AOS/VS - Since the AOS/VS Ada compiler was run in batch the error messages are first queried from the batch stream. The compilation of a sample Ada program could show the following sequence in the batch stream.

```
-) ada temp
  Command line parsed.
  Ada Compiler Rev. 02.20.00.00 6/5/85 at 12:52:11
  Reading from :USER1:ATB:AIM:FINAL_RELEASE:DEMO:TEMP.ADA
  5 syntax errors
  Procedure body TEST_ERRORS has NOT been added to the library.
  2 semantic errors
  Code generation suppressed
  Used 0:00:01 in 0:00:03
```

ENVIRONMENT EXPERIENCES Error Messages

From here, one would type out the list file (or edit it) to determine the actual syntax and semantic errors. There are two types of errors that can occur: syntax errors and semantic errors. Syntax error messages are located near the place where they occur. Semantic error messages are located at the end of the list file.

The general format for either type of message in the list file is as demonstrated in the following example.

```

3 | TYPE mine IS RANGE ;
-----
***      Syntax error with input `;' (Line 3, Column 20).
***      Replacing `;' (Line 3, Column 20) with `+'.

```

6.2.1.2.2 VAX/VMS - The VAX/VMS Ada compiler can be executed from the command line. An example output sequence compiling the same program as on the AOS/VS above is as follows.

\$ ada temp/list temp

```

1      WITH nothing
.....1
%ADAC-E-INSSEMI, (1) Inserted ";" at end of line

3      TYPE mine IS RANGE ;
.....1
%ADAC-E-IGNOREUNEXP, (1) Unexpected ";" ignored
.....2
%ADAC-I-IGNOREDECL, (2) Declaration ignored due to syntactic errors within

6      hoo_doo();
.....1
%ADAC-E-IGNOREPARENS, (1) Empty parentheses ignored

7      END test_errors;
%ADAC-F-TERMSYNTAX, Terminating compilation due to syntax error(s)
%ADAC-F-ENDABORT, Ada compilation aborted
$

```

The line number, the text line, and the error messages are given. Also, delineation is made between (I)nformation, (W)arning, (E)rror, and (F)atal problems.

```

3      TYPE mine IS RANGE ;
.....1.....2
%ADAC-E-IGNOREUNEXP, (2) Unexpected ";" ignored
%ADAC-I-IGNOREDECL, (1) Declaration ignored due to syntactic
                        errors within it

```

ENVIRONMENT EXPERIENCES

Resultant Files

6.2.1.3 Resultant Files

Both systems generate resultant files containing the needed information to support the program library, linker, debugger and other internal functions. Overall, the functioning is very similar.

6.2.1.3.1 AOS/VS - From the compiler the following files are written to the directory containing the target Ada program library. These files are created if the compile completes successfully, only the .LST file is created if the compile does not complete successfully. It is important that the user does not modify the .OB, .TREE, and .STR files during development, as they are used subsequently by the linker.

- * .OB - object code,
- * .SR - assembly code equivalent,
- * .LST - the listing,
- * .TREE - the intermediate representation of the source code in the DIANA form,
- * .STR - the string information for the DIANA tree.

6.2.1.3.2 VAX/VMS - A "program library" is a directory in the VAX/VMS Ada Compilation System. All resultant files except the .LIS file are placed into this program library directory. It is important that the contents of this directory not be changed by the user except through the use of the ACS.

The compiler generates the resultant files and updates the program library if the compile completes with a severity level less than (E)rror. The resultant files generated by the compiler are:

- * .OBJ - the object code,
- * .ACU - a "compunit" file,
- * .ADC - a copied source file,
- * .LIS - the list file.

ENVIRONMENT EXPERIENCES
Integration Into Environment

6.2.1.4 Integration Into Environment

It is important to note the degree to which the compiler is integrated into the overall operating system environment.

6.2.1.4.1 AOS/VS - To use the features of the ADE a user must first "ENTER" the ADE. A command is provided to do this. Once entered, the command prompt is changed to "-)" to indicate to the user that ADE commands are available to be executed. Once in the ADE, ADE commands and AOS/VS commands can be used. However, when not in the ADE only AOS/VS commands can be safely used. The others can be executed (in some instances) but with unpredictable results.

An example session of entering the ADE and compiling a demonstration program is given here.

```
) enter :user1:atb:aim
-) batch ada temp
Creating directory :UDD:BORGER:BATCH to hold batch job output files.
QUEUED, SEQ=18644, QPRI=127
```

```
-) baty
```

```
**** TI Ada Work Center / BATCH OUTPUT FILE ****
```

```
AOS/VS 5.03 / EXEC 5.03          5-JUN-85          12:51:58
QPRI=127          SEQ=18644
INPUT FILE -- :USER1:ATB:AIM:FINAL_RELEASE:DEMO:?008.CLI.001.JOB
LIST FILE  -- :QUEUE:BORGER.LIST.18644
```

```
-----
LAST PREVIOUS LOGON          5-JUN-1985          12:41:20
```

```
AOS/VS CLI  REV 05.01.00.00      5-JUN-85          12:51:59
) SEARCHLIST :USER1:ADE:MACROS,:MACROS,:UTIL,
              :USER1:ATB:AIM:MACROS,:TCS
) DIRECTORY :USER1:ATB:AIM:FINAL_RELEASE:DEMO
) DEFACL SYS MGR,OWARE,BORGER,OWARE,+,WRE
) enter/flat/no_news/proj?=:USER1:ATB:AIM
ADE Revision 2.20.00.00 from directory :USER1:ADE
-) ada temp
  Command line parsed.
  Ada Compiler Rev. 02.20.00.00 6/5/85 at 12:52:11
  Reading from :USER1:ATB:AIM:FINAL_RELEASE:DEMO:TEMP.ADA
  5 syntax errors
  Procedure body TEST_ERRORS has NOT been added to the library.
  2 semantic errors
  Code generation suppressed
  Used 0:00:01 in 0:00:03
```

ENVIRONMENT EXPERIENCES
Integration Into Environment

-) exit

Leaving the Data General/Rolm Ada Development Environment.

AOS/VS CLI TERMINATING 5-JUN-85 12:52:16

PROCESS 7 TERMINATED

ELAPSED TIME 0:00:16

(OTHER JOBS, SAME USERNAME)

USER 'BORGER' LOGGED OFF 5-JUN-85 12:52:16

* LIST FILE EMPTY, WILL NOT BE PRINTED

-) type temp.lst

Ada 2.20.0.0 6/5/85 at 12:52:12

:USER1:ATB:AIM:FINAL_RELEASE:DEMO:TEMP.ADA page 1

```
1 | WITH nothing
2 | PROCEDURE test_errors IS
```

```
*** Syntax error with input `procedure` (Line 2, Column 1).
*** Inserting `;` immediately before `procedure`
                                     (Line 2, Column 1).
```

```
3 | TYPE mine IS RANGE ;
```

```
*** Syntax error with input `;` (Line 3, Column 20).
*** Replacing `;` (Line 3, Column 20) with `+`.
```

```
4 | glitch;
```

```
*** Expression appears where range attribute is expected..
```

```
5 | BEGIN
```

```
6 |   hoo_doo();
```

```
*** Empty parameter list, '()', in call..
```

```
7 | END test_errors;
```

```
*** () not allowed in proc call.
```

```
*** NOTHING denotes no unit in the library.
```

```
*** HOO_DOO is undefined.
```

ENVIRONMENT EXPERIENCES
Integration Into Environment

6.2.1.4.2 VAX/VMS - The VAX/VMS Ada compiler is embedded in the Ada Compilation System (ACS). The program library must be set up before the compiler can be invoked (if it is not setup, an error message is generated and the compilation is aborted). Once the program library is setup, Ada compiles can be started from the VAX/VMS command interpreter through the ADA command.

```
$ acs set librar [.lib]
$ ada temp/list temp
```

```
1      WITH nothing
.....1
%ADAC-E-INSSEMI, (1) Inserted ";" at end of line

3      TYPE mine IS RANGE ;
.....1
%ADAC-E-IGNOREUNEXP, (1) Unexpected ";" ignored
.....2
%ADAC-I-IGNOREDECL, (2) Declaration ignored due to syntactic
                        errors within it

6      hoo_doo();
.....1
%ADAC-E-IGNOREPARENS, (1) Empty parentheses ignored

7      END test_errors;
%ADAC-F-TERMSYNTAX, Terminating compilation due to syntax error(s)
%ADAC-F-ENDABORT, Ada compilation aborted
```

```
$ type temp.lis
```

```
5-Jun-1985 12:5
6:40 VAX Ada V1.0-7
7:43 USER1:[FRENCH]TEMP.ADA;1
```

Page 1 (1)

```
1      WITH nothing
.....1
%ADAC-E-INSSEMI, (1) Inserted ";" at end of line

2      PROCEDURE test_errors IS
3      TYPE mine IS RANGE ;
.....1.....2
%ADAC-E-IGNOREUNEXP, (2) Unexpected ";" ignored
%ADAC-I-IGNOREDECL, (1) Declaration ignored due to syntactic
                        errors within it

4      glitch;
5      BEGIN
6      hoo_doo();
.....1
```

ENVIRONMENT EXPERIENCES

Integration Into Environment

%ADAC-E-IGNOREPARENS, (1) Empty parentheses ignored

7 END test_errors;
%ADAC-F-TERMSYNTAX, Terminating compilation due to syntax error(s)

COMMAND QUALIFIERS

ADA/LIST TEMP

QUALIFIERS USED

/CHECK/COPY_SOURCE/DEBUG=ALL/ERROR_LIMIT=30/LIST/NOMACHINE_CODE
/NODIAGNOSTICS/LIBRARY=ADA\$LIB
/NOTE_SOURCE/OPTIMIZE=TIME/NOSHOW/NOSYNTAX_ONLY
/WARNINGS=(NOCOMPILATION NOTES,STATUS=LIST,SUPPLEMENTAL=ALL
 ,WARNINGS=ALL,WEAK_WARNINGS=ALL)

COMPILER INTERNAL TIMING

Phase	CPU seconds	Elapsed seconds	Page faults	I. O count
Initialization	0.40	1.88	249	34
Parser *	0.39	2.72	244	24
Compiler totals *	0.79	4.61	493	58

COMPILATION STATISTICS

Weak warnings: 0
Warnings: 0
Errors: 3

Peak working set: 1000
Virtual pages used: 4593
Virtual pages free: 32031
CPU Time: 00:00:00.79
Elapsed Time: 00:00:04.61
Compilation Complete

6.2.1.5 Functional Capabilities

The following table presents an objective comparison of the more important capabilities of both compilers.

	VAX/VMS AOS/VS/ADE	
assembly language generation.	x	x
conditional compilation.	x	x
debug information generation.	x	x
enable and disable listing.	x	

ENVIRONMENT EXPERIENCES

Functional Capabilities

errors only listing.	—	x
set default directory for source.	—	x
set listing width and height.	—	x
specify different program library.	x	x
Specify main program.	—	x
disable use of SYSTEM library.	—	x
suppress all run-time checks.	x	x
Multiple files compiled at one time.	x	x
language sensitive editor support.	x	—
specifying an error limit.	x	—
enabling/disabling an error category.	x	—
enable/disable optimization.	x	—
syntax only checking.	x	—

6.2.2 Linker

Both systems contain a linker capable of linking together object modules using information stored in the program library. In this section the method of linking, error messages, and resultant files will be examined. Also, the degree to which the compiler is integrated into the environment will be explored.

Finally, a table is given showing the objective capabilities of the two linkers.

6.2.2.1 Method Of Linking

6.2.2.1.1 AOS/VS - The Ada linker is a special linker provided in the ADE to link together Ada object code into an executable image. The executable image can then be run either in or out of the ADE.

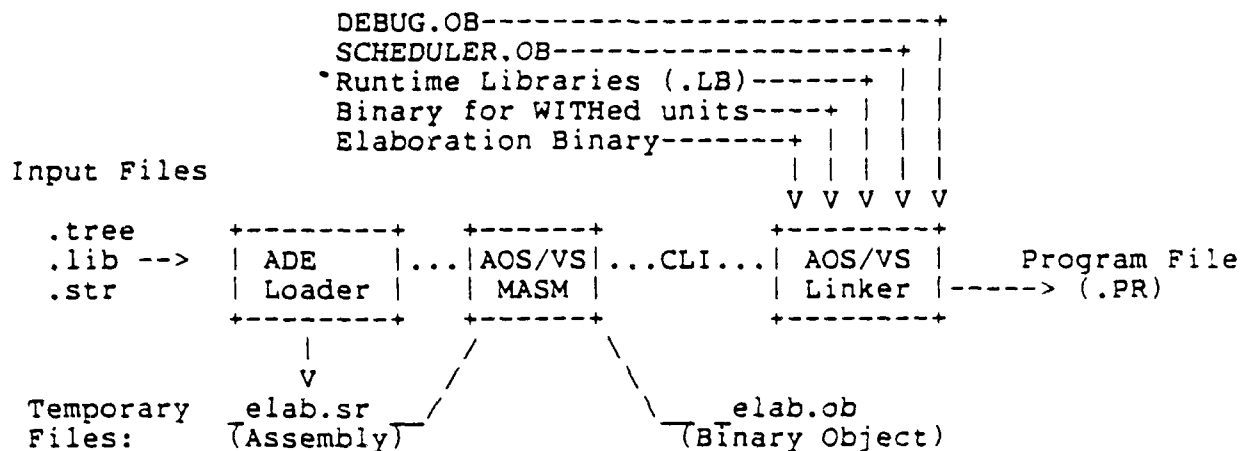
The ADE Linker produces a program file via these steps:

1. find all libraries specified when the main program was compiled,
2. check the program for Ada completeness,
3. generate an object module that contains code to elaborate the program,
4. assemble the elaboration object module, and
5. link all modules together (using the standard AOS/VS linker).

As is obvious from above, the ADE linker uses not only the AOS/VS linker, but also the Macroassembler (MASM) to produce the program file. The following is a flow diagram for the ADE linker:

ENVIRONMENT EXPERIENCES

Method Of Linking



... -- Represents control flow

The command line to invoke the ADE Ada linker is as follows:

-) ADALINK [/switch [=option]] filename

Where the filename is the name of the source file. If the ".ADA" extension is omitted, then it will be assumed.

The linker must be invoked from the batch stream. Here is an example linking a previously compiled program.

-) batch adalink/debug test1
-) baty

**** TI Ada Work Center / BATCH OUTPUT FILE ****

```

AOS/VS  5.03 / EXEC  5.03      11-JUN-85      8:31:02
QPRI=127      SEQ=18737
INPUT FILE -- :USER1:ATB:AIM:2007.CLI.001.JOB (WILL BE DELETED AFTER
PROCESSING)LIST FILE -- :QUEUE:BORGER.LIST.18737
  
```

```

-----
LAST PREVIOUS LOGON      11-JUN-1985      8:28:12
  
```

```

AOS/VS CLI  REV 05.01.00.00      11-JUN-85      8:31:04
) SEARCHLIST :USER1:ADE:MACROS,:USER1:BORGER:MACROS,:MACROS,:UTIL,
:SCRED,:USER1:ATB:AIM:MACROS,:TCS
) DIRECTORY :USER1:ATB:AIM
) DEFACL SYS_MGR,OWARE,BORGER,OWARE,+,WRE
)
  
```

ENVIRONMENT EXPERIENCES
Method Of Linking

```
) enter/flat/no_news/proj?=:USER1:ATB:AIM
ADE Revision 2.20.00.00 from directory :USER1:ADE
-) adalink/debug test1
  Command line parsed.
  Ada Loader Rev. 02.20.00.00 6/11/85 at 8:33:42
  Creating PR file :USER1:ATB:AIM:TEST1
```

```
-) exit
```

Leaving the Data General/Rolm Ada Development Environment.

```
AOS/VS CLI    TERMINATING          11-JUN-85          8:34:38
```

```
PROCESS 8 TERMINATED
ELAPSED TIME  0:03:34
(OTHER JOBS, SAME USERNAME)
USER 'BORGER' LOGGED OFF          11-JUN-85          8:34:38
```

```
****
* LIST FILE EMPTY, WILL NOT BE PRINTED
****
```

6.2.2.1.2 VAX/VMS - The ACS Ada linker is a special linker provided to link Ada object files with other Ada object files, and/or foreign object files. Once linked, the resulting executable image is executed from the VAX/VMS command language interpreter. It does not need to be run from within the ACS (and, in fact, cannot).

The ACS Linker produces an executable image via these steps:

1. issues a CHECK command to form the closure of the main program and verify that all units in the closure are present and current. If ACS detect an error, the operation is terminated before the linker is invoked.
2. creates an object file (to be linked with the program) that elaborates the library units in the proper order at run time,
3. creates a command file for the VAX/VMS linker,
4. invokes the VAX/VMS linker.

The command line to invoke the ACS Ada linker is as follows:

```
S ACS LINK [/command switches] [/parameter switches]
           main-program-name [ filename [,filename]* ]
or
```

ENVIRONMENT EXPERIENCES

Method Of Linking

```
$ ACS LINK/NOMAIN unit-name [, unit-name ] [ filename [,filename]* ]
```

"command_switches" apply to the command (and, therefore, across all of the filenames), "parameter_switches" apply to the specific parameters on which they are located.

The first form is to link a program in which the main program is an Ada unit. The second form is to link Ada units to a non-Ada main program that is specified in one of the filename parameters.

An example link session is as follows:

```
$ acs link test1
%ACS-I-CL LINKING, Invoking the VAX/VMS Linker
$SET DEFAULT USER1:[FRENCH]
$LINK := ""
$LINK-
/NOMAP-
/EXE=[ ]TEST1-
  SYSSINPUT:/OPTIONS
  USER1:[FRENCH]TEST1.OBJ;1
  SYSSCOMMON:[SYSLIB.ADALIB]IO_EXCEPTIONS.OBJ;1
  SYSSCOMMON:[SYSLIB.ADALIB]TEXT_IO.OBJ;1
  SYSSCOMMON:[SYSLIB.ADALIB]TEXT_IO.OBJ;1
  USER1:[FRENCH.LIB]PACK1.OBJ;4
  USER1:[FRENCH.LIB]PACK1.OBJ;4
  USER1:[FRENCH.LIB]TEST1.OBJ;4
$DELETE USER1:[FRENCH]TEST1.OBJ;1
$DELETE USER1:[FRENCH]TEST1.COM;1
$
```

The only true command was "\$ acs link test1". The rest was generated by the link operation. This information is normally not seen by the user.

6.2.2.2 Error Messages

From both systems error message can come from the link operation prior to submitting to the host OS linker, and from the host OS linker itself.

6.2.2.2.1 AOS/VS - Error messages in the ADE link operation are given in the batch stream as they occur. An example is shown.

```
-) adalink/debug test1
  Command line parsed.
  Ada Loader Rev. 02.20.00.00 6/11/85 at 8:37:25
*** PACK1 was compiled after body of TEST1
  Unit is not complete
```

-) exit

6.2.2.2.2 VAX/VMS - Errors in the VAX/VMS link operation conform to the standard error message format that all tools on the VAX/VMS system use. An example is:

```
%ACS-E-CL_MAIUNIKIN, Main unit PACK1 is a package body,  
                        not a subprogram body
```

This message comes from a link operation performed prior to submitting to the VAX/VMS standard linker. The VAX/VMS standard linker has the same error message format.

6.2.2.3 Resultant Files

Both systems generate executable files when the link operation successfully completes. Both also generate a map file.

6.2.2.3.1 AOS/VS - The following files are generated from the link operation:

- * .MAP - the link map file (from the AOS/VS linker)
- * .PR - the executable file
- * .ST - debugger symbol table file
- * .LOG - link messages

6.2.2.3.2 VAX/VMS - The following files are generated from the link operation:

- * .MAP - the link map file (from the VAX/VMS linker)
- * .EXE - the executable file
- * .COM - the command file that invokes the VMS linker
- * .LIS - the link messages (that are normally sent to the terminal)

6.2.2.4 Integration Into Environment

Both systems are integrated into the standard host operating system environment as much as possible. Both use the standard linker after preprocessing the library to generate elaboration code. The ADE linker generates assembly language then assembles it, and links all

ENVIRONMENT EXPERIENCES

Integration Into Environment

the object modules together. The VAX/VMS Ada linker generates object to support elaboration then links it with the other object modules. Effectively, both perform in the same manner. The VAX/VMS linker generates better, more consistent messages.

6.2.2.5 Functional Capabilities

The following table presents an objective comparison of the more important capabilities found in the two systems Ada linkers.

	VAX/VMS AOS/VS/ADE	
non-Ada link capability	x	
deferred (after a specific time)	x	
enable/disable link map generation	x	x
specify full/brief link map	x	
generate a link command file	x	
enable/disable symbol cross-reference	x	
generate debug information	x	x
enable/disable executable file creation	x	
specify batch/nobatch operation	x	
specify map filename	x	
specify object filename	x	
specify diagnostic output file	x	
enable/disable system library search	x	x
enable/disable traceback info	x	
library search capabilities	x	x
extended options capabilities	x	
sharable image support	x	
specify maximum memory		x
force load		x
enable/disable library trace		x
specify main program		x
non-Ada main program	x	

6.2.3 Ada Source Code Debugger

As the AIM (DG) implementation/integration proceeded, it became clear that, besides the compiler and linker, the DG Ada source code debugger was the single most important utility in the ADE toolset. Furthermore, when the completed AIM program was transported from the DG ADE to the VAX/VMS Ada environment, the DEC Ada source code debugger was used extensively and proved to be an invaluable tool in this rehost effort. It is apparent from our AIM implementation, integration, and rehost experiences that an Ada source code debugger is not only an invaluable APSE tool, but also mandatory for the development of complex, concurrent Ada software systems.

It is the purpose of this section to compare the Ada source code debuggers of the DG ADE and DEC Ada environment at two levels:

- * Preparatory Requirements
- * Functional Capabilities

6.2.3.1 Compiler/Linker Requirements

The preparatory requirements are similar for both the DG and VAX Ada Source Code Debuggers: all compilation units to be debugged must be compiled with the /DEBUG option, and subsequently, the main program must be linked with the /DEBUG option. The DEC Ada Environment supports a /NODEBUG option on its RUN command which allows the user to activate a program image without the debugger even if it was linked with the /DEBUG option, whereas the ADE's XECUTE command does not.

6.2.3.2 Functional Capabilities

The following functional capabilities checklist represents the majority of Ada Source Code Debugger features that are necessary to efficiently debug Ada programs.

ENVIRONMENT EXPERIENCES
Functional Capabilities

	VAX/VMS AOS/VS/ADE	
Breakpoints (set/reset) on		
statements	x	x
program units		
subprograms.	x	x
packages	x	x
tasks.	x	x
generic units.	x	x
exceptions	x	x
Tracepoints (set/reset) on		
statements	x	
program units		
subprograms.	x	x
packages	x	
tasks.	x	
generic units.	x	
exceptions	x	x
rendezvous	x	x
Watchpoints for variables.	x	
Display		
program source	x	x
history.		x
stack.	x	x
tasks.	x	x
breaks	x	x
tracepoints.	x	x
Evaluate Objects	x	x
Step		
single	x	x
by discrete amounts.	x	x
into subprograms	x	x
over subprograms	x	x
to next rendezvous		x
to end of program unit		x
Miscellaneous		
symbol abbreviation.		x
set context for program control.	x	x
input debugger command files	x	x
modify variables' value.	x	x
console interrupt.	x	x
full screen mode	x	
keypad mode for entering commands.	x	

6.2.4 Program Librarian And Library Structure

As a consequence of the Ada Language Rule: "Compilers are required to enforce the language rules in the same manner for a program consisting of several compilation units (and subunits) as for a program submitted as a single compilation." [DOD83], all Ada Programming Support Environments must maintain an Ada program library. The intent of this section is to provide a comparison between the DG ADE and DEC Ada Compilation System (ACS) program libraries and to evaluate the functional capabilities of their respective program librarians.

6.2.4.1 ADE Program Library

The ADE program library contains information to help the ADE linker find the required object code files and to provide the ADE Ada source code debugger with program unit information. Upon the successful compilation of an Ada compilation unit, the following file types are written to the directory containing the target Ada program library (which is by default the current working directory):

Filename Extension	Explanation
.OB	Relocatable binary (object code) for the compilation unit.
.SR	Assembly language equivalent of the compiled library unit.
.LST	Compilation listing file.
.TREE	Diana tree representation of compilation unit's source code.
.STR	String information relative to the Diana tree.

The program library file (.LIB) in conjunction with the ADE library manager utility provide the following information about the Ada compilation units in the program library:

- * date/time stamp for syntax checking, semantic analysis, and code generation,
- * unit completeness information,
- * names of the source and object code files corresponding to a compilation unit,
- * the type of program unit or subunit (i.e.--subprogram, task, package, generic, subprogram body, package body),
- * the names of other library units WITHing this unit, and

ENVIRONMENT EXPERIENCES

ADE Program Library

- * the names of other library units WITHed by this unit.

One last feature of the ADE program library is the program library searchlist. By using the LIBSEARCHLIST command, one may extend the list of program libraries that are searched (which always starts with the current working directory and ends with the ADE directory containing the SYSTEM package specification) in order to resolve any WITH dependencies that are not satisfied within the context of the current program library. This tends to be a very powerful feature as it promotes a tree structured project directory with program libraries located at the leaves of the tree.

6.2.4.2 ACS Program Library

As with the ADE, all Ada compilations within the ACS are performed in the context of a program library. An ACS program library is a dedicated VAX/VMS directory which consists of the following files on a per compilation unit basis:

Filename Extension	Explanation
.ALB	Library index file indentifying all files in the library.
.OBJ	Relocatable binary (object code) for the compilation unit.
.ACU	Internal representation of compilation unit.
.ADC	Identical copy of compilation unit's source code.

The VAX/VMS Ada program librarian supports most of the capabilities previously mentioned in the ADE program library section. One major difference between the two program libraries is the mechanism used for accessing library units which reside in outside the current library. The ADE uses a library searchlist technique, whereas the ACS requires that the library unit in question be either COPIED or ENTERED into the current library. The major drawback with this technique is that when an accessed library unit is recompiled it must be re-copied or re-entered into the current program library.

6.2.4.3 Functional Capabilities

The following functional capabilities checklist represents the majority of Ada program librarian features that are necessary to efficiently build Ada systems.

ENVIRONMENT EXPERIENCES
Functional Capabilities

	VAX/VMS AOS/VS/ADE	
Listing Information		
directory of unit names.	x	x
associated file names for unit	x	x
units WITHing specified unit		x
units WITHed by specified unit		x
time-stamp information	x	x
kind of compilation unit	x	x
Completeness and Currency check.	x	x
Automatic recompilation.	x	
Spawn CLI subprocess	x	x
Remove compilation unit.	x	x
Library Access Control		
Read Only.	x	x
Exclusive.	x	x

ENVIRONMENT EXPERIENCES

Configuration Management And File Structure

6.2.5 Configuration Management And File Structure

The intent of this section is to provide a comparison between the DG and DEC configuration management tools and to evaluate their functional capabilities.

6.2.5.1 ADE Configuration Management

The Text Control System (TCS) provides fundamental configuration management support within the ADE. It provides version and variation control for Ada source code and documentation files. The basic TCS commands are listed below with a cross reference back to the corresponding operation of DEC's Code Management System (VAX/CMS):

ACCESS	Creates a copy of a controlled text file version in the current working directory (CMS FETCH).
CHECKIN	Checks in a previously checked out version of a controlled text file (CMS REPLACE).
CHECKOUT	Creates a copy of the latest version of a controlled text file in the current working directory; this file is then marked to prevent other users from checking it out of the control library (CMS RESERVE).
NEWBASIS	Creates a NEW, separately controlled, variation of a controlled text file. A "basis" name is used to differentiate one variation of a file from another. (CMS REPLACE/VARIANT=)
NEWTCS	Places a new file under text control (CMS CREATE).
REVMARK	Marks a specific version of a controlled text file as belonging to a particular "revision" level (CMS INSERT).
TCSPRINT	Output information about controlled text files (CMS SHOW).

6.2.5.2 VAX/VMS Configuration Management

VAX/CMS is a library system which provides fundamental configuration management in support of software development and documentation. The basic CMS commands are listed below with a cross reference back to ADE TCS command where applicable:

ENVIRONMENT EXPERIENCES
VAX/VMS Configuration Management

ANNOTATE	Creates an annotated element listing in current working directory.
COPY ELEMENT	Copies an existing element to form a new element.
CREATE CLASS	Establish a new CMS class.
CREATE ELEMENT	Creates first generation of a file from the current working directory (TCS NEWTCS).
DELETE CLASS	Delete an established class.
DELETE ELEMENT	Delete all generations of an element in the CMS library.
DIFFERENCES	Textually compare two elements from the CMS library.
FETCH	Get a local copy of the specified element (TCS ACCESS).
INITIALIZE	Convert a normal VAX/VMS directory into a CMS library.
INSERT	Place an element into the specified class (TCS REVMARK).
REMOVE	Remove an element from the specified class.
REPLACE	Return an updated copy of a previously RESERVED element (TCS CHECKIN).
RESERVE	Reserve a copy of an element and place it into the current working directory (TCS CHECKOUT).
SET LIBRARY	Identify an existing CMS library as the current CMS library.
SHOW CLASS	Display all established CMS classes.
SHOW ELEMENT	Display a listing of all elements and their corresponding files (TCS TCSPRINT).
SHOW HISTORY	Show an entire transaction history for the CMS library.

ENVIRONMENT EXPERIENCES
VAX/VMS Configuration Management

SHOW LIBRARY Show the currently set CMS library.

SHOW RESERVATIONS Display a listing of all current reservations outstanding for the CMS library.

UNRESERVE Cancels an existing CMS reservation.

VERIFY Perform a series of consistency checks on the current CMS library.

6.2.5.3 Functional Capabilities

The following functional capabilities checklist represents the majority of Configuration Management features that are necessary to maintain the development of source code and documentation.

	VAX/VMS (CMS) AOS/VS (TCS)	
Configuration Library		
create	x	
delete	x	
verify	x	
Library Elements	x	x
create	x	x
delete	x	x
fetch.	x	x
reserve.	x	x
unreserve.	x	
replace.	x	x
differences.	x	
Element Classes		
create	x	x
delete	x	
insert element	x	x
remove element	x	
Listings		
elements	x	x
reservation.	x	x
history.	x	
annotation	x	

6.2.6 Text Editor

The intent of this section is to provide a comparison between the DG and DEC text editors and to evaluate their functional capabilities.

6.2.6.1 ADE Text Editor

The DG supplied text editor (SED) is terminal dependent as it emits DG escape sequences to perform its functioning; therefore, it is not usable on the TV970 terminals connected to our LAN. For this reason, a commercially available (terminal independent) full screen editor capable of running under the AOS/VS operating system had to be purchased. The only editor found that satisfied these requirements was called SCRED. SCRED is full screen, terminal independent text editor developed by Rational Data Systems to run under the AOS/VS operating system. Although SCRED's capabilities in no way rival those of an editor like EMACS, it does support some nice features:

- * Terminal independence via a terminal capabilities file
- * Personalized key bindings via the terminal capabilities file
- * Regular expression search and replace functions
- * Built-in HELP function
- * User defined command macros built upon the predefined command set

6.2.6.2 VAX/VMS Text Editor

The text editor used in our VAX/VMS environment is the Unipress Inc. version of Gosling's EMACS. To date, the EMACS editor is the most complex and powerful terminal independent text editor available for VAX/VMS. Some of its features include:

- * Terminal independence via a terminal capabilities file
- * Split screen capability
 - Edit several files simultaneously
 - Different portions of same file may be edited concurrently

ENVIRONMENT EXPERIENCES
VAX/VMS Text Editor

- Any user or VMS system program may be executed from within a special Shell window
- * Help facility by command name and subject matter
- * Great extensibility
 - Any key or key sequence may be re-defined by user
 - User-definable macros (keyboard or by name)
 - MLISP programming language built-in
- * Regular expression search and replace functions
- * Ada Programming mode and Ada-LRM automated access
- * Backups previous version of files and performs periodic checkpointing

6.2.6.3 Functional Capabilities

The following functional capabilities checklist represents the majority of text editor features that are necessary to efficiently develop Ada code.

ENVIRONMENT EXPERIENCES
Functional Capabilities

	VAX/VMS (EMACS)	AOS/VS (SCRED)
Cursor Movement		
Left, Right, Up, Down.	x	x
Top, Bottom.	x	x
Beginning/End of line.	x	x
Next/Previous Word	x	x
Search/Replace		
Search Forward	x	x
Search Reverse	x	x
Regular Expression Search.	x	x
Regular Expression Replace	x	x
Multiple Replace	x	x
Buffers		
Copy text to	x	x
Copy text from	x	x
Split Screen	x	
Edit multiple files.	x	
Regions		
Set mark	x	x
Kill region.	x	x
Copy region.	x	x
Move region.	x	x
File Manipulation		
Copy from file	x	x
Append to file	x	x
Macros		
Keyboard macros.	x	x
Macro language	x	
Ada Mode	x	
Ada LRM automated access	x	
Miscellaneous		
Terminal independent	x	x
On-line help facility.	x	x
Minimal redisplay algorithm.	x	
Keypad, function key re-definition	x	x
Undo Capability.		x
Spawn CLI.	x	x
Command iteration.	x	x
Command type-ahead	x	

ENVIRONMENT EXPERIENCES

Electronic Mailer

6.2.7 Electronic Mailer

The intent of this section is to provide a comparison between the DG and DEC electronic mail utilities and to evaluate their functional capabilities.

6.2.7.1 ADE Electronic Mailer

One of the most often heard complaint about the ADE was that it did not include an electronic mail utility per se. The only mailing capability available to our implementation team was that supported by a simple AOS/VS CLI command macro. Essentially this mail macro, when invoked, creates (if necessary) a text file corresponding to the sender-receiver pair and appends the new message onto that text file. Needless to say, this was unacceptable and was rarely, if ever, used by our team. Therefore, we relied solely on the VAX/VMS electronic mailer.

6.2.7.2 VAX/VMS Electronic Mailer

VAX/VMS supports personal electronic mail between system users via its MAIL utility. The AIM implementation team used this mailer for general purpose communications even though the development was being done the Data General system. This utility is a comprehensive electronic mailer supporting numerous functions:

- * send/receive messages
- * mail folders
- * reply to/forward messages
- * archive/print/delete messages
- * edit messages to be sent
- * send messages to a list of users
- * send messages across DECnet
- * on-line help facility

6.2.7.3 Functional Capabilities

The following functional capabilities checklist represents the majority of Electronic Mailer features that are necessary to support the team communication during software development.

ENVIRONMENT EXPERIENCES Functional Capabilities

	VAX/VMS (MAIL)	AOS/VS (MAIL.CLI)
Message related functions		
Send	x	x
Receive.	x	x
Immediate forwarding	x	
Immediate reply.	x	
Archive.	x	x
Print.	x	x
Search for string.	x	x
Edit message to be sent.	x	x
Read next message.	x	
Read previous message.	x	
Read first message	x	
Read last message.	x	
Position to start of current message .	x	
Miscellaneous		
Keypad support	x	
On-line help facility.	x	x
Send to distribution lists	x	x
Send across DECnet	x	
Mail folders	x	

6.2.8 Conclusions

There are various conclusions that can be drawn from our experiences in using both the DG ADE and DEC ACS development environments:

- * The two environments contained very similar tools.
- * The DEC Ada Compilation System is more integrated into the VAX/VMS host operating system.
- * The DEC ACS tools generate more consistent and informative messages.
- * The power of the editor and debugger directly affects programmer/designer productivity.
- * The DG program library mechanism, specifically the LIBSEARCHLIST concept, better supports a tree-structured approach to managing project directories.

ENVIRONMENT EXPERIENCES

Conclusions

- * It is important to have a usable electronic mail system to promote communications between programmers/designers/managers.

CHAPTER 7

LIFECYCLE ANALYSIS

7.1 INTRODUCTION

The AIM project started in October 1982 with the purpose of exploring and determining interface issues and problems with Ada Program Support Environments (APSEs). During this time, a detailed record of effort was kept. The intent of this chapter is to present an analysis of the AIM lifecycle as well as map that effort to software lifecycle and software cost models.

This chapter is organized to give an overview of the tasks, deliverables, testing methodology, and Lines-of-Code figures followed by comparisons of the AIM project to various models. A conclusions section is included to give the reader the perspective of the AIM team. Hopefully, enough data is included to allow the reader to draw his or her own conclusions.

7.2 PROJECT OVERVIEW

The AIM project was atypical in that the primary intent was to investigate APSE interfaces with the resultant tool a by-product. As a result, numerous reports were required and participation in several working groups evolved (Common APSE Interface Set (CAIS) participation by Tim Harrison and Guidelines and Conventions (GAC) participation by Stewart French). Also, the code was written in Ada for which there were few compilers and non-existent Ada project history on which to base estimates or make projections. Foresight by the project manager (John Foreman) resulted in a table of efforts to which time was charged for use in tracking the various phases of the project. These are shown in Table 7-1. This tracking scheme was begun three months into the project and as a result, some time was charged to Requirements Definition that rightfully should have been charged to System Specification and Preliminary Design.

LIFECYCLE ANALYSIS
PROJECT OVERVIEW

Table 7-1 Efforts Tracked for the AIM

Requirements Definition
Financial Accounting
System Specification
Preliminary Design
User Manual
Detailed Design
Hardware/Software Problems
Implementation
New Hire (Training)
Integration
Rehost
Quality Assurance
Testing
CAIS
GAC
Data Management
Interface Reports
Software Tools
Configuration Management
Program Management

One shortcoming of the division of efforts occurred in testing. A separate category for each testing phase (unit, integration, acceptance) would have facilitated the model comparisons. Testing included writing the Acceptance Test Plan, Acceptance Test Procedures, Computer Program Test Specification, System/Integration Test Plan, and System/Integration Test Procedures. These documents were written during the design phase of the project and charged against "Testing" shown in Table 7-1. Time was also charged against the Testing effort when integration testing began. Thus the table entry for Testing was used twice with the time gap between the corresponding groups used to distinguish each effort. Unfortunately, unit testing was not tracked.

Figure 7-1 shows the time span of each effort. Also included are markers for the Preliminary Design Review (PDR) and the Critical Design Review (CDR). The gap occurring in the summer of 1984 was caused by the expiration of the original AIM contract before an extension could be finalized and because, at that time, a satisfactory Ada compiler had not yet been obtained. The AIM restarted in September of 1984.

LIFECYCLE ANALYSIS PROJECT OVERVIEW

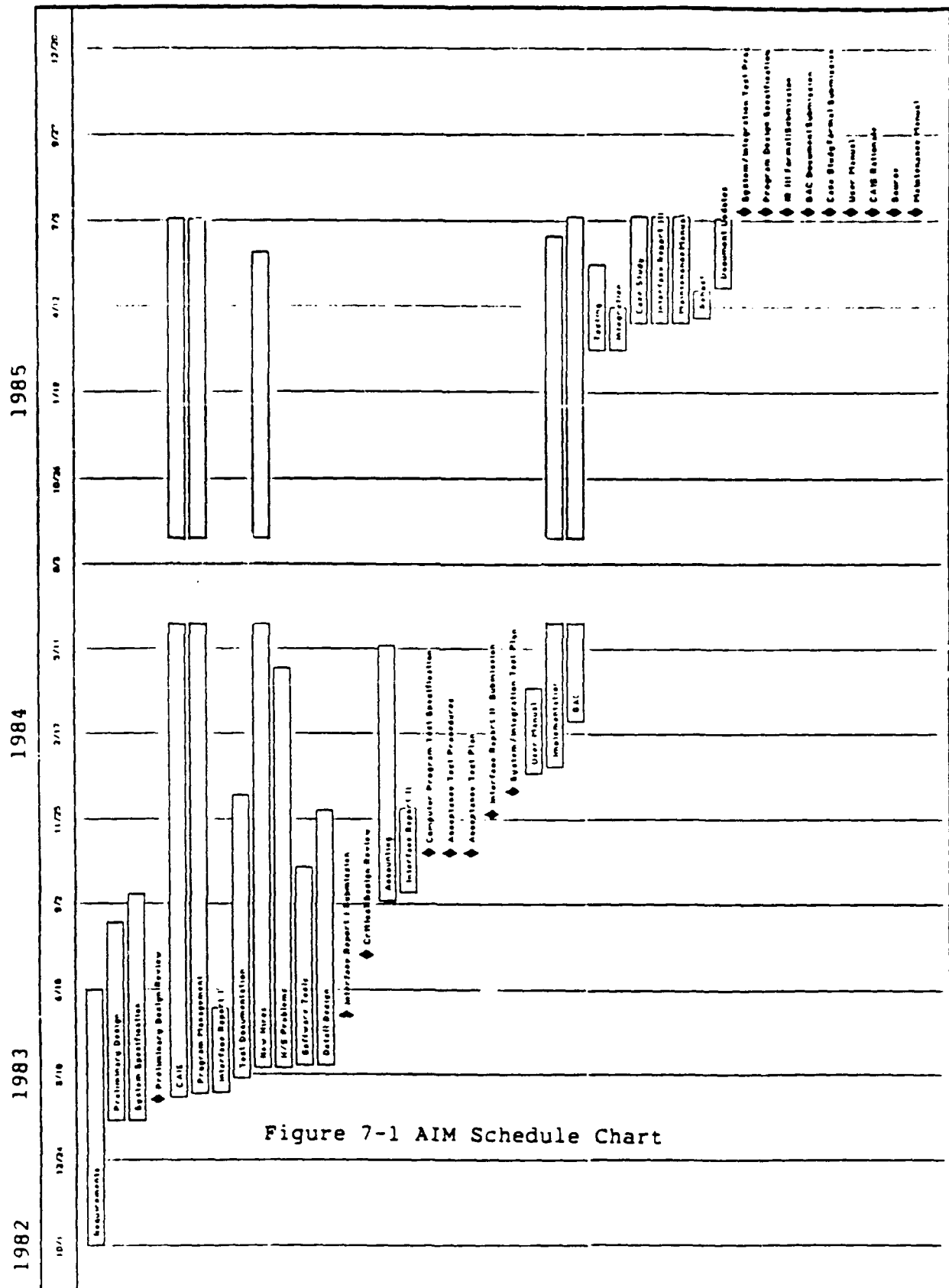


Figure 7-1 AIM Schedule Chart

LIFECYCLE ANALYSIS Systems

7.2.1 Systems

Initial development of the AIM documents was done on a TI990 minicomputer which was used primarily for documentation and communication. However, since there were not adequate communication facilities available on this a mail utility was developed to supplement communication between team members. An existing terminal emulator was then modified to allow access to the ARPANet. The ARPANet was used as the primary means of communication with the Naval Ocean Systems Center (NOSC), the project contractor. The documentation formatter was also improved to increase the number of paragraph levels allowed in a document. The time spent on these tasks was charged to the effort titled "Software Tools" in Table 7-1.

In April of 1983 the project began a migration to a VAX 11/780 which has proven significantly more reliable than the TI990 computer system. The effort titled "Hardware/Software Problems" (see Table 7-1) was used to track system crashes, reboots, access denial due to load limitations, and maintenance. By the end of 1983 all work was being performed on the VAX (All problems did not vanish with the transition but they were greatly reduced). The first task undertaken on the VAX was to write a terminal emulator similar to the existing TI990 emulator. All existing AIM documentation was then ported to the VAX from the TI990. To facilitate the transition from the TI990 text formatter to the VAX-11 Digital Standard Runoff, a quick and dirty routine was written that converted roughly 90% of the TI990 text formatter commands to Runoff commands. The remaining 10% were converted with an editor.

As mentioned earlier, a satisfactory compiler for the VAX had not yet been obtained in the summer of 1984 when implementation was to begin. The solution was to lease a Data General (DG) MV/10000 in September of 1984. The DG system was leased to take advantage of the Ada Development Environment (ADE). The finalization of the AIM contract extension preceded the arrival of the DG system by about a month. By the end of implementation and testing of the AIM on the DG system, a validated Ada compiler was available on the VAX. A rehosting effort (a no cost contract extension to include this effort) was then undertaken to move the AIM from the DG to the VAX. This effort was tracked under the title of "Rehost" in Table 7-1.

7.2.2 Personnel

The three original team members have remained involved (in one capacity or another) throughout the 33 months of the project. One of the original members became involved in the CAIS work full time under the umbrella of the AIM contract. One member became primarily the program manager for the AIM project as well as other projects. The remaining member of the original team was the project technical

LIFECYCLE ANALYSIS Personnel

leader as well as a member of the GAC working group. Other personnel have been added as needed. Each new member went through a learning curve. This effort is reflected under the title "New Hire" in Table 7-1. The personnel remained very stable throughout the project with participation varying from full time to minimal depending on the current project phase. Besides the three original members, two others who joined the project in the first six months remained until the end. Of all the personnel involved with the AIM only one person could be considered as having "turned over". Several others have contributed to specific phases of the project (e.g., one wrote the Configuration Management Plan, one was involved in documentation updates). One other person joined the project at the beginning of implementation and remained until the completion of the contract.

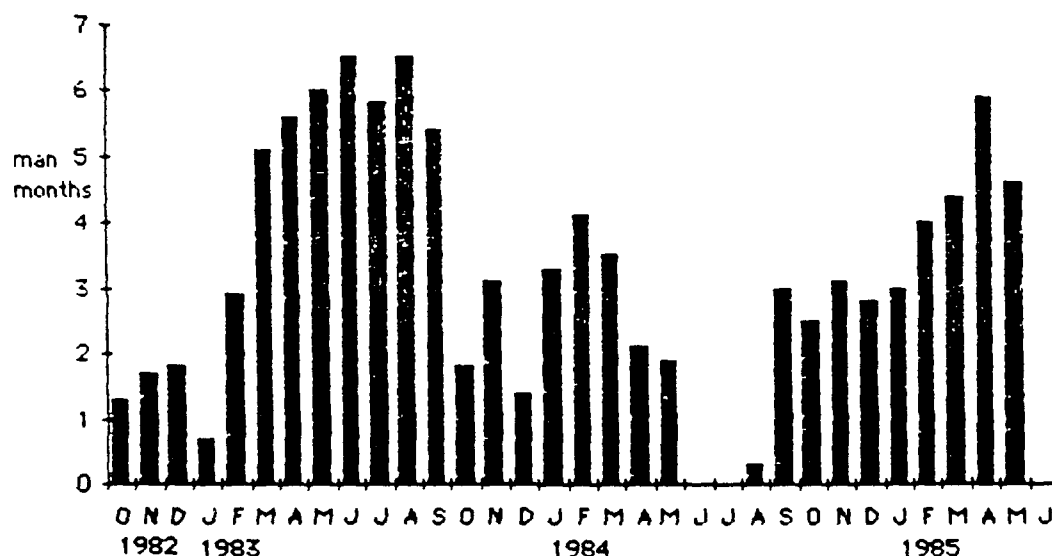


Figure 7-2 AIM Manloading

LIFECYCLE ANALYSIS

Personnel

Figure 7-2 shows the manloading for the AIM project. The numbers shown are total man-months per month. Again, note the gap for the summer of 1984. Reading from left to right on Figure 7-2 you will note an initial small peak in December of 1982 which corresponds to the Requirements Definition effort. This is followed by the largest peak representing the Design effort (this includes writing the test documentation). The peak in November of 1983 was caused mainly by the start of a second Interface Report as the test documents were being finished. The peak shown for February 1984 maps to the User's Manual effort. Starting in September of 1984 and continuing to January of 1985, the bars are fairly even. This reflects the Implementation effort. The increase beginning in February 1985 reflects the addition of one person full time doing documentation updates and the part time effort of a VAX Analyst during the Rehost effort.

7.3 AIM PROJECT EFFORTS

The AIM project included several atypical tasks besides those typically found in the software development process. Table 7-2 shows these tasks.

The typical tasks have been divided into three groups. Group 1 shows tasks whose duration is a subset of the project life span. Group 2 shows tasks whose duration covers the entire life span of the project. Group 3 are those areas discussed in the previous section that occurred for the most part early in the project (with the exception of Financial Accounting) though not necessarily restricted to that period. The logic for this breakdown will become apparent in the discussion of the models.

Table 7-2 also shows the atypical efforts associated with the AIM. The Interface Reports (2 interim and 1 final) are a series of reports on interface analysis and software engineering techniques. The CAIS is an ongoing effort to establish a common set of interfaces for all APSEs. The GAC is a working group to establish transportability guidelines and conventions. The Rehost effort was time spent porting the AIM from the DG system to the VAX and back to the DG.

Table 7-2 AIM Software Development Efforts

	Typical	Atypical
Group 1:		
	Requirements Definition	Interface Reports (3)
	System Specification	CAIS
	Preliminary Design	GAC
	Detailed Design	Rehost
	Test Plans and Procedures	
	Preliminary User's Manual	
	Implementation	
	Integration	
	Formal Testing	
	Documentation Updates	
Group 2:		
	Configuration Management (CM)	
	Quality Assurance (QA)	
	Program Management (PM)	
	Data Management (DM)	
Group 3:		
	New Hire	
	Software Tools	
	Hardware/Software Problems	
	Financial Accounting	

Figure 7-3 shows the percentage of time spent on all efforts. Two pie slices worth special mention are Program Management and CAIS. These two slices account for 21% of the total labor cost of the AIM. The CAIS percentage is a result of one engineer devoting the equivalent of one one full year over the lifetime of the AIM contract working on the CAIS. The Program Management figure is a result of the Program Manager taking a very active role in the AIM project. At the completion of the design phase, the Program Management was involved in the determination of how to implement the AIM given the current state of Ada compilers. The included such options as securing a validated Ada compiler, coding in a different language, or postponing till a future date. Once an option was selected (securing a validated compiler), time was spent extending the contract and leasing the Data General system.

The Interface Reports account for 11% percent of the time. These reports, however, are considered by most to be the primary deliverable of the AIM project.

LIFECYCLE ANALYSIS
AIM PROJECT EFFORTS

The totals for Table 7-2 Group 1 efforts are discussed later. The percentages shown in Figure 7-3 relate to the entire project (through May 1985) including the non-typical efforts.

LIFECYCLE ANALYSIS AIM PROJECT EFFORTS

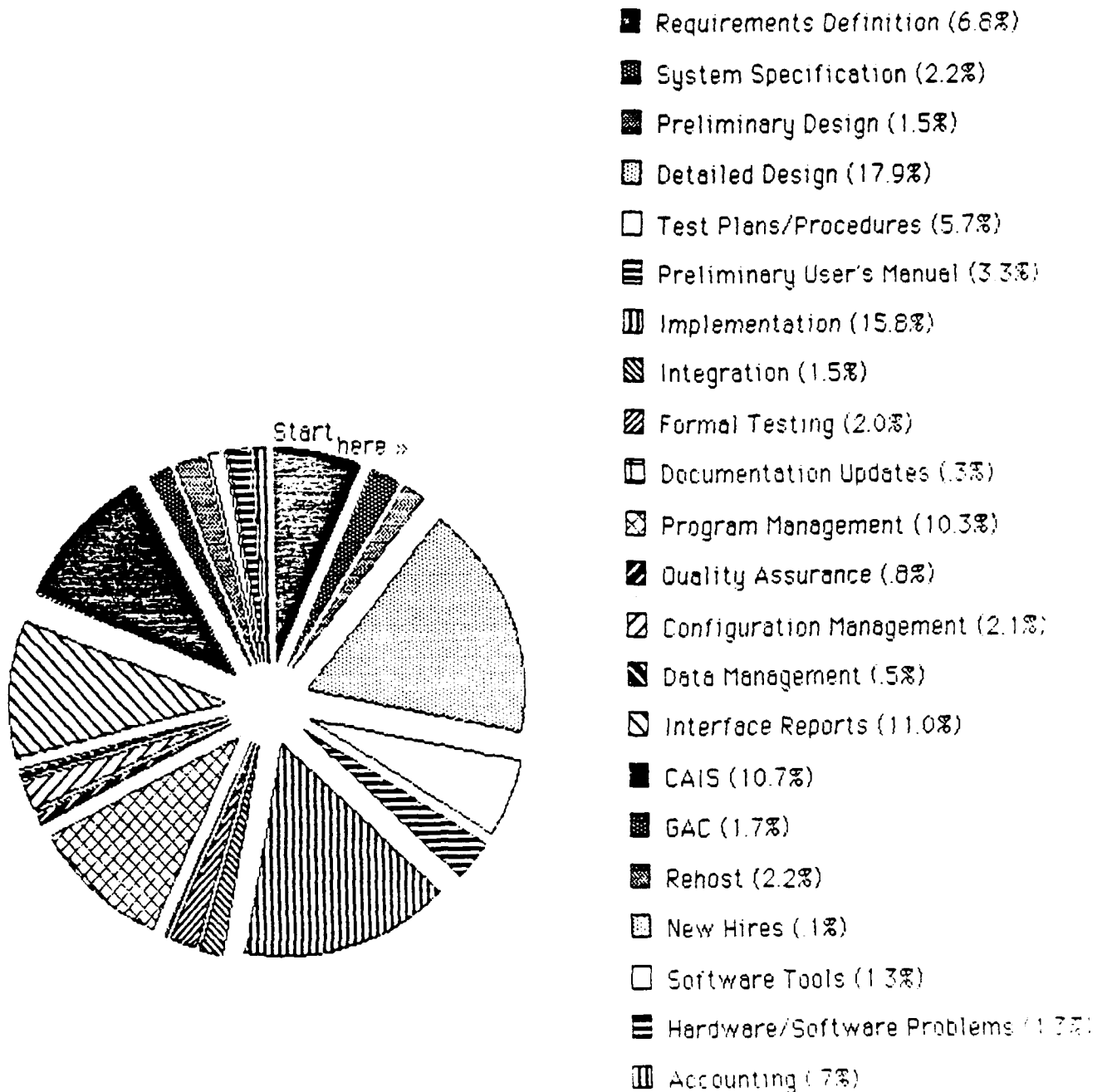


Figure 7-3 AIM Project Total Efforts

LIFECYCLE ANALYSIS TESTING METHODOLOGY

7.4 TESTING METHODOLOGY

Testing of the AIM was performed in three areas: unit, integration, and rehost. (Acceptance Testing will be performed at a later date.) Unit testing was performed by each engineer on the software for which each was responsible. Test cases and test results were reviewed by the other team members during code walk-throughs. Code walk-throughs were performed for all AIM code. Each team member wrote drivers to test individual areas of responsibility.

After all code was written and tested separately, integration of the segments was begun. Integration testing was performed using the AIM System/Integration Test Procedures. Integration testing was repeated at least four times. A record was kept of the testing results and distributed to each team member. The member responsible for any uncovered bugs would fix those bugs and a new round of integration testing would begin. This process was continued until no bugs were found. The same method was followed when the AIM was rehosted from the DG System to the VAX System.

During the Rehost effort, testing was performed mainly through the use of the VAX debugger. Once the system dependent code was written, integrated, and debugged, the test procedures were once again performed. Two new errors were then uncovered. These errors were fixed and the new code, now resident on the VAX, was rehosted to the DG. No errors were encountered in this re-Rehost.

7.4.1 Error Correction

Table 7-3 shows the bug counts and types. Three areas (INFO, HELP, CLI) were used to identify bugs locations. The INFO and HELP categories accurately denote packages where bugs were found. The CLI category is more of a catch-all for not only CLI (Command Language Interpreter) errors but also internal errors for which a clear mapping was unknown. The table shows two types of errors: Format and Logic. The Format errors are those where the data presented on the screen did not match the output specified in the test procedures. These errors were minor in nature. The Logic errors were those causing unreliable results and program crashes.

One of the Logic errors found in HELP was a result of transporting the HELP package to other tools. The other (not really a logic error) was the addition of one procedure to increase the ease of transportability.

Table 7-3 AIM Bug Count and Type

	INFO	HELP	CLI
Format Error			
1st Pass	14	3	25
2nd Pass	4	0	6
3rd Pass	0	0	0
4th Pass	0	0	0
Rehost	0	0	0
Re-Rehost	0	0	0
Logic Error			
1st Pass	0	0	2
2nd Pass	0	0	2
3rd Pass	0	2	2
4th Pass	0	0	1
Rehost	0	0	2
Re-Rehost	0	0	0

7.5 LINES OF CODE

Lines-of-Code (LOC) is an often used (and controversial) measure of productivity for software projects. There are three LOC measurements for the AIM. The first count is the total number of semi-colons in the source files. This count is 7384 lines. The second count is the total number of semi-colons and comments (11190), and the third is the total number of lines in the source files (21059). It should be obvious why LOC is a controversial measure. Table 7-4 shows productivity measures for all counts. Each count includes all type declarations.

Although the semicolon count is an accurate measure of the source code, the comment lines should not be ignored. Each comment written (3806 lines) required an initial effort and subsequent update if the source code changed.

The third count (total lines) includes statements that span more than one line but constitute only one instruction. This includes constructs that, in some cases, contain function calls or expression evaluations. Blank lines, if any, are also included in the total line count. Assuming that the blank lines are minimal, this count could be the most accurate measure of work. Considering that each If-Then-Else statement, for instance, could span several lines and contain function calls and expression evaluations, these lines might represent considerable work.

LIFECYCLE ANALYSIS LINES OF CODE

All counts are presented to allow the preferred interpretation of the reader.

Table 7-4 shows a breakdown of the LOC per day, per month, for the Group 1 tasks and for the Group 1 combined with Group 2 tasks shown in Table 7-2. As can be seen, the LOC range from a low of 4.5 per man-day to a high of 15.8 per man-day. A total with Group 3 included is not shown because the Group 3 effort included Software Tools for which an accurate LOC is not available. None of the above figures include code written for test purposes. A figure for the total effort is not included because it includes the atypical efforts.

Table 7-4 AIM LOC

	7384 Source		11190 Source & Comments		21059 Total Lines	
	Day	Month	Day	Month	Day	Month
Group 1 Only	5.5	120.0	8.4	181.8	15.8	342.2
Group 1 and 2	4.5	96.6	6.8	146.5	12.7	275.6

Unfortunately, there is not enough statistical data available from Ada projects to support meaningful conclusions on productivity. TI recently completed four tools for NOSC for which there is data [TI85H] to compare but the projects were not near the scope of the AIM. Data from those tools is shown in Table 7-5. Each tool was the result of concurrent six month contracts. The deliverables for each tool consisted of the source, acceptance test procedures, user's manual, and technical reports. Time for the technical reports was not included in the computations shown in Table 7-5. Because of the limited deliverables and small scope of the NOSC tools, the Group 1 Only figures in Table 7-4 give the most reasonable comparison. Still, the Group 1 figures include documentation that was not, for the most part, required for the NOSC tools.

The "low" figure for the AIM may be attributed to the lack of experience by all persons participating on the AIM project as well as the degree of difficulty of the AIM. Conversely, causes for the higher figures in Table 7-5 for the NOSC tools could be, to some degree, to

LIFECYCLE ANALYSIS
LINES OF CODE

- o the elevated level of Ada experience attributable to both the AIM and the increased awareness of Ada in the programming world
- o the AIM had numerous tasks whereas, except for in the Virtual Terminal, the NOSC tools had no tasking
- o the NOSC tools were much smaller in scope than the AIM
- o the documentation for the NOSC tools was on a much smaller scale than for the AIM
- o the NOSC tools were implemented on the DG system three months after the AIM implementation had begun thus allowing the NOSC engineers to draw from the AIM experience
- o the degree of reusable software imported into the NOSC tools which are included in the LOC figures. A Help package developed for the AIM was imported to the Spell Checker and Style Checker. This accounted for about 25% of the total code in each of these programs.

The AIM also imported code but to a lesser degree if measured as a percentage of the total code. A part of the Virtual Terminal (1805 source lines) was imported from the NOSC tools.

Table 7-5 NOSC Tools LOC

	Virtual Terminal	Spell Checker	Style Checker	Batch/Forms Generator
Source	2421	2743	3189	2869
LOC/MD	11.4	14.5	17.3	16.7
LOC/MM	246.0	311.0	373.0	359.5
Source & Comments	3011	4848	4681	4576
LOC/MD	14.2	25.6	25.5	26.7
LOC/MM	306.0	549.7	547.5	573.4
Total Lines	6300	7576	7880	8307
LOC/MD	29.8	40.0	42.9	48.4
LOC/MM	640.2	859.0	921.6	1041.0

LIFECYCLE ANALYSIS MODEL COMPARISONS

7.6 MODEL COMPARISONS

There are numerous models available for software projects. Some models are based on percentages and others are based on man-days or man-months. In the following discussion, Lifecycle Models are those that suggest a certain percentage of time for different phases of a project and Costing Models are those that suggest a certain length of time. The Lifecycle Models have set percentages that are expected for the phases of a project. Some degree of variation is allowed in the percentages. The Costing Models differ in that they predict the percentage of time that will be spent based on various parameters entered into the model. Each of the Costing models shown here is an interactive tool. Both types of models are intended for use as guides in the planning phase of a project. This was not the case for the AIM. Since the models were not used at the beginning of the AIM project, an attempt has been made to retro-fit the project to each model. In so doing, a comparison of the AIM is made to historical studies of software projects. Conclusions drawn from these comparisons may help predict future projects of the nature of the AIM.

7.6.1 Lifecycle Models

Lifecycle Models are based on percentages. The percentages vary from model to model depending, to some degree, on the phases identified for each model.

The Group 2 and Group 3 efforts shown in Table 7-2 will be combined and assumed to be evenly distributed for the Lifecycle Models thus the individual percentages are not changed. The reasoning for this approach is that these efforts are primarily auxiliary functions and are not solely attributable to any one phase of a project. The Atypical efforts have also been eliminated because they would not be found in the models. Figure 7-4 shows the percentage of effort for the Group 1 efforts only.

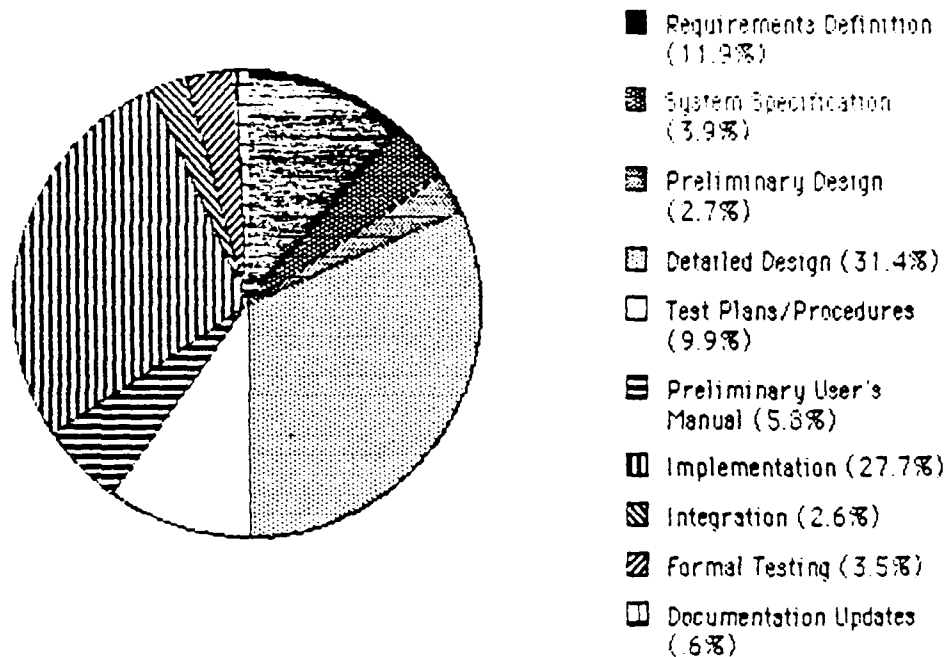


Figure 7-4 Typical Project Efforts

7.6.1.1 40-20-40 Model

The 40-20-40 Model [PRES82] is a very simple model dividing a project into three phases: Design 40%, Coding 20%, Testing 40%. This model is shown in Figure 7-5. The 40-20-40 Model places heavy emphasis on design and testing and very little on coding. In order to compare the AIM to the model, the Group 1 efforts must be divided into the three phases identified in the 40-20-40 model. This is done in Table 7-6. This categorization is consistent with [PRES82] and [DEMA79]. The reason for including the Test Plans and Procedures in the Design Efforts is to ensure that during Design the proper attention is giving to testing. Both testing and detailed design proceed from the

LIFECYCLE ANALYSIS
40-20-40 Model

Preliminary Design Specification. As requirements are designed into the program, tests against that design are written. These tests become the Test Procedures. The Test Plan must precede the Test Procedures. A Preliminary User's Manual is written to ensure that the user's viewpoint is considered during the design effort.

The inclusion of these efforts reflects the actual time frame for the AIM as well as the previously suggested time frame. (i.e., test documentation was done in parallel with design as opposed to being done later with the time then added to the design effort.) Documentation Updates are those efforts required to complete the User's Manual and to update the Test Procedures. The procedures were updated to reflect design changes that occurred since the previous release of the test document. The update effort was included in the testing efforts because the changes to the documents was driven by the testing of the AIM against the Test Procedures and against the User's Manual Tutorial.

Note that the Testing Phase is considerably under and that the Design Phase is considerably over what the 40-20-40 model suggests. This will hold true for all models. The Coding Phase differs from what the model suggest but by a smaller percentage. Explanations for these variances will be presented in the conclusions section.

Figure 7-6 shows the resulting comparison for the AIM.

Table 7-6 40-20-40 Mapping

40-20-40	AIM
Design Efforts	Requirements Definition System Specification Preliminary Design Detailed Design Test Plans and Procedures Preliminary User's Manual
Coding Efforts	Implementation
Testing Efforts	Integration Formal Testing Documentation Updates

LIFECYCLE ANALYSIS
40-20-40 Model

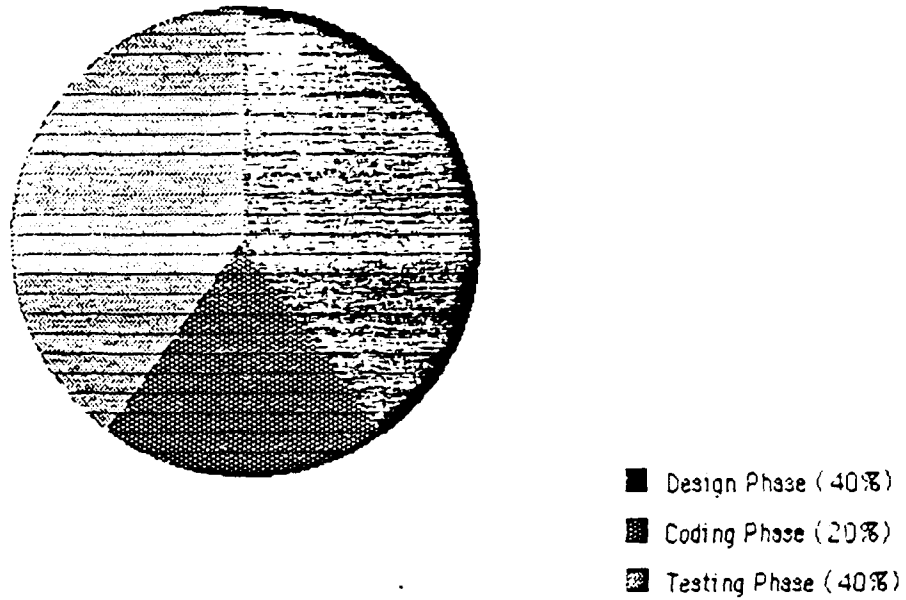


Figure 7-5 40-20-40 Model

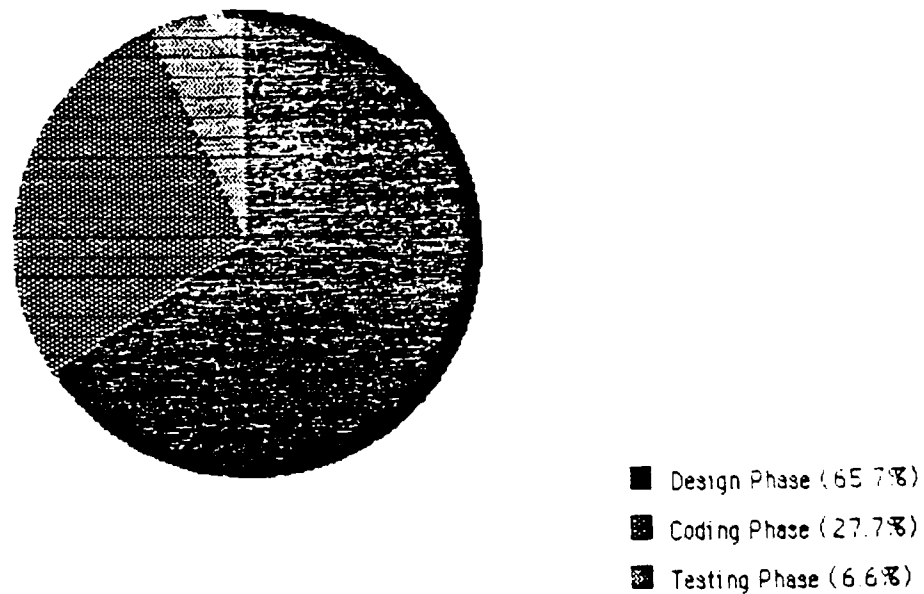


Figure 7-6 AIM vs. 40-20-40 Model

LIFECYCLE ANALYSIS
Brooks Model

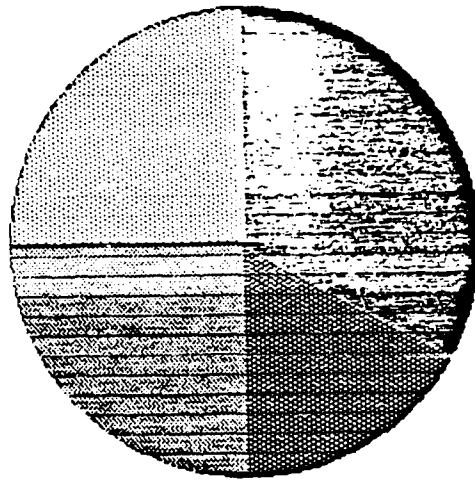
7.6.1.2 Brooks Model

The Brooks Model [BROO79] shown in Figure 7-7 differs from the 40-20-40 model only slightly. This model is the "rule of thumb" used by Frederick P. Brooks for scheduling a software task. The primary difference in the models is in the weight given to the Testing Phase in the Brooks Model. Table 7-7 shows the new mapping. The conversion from Table 7-6 to Table 7-7, reflects calling the Design Phase the Planning Phase and dividing the Testing Phase so that Integration Testing maps to Brooks' Test Phase. The rest remains the same. The result is shown in Figure 7-8. Again, the Testing Phases and the Design (or Planing) Phase are extremely off the model figures and the Coding Phase is higher.

Table 7-7 Brooks Mapping

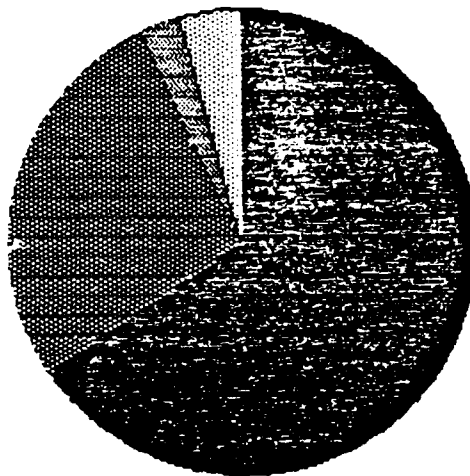
Brooks	AIM
Planning Efforts	Requirements Definition System Specification Preliminary Design Detailed Design Test Plans and Procedures Preliminary User's Manual
Coding Efforts	Implementation
Testing Efforts	Integration
System Test Efforts	Formal Testing Documentation Updates

LIFECYCLE ANALYSIS
Brooks Model



- Plan Phase (33%)
- Coding Phase (17%)
- Testing Phase (25%)
- Test System Phase (25%)

Figure 7-7 Brooks Model



- Plan Phase (65.7%)
- Coding Phase (27.7%)
- Testing Phase (2.6%)
- Test System Phase (4.1%)

Figure 7-8 AIM vs. Brooks Model

LIFECYCLE ANALYSIS

GTE Model

7.6.1.3 GTE Model

The last Lifecycle Model is the GTE Model [DALY77]. This model is shown in Figure 7-9. The GTE Model is divided into more phases than the previous models. Table 7-8 shows the GTE mapping. The following alterations have been made: Map the Requirements Definition to the Plan Phase of the GTE Model, map the System Specification to the Specifications Phase, map the rest of the Design category to the Design Phase, and split the testing as was done for the Brooks Model. The AIM then compares to the GTE Model as shown in Figure 7-10. Here again the Testing and Design Phases of the AIM are not consistent with the model. If the Plan and the Specification Phases are combined in the GTE Model and the AIM comparison, the two do not differ significantly but the Design, Coding, and Testing Phases show the same disparity as the other models.

Table 7-8 GTE Mapping

GTE	AIM
Planning Efforts	Requirements Definition
Requirement Effort	System Specification
Design Effort	Preliminary Design Detailed Design Test Plans and Procedures Preliminary User's Manual
Coding Efforts	Implementation
Testing Efforts	Integration
System Test Efforts	Formal Testing Documentation Updates

LIFECYCLE ANALYSIS
GTE Model

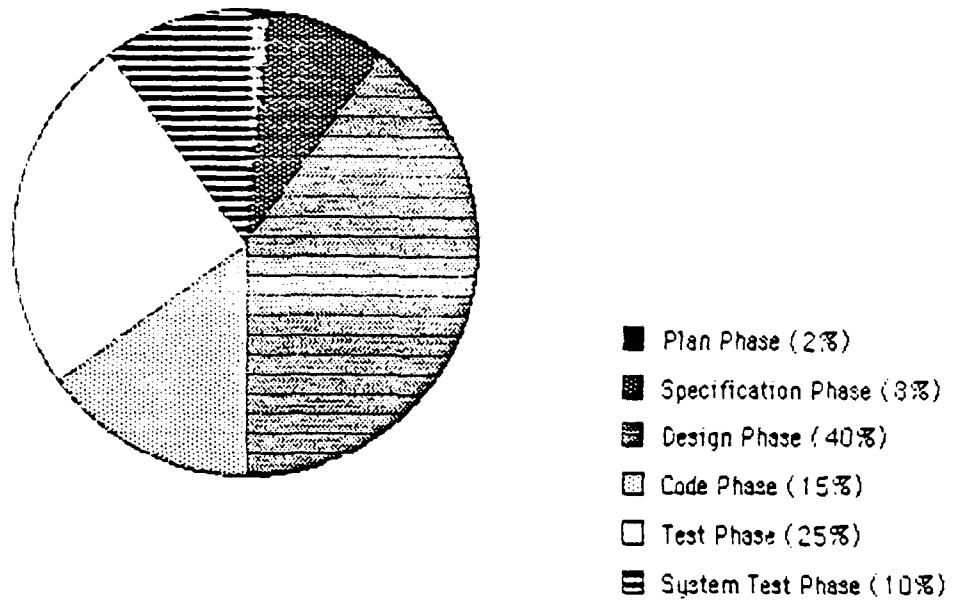


Figure 7-9 GTE Model

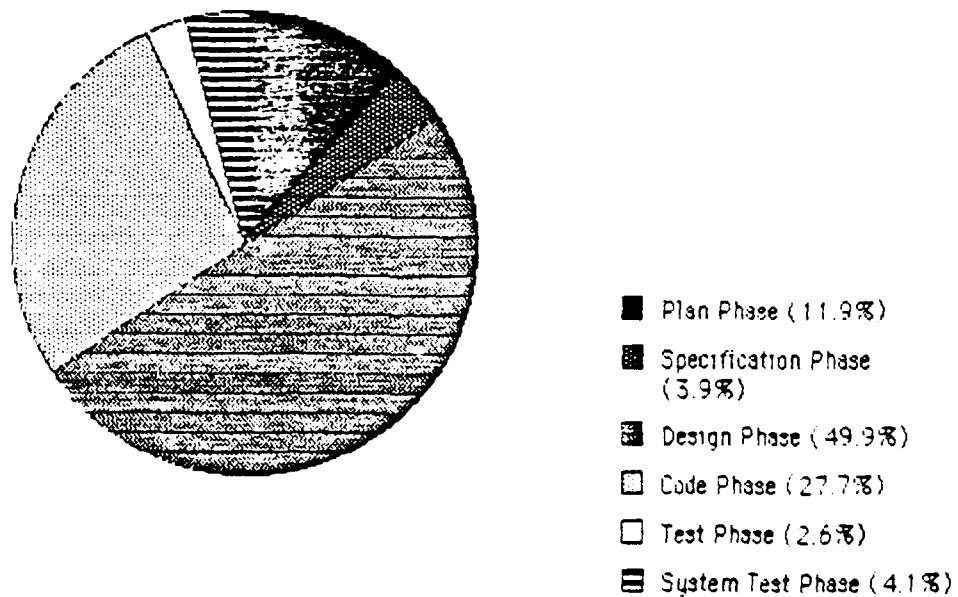


Figure 7-10 AIM vs. GTE Model

LIFECYCLE ANALYSIS

Costing Models

7.6.2 Costing Models

The Costing Models presented here predict the cost of a project. This cost can be computed in dollars, man-days, man-months, or duration depending on the model. Dollars are not used here to eliminate confusion that might arise from differences in the costing structure of different companies. Each of the following model discussions contains a table of the Typical Efforts from Table 7-2 that were used in the computations. As with the Lifecycle Models, these efforts vary from model to model.

7.6.2.1 SoftCost

The SoftCost Model [TI85G] is based on research conducted by the Jet Propulsion Laboratory in Pasadena, California [TAUS81]. This tool is used to help predict software cost in man-days and to project schedules for the different phases of the project. The AIM project data was entered into the model after the Testing phase was complete. The relevant data gathered from the model is the projected man-days for each phase of the project. Table 7-9 shows the mapping of the AIM Efforts used in this model to the efforts predicted by the model. For this model, the Group 2 and Group 3 efforts have been combined and proportionately distributed across the phases.

The SoftCost Model predicts effort based on the assumption that these efforts do take place to some degree in each of the phases of the project. For instance, Quality Assurance is assumed to occur during design, coding, and testing. Figure 7-11 shows the comparison of the actual AIM effort to the effort predicted by the SoftCost Model.

As with the Lifecycle Models, the SoftCost Model differs from the AIM in the areas of Design, Coding and Testing. Note that no entry was made for the AIM in the Initiation Phase but that if the first two phases are added for the AIM and the model, the results are very near the same. The SoftCost Model indicates that more time was expected in the Programming (Coding) Phase as opposed to the Lifecycle Models which predict a smaller percentage for coding than shown for the AIM.

Table 7-9 SoftCost Mapping

SoftCost	AIM
Initiation Phase	
Definition Phase	Requirements Definition System Specification
Design Phase	Preliminary Design Detailed Design Test Plans/Procedures User's Manual Document Updates
Programming Phase	Implementation Integration
System Test Phase	Formal Testing

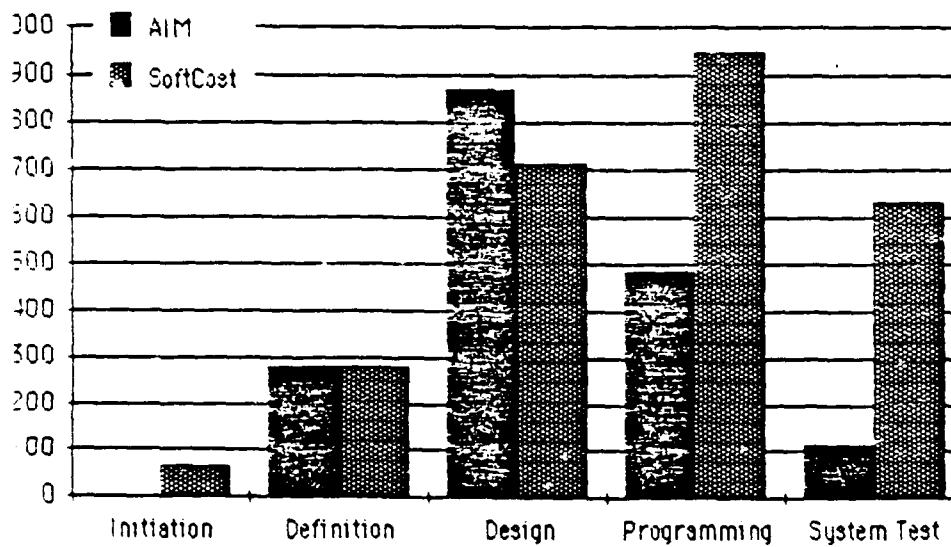


Figure 7-11 SoftCost Model Comparison (Man-Days)

LIFECYCLE ANALYSIS
Price-S

7.6.2.2 Price-S

The Price-S Model [FRIE79] is a parametric cost-modeling method. It is an interactive model that allows the user to customize the input. The model computes duration of effort (not man-months) based on the maximum manload and cost per month for the three phases: Design, Implementation, Integration and Testing. Price-S assumes that all efforts prior to Design have been completed and excludes them. Also, the model expects a continuum of effort. In order to compare the AIM, the months of no activity (Summer 1984) and the months in which no activity for the specified phases took place, (October 1982 - January 1983, October 1983 - December 1983) were removed (See Figure 7-1). The resulting time span was 22 months covering February 1983 through November 1984. The mapping of the AIM project to the Price-S model is shown in Table 7-10.

Figure 7-12 shows the comparison of the AIM to Price-S. The comparison shows each effort as a percentage of the total months. As with all previous models, Design is over and Testing is under; the predicted figures for the AIM. Also, for the first time, there is a large discrepancy in the Implementation Phase.

Table 7-10 Price-S Mapping

Price-S	AIM
Design	Preliminary Design Detailed Design
Implementation	Implementation
Integration and Testing	Integration Formal Testing

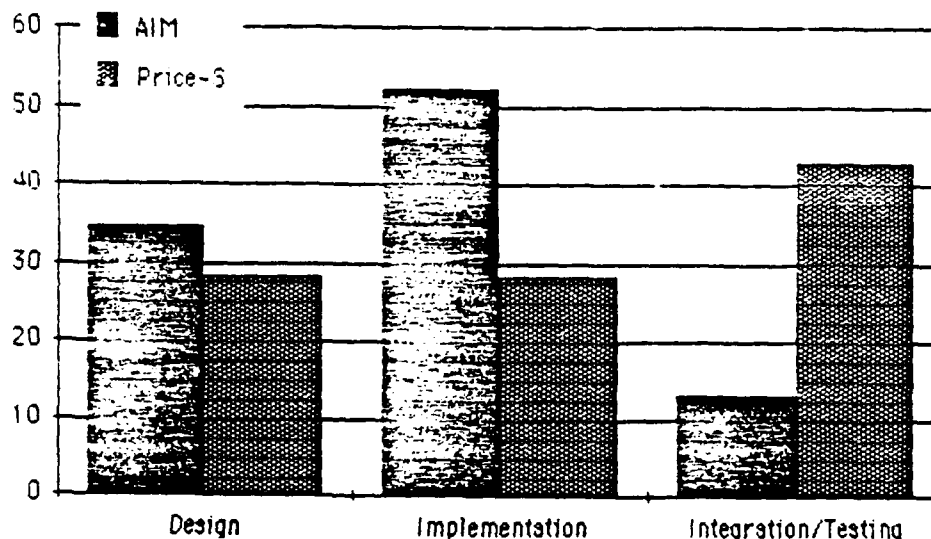


Figure 7-12 Price-S Comparison (Percent of Actual Months)

7.6.2.3 COCOMO

The Constructive Cost Model (COCOMO) [BOEH81] estimates the number of man-months required to develop lines of source code for a software product. Comment lines are excluded from the source code line count. The COCOMO Model covers the period starting with the Design Phase (after requirements acceptance) and continues through the test phase. However, given that the project was complete and the LOC known, the model can estimate the time spent during the planning and requirement specification phases. Excluded from the estimate are charges for training, accounting, software tools, and test drivers. The COCOMO Model assumes that Quality Assurance, Configuration Management, Test Planning, and manual writing are evenly distributed across all phases. Table 7-11 shows the AIM to COCOMO mapping.

Figure 7-13 shows the comparison of the AIM to the COCOMO prediction. The COCOMO Model gives results similar to the Lifecycle Models except in the Programming (Coding) Phase. Like the Price-S Model, the COCOMO Model shows a large discrepancy in predicted coding time. The COCOMO Model prediction, however, differs in the opposite direction.

LIFECYCLE ANALYSIS
COCOMO

Table 7-11 COCOMO Mapping

COCOMO	AIM
Plans and Requirements	Requirements Definition System Specification
Design Phase	Preliminary Design Detailed Design
Programming Phase	Implementation
Testing Phase	Integration Formal Test

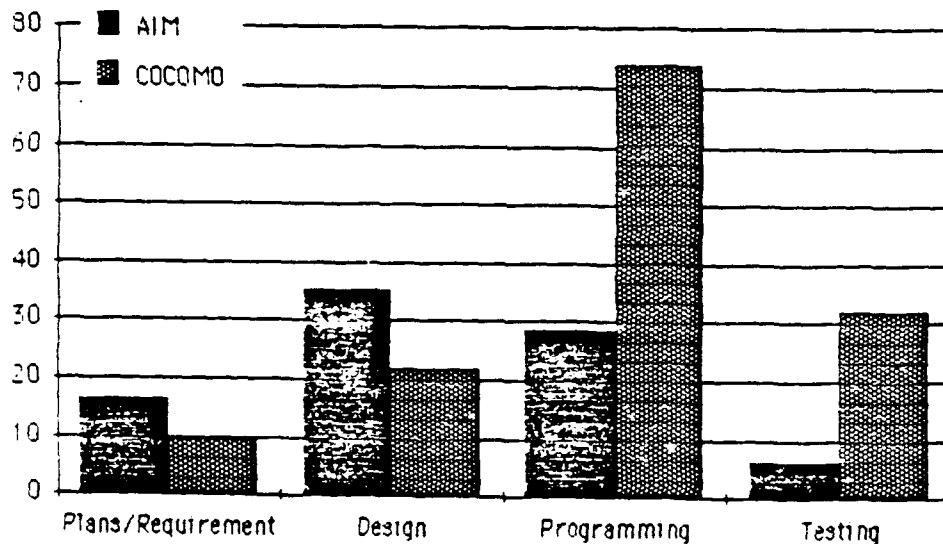


Figure 7-13 COCOMO Comparison (Man-Months)

7.7 CONCLUSIONS

The conclusions drawn from the information contained herein are divided into the following categories:

- o Design Effort
- o Implementation Effort
- o Testing Effort
- o LOC

In the discussion that follows, bear in mind that each model has slightly different groupings for the phases of a software project. The data from each model will be presented as percentages to allow for a common ground of comparison. Percentages that vary only slightly from model to model may be attributed to grouping differences. This is not an attempt to compare the models presented herein. Two types of models, and several of each, were used to provide reliable data from which conclusions could be drawn.

7.7.1 Design Effort

The Design effort for the AIM was much higher than what the models predict. Table 7-12 summarizes the models. The Costing Models results have been converted to percentages for ease of comparison. The table shows the AIM varying from nearly 10% to about 33% higher than the models for design. Several factors contributed to this high ratio. The requirements written for the AIM were reviewed by every member of the AIM during the first year of the project. Because the requirements for the program were defined by the designers of the AIM, there was a lot of redefinition and updating. As the design progressed, a requirements cross-reference was kept. The cross-reference mapped each requirement to the corresponding procedure in the design document that fulfilled that requirement.

In parallel to the writing of the design document, the test documents

- o Acceptance Test Plan
- o Acceptance Test Procedures

LIFECYCLE ANALYSIS
Design Effort

- o Computer Program Test Specification
- o System/Integration Test Plan
- o System/Integration Test Procedures

and a preliminary user's manual were written.

Table 7-12 Summary of AIM vs. Models

	40-20-40 %	Brooks %	GTE %	Softcost M-D	Price-S %	COCOMO M-M
Plan:						
AIM	0.0	0.0	11.9	0.0	0.0	0.0
Model	0.0	0.0	2.0	2.5	0.0	0.0
Specification:						
AIM	0.0	0.0	3.9	15.8	0.0	18.9
Model	0.0	0.0	8.0	10.5	0.0	7.0
Design:						
AIM	65.7	65.7	49.9	49.8	34.8	40.8
Model	40.0	33.0	40.0	27.0	28.5	17.0
Implementation:						
AIM	27.7	27.7	27.7	27.7	52.2	33.1
Model	20.0	17.0	15.0	36.0	28.5	58.2
Testing:						
AIM	6.6	6.7	6.7	6.6	13.0	7.2
Model	40.0	50.0	35.0	24.0	43.1	24.8

Writing the test documents required considerable time but helped clarify and refine each requirement. The preliminary user's manual also helped clarify the requirements by forcing the designers into a

user's viewpoint of the tool. The net result of writing these documents was a very clearly defined set of requirements and an unambiguous understanding of the desired tool by all engineers involved.

The AIM design team also used a new approach to design. This new method, Object Oriented Design, [BOO83] required time for each member to grasp individually, and as a consolidated group, as applied to the AIM. Each of these causes contributed to a prolonged design effort.

In addition, the complexity of the tool itself (windowing, tasking, sub-process management) and the Ada language extended the design effort.

7.7.2 Implementation Effort

Each of the Lifecycle Models predicted a shorter Implementation (Coding) Effort than was the actual case. These differences were considered within a reasonable range. The Cost Model predictions vary. The SoftCost Model, like the Lifecycle Models, has an acceptable variance. The SoftCost Model does, however, differ in the opposite direction. The Price-S and COCOMO Models not only are considerably off, the predictions are almost inverted! No reasonable explanation can be given for the wide discrepancy of these two models. All other models are within reason.

7.7.3 Testing Effort

Referring to Table 7-12 again, notice that the Testing effort was much lower than any of the models suggest as appropriate. Possibly the best explanation for this is code walk-throughs. During Implementation each engineer was required to schedule a code walk-through complete with test cases and results. This meant that all code was thoroughly tested at the unit level and double checked by fellow engineers. If unit testing had been separately tracked, including time spent in code walk-throughs, an increase of over 17% in testing would be necessary before the AIM matched any of the model predictions.

To facilitate the unit testing, an interactive source level debugger was used on both the DG and the VAX. The debugger decreased both time required to locate bugs and the number of drivers necessary to test the code. The importance of good tools cannot be over emphasized! The end result was fewer bugs and less time spent in the testing effort.

Another contributing factor to the low testing figures is the amount of time spent in design. The more quality time spent in design, the fewer the bugs that will exist. As mentioned previously, a great

LIFECYCLE ANALYSIS

Testing Effort

deal of effort was put in defining requirements, writing test documentation, and preparing a preliminary user's manual. Each of these reduced the number of potential errors, thus reducing the amount of time spent in testing.

The testing of the AIM did not include extensive detail such as doing memory dumps after each line of code was executed.

7.7.4 LOC

The AIM project was written in a new language that most will agree is more complicated yet more versatile than many of the more popular languages in use today. Ada is still in its infancy two and a half years after the AIM project was begun. The base of expertise required to achieve high measures of LOC has only started to accumulate.

This is one of the reasons one might conjecture as the cause for the "low" LOC count for the AIM. More specific to the AIM is the original purpose of the project. The intent was to investigate APSE interface issues and for this reason the design of the AIM had to be general enough to allow the AIM to operate in any potential APSE. Unfortunately, no APSE existed at the time the AIM was designed. Initially, time was spent investigating operating systems interfaces to get a feel for what to expect. A new direction was then chosen. Rather than anticipating what would exist, a tool was designed that would require extensive interfaces to any system on which it was installed. The result was an abnormal amount of design work for the resultant code. Since the LOC figure is computed over the entire lifecycle, this contributed to a low LOC.

7.7.5 Wrap-Up

The AIM was designed for transportability and was in fact rehosted from the DG to a VAX with a minimum of effort. In a one month period (2.4 man-months), the AIM was rehosted. This time included the writing of the system dependent code and execution of the System/Integration Test Procedures twice. As previously mentioned, only two bugs were found as a result of the rehost. Most of the Rehost time was spent gaining an understanding of the VAX internals. The AIM is a portable tool with localized system dependent code. The understanding of the internals of the host system is the hardest part of porting the AIM.

The Lifecycle Analysis of any project is filled with assumptions, unique situations, and various interpretations. LOC estimates and model comparisons are filled with pitfalls. Hopefully this report has presented enough data to allow the reader to not only understand the conclusions presented here but also facilitate in the predictions

and estimations of future projects.

CHAPTER 8

DIDS

8.1 PURPOSE

This section of the Interface Report is concerned with the problems encountered in using the contract specified DIDs for the AIM deliverables.

8.2 OVERVIEW

The deliverables were completed in the same order as presented below. Requirements, testing, and terminology will be covered.

The following documents and associated DIDs are discussed:

Program Performance Specification(PPS)	Government DID # DI-E-2136A
Acceptance Test Plan(ATP)	Government DID # DI-T-2142
Computer Program Test Specification(CPTS)	Government DID # DI-E-2143
Acceptance Test Procedures(ATPRO)	Government DID # DI-T-2144
Program Design Specification(PDS)	Government DID # DI-E-2138
System/Integration Test Plan(SITP)	Government DID # DI-T-2142
System/Integration Test Procedures(SITPRO)	Government DID # DI-T-2144

DIDS
PROGRAM PERFORMANCE SPECIFICATION (PPS)

8.3 PROGRAM PERFORMANCE SPECIFICATION (PPS)

8.3.1 Requirements

The PPS defines the requirements and the types of testing that are required for the system. The requirements were divided into four groups:

1. acceptance,
2. performance,
3. host, and
4. requirements on other tools.

The latter two categories were special cases for the AIM tool and are not relevant to this discussion. The PPS DID does not prescribe a method or rules to follow in determining a Performance or Acceptance requirement. The PPS does, however, state that the Acceptance requirements are a subset of the Performance requirements.

The Performance requirements were derived from the text of the PPS. Lacking guidelines for determining requirements, we developed our own. Any statement of action to be performed by the program or the internals of the program modules was called a Performance requirement. Those Performance requirements that could be demonstrated (i.e. results are visible to the user) in the host environment by the completed program were classified as Acceptance requirements.

8.3.2 Testing

The Quality Assurance section of the PPS DID mentions four levels of testing that may be performed:

1. subprogram,
2. program,
3. integration, and

4. acceptance.

Except for acceptance, these levels are discussed in MIL-STD-1679. Acceptance testing is discussed in the PPS DID. Program testing is referred to as Program Performance testing and integration testing is referred to as System(s) Integration testing in MIL-STD-1679. Subprogram testing is referred to as subprogram testing in both.

Acceptance testing is performed to show that the Acceptance requirements have been met. Subprogram and program testing are performed to show that the Performance requirements have been met. Integration testing is performed when the developed program is an element of a larger system involving the integration of two or more programs. Integration testing verifies that the interfaces with the other system programs have been met.

The following shows the types of testing required and the corresponding nomenclature used by each government document:

NOSC Deliverable	PPS	MIL-STD-1679
none.....	none.....	Module
none.....	Subprogram.....	Subprogram
System/Integration.....	Program.....	Program Performance
none.....	Integration.....	System(s) Integration
Acceptance.....	Acceptance.....	none

Note in the above that there are no required deliverables for module and subprogram testing. Also, note that NOSC's System/Integration maps to program performance testing and not integration testing. The AIM, although used in a larger system (the APSE), is an independent tool and therefore requires no integration testing as defined in MIL-STD-1679. Acceptance testing will ensure that the AIM works in its intended environment.

As a last note on the PPS, the DID used in writing the PPS refers to "the combat system" which indicates that the DID is more oriented toward embedded weapon systems as opposed to software environments and tools.

8.4 ACCEPTANCE TEST PLAN (ATP)

The only problem dealing with the DID in writing the ATP was that the DID is a general purpose outline for writing any type of test plan. The same DID was used for writing both the Acceptance Test Plan and the System/Integration Test Plan. The description of the ATP is

DIDS ACCEPTANCE TEST PLAN (ATP)

simple enough until considered with the descriptions of all the other test document DIDs. After reading them all, it becomes difficult to distinguish where one stops and the next starts.

8.5 COMPUTER PROGRAM TEST SPECIFICATION (CPTS)

The DID for the CPTS calls for a two-part document. The first part is called the System test. The System test addresses the complete system operation and interface. The second section is called the Function Test. The Function Test is concerned with testing the individual parts of the program. The CPTS outlined the tests to be developed in the test procedures. Although not clearly stated as such, the System tests corresponded to testing the Acceptance requirements and the Function tests corresponded to testing the Performance requirements. The tests outlined in the Function Tests section later become the System/Integration Test Procedures. The CPTS DID does not give a good feeling for the level of explanation expected for the tests described here. It was difficult determining the differences between the Function Tests of the CPTS and the System/Integration Test Procedures. Therefore, the tests in the CPTS were used as an outline for the test procedures.

8.6 ACCEPTANCE TEST PROCEDURES (ATPRO)

As mentioned in the above, the ATP and SITP used the same DID. The DID used for the ATPRO and SITPRO is also the same. Not only was it difficult to determine the differences in the two sets of test procedures, but the distinction between the descriptions of the test plan, the test specification, and the test procedures makes it even more difficult to determine.

8.7 PROGRAM DESIGN SPECIFICATION (PDS)

The PDS DID calls for a cross reference matrix of the performance requirements defined in the PPS to the paragraph in the PDS where that requirement is satisfied. The paragraph numbers were not used because throughout the development of the PDS these numbers were changing. Since only manual means were available to change the numbers, the routine name was substituted in place of the paragraph number. By using the table of contents, the actual page number could be determined. It was felt that this substitution would be more meaningful, more cost effective, and more prone to stay correct over time.

DIDS

SYSTEM/INTEGRATION TEST PLAN (SITP) AND PROCEDURES (SITPRO)

8.8 SYSTEM/INTEGRATION TEST PLAN (SITP) AND PROCEDURES (SITPRO)

By the time the SITP/SITPRO was written the distinction between the DIDS had been determined. There were no further problems encountered with the test deliverables.

8.9 SUMMARY

The DIDS used for the AIM project were sometimes vague and confusing. The testing required was not precisely defined. There was neither a method given for defining performance and acceptance requirements or a definition of performance and acceptance requirements.

The same DID was used to write the ATP and the SITP. Since there was no distinction made between the two, it was, at times, difficult to determine the differences between them. This DID was not written specifically for the purpose of writing Acceptance Test Plans (or System/Integration Test Plans) but rather for writing any test plans. The same problem occurred with the DID used in writing the Acceptance and the System/Integration test procedures.

The CPTS DID calls for two parts: System Tests and Function Tests. These terms were not used in any of the other test documents. This caused some confusion as to the relationship of these parts to the test plans and procedures. The assumption was made that the System Tests were related to the Acceptance Tests and that the Function Tests were related to the System/Integration Tests.

APPENDIX A

GLOSSARY

ADE Ada Development Environment

AIE Ada Integrated Environment

AIM APSE Interactive Monitor

ALS Ada Language System

AOS/VS Advanced Operating System/Virtual Storage, a Data General Corporation operating system for the ECLIPSE MV/Family of machines.

APSE Ada Programming Support Environment

APSE program A program that can be executed in the hosting APSE and uses only KAPSE supplied services to perform its function.

CAIS Common APSE Interface Set, the KIT/KITIA effort to standardize certain KAPSE interfaces.

CAISWG Common APSE Interface Set Working Group, a working group within the KIT/KITIA effort.

GLOSSARY

character

A member of a set of elements that is used for the organization, control, or representation of data.

character echo

The act of re-transmitting a character immediately upon receipt of it back to the entity that originally transmitted it.

character imaging device

A device that gives a visual representation of data in the form of graphic symbols using any technology, such as cathode ray tube or printer.

character stream

An unbounded sequence of ASCII characters.

character string

A bounded sequence of ASCII characters.

command script

A database file containing commands to the AIM command interpreter. The command interpreter reads commands from the command script rather than prompting the user interactively.

critical region

A section of code in a process (Ada task) which, when executed, is guaranteed mutually exclusive access to shared data.

database file

A standard file in the APSE database.

DG

Data General

display

The area for visual presentation of data on a character imaging device.

display terminal

A data communications device composed of a keyboard and a display screen (usually a cathode ray tube).

EDT

An interactive full-screen editor supported by DEC on the VAX machine.

environment-dependent

Using features which are unique to a specific Ada Program Support Environment (such as ALS or AIE).

erroneous

An Ada program which does not conform to the requirements of an APSE program. The program might execute correctly within an APSE in a given situation, but the program may not be considered entirely reliable. An APSE program must use only KAPSE services; any other services (such as host services) result in an erroneous program.

exclusive access

Control of a file (or, the terminal, in this case) which prohibits any other program besides the AIM from writing to the terminal screen.

host services

Facilities provided by the operating system of the host machine underlying the KAPSE.

image

An analog of the physical display device. The image is the entity that is mapped onto the display. Given a number of user defined images, only one at a time can be mapped onto the display. The rest exist and are updated asynchronously but are not mapped onto the display until the user requests it.

interface

The place at which independent systems meet and act on or communicate with each other.

IPC

Interprocess communication.

KAPSE

Kernel Ada Programing Support Environment.

keyboard

The physical input device.

KIT

KAPSE Interface Team.

KITIA

KAPSE Interface Team from Industry and Academia.

GLOSSARY

LALR

Lookahead Left to Right; a method for parsing grammars.

line

A set of adjacent character positions in a visual display that have the same vertical position.

mappings

The relationships managed by the AIM connecting logical representations of windows, images, and viewports to physical representations on a display device.

MIL-STD

Military Standard.

node

Pertaining to the KAPSE database, either a file or a directory in the tree-structured database.

NOSC

Naval Ocean Systems Center

pad

Two files which contain a complete history of window activity that transpires from the beginning of pad mode until it is terminated by the user or the window is destroyed. One pad, the INPUT pad, includes the input to the APSE program from the user through the keyboard. The other pad, the OUTPUT pad, logs the output to the display from the AIM and any program initiated by the AIM.

Page mode terminal

A screen-oriented display device which possesses extended two-dimensional functional capabilities. Characters are transmitted and received one at a time.

pipe

A logical connection between an output file of one program and an input file of another program.

screen

The area for visual presentation of data on any type of character imaging device, including printer and cathode ray tube device.

STANDARD IN and STANDARD OUT

Input and output files defined in the package TEXT_IO. For AIM purposes, these must be the only files used for terminal I/O,

task

An Ada program unit that operates in parallel with other program units.

terminal

A data communications device consisting of a keyboard and a character imaging device.

Terminal Capabilities File

A file which describes common terminal functions in terms of device-specific control sequences, for many different terminals.

terminal communication protocols

Sequences of characters in which the relationships between specific characters are given meanings for different types of terminals.

transmit

To send data as a data stream for purposes of information interchange.

user terminal

The terminal with which a user interacts in order to communicate with an APSE program.

VMS

Virtual Memory System, the DEC operating system for the VAX 11-780.

viewport

The portion of the window displayed in the image.

viewport header

A single highlighted line located at the top of a viewport.

window

An analog of the APSE program's view of the terminal.

APPENDIX B

REFERENCES

B.1 GOVERNMENT STANDARDS

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [DOD80] United States Department of Defense, "Requirements for Ada Programming Support Environments" ("STONEMAN"), February 1980.
- [DOD83] United States Department of Defense, "Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983," February 17, 1983.
- [DID73] Data Item Description, "Informal Technical Information, DI-S-30593," March 73.

B.2 GOVERNMENT SPECIFICATIONS

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the document referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [INT82] Intermetrics Inc., "IR-678-1 Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type B5," Wakefield, MA, November 1982.
- [KIT83] KAPSE Interface Team (Ada Joint Program Office), "Common APSE Interface Set," Version 1.1, September 1983.

REFERENCES
GOVERNMENT SPECIFICATIONS

- [KIT85] KAPSE Interface Team (Ada Joint Program Office), "Proposed Military Standard Common APSE Interface Set (CAIS)," January, 1985.
- [SOF82] SofTech Inc., Ada Problem Report 602, Waltham, MA, November 1982.
- [SOF83] SofTech Inc., "Draft Ada Language System Specification," Waltham, MA, November 28, 1983

B.3 OTHER GOVERNMENT DOCUMENTS

The following documents of the latest issue per date of this report form a part of this specification.

- [TI82] Texas Instruments, Advanced Computer Systems Laboratory, "Proposal for Development of Ada Software Tools and Interface Standards," Lewisville, TX, February 1982.
- [TI83A] Texas Instruments, "APSE Interactive Monitor (AIM) Program Performance Specification (PPS)," Contract N66001-82-C-0440, 19 September 1983.
- [TI83B] Texas Instruments, "APSE Interactive Monitor (AIM) Software Development Plan (SDP)," Contract N66001-82-C-0440, 10 July 1983.
- [TI83C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Plan (SITP)," Contract N66001-82-C-0440, 23 December 1983.
- [TI83D] Texas Instruments, "APSE Interactive Monitor (AIM) Software Quality Assurance Plan (QA)," Contract N66001-82-C-0440, 28 March 1983.
- [TI83E] Texas Instruments, "APSE Interactive Monitor (AIM) Computer Program Test Specification (CPTS)," Contract N66001-82-C-0440, 15 September 1983.
- [TI83F] Texas Instruments, "APSE Interactive Monitor (AIM) Configuration Management Plan (CM)," Contract N66001-82-C-0440, 28 March 1983.
- [TI83G] Texas Instruments, "Interim Report on Interface Analysis and Software Engineering Techniques," Contract N66001-82-C-0440, May 1983.

REFERENCES
OTHER GOVERNMENT DOCUMENTS

- [TI83H] Texas Instruments, "Interim Report on Interface Analysis and Software Engineering Techniques," Contract N66001-82-C-0440, December 1983.
- [TI85A] Texas Instruments, "APSE Interactive Monitor (AIM) User's Manual (UM)," Contract N66001-82-C-0440, July 1985.
- [TI85B] Texas Instruments, "APSE Interactive Monitor (AIM) Program Design Specification (PDS)," Contract N66001-82-C-0440, July 1985.
- [TI85C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Procedures (SITPRO)," Contract N66001-82-C-0440, July 1985.
- [TI85D] Texas Instruments, "CAIS Rationale," Contract N66001-82-C-0440, July 1985.
- [TI85E] Texas Instruments, "Transportability Guide," Contract N66001-82-C-0440, July 1985.
- [TI85F] Texas Instruments, "Installation and Maintenance Guide for the APSE Interactive Monitor (AIM)," Contract N66001-82-C-0440, July 1985.
- [TI85I] Texas Instruments, "APSE Interactive Monitor (AIM) Acceptance Test Plan (ATP)," Contract N66001-82-C-0440, July 1985.
- [TI85J] Texas Instruments, "APSE Interactive Monitor (AIM) Acceptance Test Procedures (ATPRO)," Contract N66001-82-C-0440, July 1985.

B.4 SPECIAL SOURCES

- [TT83] Verbal communications with Tucker Taft of Intermetrics, Inc., Jan 26, 1983 at the San Diego KIT meeting.
- [TT83A] Verbal communications with Tucker Taft of Intermetrics, Inc., April 21, 1983 at the Willow Grove, PA KIT meeting.
- [RT83] Verbal communications with Rich Thall of SofTech, Inc., Jan 26, 1983 at the San Diego KIT meeting.

REFERENCES
SPECIAL SOURCES

- [RT83A] Verbal communications with Rich Thall of SofTech, Inc., April 20, 1983 at the Willow Grove, PA KIT meeting.

B.5 OTHER PUBLICATIONS

- [ABB82] Abbott, Russell J., "Program Design by Informal English Descriptions," Unpublished.
- [KIN81] Akin, T. Allen, "Virtual Terminal Handler Preliminary Quick Reference," School of Information and Computer Science, Georgia Institute of Technology, April 1981.
- [ANSI73] American National Standards Institute, "American National Standard Graphic Representation of the Control Characters of American National Standard Code for Information Interchange (ANSI Standard X3.32-1973)," July 1973.
- [ANSI77] American National Standards Institute, "American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)," June 1977.
- [ANSI79] American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)," July 1979.
- [APSE82] "Working Paper: Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for Standard Interface Specifications," Not Approved, October 1982.
- [BOEH81] Boehm, Barry W., Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J., 1981.
- [BOO83] Booch, Grady., Software Engineering with Ada, Benjamin Cummings Publishing Company, Menlo Park, CA., Copyright 1983.
- [BOR85] Borger, Mark W., "Software Design Issues in Ada," Journal of Pascal, Ada, and Modula2, Volume 3, Number 3, March-April 1985.
- [BROO79] Brooks, Fredrick P., The Mythical Man-Month, Addison-Wesley, Reading, Mass., 1979.

REFERENCES
OTHER PUBLICATIONS

- [BUH84] Buhr, R. J. A., System Design with Ada, Prentice-Hall, Inc., 1984.
- [COX83] Cox, Fred, "KAPSE Support for Program/Terminal Interaction," Working paper for KITIA/ Working Group 1, February 1983.
- [CSC82A] Computer Sciences Corporation, "Configuration Management System Program Performance Specification (Draft)," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract N00123-80-D-0364.
- [CSC82B] Computer Sciences Corporation, "Configuration Management System Interim Report on Interface Analysis," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract N00123-80-D-0364.
- [DALY77] Daly, Edmund B., "Management of Software Development," IEEE Transactions on Software Engineering, May 1977.
- [DAT83A] Data General Corporation, "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual Volume 1 System Concepts," Westborough, Massachusetts, March 1983.
- [DAT83B] Data General Corporation, "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual Volume 2 System Calls," Westborough, Massachusetts, March 1983.
- [DAT84] Data General Corporation, "Ada Development Environment (ADE) (AOS/VS) User's Manual," Westborough, Massachusetts, March 1983.
- [DEC82] Digital Equipment Corporation, "VAX/VMS I/O User's Guide (volume 1)," Maynard, Massachusetts, May 1982.
- [DEMA79] DeMarco, Tom, Structured Analysis and System Specification, Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [DP82] Datapro Reports on Data Communications, vol 2., Sept 1982, "Display Terminals", n C25-10-101.
- [ELS73] Elson, Mark., Concepts of Programming Languages, Science Research Associates, Inc. Paris, France 1973.
- [FH83] French, Stewart and Harrison, Tim, "The APSE Interactive Monitor," Texas Instruments, Inc., March 1983.

REFERENCES
OTHER PUBLICATIONS

- [FOR83] Foreman, John, "Experiences With Object-Oriented Design," AdaTEC, Cherry Hill, NJ, June 1983.
- [FRA] Franck, R., "Design and Implementation of a Virtual Terminal for a Real-time Application System"
- [FRE83] French, Stewart L., "A Virtual Terminal Specification and Rationale," IEEE Proceedings, 7th International Computer Software and Applications Conference. COMPSAC 83, November 7-11, 1983.
- [FRIE79] Frieman, Frank R. and Park, Robert E., "Parametric Cost Models," RCA PRICE Systems, RCA Corporation, Cherry Hill, N.J., October 1979.
- [GOL83] Goldberg, A. and Robson, D., SMALLTALK-80 The Language and its Implementation, Addison-Wesley Publishing Company, Reading, MA, 1983.
- [GOO75] Goodenough, John B., "Exception Handling Design Issues," ACM SIGPLAN Notices, July 1975, pp 41-45. Association for Computing Machinery, Inc.
- [GREN80] Greninger, Lars and Roberts, Roger, "Considerations for a Local Virtual Terminal Interface," Presented at IEEE Conference, September 1980.
- [GRR80] Groves, L.J. and Rogers, W.J., "The Design of a Virtual Machine for Ada", Communications of the ACM, 1980.
- [HAP83] Habermann, A.N., and Perry, D.E., Ada For Experienced Programmers, Addison-Wesley Publishing Company, 1983.
- [HOA81] Hoare, C.A.R., "The Emperor's Old Clothes," 1980 ACM Turing Award Lecture, Communications of the ACM, Vol 24 No 2, Feb 1981.
- [ISO642] International Standards Organization, Standard number: ISO DP 6429, "Additional Control Functions for Character Imaging Devices (Draft)," Not approved, April 1982.
- [JOY81] Joy, W. and Horton, M., "TERMCAP," UNIX Programmer's Manual, Seventh Edition, Berkley release 4.1, June 1981.
- [LAN79A] Lantz, Keith A., et.al., "RIG: An Overview, Working Paper," University of Rochester, Rochester, NY, 1979.

REFERENCES
OTHER PUBLICATIONS

- [LAN79] Lantz, Keith and Rashid, Richard, "Virtual Terminal Management in a Multiple Process Environment," Proceedings of the Seventh Symposium on Operating Systems Principles, (December 10-12, 1979).
- [LAW78] Lawson, James T. and Mariani, Michael P., "Distributed Data Processing System Design - A Look at the Partitioning Problem," IEEE Press, 1978.
- [LOV81] Loveman, David., "Ada Resolves the Unusual with 'Exceptional' Handling," Electronic Design, January 22, 1981.
- [MAC81] MacEwen, Glen H. and Martin, T. Patrick, "Abstraction Hierarchies in Top-Down Design," The Journal of Systems and Software 2, 213-224(1981), Elsevier Science Publishing Co.
- [MAG79] Magnee, F., Endrizzi, A., and Day, J., "A Survey of Terminal Protocols," Computer Networks, 1979, pp 299-314.
- [MEY81] Meyrowitz, Norman and Moser, Margaret, "BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems," Department of Computer Science, Brown University, December 1981.
- [OLS83] Olsen, Eric W. and Whitehall, Stephen B., Ada for Programmers, Reston Publishing, Inc., 1983.
- [PAR72] Parnas, D.L., "On the Criteria to Be Used in Decomposing Systems into Modules," Communications of the ACM, Volume 15 Number 12, December 1972.
- [PER83] Perry, John W., "Are We Wearing the Emperor's Old Clothes?", INFO-ADA ARPAnet message, 4 Nov 1983.
- [PRES82] Pressman, Roger S., Software Engineering: A Practitioner's Approach, McGraw-Hill, New York City, New York, 1982.
- [SCH78] Schicker, P. and Duenki, A., "The Virtual Terminal Definition," Computer Networks, 1978, pp 429-441.
- [SIM76] DEC-System 10 Simula Language Handbook: Part 1: The Programming Language Simula. Report no. C8398. Part 2: DEC-System 10 Dependent Information, Debugging. Report no. C8399. Part 3: Utility Library. Report no. C10045. Rapportcentralen, FOA 1, S-104 50 Stockholm 80 Sweden.

REFERENCES
OTHER PUBLICATIONS

- [SPE81] Spencer, P.D. and Gordon, D., "Software Development Methods For Use With the IAPX432 Microprocessor," EUROMICRO, 1981, North-Holland Publishing Co.
- [STE81] Stenning, Vic, Et Al., "The Ada Environment: A Perspective," Computer, Volume 14, number 6, June 1981, pp 26-34, 36.
- [SUK81] Sukamar, Srinivas and Wiese, John D, "Hardware and Firmware Support for Four Virtual Terminals in One Display Station," Hewlett-Packard Journal, March 1981.
- [TAF82] Taft, S. Tucker, "Portability and Extensibility in the Kernel and Database of a Programming Support Environment," Intermetrics, March 1982.
- [TAJ79] Tajima, Takashi and Katsuyama, Yoshiki, "Layered and Parametric Approach to Terminal Virtualization," Presented at International Conference on Communications, Boston, MA, June 1979.
- [TAUS81] Tausworthe, Robert C., "Deep Space Network Software Cost Estimation Models," Jet Propulsion Laboratory, Pasadena, California, April 1981.
- [TI81A] Texas Instruments, "Ada Integrated Environment," Lewisville, TX, March 1981. Prepared for Rome Air Development Center (RADC) under DoD Contract F30602-80-C-0293.
- [TI85G] Texas Instruments, "SoftCost Software Price Estimation Model User's Guide," Version 3.4, 1985.
- [TI85H] Texas Instruments, "Technical Report on Tools Designed by Texas Instruments," Contract N660001-84-R-0030, 15 April 1985.
- [THA82] Thall, Richard, "The KAPSE for the Ada Language System," SofTech Inc, Proceedings of the AdaTEC conference on Ada, October 1982.
- [WEB80] Websters New Collegiate Dictionary, G. and C. Merriam Company, Springfield, MA, 1980.
- [WOL81] Wolfe, Martin I., et al., "The Ada Language System," Computer, Volume 14, number 6, June 1981, pp 37-45.

APSE
INTERACTIVE MONITOR

Final Report on Interface
Analysis and Software
Engineering Techniques

VOLUME 3

Transporting an Ada
Software Tool : A
Case Study

Prepared for:

NAVAL OCEAN SYSTEMS CENTER (NOSC)
United States Navy
San Diego, CA 92152

Contract No. N66001-82-C-0440
CDRL No. A012

Equipment Group - ACST
P.O. Box 801, M.S. 8007
McKinney, Texas 75069
15 July 1985

TEXAS INSTRUMENTS
INCORPORATED

Ada is a registered trademark of the U.S. Government, Ada Join.
Program Office (AJPO).

ADE is a trademark of ROLM Corporation.

DEC is a trademark of Digital Equipment Corporation.

ROLM is a registered trademark of ROLM Corporation.

VAX is a trademark of Digital Equipment Corporation.

VMS is a trademark of Digital Equipment Corporation.

CONTENTS

1	INTRODUCTION	1
1.1	The AIM	1
1.2	Design For Transportability	5
1.3	The Environments	5
2	PROCEDURES FOLLOWED	6
2.1	Physical Transfer	6
2.2	System Dependencies	6
2.2.1	Computer Terminal Control And Communications	7
2.2.2	Process Control And Communications	8
2.2.3	Environment Variables	9
2.2.4	Ada Length Representation Clause	9
2.3	Order Of Integration During Rehost	9
2.4	Using A Debugger	10
2.5	Formal Testing	10
2.6	A Feedback Loop	11
3	REHOST PROBLEMS	11
3.1	Code Work-Arounds	11
3.2	Program Termination	12
3.3	System Dependencies	13
3.3.1	Enforcing A Model (and Making Assumptions About It)	13
3.3.2	Terminal Communications	14
3.3.3	The Process Model	16
3.3.4	Complexities Relating To System Services	18
3.3.5	Representation Clauses	20
3.4	Module Testing Baggage	21
4	CONCLUSIONS	21

APPENDIX A GLOSSARY

APPENDIX B REFERENCES

B.1	GOVERNMENT STANDARDS	B-1
B.2	GOVERNMENT SPECIFICATIONS	B-1
B.3	OTHER GOVERNMENT DOCUMENTS	B-2
B.4	OTHER PUBLICATIONS	B-3

1 INTRODUCTION

This document presents a case study of transporting an Ada(tm) software tool from one environment, the Data General AOS/VS Ada Development Environment (ADE)(tm), into another environment, the Digital Equipment Corporation VAX/VMS(tm) environment (with the DEC(tm) Ada compiler and tools).

The APSE Interactive Monitor (AIM) was developed in Ada in the DG ADE. The AIM was transported to the VAX/VMS in 2.4 man-months. The transport turned up many issues including:

- * how to deal with compiler bugs,
- * problems with run-time storage allocation schemes,
- * problems with scheduling and task blocking schemes.
- * how inappropriate assumptions can be made with regard to low-level models of the operating system functions,
- * how inappropriate reliance can exist on operating system services,
- * the positive and negative aspects of techniques that improve transportability,
- * problems relating to debugging a transported tool.

The use of Ada can promote the transportability of source code. This case study shows that, with appropriate transportability guidelines, and attention paid to the details, a software tool written totally in Ada can be moved from one system to another with minor difficulties.

1.1 The AIM

The AIM is a software tool written in Ada developed for the Naval Oceans System Center (NOSC) under contract number N66001-82-C-0440 [TI82][TI83A-J] [TI85A-F]. This contract had the following goals:

- * Identify and investigate operating system interfaces that affect transportability of software tools,
- * Provide support and feedback to NOSC and Ada Joint Program Office (AJPO) KAPSE Interface Team (and KAPSE Interface Team for Industry and Academia) KIT/KITIA. The KIT/KITIA is attempting to develop a common set of operating system interfaces to promote interoperability and transportability of software tools, and provide the interfaces in Ada.

INTRODUCTION

The AIM

- * Provide a useful operational tool, and,
- * Investigate techniques, tools and transportability issues pertinent to developing in Ada.

The AIM is a tool that a user runs from a computer terminal. It provides the user with the capabilities to define windows, images, viewports, and pads. Where:

- * window - A host OS process' view of the computer terminal display screen (that is, some other process than the AIM),
- * image - The users' view of the computer terminal display screen,
- * viewport - A mapping from a window(s) to an image, where many windows can be mapped onto one image, and one window can be mapped onto more than one image (however, one window may not be mapped more than once on the same image).
- * pad - A file containing a transcript of the text in a given window.

The AIM supports running multiple host OS command interpreters (one per window). Please refer to Figure 1 for a visual representation of these definitions.

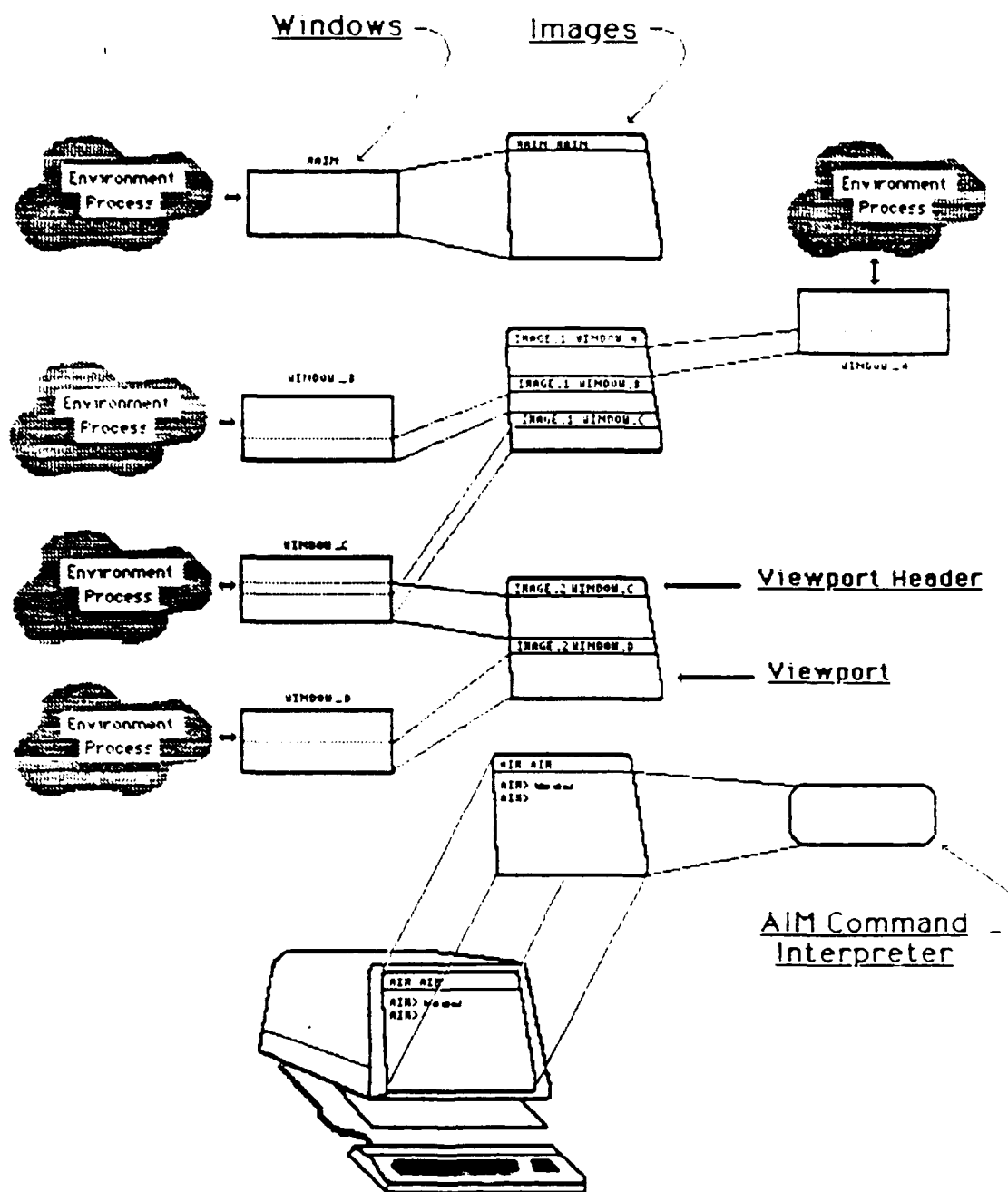


Figure 1. The Elements of the AIM

INTRODUCTION

The AIM

The AIM exercises most Ada Language capabilities of Ada including:

- * packages,
- * tasks,
- * generics,
- * types,
- * aggregates,
- * slices,
- * interfaces to other languages,
- * representation specifications,
- * text I/O,
- * separate compilation.

It does not use:

- * floating or fixed point,
- * task families,
- * interrupts.

Within the AIM there are many sources of asynchronous data (that is, data that appears at random times that must be handled by the AIM):

- * the terminal keyboard,
- * each host OS process (that is running the host command interpreter),
- * internal AIM script files.

The AIM is 22,000 text lines of source code. It is contained in 240 separately compiled text files. When the AIM is running, the number of concurrent Ada tasks range from a minimum of 15 to a maximum of approximately 180.

1.2 Design For Transportability

A great effort was made to increase the transportability of the source code. The following techniques were used:

- * Isolation - the system dependent parts of the AIM were minimized and placed in packages. A model was developed for each system dependency, and interfaces developed (the package specification). The implementation of each interface would need to be rewritten when moving from one system to another.
- * Encapsulation - Object oriented design was used in an attempt to identify the objects, their attributes, and the operations that were to be performed on the objects. These objects were placed in packages. As each package was developed, particular emphasis was placed on minimizing inter-package dependencies.
- * ANSI Ada was used. Chapter 13 issues (optional items) were isolated into the system dependent packages as much as possible.

1.3 The Environments

The DG Ada Development Environment (ADE) [DAT84] is a complete environment that is entered from the standard DG AOS/VS operating system. Within the environment all of the AOS/VS command interpreter commands are available as well as specific ADE commands to support Ada program development.

The DEC VAX/VMS environment has integrated the Ada compiler into the standard VAX/VMS program support environment. The compiler is run like all other integrated compilers on the system. A support system for automatic recompilation, program library management, and other support functions is provided through the Ada Compilation System (ACS).

Both environments include:

- * full ANSI Ada compiler,
- * Ada linker,
- * Ada program librarian,
- * Ada source level debugger,
- * project management capabilities,

INTRODUCTION

The Environments

- * text formatter,
- * editor(s),
- * configuration management support,
- * support capabilities (including file creation, deletion, renaming, copying, printing, etc).

2 PROCEDURES FOLLOWED

This section documents the procedures followed when rehosting the AIM from the ADE into the VMS environment.

2.1 Physical Transfer

The AIM was transferred from the ADE into the VMS environment at the source code level. The two environments have completely different (and proprietary) formats for object and executable files, and no technique exists for converting between them. Since the AIM consists of 240 separate source files, and these files resided in different branches of the AOS/VS tree structured file system, it was necessary to use a transport tool known as Pager. Pager is a transportability tool located on the Ada software repository on the ARPAnet node SIMTEL-20. Pager runs on a variety of machines including the Data General and VAX. It is written entirely in Ada. Pager takes groups of source (more generally, any text) files and groups them into one text file, with separations between the start of one and the end of another. This one file is transferred from the host to the destination, then Pager is run again on the destination computer to put the source back into the separate files from which it came. Pager can also be used to maintain (and reconstruct) the tree level structure of the original directories where the source resided.

Magnetic tape was used to transfer the single paged file from the ADE to VMS. ANSI magnetic tape support exists in both environments.

The entire transport, including paging, dumping the file to tape, reloading the tape onto the destination computer, and unpaging into its original directory structure took one afternoon.

2.2 System Dependencies

The AIM has four interface areas which are dependent on the host operating system:

- * computer terminal control and communications,
- * process control and communications,
- * environment variables,
- * Ada "length" representation clause.

These operating system dependent interfaces were developed in VMS over a period of about one man-month. A rather complete description of these is presented to promote

greater understanding when the problems and issues are discussed in section 3.

2.2.1 Computer Terminal Control And Communications

The AIM computer terminal control and communications package is known as SYSDEP. This package provides very elementary interfaces for controlling and communicating with the computer terminal. The interfaces are:

- * Open the computer terminal - When the terminal is open all characters written to it will be sent directly to the terminal immediately (no buffering), and there will be no translation performed by the host operating system. If the computer terminal cannot be opened, then an exception is raised.
- * Close the computer terminal - Reset the computer terminal back to the characteristics it had before OPEN was called. If there are any outstanding I/O requests pending on the terminal, they will be dequeued immediately.
- * Read data from the computer terminal's keyboard - At least one character is read at a time. There is no translation done on the characters before they are passed back to the calling program. No echo is performed before passing the characters back to the calling program (that responsibility is held by the calling program). This "no echo" characteristic can be setup in the OPEN on some systems. Also, a call on READ must not block the entire process that contains both the reader and the SYSDEP package, only the task calling the read should be blocked. In this manner, tasks can be fired up to infinitely loop reading data, rendezvousing with a buffer task and passing the characters on, eventually to be read by another task.
- * Write data to the computer terminal - A call on WRITE causes a string to immediately be sent to the computer terminal. When the call returns the string either must be on the screen or queued to

PROCEDURES FOLLOWED

System Dependencies

the screen (via host operating system services).

- * Determine the name of the terminal - This package must find a way to determine a unique name of the computer terminal. The AIM expects to have a name that it can then look up in a terminal capabilities database to figure out the escape character sequences appropriate for the particular terminal on which the AIM is running. This is a text string.
- * Determine the name of the terminal capabilities database - filename. This database is a text file that, when given the name, can be manipulated with TEXT_IO.

2.2.2 Process Control And Communications

The process control and communication package is called SYSDEP_PROCESS. This package defines the following interfaces:

- * Create a new son process - A process is an operating system entity that runs as if it is executing in its own computer. The process created will be running the host operating systems' standard command interpreter. Interprocess communication channels will be created and passed into the newly created son process as its standard input and output files. The AIM, therefore, will be able to communicate with the son process through these channels.
- * Destroy a son process - When called the son is immediately destroyed. Anything running in the process is stopped unconditionally. The IPC files are shut down and deleted. Any pending I/O requests are dequeued.
- * Read a line from a process - The calling task is blocked until a line is available to be read (but not the calling process, as in the terminal control and communication section above).
- * Write a text line to the process standard input - This is a text line containing a command that will be interpreted by the program running in the son process. The calling task will not wait for the son process to read the text line from the IPC file. The text line will be queued to the IPC file.
- * Various status and information queries

2.2.3 Environment Variables

An environment variable is a mapping from an AIM internal entity that names a host operating system entity, to an associated host operating system entity. The AIM has four requirements for environment variables. These are:

- * getting the terminal name,
- * getting the name of the terminal capabilities file (TCF),
- * getting initial script file name,
- * getting the parse table initialization file,
- * getting the help file.

On the Data General these are implemented using files. There must be files (or links to files) named TERM, TCF, AIM_INIT_SCRIPT_FILE, and AIM_HELP_FILE on the user's search path.

On the VAX, "logical names" are used for these environment variables. A logical name TERM must exist and contain the name of the terminal. Also, the logical names TCF, AIM_INIT_SCRIPT_FILE, and AIM_HELP_FILE must exist and point to valid filenames.

From the point of view of the calling program there is no difference. The implementation is hidden in the packages SYSDEP and DATABASE_SUPPORT.

2.2.4 Ada Length Representation Clause

To get the tasks to run on the Data General it was necessary to expand the task memory size using the Ada representation length clause (see the Ada LRM section 13.2):

```
for TASK_NAME'SORAGE_SIZE use TASK_SIZE;
```

This clause can have different meaning on different systems. Also, this clause had to be placed in a specific place in the source code.

2.3 Order Of Integration During Rehost

- * The AIM was a debugged, running system, when the rehost was attempted. After the AIM system dependent parts were developed and debugged, the simplest technique for debugging the AIM was simply to compile it all, link it, and run it. It did NOT run the first time. A technique had to be developed to debug the AIM. However, the debugging information had been removed much earlier in the module

Pages 3-368 and 3-369
not available

Page 15
not available

AIM.

3.2 Program Termination

The AIM is a program that has many tasks (up to 180) running at the same time and performing many different functions, including:

- * infinitely looping monitoring an operating system I/O request,
- * managing a buffer, such as accepting reads and writes,
- * infinitely looping reading from one buffer and placing data into a queue, and,
- * managing queues.

The two basic building blocks were infinite loops with no incoming rendezvous' and infinite loops with select blocks.

The major problem in terminating the AIM was this: It could not be determined how to dequeue an I/O request to a device or process. When an infinite loop task was started, reading from the terminal or process and, when the read completed, writing the information to a buffer task, the task blocked at the call to read from the device or process. The AIM would be ready to shut down except for these tasks and the tasks that were dependent on messages from these tasks. The task would be waiting on an I/O request that would never occur. In the case of queued terminal I/O requests the terminating character could simply be typed at the terminal, and the system could shut down gracefully. But this was an inappropriate answer, reacting to the symptom, not really solving the problem (although it was used for a while). The problem was more severe in the case of queued I/O requests to the IPC files associated with the external processes. Shutting down the process, or deleting the IPC file did not seem to dequeue the I/O request.

A system service was discovered, "?TERM". With an appropriate call on "?TERM" the entire AIM process could be deleted and control returned to the host command interpreter. By making use of this system service the AIM could be shut down immediately, regardless of the state it was in.

This solution was used throughout the remaining development, integration, and testing of the AIM, and it worked just fine for the Data General implementation. However, it caused problems during the rehost. The only similar service that could be called from the DEC side was "SEXIT". This system service had problems when tasks were running. It did not work consistently. The DEC, on the other hand,

had the facility to dequeue an I/O request [DEC82]. So, the termination code was reworked to allow the AIM to terminate correctly. That is, the tasks shut themselves down or were waiting at terminate alternatives when the AIM program completed.

By doing this, we diverged from the original implementation. Luckily, a Data General systems programmer gave us the answer to the dequeuing an I/O request. By ABORTing the tasks that were doing the operating system I/O service calls, the queued requests would become dequeued.

By careful encapsulation and isolation, the ABORT statements were embedded in the system dependent code.

3.3 System Dependencies

There was a variety of problems that surfaced in the area of system dependencies. Without performing a detailed analysis of the various systems that you will be rehosting to, you cannot be sure that your model will work in all the systems.

3.3.1 Enforcing A Model (and Making Assumptions About It)

The design of the AIM left nebulous the exact details of the models for terminal control and communications and for process control and communication. During implementation these models firmed up and took shape. The models were implemented as described in section A.2.2.

As it turned out, these were all very reasonable design criteria, and almost every one had problems. They will be addressed in the next two sections in order.

For communications with the terminal the following capabilities will be discussed:

- * read every character from the terminal with no translation,
- * read at least one character at a time,
- * exclusive access to the terminal.

For control and communication with processes, the following capabilities will be discussed:

- * Spawn a son and pass it standard input and standard output files
- All terminal directed output would be intercepted by the AIM through the process' standard output file. All process directed input (that would normally come from the terminal) would be

REHOST PROBLEMS

System Dependencies

supplied by the AIM through the process' standard input.

- * Deleting a process would shut it down immediately regardless of what it was doing -
- * Detect and control any son processes of a son process (grandson processes) - It was felt important to both detect and prune these processes from the process tree to control the operation of the AIM.

3.3.2 Terminal Communications

- * read every character from the terminal with no translation - What about characters such as XON and XOFF? These characters are used for flow control in some systems, and can be generated from the terminal by pressing CONTROL-S and CONTROL-Q. Other problem characters can include: CONTROL-C, CONTROL-Y, CONTROL-O. These characters are intercepted by the different operating systems for different purposes and may not get to the program.

Ways were found around all of the problem control characters except CONTROL-S and CONTROL-Q. It is undesirable to handle flow control. This is an extremely difficult task best left to the operating system. But this problem is indicative of a larger problem.

Consider the use of Local Area and Long Haul Networks. At Texas Instruments Ungermann-Bass LANs are used. These LANs use the tilde character for special purposes. It is intercepted by the LAN. Complicating the situation further, four tildes in a row is a disconnect sequence that will log a user off. Due to this, any sequence of tildes must be followed by another character, at which time all characters will be sent to the host. For example, a tilde typed at the terminal will not be sent to the host, nor two tildes in a row, nor three or four. However a tilde followed by any other character (say an 'A') will, after the 'A' is typed, cause two characters, the tilde and the 'A' to be transmitted to the host.

Going over a TAC to the ARPAnet causes the character '@' to be treated similarly. If a user calls the computer from a remote site using a Novation smart cat 1200 baud modem the '%' key is reserved. On a Hayes modem three quick pluses ("+++") are significant to the modem. Consider the following case: A user at TI would call into the TI LAN from home using a Novation smartcat 1200 baud modem. The LAN talks with the VAX/VMS at 9600 baud. So, for a 1200 baud terminal to talk with VAX tremendous

flow control problems are handled using XON/XOFF all along the communication path. A Hayes 1200 baud modem is connected to one of the ports on the VAX allowing any user logged in to run a special program and communicate out to the Hayes modem. This Hayes modem is used to call a local ARPAnet TAC to gain access to a host where contract accounts exist. The modem support program has a special character that means "ATTENTION" and allows a user to log the data to a file, or exit the program. So, the path is:

User terminal <--> Novation modem <--> LAN <--> VAX <--> Hayes modem
 <--> TAC <--> ARPAnet host

The characters that the user must concern themselves with are:

- * XON/XOFF (CONTROL-S and CONTROL-Q),
- * percent,
- * tilde,
- * at sign '@',
- * modem attention characters (which typically are setable).

The problem can be severe. One solution (which the AIM uses) is to define an interface to the terminal system dependent package which allows a calling program to query about the characters that are important. Passing a CONTROL-S into the function will return a boolean identifying whether you can expect to see that key coming from the keyboard, or be able to send that key to the display screen. This solution is not graceful because it cannot be determined completely which characters are valid at a given instant in time due to the variability of the communication medium that is being used.

- * Read at least one character at a time - In the Data General AOS/VS operating system, characters from the computer terminal can be read only one character at a time. Because of this restriction, code written to support this aspect of the model, that is, handling more characters than one at a time, was not tested until the rehost. The VAX supports reading multiple characters at a time, and the code was checked out on the VAX, then eventually moved back to the DG.
- * Exclusive access to the terminal - This was possible on both systems, but was found to be not desirable. System messages such as "Going Down in 5 Minutes" would never "break through" the AIM to the terminal and could not be intercepted by the AIM on the DG

REHOST PROBLEMS

System Dependencies

or the VAX if exclusive access were enabled. By simply adding a repaint screen procedure, this requirement was eliminated.

A subtle problem was discovered late in the rehost. On the VAX an infinite loop task that makes the operating system I/O call will rendezvous with a buffer task that takes the information and places it into a buffer, from which another task can extract it (at another accept statement in the select clause). Typically, the task rendezvousing with the buffer task to get data from the buffer is suspended until data appears from the terminal and is placed into the buffer task. When the AIM is shut down the queued I/O request is dequeued and the task infinitely looping passes a message to the buffer task telling it to shut down. When the buffer task exits its infinite loop there may be an entry call queued to the READ accept statement. Typically a message could be passed back to the caller before exiting the infinite loop telling it that things are shutting down. But due to problems with the DG implementation, the only reaction that worked was the raising of TASKING_ERROR.

The elegant answer could not be used due to problems with the DG, so the VAX version had to be modified to react with exactly the same semantic meaning that the DG had when shutting down; that is, raising TASKING_ERROR. This problem could not be anticipated when the model was developed, and it affected both systems.

3.3.3 The Process Model

The process model developed had some interesting problems. The requirements of interest are:

- * Spawn a son and pass it standard input and standard output files - This was possible in both systems. When the system was being implemented on the DG it was discovered that the IPC channels were actually files. As the AIM wrote to the process' standard input it went into the IPC file and queued there. So writes were non-blocking (asynchronous). On the VAX it was a different story. The IPC channels (called Mailboxes) were completely synchronous. When a WRITE was issued the calling task was suspended until a process read the message from the IPC channel. This is a completely different IPC model than was used to develop the AIM. Luckily, asynchronous IPC communications can be simulated with synchronous IPC channels by buffering the writes with an Ada queueing package. The asynchronous nature of the IPC channels was simulated in Ada in the SYSDEP_PROCESS package. This was unanticipated but effective. If the AIM had been developed on the VAX and the model had been synchronous IPC channels, it would not have been possible to simulate the synchronous IPC channels using asynchronous IPC channels for the Data General implementation. This was a completely unanticipated

problem with the model.

Another problem that affected the implementation of the model was the passing of standard input and standard output into a son process of the son process (a grandson) of the AIM. On the Data General this simply did not work. The IPC files were passed in and the process was created. But the information from and to the process could not be tracked down. It seemed to disappear. System engineers at Data General studied the problem and told us that that functionality was not supported and would never be supported. They said that it was a design decision made when the operating system was being developed. On the VAX this worked just fine.

- * Deleting a sub-process would shut it down immediately regardless of what it was doing - This is NOT the suicide call described in A.3.2. The system service on the DG "?TERM" is supposed to be able to do this. However, it could not be made to work. A similar facility on the VAX worked just fine. So, in this example, there exists a semantically different meaning for the interface. Contrary to the terminal communication termination problem, where the choice was made to make the semantic meaning the same to preserve the model, here the choice was made to correctly implement the model on the VAX even though it made the two models different. In this case it is preferred that the interface react correctly rather than being consistent and acting incorrectly.
- * Detect and control any son processes of a son process (grandson processes) - This was possible to do on both systems. However, when attempting the implementation an interesting problem arose. The Data General AOS/VS operating system uses processes much more extensively to perform operations than the VAX does. Whenever a command is issued to the command interpreter to invoke a program it spawns a new process in which to run the program. These subprocesses then have the option of spawning new ones (and typically in an unpredictable manner) and thus treeing down. Not all commands to the command interpreter cause a subprocess to be spawned, but most do. Users do not see the spawning of the subprocesses (unless they look for it). Typically there is no limit on the number of subprocesses that a user can spawn. They are resources that are not that precious.

On the VAX, processes are used much less. When a command is issued to the command interpreter, it either does the command or replaces itself in the current process with the program to be run to perform the command. When the command is completed, the command interpreter is brought back into the same process. It is expensive and slow to spawn a subprocess on the VAX. Typically,

REHOST PROBLEMS

System Dependencies

a subprocess quota of 2 or 3 is sufficient for most users.

This is an inherent and subtle difference in the models presented by the two operating systems to the programmer. The difference involves not the presentation of the processes, but their availability and cost.

3.3.4 Complexities Relating To System Services

Interfacing to system services on the Data General AOS/VS operating system is difficult due to the lack of documentation on the supplied interfaces [DAT83A,B]. A set of packages are supplied to perform the necessary operations including:

- * SYS_CALLS - which is the primary package,
- * TTY_IO - to simplify I/O to the terminal,
- * FILE_IO - to simplify I/O to the file system,
- * a file called PARU.32 which contains assembly language source with the error messages that can come out of the system service calls.

TTY_IO and FILE_IO use SYS_CALLS in performing its function. The specifications for these interfaces are quite short and can be scanned easily in an afternoon.

Armed with these interfaces, some small knowledge of how the services are called (mechanically) and the AOS/VS system services manuals (there are two of them), it is possible to get the system service calls figured out.

The VAX system services conform to the standard VAX system service interface techniques. There is a set of packages:

- * STARLET - the entire set of specifications for the system services,
- * CONDITION_HANDLING - a support package containing the constants and types for handling the errors that result from calling a system service,
- * TASKING_SERVICES - supplying interfaces similar to STARLET except supporting non-process blocking I/O,

- * RMS_ASYNC_SERVICES - supplying interfaces similar to STARLET except supporting non-process blocking access to VAX/VMS RMS services;
- * SYSTEM - the predefined package, which has been expanded significantly to support the calling of system services.

Figure 2 shows their WITHing structure.

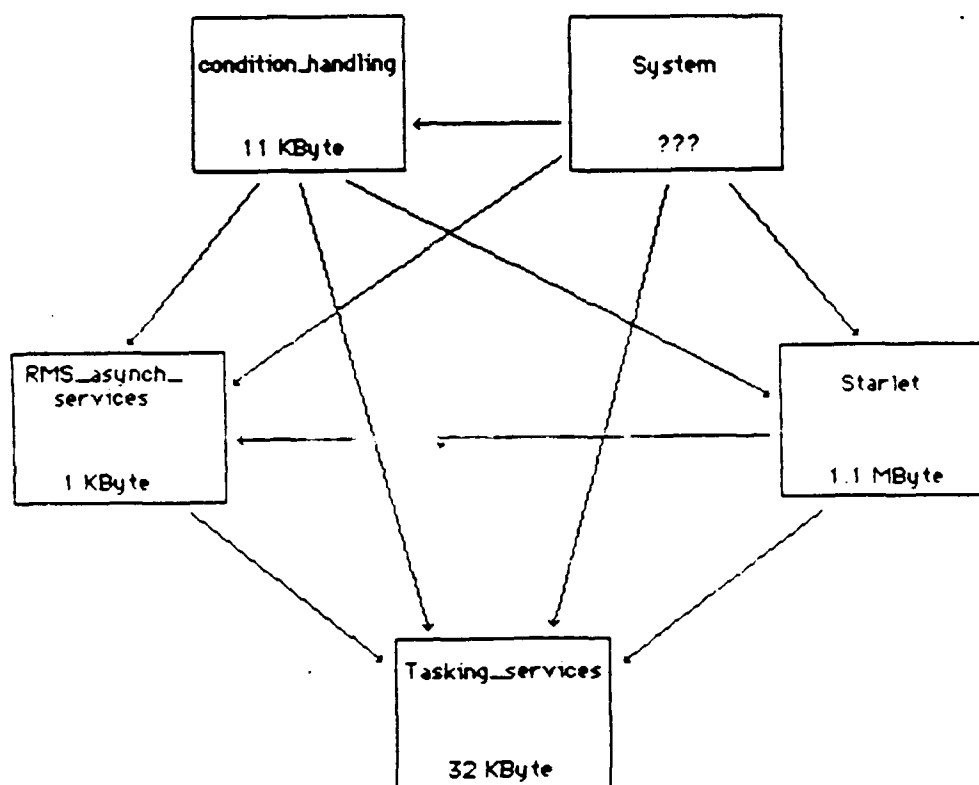


Figure 2. VAX/VMS System Service Package Structure

REHOST PROBLEMS

System Dependencies

These packages use WITH/USE/RENAME for many, many purposes, creating essentially unrecognizable types across the different packages.

The packages are the following sizes:

- * STARLET - 1.1 MByte
- * CONDITION_HANDLING - 11 KByte
- * TASKING_SERVICES - 32 KByte
- * RMS_SYNCH_SERVICES - 1 KByte

After examining and studying them for about a week, the following conclusion was reached:

This is a perfect example of how to make an interface completely Ada-like and almost totally unusable.

Luckily, the source for the specifications of all of the packages (except SYSTEM) was on-line and could be searched with an editor (Emacs in this case). Without that capability, it would have been very difficult making this interface work.

Based on this experience, the following guideline should be used by all software designers with transportability goals:

KEEP IT SIMPLE

3.3.5 Representation Clauses

To get the tasks to run on the Data General it was necessary to expand the task run-time memory size using the Ada representation length clause (see the Ada LRM section 13.2):

for TASK_NAME'SORAGE_SIZE use TASK_SIZE;

The use of this length clause was required due to an unusual memory allocation scheme that the ADE supported. This clause had to appear in the source immediately within the declarative part where the task (or task type) was declared.

On the VAX this clause was not needed. It turned out that it did no harm to be there, however there may be compilers (and their associated run-times) where clauses such as these cannot be made transportable, and the techniques of encapsulation and isolation less usable due to placement requirements within the source code.

3.4 Module Testing Baggage

An analogy between module testing and a rocket taking off can be effective in demonstrating this issue. A multi-stage rocket has a variety of lower stages that are used to boost the payload into orbit. The goal is to obtain orbit. As the lower stages perform their function they are discarded. In some more modern systems the lower stages are recovered, completely refurbished and eventually (if all goes well) re-used.

Module development and testing works similarly. As the modules are debugged and eventually integrated into the whole, the testing baggage becomes obsolete. If a system is to be rehosted, the question arises: Can the modules testing code itself be re-used? When rehosting, one works backwards from the original development. The working system exists and runs correctly on one host. On the new host, however, it may not work. As outlined above, should the testing involve debugging the system as a whole, using a debugger, making changes to the individual modules, then recompiling and re-linking? Or, should an attempt be made to adapt the old module tests? This has some interesting issues associated with it:

- * Can the modules be moved with the source code onto the new host?
- * Should consideration be given to the transportability of the developed module tests?
- * Should the design and documentation of the system reflect the requirements and methods of rehosting the module tests and the data required for these tests?

Lastly, another approach is possible. After it has been determined that the system does not work on the new host, break it down, stubbing out low level modules and developing brand new module tests. This implies a whole new technique for testing which is out of the scope of this document.

4 CONCLUSIONS

The AIM moved from the DG ADE into the VAX/VMS environment in 2.4 man-months. Most of the problems were due to:

- * compiler bugs,
- * inappropriate assumptions made with regard to the low-level models of terminal and process, control and communications,

CONCLUSIONS

- * inappropriate reliance on AOS/VS services to terminate the AIM,

The transport was assisted by:

- * using an ACVC validated Ada compilers,
- * designing in transportability by using the techniques of encapsulation and isolation, and
- * using the source level debugger to identify the problem areas.

APPENDIX A

GLOSSARY

ACS

Ada Compilation System. A DEC product.

ADE

Ada Development Environment. A Data General product.

AIM

APSE Interactive Monitor

AOS/VS

Advanced Operating System/Virtual System. A Data General operating system that runs on the Eclipse model computer systems.

APSE

Ada Programming Support Environment

character

A member of a set of elements that is used for the organization, control, or representation of data.

echo

The act of re-transmitting a character immediately upon receipt of it back to the entity that originally transmitted it.

character string

A bounded sequence of ASCII characters.

command script

A database file containing commands to the AIM command interpreter. The command interpreter reads commands from the command script rather than prompting the user interactively.

GLOSSARY

DG

Data General

DEC

Digital Equipment Corporation

display

The area for visual presentation of data on a character imaging device.

environment variable

A mapping from an AIM internal entity that names a host operating system entity, to an associated host operating system entity.

exclusive access

Control of a file (or, the terminal, in this case) which prohibits any other program besides the AIM from writing to the terminal screen.

flow control

The control of communications to prevent the loss of information due to the receiver being unable to accept it.

image

An analog of the physical display device. The image is the entity that is mapped onto the display. Given a number of user defined images, only one at a time can be mapped onto the display. The rest exist and are updated asynchronously but are not mapped onto the display until the user requests it.

interface

The place at which independent systems meet and act on or communicate with each other.

interoperability

The ability for systems to exchange information.

KAPSE

Kernel Ada Programing Support Environment.

keyboard

The physical input device.

KIT

KAPSE Interface Team.

KITIA

KAPSE Interface Team for Industry and Academia.

LAN

Local Area Network

mapping

The relationships managed by the AIM connecting logical representations of windows, images, and viewports to physical representations on a display device.

NOSC

Naval Ocean Systems Center

pad

Two files which contain a complete history of window activity that transpires from the beginning of pad mode until it is terminated by the user or the window is destroyed. One pad, the INPUT pad, includes the input to the APSE program from the user through the keyboard. The other pad, the OUTPUT pad, logs the output to the display from the AIM and any program initiated by the AIM.

pipe

A logical connection between an output file of one program and an input file of another program.

process

An operating system level flow of control that runs as if it is executing on a separate computer.

screen

The area for visual presentation of data on any type of character imaging device.

son process/sub-process

A process which depends in some manner on the process that created it.

standard input/standard output

Input and output files defined in the package TEXT_IO. For AIM purposes, these must be the only files used for terminal I/O.

system dependent

Using features which are unique to a specific Ada Program Support Environment (such as ADE or VMS).

GLOSSARY

task

An Ada program unit that operates in parallel with other program units.

terminal

A data communications device consisting of a keyboard and a character imaging device.

Terminal Capabilities File (Database)

A file which describes common terminal functions in terms of device-specific control sequences, for many different terminals.

transmit

To send data as a data stream for purposes of information interchange.

transportability

The degree to which a software tool or system can be moved from one environment to another.

user terminal

The terminal with which a user interacts in order to communicate with an APSE program.

VMS

Virtual Memory System, a DEC operating system for the VAX 11 family of minicomputers.

viewport

The portion of the window displayed in the image.

viewport header

A single highlighted line located at the top of a viewport.

window

An analog of the APSE program's view of the terminal.

APPENDIX B

REFERENCES

B.1 GOVERNMENT STANDARDS

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [DOD80] United States Department of Defense, "Requirements for Ada Programming Support Environments" ("STONEMAN"), February 1980.
- [DOD83] United States Department of Defense, "Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983," February 17, 1983.
- [DID73] Data Item Description, "Informal Technical Information, DI-S-30593," March 73.

B.2 GOVERNMENT SPECIFICATIONS

The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the document referenced herein and the contents of this specification, the contents of this specification shall be considered a superceding requirement.

- [INT82] Intermetrics Inc., "IR-678-1 Computer Program Development Specification for Ada Integrated Environment: KAPSE/Database Type B5," Wakefield, MA, November 1982.
- [KIT83] KAPSE Interface Team (Ada Joint Program Office), "Common APSE Interface Set", Version 1.1, September 1983.

REFERENCES
GOVERNMENT SPECIFICATIONS

- [KIT85] KAPSE Interface Team (Ada Joint Program Office), "Proposed Military Standard Common APSE Interface Set (CAIS)", January, 1985.
- [SOF82] SofTech Inc., Ada Problem Report 602, Waltham, MA, November 1982.
- [SOF83] SofTech Inc., "Draft Ada Language System Specification," Waltham, MA, November 28, 1983

B.3 OTHER GOVERNMENT DOCUMENTS

The following documents of the latest issue per date of this report form a part of this specification.

- [TI82] Texas Instruments, Advanced Computer Systems Laboratory, "Proposal for Development of Ada Software Tools and Interface Standards," Lewisville, TX, February 1982.
- [TI83A] Texas Instruments, "APSE Interactive Monitor (AIM) Program Performance Specification (PPS)," Contract N66001-82-C-0440, 19 September 1983.
- [TI83B] Texas Instruments, "APSE Interactive Monitor (AIM) Software Development Plan (SDP)," Contract N66001-82-C-0440, 10 July 1983.
- [TI83C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Plan (SITP)," Contract N66001-82-C-0440, 23 December 1983.
- [TI83D] Texas Instruments, "APSE Interactive Monitor (AIM) Software Quality Assurance Plan (QA)," Contract N66001-82-C-0440, 28 March 1983.
- [TI83E] Texas Instruments, "APSE Interactive Monitor (AIM) Computer Program Test Specification (CPTS)," Contract N66001-82-C-0440, 15 September 1983.
- [TI83F] Texas Instruments, "APSE Interactive Monitor (AIM) Configuration Management Plan (CM)," Contract N66001-82-C-0440, 28 March 1983.
- [TI83G] Texas Instruments, "Interim Report on Interface Analysis and Software Engineering Techniques," Contract N66001-82-C-0440, May 1983.

REFERENCE
OTHER GOVERNMENT DOCUMENTS

- [TI83H] Texas Instruments, "Interim Report on Interface Analysis and Software Engineering Techniques," Contract N66001-82-C-0440, December 1983.
- [TI85A] Texas Instruments, "APSE Interactive Monitor (AIM) User's Manual (UM)," Contract N66001-82-C-0440, July 1985.
- [TI85B] Texas Instruments, "APSE Interactive Monitor (AIM) Program Design Specification (PDS)," Contract N66001-82-C-0440, July 1985.
- [TI85C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Procedures (SITPRO)," Contract N66001-82-C-0440, July 1985.
- [TI85D] Texas Instruments, "CAIS Rationale," Contract N66001-82-C-0440, July 1985.
- [TI85E] Texas Instruments, "Ada Tool Transportability Guide," Contract N66001-82-C-0440, July 1985.
- [TI85F] Texas Instruments, "Installation and Maintenance Guide for the APSE Interactive Monitor (AIM)," Contract N66001-82-C-0440, July 1985.
- [TI85G] Texas Instruments, "APSE Interactive Monitor (AIM) Acceptance Test Plan (ATP)," Contract N66001-82-C-0440, July 1985.
- [TI85H] Texas Instruments, "APSE Interactive Monitor (AIM) Acceptance Test Procedures (ATPRO)," Contract N66001-82-C-0440, July 1985.

B.4 OTHER PUBLICATIONS

- [ABB82] Abbott, Russell J., Program Design by Informal English Descriptions, Unpublished.
- [AKIN81] Akin, T. Allen, "Virtual Terminal Handler Preliminary Quick Reference," School of Information and Computer Science, Georgia Institute of Technology, April 1981.
- [ANSI73] American National Standards Institute, "American National Standard Graphic Representation of the Control Characters of American National Standard Code for Information Interchange (ANSI Standard X3.32-1973)," July 1973.

REFERENCES
OTHER PUBLICATIONS

- [ANSI77] American National Standards Institute, American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)," June 1977.
- [ANSI79] American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)," July 1979.
- [APSE82] "Working Paper: Ada Programming Support Environment (APSE) Requirements for Interoperability and Transportability and Design Criteria for Standard Interface Specifications," Not Approved, October 1982.
- [BAR80] Barnes, J.G.P. "An Overview of Ada", Software-Press.
- [BOO83] Booch, Grady., Software Engineering with Ada. Benjamin Cummings Publishing Company, Menlo Park, CA. Copyright 1983.
- [BOR85] Borger, Mark W., "Software Design Issues in Ada," Journal of Pascal, Ada, and Modula2, Volume 3, Number 3, March-April 1985.
- [BUH84] Buhr, R. J. A., System Design with Ada, Prentice-Hall, Inc., 1984.
- [COX83] Cox, Fred, "KAPSE Support for Program/Terminal Interaction", Working paper for KITIA/ Working Group 1, February 1983.
- [CSC82A] Computer Sciences Corporation, "Configuration Management System Program Performance Specification (Draft)," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract N00123-80-D-0364.
- [CSC82B] Computer Sciences Corporation, "Configuration Management System Interim Report on Interface Analysis," Falls Church, VA, August 1982. Prepared for Naval Ocean Systems Center under contract N00123-80-D-0364.
- [DAT83A] Data General Corporation, "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual Volume 1 System Concepts", Westborough, Massachusetts, March 1983.
- [DAT83B] Data General Corporation, "Advanced Operating System/Virtual Storage (AOS/VS) Programmer's Manual Volume 2 System Calls", Westborough, Massachusetts, March 1983.

REFERENCES
OTHER PUBLICATIONS

- [DAT84] Data General Corporation, "Ada Development Environment (ADE) (AOS/VS) User's Manual", Westborough, Massachusetts, March 1983.
- [DEC82] Digital Equipment Corporation, "VAX/VMS I/O User's Guide (Volume 1)", Maynard, Massachusetts, May 1982.
- [DP82] Datapro Reports on Data Communications, vol 2., Sept 1982, "Display Terminals", p C25-10-101
- [ELS73] Elson, Mark. Concepts of Programming Languages, Science Research Associates, Inc. Paris, France 1973.
- [FH83] French, Stewart and Harrison, Tim, "The APSE Interactive Monitor" Texas Instruments, Inc., March 1983.
- [FOR83] Foreman, John, Experiences With Object-Oriented Design, AdaTEC, Cherry Hill, NJ, June 1983.
- [FRA] Franck, R., "Design and Implementation of a Virtual Terminal for a Real-time Application System"
- [FRE83] French, Stewart L., "A Virtual Terminal Specification and Rationale," IEEE Proceedings, 7th International Computer Software and Applications Conference, COMPSAC 83, November 7-11, 1983.
- [GOL83] Goldberg, A. and Robson, D., SMALLTALK-80 The Language and its Implementation, Addison-Wesley Publishing Company, Reading, MA, 1983.
- [GOO75] Goodenough, John B. "Exception Handling Design Issues", ACM SIGPLAN Notices, July 1975, pp 41-45. Association for Computing Machinery, Inc.
- [GREN80] Greninger, Lars and Roberts, Roger, "Considerations for a Local Virtual Terminal Interface," Presented at IEEE Conference, September 1980.
- [GRR80] Groves, L.J. and Rogers, W.J. "The Design of a Virtual Machine for Ada", Communications of the ACM, 1980.
- [HAP83] Habermann, A.N., and Perry, D.E. Ada For Experienced Programmers, Addison-Wesley Publishing Company, 1983.
- [HOA81] Hoare, C.A.R. "The Emperor's Old Clothes", 1980 ACM Turing Award Lecture, Communications of the ACM, Vol 24 No 2, Feb 1981.

REFERENCES
OTHER PUBLICATIONS

- [ISO642] International Standards Organization, Standard number: ISO DP 6429, "Additional Control Functions for Character Imaging Devices (Draft)," Not approved, April 1982.
- [JOY81] Joy, W. and Horton, M., "TERMCAP," UNIX Programmer's Manual, Seventh Edition, Berkeley release 4.1, June 1981.
- [LAN79A] Lantz, Keith A., et.al., RIG: An Overview, Working Paper, University of Rochester, Rochester, NY, 1979.
- [LAN79B] Lantz, Keith and Rashid, Richard, Virtual Terminal Management in a Multiple Process Environment, Proceedings of the Seventh Symposium on Operating Systems Principles, (December 10-12, 1979).
- [LAW78] Lawson, James T. and Mariani, Michael P., Distributed Data Processing System Design - A Look at the Partitioning Problem, IEEE Press, 1978.
- [LOV81] Loveman, David. "Ada Resolves the Unusual with 'Exceptional' Handling", Electronic Design, January 22, 1981.
- [MAC81] MacEwen, Glen H. and Martin, T. Patrick, Abstraction Hierarchies in Top-Down Design, The Journal of Systems and Software 2, 213-224(1981), Elsevier Science Publishing Co.
- [MAG79] Magnee, F., Endrizzi, A., and Day, J, "A Survey of Terminal Protocols," Computer Networks, 1979, pp 299-314.
- [MEY81] Meyrowitz, Norman and Moser, Margaret, "BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems," Department of Computer Science, Brown University, December 1981.
- [OLS83] Olsen, Eric W. and Whitehall, Stephen B., Ada for Programmers, Reston Publishing, Inc., 1983.
- [PAR72] Parnas, D.L., On the Criteria to Be Used in Decomposing Systems into Modules, Communications of the ACM, Volume 15 Number 12, December 1972.
- [PER83] Perry, John W. "Are We Wearing the Emperor's Old Clothes?" INFO-ADA ARPAnet message, 4 Nov 1983.
- [SCH78] Schicker, P. and Duenki, A., "The Virtual Terminal Definition," Computer Networks, 1978, pp 429-441.

REFERENCES
OTHER PUBLICATIONS

- [SIM76] DEC-System 10 Simula Language Handbook: Part 1: The Programming Language Simula. Report no. C8398. Part 2: DEC-System 10 Dependent Information, Debugging. Report no. C8399. Part 3: Utility Library. Report no. C10045. Rapportcentralen, FOA 1, S-104 50 Stockholm 80 Sweden.
- [SPE81] Spencer, P.D. and Gordon, D., Software Development Methods For Use With the IAPX432 Microprocessor, EUROMICRO, 1981, North-Holland Publishing Co.
- [STE81] Stenning, Vic, Et Al., "The Ada Environment: A Perspective," Computer, Volume 14, number 6, June 1981, pp 26-34, 36.
- [SUK81] Sukamar, Srinivas and Wiese, John D, "Hardware and Firmware Support for Four Virtual Terminals in One Display Station," Hewlett-Packard Journal, March 1981.
- [TAF82] Taft, S. Tucker, "Portability and Extensibility in the Kernel and Database of a Programming Support Environment," Intermetrics, March 1982.
- [TAJ79] Tajima, Takashi and Katsuyama, Yoshiki, "Layered and Parametric Approach to Terminal Virtualization," Presented at International Conference on Communications, Boston, MA, June 1979.
- [TI81A] Texas Instruments, "Ada Integrated Environment," Lewisville, TX, March 1981. Prepared for Rome Air Development Center (RADC) under DoD Contract F30602-80-C-0293.
- [THA82] Thall, Richard, "The KAPSE for the Ada Language System," SofTech Inc, Proceedings of the AdaTEC conference on Ada, October 1982.
- [WEB80] Websters New Collegiate Dictionary, G. and C. Merriam Company, Springfield, MA, 1980.
- [WOL81] Wolfe, Martin I., et al., "The Ada Language System," Computer, Volume 14, number 6, June 1981, pp 37-45.

DoD
Requirements
and
Design Criteria
for the Common APSE Interface Set (CAIS)

13 September 1985

Prepared and approved by the
KAPSE Interface Team (KIT)
and the
KIT-Industry-Academia (KITIA)

for the
Ada* Joint Program Office
Washington, D.C.

* Ada is a Registered Trademark of the U.S. Government,
Ada Joint Program Office

Table of Contents

PREFACE

1. INTRODUCTION	1-1
1.1 Scope.	1-1
1.2 Terminology.	1-1
1.3 Relationship to Specifications & Implementations.	1-2
1.4 Reference Documents.	1-2
2. GENERAL DESIGN OBJECTIVES	2-1
2.1 Scope of the CAIS.	2-1
2.2 Basic Services.	2-1
2.3 Implementability.	2-1
2.4 Modularity.	2-1
2.5 Extensibility.	2-1
2.6 Technology Compatibility.	2-1
2.7 Uniformity.	2-2
2.8 Security.	2-2
3. GENERAL SYNTAX AND SEMANTICS	3-1
3.1 Syntax	3-1
3.1A General Syntax.	3-1
3.1B Uniformity.	3-1
3.1C Name Selection.	3-1
3.1D Pragmatics.	3-1
3.2 Semantics	3-1
3.2A General Semantics.	3-1
3.2B Responses.	3-2
3.2C Exceptions.	3-2
3.2D Consistency.	3-2
3.2E Cohesiveness.	3-2
3.2F Pragmatics.	3-2
4. ENTITY MANAGEMENT SUPPORT	4-1
4.1 Entities, Relationships, and Attributes	4-2
4.1A Data.	4-2
4.1B Elementary Values.	4-2
4.1C System Integrity.	4-2
4.2 Typing	4-2
4.2A Types.	4-3
4.2B Rules about Type Definitions.	4-3
4.2C Type Definition.	4-3
4.2D Changing Type Definitions.	4-3
4.2E Triggering.	4-3
4.3 Identification	4-4
4.3A Exact Identities.	4-4

4.3B Identification.	4-4
4.3C Identification Methods.	4-4
4.4 Operations	4-5
4.4A Entity Operations.	4-5
4.4B Relationship Operations.	4-5
4.4C Attribute Operations.	4-5
4.4D Exact Identity Operations.	4-5
4.4E Uninterpreted Data Operations.	4-5
4.4F Synchronization.	4-6
4.4G Access Control.	4-6
4.5 Transaction.	4-6
4.5A Transaction Mechanism.	4-6
4.5B Transaction Control.	4-6
4.5C System Failure.	4-6
4.6 History.	4-6
4.6A History Mechanism.	4-6
4.6B History Integrity.	4-7
4.7 Robustness and Restoration.	4-7
4.7A Robustness and Restoration.	4-7
5. PROGRAM EXECUTION FACILITIES	5-1
5.1 Activation of Program	5-2
5.1A Activation.	5-2
5.1B Unambiguous Identification.	5-2
5.1C Activation Data.	5-2
5.1D Dependent Activation.	5-2
5.1E Independent Activation.	5-2
5.2 Termination	5-2
5.2A Termination.	5-2
5.2B Termination of Dependent Processes.	5-2
5.2C Termination Data.	5-3
5.3 Communication	5-3
5.3A Data Exchange.	5-3
5.4 Synchronization	5-3
5.4A Task Waiting.	5-3
5.4B Parallel Execution.	5-3
5.4C Synchronization.	5-3
5.4D Suspension.	5-3
5.4E Resumption.	5-3
5.5 Monitoring	5-3
5.5A Identify Reference.	5-3
5.5B RTS Independence.	5-3
5.5C Instrumentation.	5-4
6. INPUT/OUTPUT	6-1
6.1 Virtual I/O Devices: Data Unit Transmission	6-2

6.1A Hardcopy Terminals.	6-2
6.1B Page Terminals.	6-2
6.1C Printers.	6-2
6.1D Paper Tape Drives.	6-2
6.1E Graphics Support.	6-2
6.1F Telecommunications Support.	6-2
6.2 Virtual I/O Devices: Data Block Transmission	6-2
6.2A Block Terminals.	6-2
6.2B Tape Drives.	6-2
6.3 Datapath Control	6-2
6.3A Interface Level.	6-2
6.3B Timeout.	6-3
6.3C Exclusive Access.	6-3
6.3D Datastream Redirection.	6-3
6.3E Datapath Buffer Size.	6-3
6.3F Datapath Flushing.	6-3
6.3G Output Datapath Processing.	6-3
6.3H Input/Output Sequencing.	6-3
6.4 Data Unit Transmission	6-3
6.4A Data Unit Size.	6-3
6.4B Raw Input/Output.	6-3
6.4C Single Data Unit Transmission.	6-3
6.4D Padding.	6-4
6.4E Filtering.	6-4
6.4F Modification.	6-4
6.4G Input Sampling.	6-4
6.4H Transmission Characteristics.	6-4
6.4I Type-Ahead.	6-4
6.4J Echoing.	6-4
6.4K Control Input Datastream.	6-4
6.4L Control Input Trap.	6-5
6.4M Trap Sequence.	6-5
6.4N Data Link Control.	6-5
6.5 Data Block Transmission	6-5
6.5A Data Block Size.	6-5
6.6 Data Entity Transfer	6-5
6.6A Common External Form.	6-5
6.6B Transfer.	6-5
6.7 General Input/Output	6-5
6.7A Waiting.	6-5
6.7B Unsupported Features.	6-5

PREFACE

The KAPSE Interface Team (KIT), and its companion Industry-Academia team (KITIA), were formed by a Memorandum of Agreement (MOA) signed by the three services and the Undersecretary of Defense in January, 1982. Their purpose is to contribute to the achievement of Interoperability of application databases and Transportability of software development tools ("I&T"). These are economic objectives identified at the outset of the DoD common language initiative in the mid-1970's. Progress toward fulfilling these objectives is now acknowledged to require a level of commonality among Ada Programming Support Environments (APSEs), in addition to the standard language Ada [Ada83]. The core of the KIT/KITIA strategy to fulfill I&T objectives is to define a standard set of APSE interfaces ("CAIS" for "Common APSE Interface Set"), which augment the Ada language with the functionality needed to implement tools, thus improving the ability to share tools and databases between conforming APSEs. Note that a number of these interfaces are at the Kernel APSE (KAPSE) level, while others address a higher level of functionality. This document establishes requirements and design objectives (called "criteria") on the definition of a CAIS.

This document *refines some of the DoD "Stoneman" Requirements for Ada Programming Support Environments [Buxton80]* and imposes them upon a CAIS specification. The DoD "Steelman" Requirements for High Order Computer Programming Languages [Fisher78] and the several sets of ANSI "OSCRL" requirements and design objectives for Operating System Command and Response Languages [OSCRL82] have also influenced this document.

1. INTRODUCTION

1.1 Scope. This document provides the Department of Defense's requirements and design criteria for the definition and specification of a Common APSE Interface Set (CAIS) for Ada Programming Support Environments (APSEs).

1.2 Terminology. Precise and consistent use of terms has been attempted throughout the document.

Potentially ambiguous terms used in the document are defined in the Glossary of KIT/KITLA Terminology [KK85]. Some definitions tailored to the context of this document are provided in the sections of the document where they are used.

Additionally, the following verbs and verb phrases are used throughout the document to indicate where and to what degree individual constraints apply. Any sentence not containing one of the following verbs or verb phrases is a definition, explanation or comment.

"SHALL" indicates a requirement on the definition of the CAIS; sometimes "shall" is followed by "provide" or "support," in which cases the following two definitions supersede this one.

"SHALL PROVIDE" indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities.

"SHALL SUPPORT" indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities or for CAIS definers to demonstrate that the capability may be constructed from CAIS interfaces.

"SHOULD" indicates a desired goal but one for which there is no objective test.

1.3 Relationship to Specifications & Implementations. This document specifies functional capabilities which are to be provided in the semantics of a CAIS specification and are therefore to be provided by conforming CAIS implementations. In general, the specifications of software fulfilling those capabilities (and decisions about including or not including CAIS interfaces for certain capabilities as suggested by the "shall support" definition in the previous section) are delegated to the CAIS definers. If a CAIS implementor determines that it is feasible, then the CAIS implementor may provide a particular specified CAIS facility by reusing other CAIS facilities, thereby achieving a "layered implementation" of the CAIS. Therefore, the realization of a specific CAIS implementation is the result of intentionally divided decision-making authority among 1) this requirements document, 2) CAIS definers, and 3) CAIS implementors.

1.4 Reference Documents.

MILITARY STANDARDS

[Ada83] Reference Manual for the Ada Language, ANSI/MIL-STD-1815A, January 1983.

OTHER GOVERNMENT DOCUMENTS

[Buxton80] "Stoneman" DoD Requirements for Ada Programming Support Environments, February 1980.

[Fisher78] "Steelman" DoD Requirements for High Order Computer Programming Languages, June 1978.

[KK85] Glossary of KIT/KITIA Terminology, draft 1985.

[TCSEC83] Trusted Computer System Evaluation Criteria, CSC-STD-001-83. DoD Computer Security Center, August 15, 1983.

NON-GOVERNMENT DOCUMENTS

[OSCRL82] Operating System Command and Response Languages, proposed ANSI standard drafts, 1982.

2. GENERAL DESIGN OBJECTIVES

2.1 Scope of the CAIS. The CAIS shall provide interfaces sufficient to support the use of APSEs for wide classes of projects throughout their lifecycles and to promote I&T among APSEs.

2.2 Basic Services. The CAIS should provide simple-to-use mechanisms for achieving common, simple actions. Features which support needs of less frequently used tools should be given secondary consideration.

2.3 Implementability. The CAIS specification shall be machine independent and implementation independent. The CAIS shall be implementable on bare machines and on machines with any of a variety of operating systems. The CAIS shall contain only interfaces which provide facilities which have been demonstrated in existing commercial or military software systems. CAIS features should be chosen to have a simple and efficient implementation in many machines, to avoid execution costs for unneeded generality, and to ensure that unused portions of a CAIS implementation will not add to execution costs of a non-using tool. The measures of the efficiency criterion are, primarily, minimum interactive response time for APSE tools and, secondarily, consumption of resources.

2.4 Modularity. Interfaces should be partitioned such that the partitions may be understood independently and they contain no undocumented dependencies between partitions.

2.5 Extensibility. The design of the CAIS should facilitate development and use of extensions of the CAIS; i.e., CAIS interfaces should be reusable so that they can be combined to create new interfaces and facilities.

2.6 Technology Compatibility. The CAIS shall adopt existing standards where applicable. For example, recognized standards for device characteristics are provided by ANSI, ISO, IEEE, and DoD.

2.7 Uniformity. All CAIS features should uniformly address aspects such as status returns, exceptional conditions, parameter types, and options. Different modules within the CAIS should be specified to the same logical level, and a small number of unifying conceptual models should underlie the CAIS.

2.8 Security. The CAIS shall provide interfaces to allow tools to operate within a Trusted Computer System (TCS) that meets the Class B3 criteria as defined in [TCSEC83]. Specifically:

- a. It shall be possible to implement the CAIS within a TCS.
- b. When implemented within a TCS, the CAIS shall support the use of the security facilities provided by the Trusted Computing Base (TCB) to applications programs.
- c. When not implemented within a TCS, the CAIS interfaces sensitive to security shall operate as a dedicated secure system (i.e., all data at a single security level, and all subjects cleared to at least that level).

3. GENERAL SYNTAX AND SEMANTICS

3.1 Syntax

3.1A General Syntax. The syntax of the CAIS shall be expressed as Ada package specifications. The syntax of the CAIS shall conform to the character set as defined by the Ada standard (section 2.1 of ANSI/MIL-STD-1815A [Ada83]).

3.1B Uniformity. The CAIS should employ uniform syntactic conventions and should not provide several notations for the same concept. CAIS syntax issues (including, at least, limits on name lengths, abbreviation styles, other naming conventions, relative ordering of input and output parameters, etc.) should be resolved in a uniform and integrated manner for the whole CAIS.

3.1C Name Selection. The CAIS should avoid coining new words (literals or identifiers) and should avoid using words in an unconventional sense. Ada identifiers (names) defined by the CAIS should be natural language words or industry accepted terms whenever possible. The CAIS should define Ada identifiers which are visually distinct and not easily confused (including, at least, that the CAIS should avoid defining two Ada identifiers that are only a 2-character transposition away from being identical). The CAIS should use the same name everywhere in the interface set, and not its possible synonyms, when the same meaning is intended.

3.1D Pragmatics. The CAIS should impose only those restrictive rules or constraints required to achieve I&T. CAIS implementors will be required to provide the complete specifications of all syntactic restrictions imposed by their CAIS implementations.

3.2 Semantics

3.2A General Semantics. The CAIS shall be completely and unambiguously defined. The specification of semantics should be both precise and understandable. The semantic specification of each CAIS interface shall include a precise statement of assumptions (including execution-time preconditions for calls), effects on global data and packages, and interactions with other interfaces.

3.2B Responses. The CAIS shall provide responses for all interface calls, including informative non-null responses (return value or exception) for unsuccessful completions. All responses returned across CAIS interfaces shall be defined in an implementation-independent manner. Every time a CAIS interface is called under the same circumstances, it should return the same response.

3.2C Exceptions. The CAIS interfaces shall employ the mechanism of Ada exceptions to report exceptional situations that arise in the execution of CAIS facilities. The CAIS specification shall include exceptions (with visible declarations) for all situations that violate the preconditions specified for the CAIS interfaces. The CAIS specification shall include exceptions (with visible declarations) that cover all violations of implementation-defined restrictions.

3.2D Consistency. The description of CAIS semantics should use the same word or phrase everywhere, and not its possible synonyms, when the same meaning is intended.

3.2E Cohesiveness. Each CAIS interface should provide only one function.

3.2F Pragmatics. The CAIS specification shall enumerate all aspects of the meanings of CAIS interfaces and facilities which must be defined by CAIS implementors. CAIS implementors will be required to provide the complete specifications for these implementation-defined semantics.

4. ENTITY MANAGEMENT SUPPORT

Access controls and security rights will apply to all CAIS facilities required in this section.

The general requirements for the CAIS entity management support are the following.

- a. There shall be a means for retaining data.
- b. There shall be a way for retaining relationships among and properties of data.
- c. There shall be a way of operating upon data, deleting data, and creating new data.
- d. There shall be a means for defining certain operations and conditions as legal, for enforcing the definitions, and for accepting additional definitions of legality.
- e. There shall be a means to describe data, and there shall be a means to operate upon such descriptions. Descriptions of the data shall be distinguished from the data described.
- f. There shall be a way to develop new data descriptions by inheriting (some of) the properties of existing data descriptions.
- g. The relationships and properties of data shall be separate from the existence of the data instances.
- h. The descriptions of data and the instances of data shall be separate from the tools that operate upon them.

This characterization (subsections 4.1 - 4.7) of Entity Management Support is based on the STONEMAN requirements for a database, using a model based on the entity-relationship concept. Although a CAIS design meeting these requirements is expected to demonstrate the characteristics and capabilities reflected here, it is not necessary that such a design directly employ this entity-relationship model.

The entity-relationship model, for which definitions and requirements follow in 4.1 - 4.7, fulfills these requirements, and any alternative data model shall fulfill these requirements and shall also fulfill the equivalent of the requirements in 4.1 through 4.7.

4.1 Entities, Relationships, and Attributes The following definitions, used in this subsection, pertain to all the rest of section 4 also:

ENTITY a representation of a person, place, event or thing.

RELATIONSHIP an ordered connection or association among entities. A relationship among N entities (not necessarily distinct) is known as an "N-ary" relationship.

ATTRIBUTE an association of an entity or relationship with an elementary value.

ELEMENTARY VALUE
one of two kinds of representations of data: interpreted and uninterpreted.

INTERPRETED DATA
a data representation whose structure is controlled by CAIS facilities and may be used in the CAIS operations. Examples are representations of integer, string, real, date and enumeration data, and aggregates of such data.

UNINTERPRETED DATA
a data representation whose structure is not controlled by CAIS facilities and whose structure is not used in the CAIS operations. Examples might be representations of files, such as requirements documents, program source code, and program object code.

4.1A Data. The CAIS shall provide facilities for representing data using entities, attributes or binary relationships. The CAIS may provide facilities for more general N-ary relationships, but it is not required to do so.

4.1B Elementary Values. The CAIS shall provide facilities for representing data as elementary values.

4.1C System Integrity. The CAIS facilities shall ensure the integrity of the CAIS-managed data.

4.2 Typing The following definition, used in this subsection, pertains to all the rest of section 4 also:

TYPING an organization of entities, relationships and attributes in which they

are partitioned into sets, called entity types, relationship types and attribute types, according to designated type definitions.

4.2A Types. The facilities provided by the CAIS shall enforce typing by providing that all operations conform to the type definitions. Every entity, relationship and attribute shall have one and only one type.

4.2B Rules about Type Definitions. The CAIS type definitions shall

- specify the entity types and relationship types to which each attribute type may apply
- specify the type or types of entities that each relationship type may connect and the attribute types allowed for each relationship type
- specify the set of allowable elementary values for each attribute type
- specify the relationship types and attribute types for each entity type
- permit relationship types that represent either functional mappings (one-to-one or many-to-one) or relational mappings (one-to-many or many-to-many)
- permit multiple distinct relationships among the same entities
- impose a lattice structure on the types which includes inheritance of attributes, attribute value ranges (possibly restricted), relationships and allowed operations.

4.2C Type Definition. The CAIS shall provide facilities for defining new entity, relationship and attribute types.

4.2D Changing Type Definitions. The CAIS shall provide facilities for changing type definitions. These facilities shall be controlled such that data integrity is maintained.

4.2E Triggering. The CAIS shall provide a conditional triggering mechanism so that prespecified procedures or operations (such as special validation techniques employing multiple attribute value checking) may be invoked whenever values of indicated attributes change. The CAIS shall provide facilities for defining such triggers and the operations or procedures which are to be invoked.

4.3 Identification The following definitions, used in this subsection, pertain to all the rest of section 4 also:

EXACT IDENTITY

a designation of an entity (or relationship) that is always associated with the entity (or relationship) that it designates. This exact identity will always designate exactly the same entity (or relationship), and it cannot be changed.

IDENTIFICATION

a means of specifying the entities, relationships and attributes to be operated on by a designated operation.

4.3A Exact Identities. The CAIS shall provide exact identities for all entities. The CAIS shall support exact identities for all relationships. The exact identity shall be unique within an instance of a CAIS implementation, and the CAIS shall support a mechanism for the utilization of exact identities across all CAIS implementations.

4.3B Identification. The CAIS shall provide identification of all entities, attributes and relationships. The CAIS shall provide identification of all entities by their exact identity. The CAIS shall support identification of all relationships by their exact identity.

4.3C Identification Methods. The CAIS shall provide identification of entities and relationships by at least the following methods:

- identification of some "start" entity(s), the specification of some relationship type and the specification of some predicate involving attributes or attribute types associated with that relationship type or with some entity type. This method shall identify those entities which are related to the identified start entity(s) by relationships of the given relationship type and for which the predicate is true. Subject to the security constraints of section 2.8, all relationships and entities shall be capable of identification via this method, and all attributes and attribute types (except uninterpreted data) shall be permitted in the predicates.
- identification of an entity type or relationship type and specification of some predicate on the value of any attribute of the entity type or relationship type. This method shall identify those entities or relationships of the given type for which the predicate is true. Subject to the security constraints of section 2.8, all attributes (except uninterpreted data) shall be permitted in the predicates.

4.4 Operations

4.4A Entity Operations. The CAIS shall provide facilities to:

- create entities
- delete entities
- examine entities (by examining their attributes and relationships)
- modify entities (by modifying their attributes)
- identify entities (as specified in Section 4.3)

4.4B Relationship Operations. The CAIS shall provide facilities to:

- create relationships
- delete relationships
- examine relationships (by examining their attributes)
- modify relationships (by modifying their attributes)
- identify relationships (as specified in Section 4.3)

4.4C Attribute Operations. The CAIS shall provide facilities to:

- examine attributes
- modify attributes

4.4D Exact Identity Operations. The CAIS shall provide facilities to:

- pass exact identities between processes
- compare exact identities

4.4E Uninterpreted Data Operations. The CAIS shall provide that use of the input-output facilities of the Ada language (as defined in Chapter 14 of ANSI/MIL-STD-1815A [Ada]) results in reading/writing an uninterpreted data attribute of an entity. The facilities of Section 6 shall then apply.

4.4F Synchronization. The CAIS shall provide dynamic access synchronization mechanisms to individual entities, relationships and attributes.

4.4G Access Control. The CAIS shall provide selective prohibition of operations on entities, relationships, and attributes being requested by an individual.

4.5 Transaction. The following definition, used in this subsection, pertains to all the rest of section 4 also:

TRANSACTION a grouping of operations, including a designated sequence of operations, which requires that either all of the designated operations are applied or none are; e.g., a transaction is uninterruptible from the user's point of view.

4.5A Transaction Mechanism. The CAIS shall support a transaction mechanism. The effect of running transactions concurrently shall be as if the concurrent transactions were run serially.

4.5B Transaction Control. The CAIS shall support facilities to start, end and abort transactions. When a transaction is aborted, all effects of the designated sequence of operations shall be as if the sequence were never started.

4.5C System Failure. System failure while a transaction is in progress shall cause the effects of the designated sequence of operations to be as if the sequence were never started.

4.6 History. The following definitions, used in this subsection, pertain to all the rest of section 4 also:

HISTORY a recording of the manner in which entities, relationships and attribute values were produced and of all information which was relevant in the production of those entities, relationships or attribute values.

4.6A History Mechanism. The CAIS shall support a mechanism for collecting and utilizing history. The history mechanism shall provide sufficient information to support comprehensive configuration control.

4.6B History Integrity. The CAIS shall support mechanisms for ensuring the fidelity of the history.

4.7 Robustness and Restoration. The following definitions, used in this subsection, pertain to all the rest of section 4 also:

BACKUP a redundant copy of some subset of the CAIS-managed data. The subset is capable of restoration to active use by a CAIS implementation, particularly in the event of a loss of completeness or integrity in the data in use by implementation.

ARCHIVE a subset of the CAIS-managed data that has been relegated to backing storage media while retaining the integrity, consistency and availability of all information in the entity management system.

4.7A Robustness and Restoration. The CAIS shall support facilities which ensure the robustness of and ability to restore CAIS-managed data. The facilities shall include at least those required to support the backup and archiving capabilities provided by modern operating systems.

5. PROGRAM EXECUTION FACILITIES

Access controls and security rights will apply to all CAIS facilities required by this section.

The following definitions pertain specifically to this section:

PROCESS	the CAIS facility used to represent the execution of any program.
PROGRAM	a set of compilation units, one of which is a subprogram called the "main program." Execution of the program consists of execution of the main program, which may invoke subprograms declared in the compilation units of the program.
RESOURCE	any capacity which must be scheduled, assigned, or controlled by the operating system to assure consistent and non-conflicting usage by programs under execution. Examples of resources include: CPU time, memory space (actuals and virtual), and shared facilities (variables, devices, spoolers, etc.).
ACTIVATE	to create a CAIS process. The activation of a program binds that program to its execution environment, which are the resources required to support the process's execution, and includes the program to be executed. The activation of a process marks the earliest point in time which that process can be referenced as an entity within the CAIS environment.
TERMINATE	to stop the execution of a process such that it cannot be resumed.
DEACTIVATE	to remove a terminated process so that it may no longer be referenced within the CAIS environment.
SUSPEND	to stop the execution of a process such that it can be resumed. In the context of an Ada program being executed, this implies the suspension of all tasks, and the prevention of the activation of any task until the process is resumed. It specifically does not imply the release of any resources which a process has assigned to it, or which it has acquired to support its execution.
RESUME	to resume the execution of a suspended process.
TASK WAIT	delay of the execution of a task within a process until a CAIS service

requested by this task has been performed. Other tasks in the same process are not delayed.

5.1 Activation of Program

5.1A Activation. The CAIS shall provide a facility for a process to create a process for a program that has been made ready for execution. This event is called activation.

5.1B Unambiguous Identification. The CAIS shall provide facilities for the unambiguous identification of a process at any time between its activation and deactivation; one such capability shall be as an indivisible part of activation. This act of identification establishes a reference to that process. Once such a reference is established, that reference will refer to the same process until the reference is dissolved. A reference is always dissolved upon termination of the process that established the reference. A terminated process may not be deactivated while there are references to that process.

5.1C Activation Data. The CAIS shall provide a facility to make data available to a program upon its activation.

5.1D Dependent Activation. The CAIS shall provide a facility for the activation of programs that depend upon the activating process for their existence.

5.1E Independent Activation. The CAIS shall provide a facility for the activation of programs that do not depend upon the activating process for their existence.

5.2 Termination

5.2A Termination. The CAIS shall provide a facility for a process to terminate a process. There shall be two forms of termination; the voluntary termination of a process (termed completion) and the abnormal termination of a process. Completion of a process is always self-determined, whereas abnormal termination may be initiated by other processes.

5.2B Termination of Dependent Processes. The CAIS shall support clear, consistent rules defining the termination behavior of processes dependent on a terminating process.

5.2C Termination Data. The CAIS shall provide a facility for termination data to be made available. This data shall provide at least an indication of success or failure for processes that complete. For processes that terminate abnormally the termination data shall indicate abnormal termination.

5.3 Communication

5.3A Data Exchange. The CAIS shall provide a facility for the exchange of data among processes.

5.4 Synchronization

5.4A Task Waiting. The CAIS shall support task waiting.

5.4B Parallel Execution. The CAIS shall provide for the parallel execution of processes.

5.4C Synchronization. The CAIS shall provide a facility for the synchronization of cooperating processes.

5.4D Suspension. The CAIS shall provide a facility for suspending a process.

5.4E Resumption. The CAIS shall provide a facility to resume a process that has been suspended.

5.5 Monitoring

5.5A Identify Reference. The CAIS shall provide a facility for a process to determine an unambiguous identity of a process and to reference that process using that identity.

5.5B RTS Independence. CAIS program execution facilities shall be designed to require no additional functionality in the Ada Run-Time System (RTS) from that provided by Ada semantics. Consequently, the implementation of the Ada RTS shall be independent of the CAIS.

5.5C Instrumentation. The CAIS shall provide a facility for a process to inspect and modify the execution environment of another process. This facility is intended to promote support for portable debuggers and other instrumentation tools.

6. INPUT/OUTPUT

Access controls and security rights will apply to all CAIS facilities required by this section.

The requirements specified in this section pertain to input/output between/among objects (e.g. processes, data entities, communication devices, and storage devices) unless otherwise stated. All facilities specified in the following requirements are to be available to non-privileged processes, unless otherwise specified.

The following definitions pertain specifically to this section:

BLOCK TERMINAL

a terminal that transmits/receives a block of data units at a time.

CONSUMER an entity that is receiving data units via a datapath.

DATA BLOCK a sequence of one or more data units which is treated as an indivisible group by a transmission mechanism.

DATA UNIT a representation of a value of an Ada discrete type.

DATAPATH the mechanism by which data units are transmitted from a producer to a consumer.

DATASTREAM the data units flowing from a producer to a consumer (without regard to the implementing mechanism).

HARDCOPY TERMINAL

a terminal which transmits/receives one data unit at a time and does not have an addressable cursor.

PAGE TERMINAL

a terminal which transmits/receives one data unit at a time and has an addressable cursor.

PRODUCER an entity that is transmitting data units via a datapath.

TERMINAL an interactive input/output device.

TYPE-AHEAD the ability of a producer to transmit data units before the consumer requests the data units

6.1 Virtual I/O Devices: Data Unit Transmission

6.1A Hardcopy Terminals. The CAIS shall provide interfaces for the control of hardcopy terminals.

6.1B Page Terminals. The CAIS shall provide interfaces for the control of page terminals.

6.1C Printers. The CAIS shall provide interfaces for the control of character-imaging printers and bit-map printers.

6.1D Paper Tape Drives. The CAIS shall provide interfaces for the control of paper tape drives.

6.1E Graphics Support. The CAIS shall support the control of interactive graphical input/output devices.

6.1F Telecommunications Support. The CAIS shall support a telecommunications interface for data transmission.

6.2 Virtual I/O Devices: Data Block Transmission

6.2A Block Terminals. The CAIS shall provide interfaces for the control of character-imaging block terminals.

6.2B Tape Drives. The CAIS shall provide interfaces for the control of magnetic tape drives.

6.3 Datapath Control The requirements and criteria in this section pertain to both data unit transmission and block transmission.

6.3A Interface Level. The datapath control facilities of the CAIS shall be provided at a level comparable to that of Ada Reference Manual's File I/O. That is, control of datapaths shall be provided via subprogram calls rather than via the data units transmitted to the device.

6.3B Timeout. The CAIS shall provide facilities to permit timeout on input and output operations.

6.3C Exclusive Access. The CAIS shall provide facilities to obtain exclusive access to a producer/consumer; such exclusive access does not prevent a privileged process from transmitting to the consumer.

6.3D Datastream Redirection. The CAIS shall provide facilities to associate at execution time the producer/consumer of each input/output datastream with a specific device, data entity, or process.

6.3E Datapath Buffer Size. The CAIS shall provide facilities for the specification of the sizes of input/output data path buffers during process execution.

6.3F Datapath Flushing. The CAIS shall provide facilities for the removal of all buffered data from an input/output datapath.

6.3G Output Datapath Processing. The CAIS shall provide facilities to force the output of all data in an output datapath.

6.3H Input/Output Sequencing. The CAIS shall provide facilities to ensure the servicing of input/output requests in the order of their invocation.

6.4 Data Unit Transmission

6.4A Data Unit Size. The CAIS shall provide input/output facilities for communication with devices requiring 5-bit, 7-bit, and 8-bit data units, minimally.

6.4B Raw Input/Output. The CAIS shall provide the ability to transmit/receive data units and sequences of units without modification. (Examples of modification are transformation of units, addition of units, and removal of units).

6.4C Single Data Unit Transmission. The CAIS shall provide facilities for the input/output of single data units. The completion of this operation makes the data unit available to its consumer(s) without requiring another input/output event, including the receipt of a termination or escape sequence, the filling of a buffer, or the invocation of an operation to force input/output.

6.4D Padding. The CAIS shall specify the set of data units and sequences of units (including the null set) which can be added to an input/output datastream. The CAIS shall provide facilities permitting a process to select/query at execution time the subset of data units and sequences of units which may be added (including the null set).

6.4E Filtering. The CAIS shall specify the set of data units and sequences of units (including the null set) which may be filtered from an input or output datastream. The CAIS shall provide facilities permitting a process to select/query at execution time the subset of data units and sequences of units which may be filtered (including the null set).

6.4F Modification. The CAIS shall specify the set of modifications that can occur to data units in an input/output datastream (e.g., mapping from lower case to upper case). The CAIS shall provide facilities permitting a process to select/query at execution time the subset of modifications that may occur (including the null set).

6.4G Input Sampling. The CAIS shall provide facilities to sample an input datapath for available data without having to wait if data are not available.

6.4H Transmission Characteristics. The CAIS shall support control at execution time of host transmission characteristics (e.g., rates, parity, number of bits, half/full duplex).

6.4I Type-Ahead. The CAIS shall provide facilities to disable/enable type-ahead. The CAIS shall provide facilities to indicate whether type-ahead is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable type-ahead in those implementations that do not support type-ahead (e.g., null-effect or exception raised).

6.4J Echoing. The CAIS shall provide facilities to disable/enable echoing of data units to their source. The CAIS shall provide facilities to indicate whether echo-suppression is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable echoing in those implementations that do not support echo-suppression (e.g., null effect or exception raised).

6.4K Control Input Datastream. The CAIS shall provide facilities to designate an input datastream as a control input datastream.

6.4L Control Input Trap. The CAIS shall provide the ability to abort a process by means of trapping a specific data unit or data block in a control input datastream of that process.

6.4M Trap Sequence. The CAIS shall provide facilities to specify/query the data unit or data block that may be trapped. The CAIS shall provide facilities to disable/enable this facility at execution time.

6.4N Data Link Control. The CAIS shall support facilities for the dynamic control of data links, including, at least, self-test, automatic dialing, hang-up, and broken-link handling.

6.5 Data Block Transmission

6.5A Data Block Size. The CAIS shall provide facilities for the specification of the size of a sequence of units during program execution.

6.6 Data Entity Transfer

6.6A Common External Form. The CAIS shall specify a representation on physical media of a set of related data entities (referred to as the Common External Form).

6.6B Transfer. The CAIS shall provide facilities using the Common External Form to support the transfer among CAIS implementations of sets of related data entities such that attributes and relationships are preserved.

6.7 General Input/Output

6.7A Waiting. The CAIS shall cause only the task requesting a synchronous input/output operation to await completion.

6.7B Unsupported Features. The CAIS should provide facilities to control the consequences when the physical device does not have all of the features of the virtual device.

RAC Comment Form

!section:

!RAC version: 13 Sept 1985

!submitter:

!date:

!1-line topic/subject:

!extended comment or recommendation:

!rationale for recommendation:

!disposition by RACWG:

[Send via ARPA/MILNET to POberndorf@ECLB & HMumm@ECLB,
or via U.S. Mail to "Patricia Oberndorf/Hans Mumm,
Code 423, NOSC, San Diego, CA 92152"]

Rationale

for the

DoD
Requirements
and
Design Criteria

for the Common APSE Interface Set (CAIS)

13 September 1985

Draft

Prepared for review by the
KAPSE Interface Team (KIT)
and the
KIT-Industry-Academia (KITIA)

and the
Ada* Joint Program Office
Washington, D.C.

* Ada is a Registered Trademark of the U.S. Government,
Ada Joint Program Office

5.4E Resumption.	5-11
5.5 Monitoring	5-11
5.5A Identify Reference.	5-11
5.5B RTS Independence.	5-12
5.5C Instrumentation.	5-12
6. INPUT/OUTPUT	6-1
6.1 Virtual I/O Devices: Data Unit Transmission	6-2
6.1A Hardcopy Terminals.	6-3
6.1B Page Terminals.	6-3
6.1C Printers.	6-3
6.1D Paper Tape Drives.	6-3
6.1E Graphics Support.	6-4
6.1F Telecommunications Support.	6-4
6.2 Virtual I/O Devices: Data Block Transmission	6-4
6.2A Block Terminals.	6-5
6.2B Tape Drives.	6-5
6.3 Datapath Control	6-6
6.3A Interface Level.	6-6
6.3B Timeout.	6-6
6.3C Exclusive Access.	6-7
6.3D Datastream Redirection.	6-7
6.3E Datapath Buffer Size.	6-7
6.3F Datapath Flushing.	6-8
6.3G Output Datapath Processing.	6-8
6.3H Input/Output Sequencing.	6-8
6.4 Data Unit Transmission	6-9
6.4A Data Unit Size.	6-9
6.4B Raw Input/Output.	6-9
6.4C Single Data Unit Transmission.	6-10
6.4D Padding.	6-10
6.4E Filtering.	6-10
6.4F Modification.	6-11
6.4G Input Sampling.	6-11
6.4H Transmission Characteristics.	6-11
6.4I Type-Ahead.	6-12
6.4J Echoing.	6-12
6.4K Control Input Datastream.	6-12
6.4L Control Input Trap.	6-13
6.4M Trap Sequence.	6-13
6.4N Data Link Control.	6-13
6.5 Data Block Transmission	6-14
6.5A Data Block Size.	6-14
6.6 Data Entity Transfer	6-14
6.6A Common External Form.	6-14
6.6B Transfer.	6-14
6.7 General Input/Output	6-15
6.7A Waiting.	6-15
6.7B Unsupported Features.	6-15

PREFACE

This Rationale document accompanies the document titled "DoD Requirements and Design Criteria for the Common APSE Interface Set" (RAC), dated 13 September 1985.

The purpose of this document is to provide, for each requirement and design criterion in the RAC, any or all of the following as appropriate:

- explanations or clarifications of KIT/KITLA intent
- exposition of alternatives considered and the reasoning for the requirement or design criterion in the RAC
- examples
- identification of known constraints on CAIS specifications and implementations.

The structure (outline) of this Rationale document is identical to that of the RAC document of the same date. Each requirement and design criterion from the RAC appears in *italics* in this Rationale document. Rationale for each is in normal type.

1. INTRODUCTION

1.1 Scope. *This document provides the Department of Defense's requirements and design criteria for the definition and specification of a Common APSE Interface Set (CAIS) for Ada Programming Support Environments (APSEs).*

1.2 Terminology. *Precise and consistent use of terms has been attempted throughout the document.*

Potentially ambiguous terms used in the document are defined in the Glossary of KIT/KITIA Terminology [KK85]. Some definitions tailored to the context of this document are provided in the sections of the document where they are used.

Additionally, the following verbs and verb phrases are used throughout the document to indicate where and to what degree individual constraints apply. Any sentence not containing one of the following verbs or verb phrases is a definition, explanation or comment.

"SHALL" *indicates a requirement on the definition of the CAIS; sometimes "shall" is followed by "provide" or "support," in which cases the following two definitions supersede this one.*

"SHALL PROVIDE" *indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities.*

"SHALL SUPPORT" *indicates a requirement for the CAIS to provide interface(s) with prescribed capabilities or for CAIS definers to demonstrate that the capability may be constructed from CAIS interfaces.*

"SHOULD" *indicates a desired goal but one for which there is no objective test.*

1.3 Relationship to Specifications & Implementations. This document specifies functional capabilities which are to be provided in the semantics of a CAIS specification and are therefore to be provided by conforming CAIS implementations. In general, the specifications of software fulfilling those capabilities (and decisions about including or not including CAIS interfaces for certain capabilities as suggested by the "shall support" definition in the previous section) are delegated to the CAIS definers. If a CAIS implementor determines that it is feasible, then the CAIS implementor may provide a particular specified CAIS facility by reusing other CAIS facilities, thereby achieving a "layered implementation" of the CAIS. Therefore, the realization of a specific CAIS implementation is the result of intentionally divided decision-making authority among 1) this requirements document, 2) CAIS definers, and 3) CAIS implementors.

1.4 Reference Documents.

MILITARY STANDARDS

[Ada89] Reference Manual for the Ada Language, ANSI/MIL-STD-1815A, January 1989.

OTHER GOVERNMENT DOCUMENTS

[Buxton80] "Stoneman" DoD Requirements for Ada Programming Support Environments, February 1980.

[Fisher78] "Steelman" DoD Requirements for High Order Computer Programming Languages, June 1978.

[KK85] Glossary of KIT/KITIA Terminology, draft 1985.

[TCSEC89] Trusted Computer System Evaluation Criteria, CSC-STD-001-89, DoD Computer Security Center, August 15, 1989.

NON-GOVERNMENT DOCUMENTS

[OSCR82] Operating System Command and Response Languages, proposed ANSI standard drafts, 1982.

2. GENERAL DESIGN OBJECTIVES

2.1 Scope of the CAIS. *The CAIS shall provide interfaces sufficient to support the use of APSEs for wide classes of projects throughout their lifecycles and to promote I&T among APSEs.*

2.2 Basic Services. *The CAIS should provide simple-to-use mechanisms for achieving common, simple actions. Features which support needs of less frequently used tools should be given secondary consideration.*

2.3 Implementability. *The CAIS specification shall be machine independent and implementation independent. The CAIS shall be implementable on bare machines and on machines with any of a variety of operating systems. The CAIS shall contain only interfaces which provide facilities which have been demonstrated in existing commercial or military software systems. CAIS features should be chosen to have a simple and efficient implementation in many machines, to avoid execution costs for unneeded generality, and to ensure that unused portions of a CAIS implementation will not add to execution costs of a non-using tool. The measures of the efficiency criterion are, primarily, minimum interactive response time for APSE tools and, secondarily, consumption of resources.*

2.4 Modularity. *Interfaces should be partitioned such that the partitions may be understood independently and they contain no undocumented dependencies between partitions.*

2.5 Extensibility. *The design of the CAIS should facilitate development and use of extensions of the CAIS; i.e., CAIS interfaces should be reusable so that they can be combined to create new interfaces and facilities.*

2.6 Technology Compatibility. *The CAIS shall adopt existing standards where applicable. For example, recognized standards for device characteristics are provided by ANSI, ISO, IEEE, and DoD.*

2.7 Uniformity. *All CAIS features should uniformly address aspects such as status returns, exceptional conditions, parameter types, and options. Different modules within the CAIS should be specified to the same logical level, and a small number of unifying conceptual models should underlie the CAIS.*

2.8 Security. *The CAIS shall provide interfaces to allow tools to operate within a Trusted Computer System (TCS) that meets the Class B3 criteria as defined in [TCSEC89]. Specifically:*

- a. *It shall be possible to implement the CAIS within a TCS.*
- b. *When implemented within a TCS, the CAIS shall support the use of the security facilities provided by the Trusted Computing Base (TCB) to applications programs.*

- c. When not implemented within a TCS, the CAIS interfaces sensitive to security shall operate as a dedicated secure system (i.e., all data at a single security level, and all subjects cleared to at least that level).

This requirement recognises that the CAIS must co-exist within the DoD policy for the handling of classified and sensitive information. The "DoD Trusted Computer System Evaluation Criteria" [TCSEC83] is one of the baseline technical documents that delineate the security requirements for future DoD (and DoD contractor) systems that will handle sensitive data. Ada programming support environments used to develop and maintain mission critical systems clearly fall into this category.

The B3 level was chosen because it is the level that has the most complete set of functional security requirements. This mandate forces the CAIS to accommodate a formal notion of security (specifically, but not exclusively, the DoD security model).

It is important to note that it is not necessary that all conforming implementations meet the B3 level security criteria. However, it is necessary that a system that does meet the B3 level security criteria be able to host a conforming CAIS implementation. [unclear?] Additionally, the CAIS should accommodate implementations that coexist with (without compromising) and operate within a variety of security mechanisms. In other words, a principal intention of these security requirements is to constrain the design of the CAIS in such a manner so as to not technically preclude a B3 TCB as a host, and to permit such a CAIS implementation itself to achieve a B3 certification.

The following is a specific example of a constraint on the specification (design) of the CAIS, without which attaining a B3 certification is impossible:

- The CAIS shall be designed to mediate all tool access to underlying system services.

I.e., "by-passing" the conforming CAIS implementation is unnecessary to implement any APSE function. This should not be interpreted as making it impossible for tool builders to "by pass" the CAIS for underlying system services -- only that any tool builder who wishes to build a more transportable, interoperable tool will find sufficient interfaces in the CAIS to avoid "by passing".

It is inappropriate for the RAC to specifically "list" security features or requirements. There are two reasons:

- a. [TCSEC83] already serves as a complete reference for such guidelines, and
- b. any such list might be incomplete and thus fall short of meeting the intent of supporting M/S-TCB hosts as defined in [TCSEC83].

There are several DoD programming support environments in operation today that accommodate various forms of multi-level modes of processing by recognising security labels on system objects. Some will, to a degree, enforce a mandatory policy. The B3 criteria delineated in [TCSEC83] embody the "best technical" requirements for future DoD APSE hosts.

3. GENERAL SYNTAX AND SEMANTICS

3.1 Syntax

3.1A General Syntax. *The syntax of the CAIS shall be expressed as Ada package specifications. The syntax of the CAIS shall conform to the character set as defined by the Ada standard (section 2.1 of ANSI/MIL-STD-1815A [Ada89]).*

3.1B Uniformity. *The CAIS should employ uniform syntactic conventions and should not provide several notations for the same concept. CAIS syntax issues (including, at least, limits on name lengths, abbreviation styles, other naming conventions, relative ordering of input and output parameters, etc.) should be resolved in a uniform and integrated manner for the whole CAIS.*

3.1C Name Selection. *The CAIS should avoid coining new words (literals or identifiers) and should avoid using words in an unconventional sense. Ada identifiers (names) defined by the CAIS should be natural language words or industry accepted terms whenever possible. The CAIS should define Ada identifiers which are visually distinct and not easily confused (including, at least, that the CAIS should avoid defining two Ada identifiers that are only a 2-character transposition away from being identical). The CAIS should use the same name everywhere in the interface set, and not its possible synonyms, when the same meaning is intended.*

3.1D Pragmatics. *The CAIS should impose only those restrictive rules or constraints required to achieve I&T. CAIS implementors will be required to provide the complete specifications of all syntactic restrictions imposed by their CAIS implementations.*

3.2 Semantics

3.2A General Semantics. *The CAIS shall be completely and unambiguously defined. The specification of semantics should be both precise and understandable. The semantic specification of each CAIS interface shall include a precise statement of assumptions (including execution-time preconditions for calls), effects on global data and packages, and interactions with other interfaces.*

3.2B Responses. *The CAIS shall provide responses for all interface calls, including informative non-null responses (return value or exception) for unsuccessful completions. All responses returned across CAIS interfaces shall be defined in an implementation-independent manner. Every time a CAIS interface is called under the same circumstances, it should return the same response.*

3.2C Exceptions. *The CAIS interfaces shall employ the mechanism of Ada exceptions to report exceptional situations that arise in the execution of CAIS facilities. The CAIS specification shall include exceptions (with visible declarations) for all situations that violate the preconditions specified for the CAIS interfaces. The CAIS specification shall include exceptions (with visible declarations) that cover all violations of implementation-defined restrictions.*

3.2D Consistency. *The description of CAIS semantics should use the same word or phrase everywhere, and not its possible synonyms, when the same meaning is intended.*

3.2E Cohesiveness. *Each CAIS interface should provide only one function.*

3.2F Pragmatics. *The CAIS specification shall enumerate all aspects of the meanings of CAIS interfaces and facilities which must be defined by CAIS implementors. CAIS implementors will be required to provide the complete specifications for these implementation-defined semantics.*

4. ENTITY MANAGEMENT SUPPORT

4. ENTITY MANAGEMENT SUPPORT

The general capabilities of the model specified by the CAIS are the following. The entity-relationship model, for which definitions and requirements follow in 4.1 .. 4.7, provides these capabilities and any alternative model of CAIS requirements must also.

a. There shall be a means for retaining data

b. There shall be a way of retaining relationships among and properties of data.

This section uses the term data in a general (or generic) sense, to set out general requirements on the CAIS for facilities for data. In the following sections, a specific model is used for the structuring of that data in order to specify more specific requirements.

Software projects deal with a great deal of data. Facilities for management of this data are a central feature of the CAIS. The CAIS acts as the repository for all information associated with each project throughout the project life-cycle.

A software project involves many pieces or items of data. Note that there is no intention here to imply any particular granularity of the treatment of data, by the use of the words, pieces or items. These terms are used in a simple colloquial sense. Pieces of data may or may not be composed of other pieces, and so until the ultimate binary digit (bit) of data is reached. Similarly pieces of data might be aggregated into other pieces, and these aggregations may or may not intersect. The structure of data is considered in subsequent sections.

Data might include the text of a piece of program source, or test data, or documentation or a schedule. Data might also be the date a piece of test was created, the name of the author, the permitted areas, or information as to what piece of data some object code was compiled from, or which pieces of data contain other pieces of data.

The data model supports the way that the environment user and tools view project data. It is important that the capabilities provided to do this support a natural expression of the data that closely models the user's understanding of the problem that he is working on. This includes the ability to represent the objects that the user is interested in and to represent the relationships among these objects to describe the dependencies between these objects.

c. There shall be a way of operating upon data, deleting data, and creating new data.

There are many kinds of operations on data - data may be created, changed, executed, written to and used. These terms are used here in a general colloquial sense to illustrate the kinds of things that will be required. More specific operations are considered in terms of the model discussed in the next section.

Specific operation on data, and sets of specific operations, are an important constituent of the CAIS.

d. There shall be a means for defining certain operations and conditions as legal, for enforcing the definitions and for accepting additional definitions of legality.

e. There shall be a means to describe data, and there shall be a means to operate upon such descriptions. Descriptions of the data shall be distinguished from the data described.

An important aspect of data management, which is more widely recognised as a crucial aspect of modern programming languages, is the separation of the structure and rules about data from the data itself. This concept is so widely accepted for programming languages that it is not normally felt necessary to justify it, however some of the main reasons are rehearsed here.

First, data is not normally operated on by only one user, but is operated on by many users. Making its structure, and the rules about the data explicit mean that the several users have a single common understanding about the nature of the data.

Secondly, in any reasonable software project, there will be a large number of different kinds of data and a large number of specific operations. The great majority of the operations will "make sense" from a human viewpoint only when applied to a very small number of the kinds of data. It may not be unduly restrictive to specify that individual operations must apply to a single kind of data. Unfortunately, a user may by mistake, or with malice, request any operation on any piece of data. Thus typos, misspellings, and other slips of the fingers or mind can result in requests to compile a tape drive or sort a progress report. An important goal of the facilities offered by the CAIS interface is to minimise the effects of human fallibility by refusing to perform operations that do not make sense. Note that the specific examples of inappropriate operations given above are only intended as illustrations. It is not intended to imply here that these particular operations either should or should not be allowed. Specific requirements on enforcement of legality are considered later.

The definitions of what is legal in a system originate entirely with the people who design and build it; the system really only enforces their decisions. In most software systems, the rules are complex, contradictory, and largely unknown because they are created as by-products or unforeseen effects of decisions about efficiency, usability, or other concerns. It is a goal of the CAIS to make the rules governing the data and operations of a programming project as explicit and straight-forward as possible, while allowing them to be specified on an individual, company or project level.

Third, there is a need to selectively allow or prohibit certain operations on certain data requested by certain users or processes. Some specific examples of the kind of thing intended are given below, without intending any specific implication that exactly these facilities either should or should not be supplied.

We may consider that Configuration Control includes the ability to prevent change operations on particular data by particular users or processes. That is, it is the ability to "freeze" parts of the developing system such that work can proceed on other parts without needing to be concerned about changes in the frozen parts. For example, a working version of the file manager of a system may be frozen so that applications code that uses the file manager can be written and tested.

Prohibition of operations rarely needs to be universal - applied to all users or processes. It should not generally make it impossible to perform a particular operation if that operation would otherwise be legal; there should always be at least one user/process who can change the data, or at least change the prohibitions or otherwise do some operations that make change possible.

Security includes the ability to prevent change, transformation or use of certain data by certain users or processes. That is, it is the ability to prevent their finding out anything about the data that they do not already know. This may include the fact of the data item's existence as well as the data itself.

Note that a number of different aspects of the structure of, and rules about, data are intentionally considered here as different aspects of a single underlying concept. For example, the idea that a read-integer operation is not allowed on a piece of data which represents an author-name is a simple form of what may be called data typing. This would not be allowed irrespective of the user or process which attempted the operation, and may perhaps in some situations be simply thought of, in terms of Ada, in the syntactical form of a procedure specification. Security, or access control is based on the piece of data to be accessed and the user or process which is accessing it, and is less static than what was called data typing, but is nevertheless not algorithmically complex. The system can provide a fixed set of facilities, which are parameterised, by users/processes as they wish, to implement these controls. Still other aspects of rules may be based on fixed sets of rules, parameterised as needed, while finally for some rules, facilities for specifying the algorithms which constitute the rules may be needed. These facilities are all regarded as different aspects of a single underlying concept, and the way in which they are provided is regarded as a specification (or even implementation) detail of the CAIS.

- f. There shall be a way to develop new data descriptions by inheriting (some of) the properties of existing data descriptions.*

It is desirable to be able to derive new descriptions from existing ones. This results from the observation that there are natural ways in which some items of data are related to others and the conviction that the support mechanisms should conform to this natural "way of the world". While it is possible to develop such new descriptions independently of the existing ones, there are many advantages to providing support for this within the entity management mechanisms. If there is an orderly, supported means for deriving such related descriptions, the process will not be an ad hoc one prone to the errors and problems of ad hoc processes. The ability to inherit properties of the descriptions also will reduce the proliferation of independent relationships and properties in the entity base. Perhaps most importantly, this capability will make it easier for users and projects to tailor the entity system to their own needs and to organise the structures of their data in natural ways.

- g. The relationships and properties of data shall be separate from the existence of the data instances.*

This requirement is to establish the distinction between the abstract concept of a particular item of data, and the particular relationships in which it participates, or the particular values of any of its properties.

As an example, we may have a data instance which represents a particular person. This requirement requires that the particular data instance continues to exist (and therefore to represent the same person) even if particular recorded properties of that person (e.g. his name, age, address, date of birth) are changed, in fact even if every piece of information about the person is changed. Thus the identity and persistence of an item of data is distinguished from, and potentially lasts longer in time than, the particular values of any of its properties or relationships. That is, the values of the relationships and properties may change over time without changing the data item itself.

Another example is that data in a file might be changed without changing the identity of the file (of course in a particular system, other rules such as security, consistency or integrity rules may prevent this).

The requirement says nothing about how the identity of a particular data instance is to be established. One way might be to have pointers or handles within a particular execution of a program which can refer to data instances, and to be able to enquire whether the data instance referred to by one handle is the same as that referred to by another handle. Another way might be to have system assigned unchangeable unique keys for each data instance (these are known in the database world as surrogates). Note that these examples are intended only to aid understanding the concepts intended here, and no conclusions should be drawn from the use of these examples.

h. The descriptions and the instances of data shall be separate from the tools which operate upon them.

This requirement is to ensure that the information and knowledge regarding relationships and properties and their interpretations is controlled within the EMS and is not embedded in tools which are external to the EMS. This is in some ways a further constraint on requirement e. Requirement e states that the descriptions of data should be distinguished from the data, and this says that both the data and the descriptions should be separate from the tools.

All knowledge of the structure of the data items must be retained within the EMS, as opposed to any external tools. It may always be the case that tools may ascribe some additional meaning to a particular property or relationship, but the relationships and properties of general interest are not permitted to be defined strictly in tools which are external to the EMS. This constitutes a decision that all information about the data will be retained and controlled by the EMS.

The following characterization (subsections 4.1 - 4.7) of Entity Management Support (EMS) is based on the STONEMAN requirements for a database, using a model based on the entity-relationship concept. Although a CAIS design meeting these requirements is expected to demonstrate the characteristics and capabilities reflected here, it is not necessary that such a design directly employ this entity-relationship model. The entity-relationship model for which definitions and requirements follow in 4.1 - 4.7 fulfills these requirements (a-h above), and any alternative data model shall fulfill these requirements and shall also fulfill the equivalent of the requirements in subsections 4.1 through 4.7.

In order to state some more specific requirements that the CAIS shall satisfy, it is necessary to specify a particular data model within which those requirements may be expressed.

Note that the term data model is sometimes used in two distinct senses. First, it is used to refer to the way in which data is structured, for example the network model, the hierarchical model, the relational model and the E-R model. Second, it is used to refer to a particular schema to represent the things of interest in the environment, which are modeled within the facilities of a data model of the first sort. The term data model is used in the first sense here.

It is important to note that it is intended that the data model used should only be regarded as a model or notational scheme, and it is not intended that it should constrain the CAIS specification to follow the same model.

The data storage and manipulation needs of a software project can be modeled as an "Entity Management System."

Software projects deal with a great many collections of data, devices, people, and other things which need to be treated as single units; the representations of these in the computer are given the abstract title "entities." A computer-based system for storing, naming, and manipulating entities is called an "Entity Management System."

Much of the CAIS work has had the underlying assumption that the typical flat or hierarchical file system found in a modern computer system is inadequate for the needs of software development projects. This assumption originates in the fact that most projects and companies are forced to supplement the file system's facilities with additional functions, tools, and conventions to be able to do their job. For example, an attempt is often made to give files a "type" that indicates the general form of their contents by establishing a naming convention. Files containing Pascal source code are required to have a name of the form xxxx.pas, where xxxx is the user-meaningful name of the file. It may be that the Pascal compiler will only accept for input files whose name is of this form. The convention reduces by four then number of characters that a user can use to make a file name meaningful (".pas") and is far from foolproof. The RAC makes typing an intrinsic part of the file system, and thus eliminates these and other problems. The knowledge about the structure of the data is controlled within the EMS and is not embedded in the tools or anywhere else.

This section of the RAC discusses the use of an EMS model to present the general requirements of software development and maintenance in this area. It is recognised that use of a model in stating requirements could tend to bias the developers in undesirable ways toward implementations similar to the model. However, after great effort, it was felt that a model-less, bias-free statement of the requirements was beyond our abilities; the resulting statements became so general as to lose all meaning.

4.1 Entities, Relationships, and Attributes

In order to express the requirements for the CAIS, it proved necessary to have a very precise model of the way terms were used.

We make two orthogonal distinctions, leading to four separate concepts.

First, we distinguish between things in the real world, and their representation in the computer.

Second, we distinguish between the abstract concept of a thing and its observed or manipulable information. (This distinction has been known for centuries - for example Aristoteles, *Peri Hermeneias* (de Interpretatione) approx 350 BC, but unfortunately it seems to be necessary for each generation to rediscover it for itself, see Borgida, IEEE Software Vol 2 No 1 pp 63-72)

Considering the four categories in turn, we have first, things in the real world, for example a person named "John Smith". Note that we are concerned here with the abstract concept of this particular person. He may change his name, he may or may not be physically present when we talk about him, but the abstract concept of this particular person is quite separate from the information we may have about him. Similarly, we may have the abstract concept of "the piece of Ada Source code that I typed in at 11am yesterday". Although the Ada source exists in the machine, the concept exists (also) in the real world. Thus we may know many more facts about the real world thing than are actually recorded in the computer (for example, we may know that we had a feeling that the code was wrong when we typed it in, or that we copied it from a particular book, but these facts may not be recorded in the computer). Similarly, we may have a particular project as a real world conceptual thing.

Next, there are facts that we can observe in the real world about these things, such as a particular person's name, age, height, weight etc., or who typed in a particular piece of software.

Next, we have the representation of the abstract concept of a thing in the computer. This is known as an "entity". This may be considered as an instance of an abstract data type. It is quite separate from the data values that may be recorded about it, indeed you can do nothing with such instances, except apply the operations of the type.

Finally, we have the facts recorded in the computer system about entities - these are known as attributes.

Note that relationships are a quite separate concept from the four way distinction made here.

The following definitions, used in this subsection, pertain to the remainder of section 4 also.

ENTITY A representation of a person, place, event or thing.

Entities in a software project may be representations of such things in the real world as people, places, machines, source code, test data, documentation or schedules.

RELATIONSHIP An ordered connection or association among entities. A relationship among N entities (not necessarily distinct) is known as an "N-ary" relationship.

In a software project, "compiled from", "referenced in", or "contains" may be examples of relationships between entities.

The EMS holds relevant information concerning entities and relationships in which the users of the environment is interested. A complete description of the real-world item represented by an entity or relationship might not be recorded within the EMS. It is impossible (and, perhaps, unnecessary) to record every potentially available piece of information about entities and relationships. From now on, we shall consider only the entities and relationships (and the information concerning them) which are to enter into the design of the data held by the EMS.

ATTRIBUTE An association of an entity or relationship with an elementary value.

The information about an entity or a relationship is obtained by observation or measurement and is expressed by associations with values. "3", "red", "Johnson", "3.14159", and


```
"declare
  x:  INTEGER
begin
  x:  =0
end"
```

are all examples of values.

ELEMENTARY VALUE *One of two kinds of representations of data: interpreted and uninterpreted.*

INTERPRETED DATA *data representation whose structure is controlled by EMS facilities and may be used in the EMS operations. Examples are representations of integer, string, real, date, and enumeration data, and aggregates of such data.*

UNINTERPRETED DATA *A data representation whose structure is not controlled by EMS facilities and whose structure is not used in the EMS operations. Examples might be representations of files such as requirements documents, program source code, and program object code.*

Regarding UNINTERPRETED DATA: This is a special category where we permit(admit) data structure knowledge to be maintained outside of EMS.

4.1A Data.

The EMS shall provide facilities for representing data using entities, attributes, or binary relationships. The EMS may provide facilities for more general N-ary relationships, but it is not required to do so.

A software project involves many kinds of entities, relationships of entities, and attributes of entities and relationships, all of which must be stored and made available. For example, program source, test data, documentation, and schedules may all be represented by entities; date created, author, storage format, and access allowed may be attributes of an entity; "compiled from," "referenced in," and "contains" may be relationships between entities; and "date compiled" may be an attribute of a relationship. In current projects, attributes and relationships of a file are often embedded in its contents or name, and thus are vulnerable to editors and renaming utilities. For example, projects often require that the names of programmers who have modified a source file be kept as comments near its beginning. An EMS-based system could keep them as attributes or, if programmers are represented by entities in the system, as relationships. This would be both more convenient and less vulnerable.

Attributes are "modifiers," "descriptors," or "explainers" of an entity or relationship. Examples are the date an entity or relationship was created, the name of the creator, or the reason it was created. Attributes have a name and a value; values are numbers, names (enumeration variables), or character sequences, or lists thereof. (However, see the discussion of "interpreted" and "uninterpreted" data in 4.1A below.)

Relationships are "connections" or "associations" from one entity to another. Examples are source code to object code, old to new revision of a document, and owner user to owned entity. Relationships may be functional mappings (one-to-one or many-to-one) or relational mappings (one-to-many or many-to-many) and may have attributes.

Deleting an entity also deletes all of those attributes that belong to it. The definition of a relationship as an ordered connection or association among entities automatically implies that when an entity is deleted, the relationship ceases to exist, as the entities which were associated are no longer present.

At least some entities represent those collections of data that we normally think of as "files" or "data sets." They may consist of multiple records in a given format or of undifferentiated sequences of characters or bits. Examples are source files, test results, cross-references, and schedules. These entities have "contents" and may have attributes and relationships. Other entities may represent hardware devices, either actual or virtual, groupings of entities for purposes like naming, users of the system, and units of execution ("jobs" or "processes").

This paragraph uses the notion of "contents," which is not properly part of the RAC. It suggests that "files" are a "property" of entities in a different way to attributes. In contrast, the RAC uses the concept of "attribute" to represent all values: the source text of a program is held as an attribute whose elementary value may be uninterpreted data, while a date may be held in an attribute whose elementary value is interpreted data.

4.1B Elementary Values

The EMS shall provide facilities for representing data as elementary values.

"Interpreted data" might be thought of as "system data" and "uninterpreted" as "user data." The differentiation is made to point out that the former kind is "looked at" by the EMS and the latter is not, but instead is accepted, stored, and supplied from/to user programs without interpretation. Uninterpreted data might also be called "bulk data" or "contents;" an entity that has an attribute whose value is uninterpreted data is similar in many ways to a file in an operating system. Note, however, that an entity can have any number of such attributes. Also, an entity could be the representation of an I/O device or running process which accepts input as uninterpreted data written to one (or more) of its attributes or produces output to be read from an attribute. Current systems often use separate mechanisms to resolve the names of files, devices, processes, and users; in effect, the four kinds of things are in separate "name spaces." The RAC Consistency requirement (2.7) suggests that these and other kinds of things should all be handled by a single underlying mechanism, that of the EMS.

4.1C System Integrity.

The EMS facilities shall ensure the integrity of the EMS-managed data.

This requirement is in some ways a restatement of the terse requirements:

- d. There shall be a means for defining certain operations and conditions as legal, for enforcing the definitions and for accepting additional definitions of legality.*
- e. There shall be a means to describe data, and there shall be a means to operate upon such descriptions.*

The requirement for data integrity must include recognition of the fact that systems do fail unexpectedly, due to hardware and software faults. The EMS must be implementable on current (c. 1985) hosts; therefore it is unacceptable to specify fault-tolerant or multiply-redundant hardware. The EMS design must actively support recovery from unexpected failures.

There are several basic approaches to this problem: modern data base managers usually implement the "transaction," a unit of work that appears, from the viewpoint of the rest of the system and of unexpected faults, to happen "all at once." This is sometimes implemented by a journaling system, in which the data base is recorded at a given time and subsequent transactions are recorded in a journal. After a crash, the data base is restored from the recording and the transactions from the journal are re-applied to it.

Modern operating systems often build a certain amount of redundancy into their data structures and include a "scavenger" program that can scan the structure after a crash and use the redundancy to correct any inconsistencies in it. This is sometimes formalized as a system of "truths" and "hints," in which the truths are handled in such a way that they cannot be made inconsistent by a crash. Hints are used by the OS in its normal operation, for instance to speed access, but are always checked against the truths. When a hint and truth conflict, the hint is discarded. A scavenger program can recreate the hint system from the truths.

The CAIS has aspects of both data base and operating system, and both of the above approaches are possible implementations. In fact, they are quite similar at a basic level, differing mostly in the terminology used. The CAIS contractor is not constrained to adopt either approach. However, data integrity is a very important requirement, so much so that it may "set the tone" for the entire CAIS design.

4.2 Typing

The following definition, used in this subsection, pertains to the remainder of section 4 also.

TYPING *An organization of entities, relationships, and attributes in which they are partitioned into sets, called entity types, relationship types, and attribute types, according to designated type definitions.*

Typing is a powerful mechanism for detecting and preventing user error.

Entities have a type; entity types might, for example, determine the types and number of each type of attributes and relationships required of an entity of that type.

Attributes have a type; attribute types might, for example, determine the form, format, number, and range of values required of attributes of that type.

Relationships have a type; relationship types might, for example, determine the types and number of the entities participating in relationships of that type and the types and number of each type of attribute required of a relationship of that type.

In the abstract, an entity type defines the set of components required of entities if they are to be of that type. To differentiate and allow access to those components, each must have a name; the names are also part of the type.

For example, type "test-plan" might specify an attribute named "plan-text" of type "string(*)" (multiple strings), an attribute named "creation-date" of type "date", a relationship named "creator" of type "user" (i.e. pointing to an entity of type "user"), and a relationship named "progress" of type "reports(*)". (An actual, usable type would probably have many more components.) All entities of type "test-plan" would have to have these components.

4.2A Types.

The facilities provided by the EMS shall enforce typing by providing that all operations conform to the type definitions. Every entity, relationship, and attribute shall have one and only one type.

The type of a thing cannot be discussed usefully in isolation from the functions that access the thing. These functions EMBODY the type definition; without them, the concept of type is of interest only to philosophers.

Typing is necessary to:

- interpret the bit-string containing a value in order to do operations on it;
- convert between representations of the same value on different hardware. This will be necessary if the distributed EMS supports mixed models of CPUs and for transport of information from one EMS to another.
- restrict operations — establish discipline and structure. This concept is often called "integrity."

At the time we wrote the RAC, there appeared to be two models: (1) every object has exactly one type and types are arranged in a lattice or (2) objects could be of several types. We concluded that the requirements we wanted to express could be expressed in the two models in equivalent ways, but that it was easier to express them in the first model; thus this requirement. The CAIS designers may adopt either model.

4.2B Rules about Type Definitions.

The EMS type definitions shall

- *specify the entity types and relationship types to which each attribute type may apply.*
- *specify the type or types of entities that each relationship type may connect and the attribute types allowed for each relationship type.*
- *specify the set of allowable elementary values for each attribute type.*
- *specify the relationship types and attribute types for each entity type.*
- *permit relationship types that represent either functional mappings (one-to-one or many-to-one) or relational mappings (one-to-many or many-to-many).*
- *permit multiple distinct relationships among the same entities.*
- *impose a lattice structure on the types which includes inheritance of attributes, attribute value ranges (possibly restricted) relationships, and allowed operations.*

Note that the last bullet above is a major departure from the kind of typing found in the Ada language.

It is clear that: (1) we must have extensibility and (2) users and projects will want to particularize the data structures to their needs. Users need to be able to define new types that are extensions of existing types, such that operations and tools that expect entities of the old (base) type will also accept and work correctly on entities of the new type. The new type must therefore have the same attributes and relationships that the base does, plus any additional ones. Attribute types may have more restricted ranges in the new type than they did in the base, but obviously may not be less restricted or different in any other way. The tool or operation should not need any kind of recompilation or other preparation to operate on the new type. A type mechanism that works in this way is said to have a "lattice structure."

It should also be possible to merge extensions such that any tools that operate on the base type and either of the (first level) extensions will also work on the new (second level) extension type. For example, if A is the base type and B and C are both extensions of it, it should be possible to define a D which is an extension of both B and C. A tool that accepts A would also accept B, C, and D; a tool that accepts B would also accept D; and a tool which accepts C would also accept D. There may be conditions under which D cannot be formed, such as B and C having an attribute with the same name and different types.

To give a concrete example, there might be an object type named "ProgressReport" that has a text attribute in a certain format, a set of attributes, and a set of relationships. Some number of tools may exist that manipulate these objects to produce summaries, update Pert charts, and so forth. A manager named Krutar may want to add a couple of attributes, for data that he's particularly interested in. He should be able to define a new type named KrutarPR such that objects of that type can still be

processed by the existing tools and by new tools that need the new attributes. These new tools would not necessarily be able to process objects of type `ProgressReport`.

Another manager, Munck, might want his progress reports to participate in relationships to all program source and documentation mentioned in the text, so that he can look at it easily while reading the report. He could define a type named `MunckPR` that participates in those relationships.

Finally, because progress reports are generally kept for the life of a project and managers are not, the day may come when manager Hart must take over for Krutar and Munck. He may build new tools that operate on a progress report type that has both Krutar's additional attributes and Munck's additional relationships, called `HartPR`. Moreover, some of his tools may have to handle old objects of type `MunckPR` or `KrutarPR`, or even `ProgressReport`.

4.2C Type Definition.

The EMS shall provide facilities for defining new entity, relationship, and attribute types.

There are no explicit requirements for:

- naming of type definitions
- storage of type definitions in a single collection
- restrictions on the creation of new definitions

All decisions in this area are intentionally left up to the CAIS designers.

4.2D Changing Type Definitions.

The EMS shall provide facilities for changing type definitions. These facilities must be controlled such that data integrity is maintained.

A user or project may determine that the type definition of a number of existing entities is inadequate for his purposes and may wish to add new attributes and/or relationships to existing entities. An integrity clause is a reminder to the CAIS designer that there may exist entities of the old type at the time of the change. E.g., if a new attribute is added, all existing entities of the type will now have the type and probably it must appear as though these now have the attribute. Also, consider the `DELETION` of components of a type. This is being intentionally left to the CAIS designer. The EMS must support the user in making such a change and in updating the data accordingly.

A software engineering environment must support the evolution of the kinds and organisation of information that pertain to projects, to support the integration of new tools, and to accommodate changing project needs. The ability to change the type definitions of the data has a number of implications on the EMS support for data. For example, one may want to delete, add or modify an entity attribute. Deleting an attribute in an entity type will result in the deletion of that attribute in all instances of that type. Similarly, adding an attribute will result in increasing the storage for instances of that type with the possibility of initialising the storage for that attribute. Modifying an attribute (i.e. changing a range constraint) will require that all instances be checked to ensure that the values of these instances conform to these new constraints.

The ability to define new types through derivation is not sufficient to support the need to change type definitions in that one may want to modify existing types that are known to a set of tools without modifying the tools that use the type.

Type changes also includes the ability to redefine the structural characteristics of the data. For example, one may want to change a relationship from being optional to mandatory in which case the EMS must ensure that all entities that can be related by this relationship in fact are related.

Another consideration of type modification is changing the set of entity attributes (keys) that are used for identification. For example, one may change the key that uniquely identifies instances of an entity from 2 attributes to 3 attributes. The EMS must ensure that all instances conform to this change. This change will potentially impact applications that use these entities.

4.2E Triggering.

The EMS shall provide a conditional triggering mechanism so that prespecified procedures or operations (such as special validation techniques employing multiple attribute value checking) may be invoked whenever values of indicated attributes change. The EMS shall provide facilities for defining such triggers and the operations or procedures which are to be invoked.

One of the objectives of a software engineering environment is to provide a context for the greater integration of the management functions with the technical functions. One of the great weaknesses in the support systems which are in use today is that managerial and technical functions and activities are largely treated separately, even though they have an intimate relationship with one another. The existence of all of the project data together in an entity management system provides the opportunity to unify the technical events with the managerial ones and thus to provide yet another avenue for improving the quality of the systems that are produced and the ability to manage and control the development of those systems in such a way as to keep them within budget and schedule.

A key to providing this sort of unification of managerial and technical functions and activities is to support a link within the entity management system between events in one arena and those in the other. One motivation behind the requirement for triggering, then, is to provide a mechanism by which the occurrence of a pre-defined technical event will result in some corresponding, desired action and/or notification in the managerial realm. Of course, such mechanisms also have application within one of these arenas, such as when one programmer takes an action of which another one should be aware. In

general, triggering mechanisms are a means to provide increased frequency and accuracy of the sort of internal project communication on which projects depend for success.

4.3 Identification

The following definitions, used in this subsection, pertain to all the rest of section 4 also.

EXACT IDENTITY A designation of an entity (or relationship) that is always associated with the entity (or relationship) that it designates. This exact identity will always designate exactly the same entity (or relationship), and it cannot be changed.

IDENTIFICATION A means of specifying the entities, relationships and attributes to be operated on by a designated operation.

In order to carry out operations on representations of entities, or representations of relationships, or on attributes, it is necessary to specify the entity, relationship, or attributes (or sets of these) to be operated upon, and to specify the operation to be performed. The entities, relationships, and attributes to be operated on are specified by a process known as identification.

4.3A Exact Identities.

The EMS shall provide exact identities for all entities. The EMS shall support exact identities for all relationships. The exact identity shall be unique within an instance of a CAIS implementation, and the EMS shall support a mechanism for the utilization of exact identities across all CAIS implementations.

The EMS shall provide unforgeable names that cannot be separated from the exact entity that they name.

It must be possible to name an entity such that use of that name will always access exactly that same entity or will fail if the entity no longer exists. This facility, combined with the ability to prevent change of an entity, makes possible implementation of a convenient configuration management system on top of the EMS. Current systems sometimes use absolute disk addresses or checksums of the contents of a file to implement an exact identity.

The exact name of an entity is not necessarily the same as the user-specified name. For example, in a system that implements file revisioning, the user-specified name might identify the latest revision; the exact name would specify a given revision, not necessarily the latest. In a system that uses relationship structure for user naming, it would probably be desirable to be able to delete an entity and later create another at the same "place" in the structure, which would therefore have the same user name. Use of the

deleted entity's exact name must fail in that case. An exact identity is not necessarily human-readable; however, the user must be able to specify that the exact identity be determined and used or stored when s/he enters the user name. For example, to specify the pieces of a new baseline, the user would enter user names, which would be transformed into exact names and stored. On the other hand, the specification of a particular test procedure would store the user names, so that the current revisions of the entities will be used when the procedure is run. It may be desirable to be able to transform exact names into user names, for example when listing a baseline.

An EMS that meets the distribution or reliability (backup and redundancy) requirements (Section 1.8) may need to store multiple copies of an entity. It would then need to be able to use the same exact name for each copy, which makes it necessary that an entity cannot be changed and retain the same exact name. The same arguments apply to entities which may be copied or moved from one instance of the EMS to another.

As Ada begins to fulfill its purpose of making software portable and reusable, there will be increased movement of software between projects, contracts, organizations, and services. For example, some services will begin to build libraries of Ada packages that are generally useful in some application area. This will introduce the need to identify pieces of software uniquely on a service- or DoD-wide basis. To meet this need, the service or DoD will have to administer the assignment of exact names, such that at least some of them can be unique across all EMSs. A current example is the need to identify exactly the constituents of a compiler that has passed validation.

4.3B Identification.

The EMS shall provide identification of all entities, attributes and relationships. The EMS shall provide identification of all entities by their exact identity. The EMS shall support identification of all relationships by their exact identity.

The EMS shall provide for naming of individual and sets of entities, attributes, and relationships.

The user must also be able to manipulate attributes and relationships, in addition to entities. Both attributes and relationships must therefore have user names, but do not necessarily have to have exact names. In fact, it may be that entities have only exact names and attributes and relationships have only user names. The user name of an entity could be the user name of a parent entity plus the name of the child relationship that points to it.

4.3C Identification Methods.

The EMS shall provide identification of entities and relationships by at least the following methods:

- identification of some "start" entity(s), the specification of some relationship type and the specification of some predicate involving attributes or attribute types associated with that relationship type or with some entity type. This method shall identify those entities which are related to the identified start entity(s) by relationships of the given relationship type and for which the predicate is true. Subject to the security constraints of section 2.8, all relationships and entities shall be capable of identification via this method, and all attributes and attribute types (except uninterpreted data) shall be permitted in the predicates.
- identification of an entity type or relationship type and specification of some predicate on the value of any attribute of the entity type or relationship type. This method shall identify those entities or relationships of the given type for which the predicate is true. Subject to the security constraints of section 2.8, all attributes (except uninterpreted data) shall be permitted in the predicates.

A major function of the EMS is to provide a Name Space for the entities that it manages.

Human users of the EMS need to be able to refer to entities and operations by using names that have some mnemonic or connotative meaning to them. The EMS establishes a domain or region over which names are applied or valid, which may in fact span several physical computer systems. There may be syntactic rules that determine the validity of a name.

Because there is only a limited number of meaningful names and users tend to duplicate each other's choices in naming their own entities, the name space must have some internal organization or structuring by which names are mapped to entities. That is, a user must be able to specify a local area of the complete name space into which his names are mapped. Another user in another local area would then be able to use the same names for another set of entities.

The EMS shall support human-usable, exact, and flexible naming of its contents.

- "Human-usable" naming of items in the EMS is vital to the productivity of its users.
- "Exact" naming is vital to the successful implementation of strong control mechanisms such as configuration control.
- "Flexible" naming is vital to the adaptability of the EMS to a particular organization's needs.

These three criteria are apt to conflict with each other. NOTE: there is no significant difference in meaning between the words "name" and "identity" as used here and in the RAC.

The problem with the use of the term "name" and the reason why this term is avoided in the RAC is the implication that the concept is associated with a character string. In contrast, the terms "identity" or "identification" are used to imply that selection of something may depend not only on a character string (for example), but also on the context in which it is used. There is an analogy here with block structured languages; a name, for example "I", may be used in different places, and may refer unambiguously to different things, depending on the scope rules etc.

The names by which users refer to items in the EMS shall be easy to remember and easy to type.

Choice of the form of user-specified names should be biased heavily toward user convenience, at the possible expense of additional processing needs in the EMS. "User-friendly" features such as lengths on

the order of a full line, mixed case that is retained but not significant, and embedded blanks in composite names should be considered. The typical user will deal with a great many entities in the course of a day's work; s/he must be able to assign names that are both meaningful and easy to recall.

The EMS may provide particular entity types, attributes and relationships whose purpose is the support of user naming. For example, the common concept of a system of tree-structured directories and files could be implemented by directory entities and child relationships from them to other directories or to files.

The form of user-specified names is not necessarily a consideration of the CAIS; there need not be a direct mapping between how names are specified by the user to tools and how those tools specify them to the EMS interface. For example, a vanilla tree-structure implementation might have user-specified names in UNIX form: a top-down list of node-names separated by slashes. The corresponding EMS interface might specify names to be variable-length arrays of simple name strings. The tools built on this EMS would have to parse the former format and map it into the latter.

If the CAIS designers choose to specify the external form of names, they should try to avoid restrictions based on pre-conceptions of the form of the user interface. For example, the idea of embedded blanks in names is horrifying to most people because of the problems involved in parsing a command string. HOWEVER, in a menu-selection or icon-based user interface, these problems do not arise. Names are usually pointed to, not typed; when they are typed, it is in response to a prompt for a name, to be terminated by a CR that makes it possible to isolate the name despite embedded blanks. (There was an experimental system that allowed users to store and retrieve files by placing them on imaginary shelves in the (physical) space around them. Naming was by pointing with a finger.)

Users need to be able to name groups of things, such as "all files in directory x." As with the form of user names, it is not clear if this need will be fulfilled by the EMS interface or by higher-level support.

The general requirements for human useable and flexible identification have been set out above. In terms of the specific data model, two methods for identification of representations of entities and relationships are specifically required here.

First, the EMS must provide navigation from one entity to another, along a relationship connecting the two. This identifies either the other entity involved in the relationship (i.e. the entity at the other end of the relationship), or the relationship itself. A given entity may be involved in a number of relationships, each one of which potentially involves a different related entity. The EMS must therefore provide the means to specify start entities, relationships, and predicates.

Second, no matter what naming scheme is selected, an alternative scheme must be provided that supports selection of items from the EMS based on relationship names, attribute names, attribute values, or the existence of particular relationships or attributes. The naming scheme will probably be based on the "main" or most-used way that entities are retrieved; the alternative will support lesser-used retrievals. For example, a software metric tool might need to retrieve "all source files." This requirement is a generalisation of the "wild card" feature of many current command processors.

4.4 Operations

Specific, named operations on entities, attributes, and relationships, and sets of specific operations, are an important constituent of an EMS. A major function of the EMS is to regulate the interaction of entities and operations, such that a defined set of rules is not violated.

Operations may themselves be represented in the EMS by "code" or "runnable module" entities containing their executable code or may be "built in" to the EMS or a command interpreter. It should be easy for managers and programmers to add new tools in the form of entity operations. The CAIS designers may choose either to provide functionality in the EMS for defining and enforcing interaction rules for user-written operations, or may structure the system so that the operations enforce their own rules. The latter choice, while probably providing the more flexible and easier to implement approach, must deal with the problem of "untrustworthy tools."

4.4A Entity Operations.

The EMS shall provide facilities to:

- *create entities*
- *delete entities*
- *examine entities (by examining their attributes and relationships)*
- *modify entities (by modifying their attributes)*
- *identify entities (as specified in Section 4.3)*

4.4B Relationship Operations.

The EMS shall provide facilities to:

- *create relationships*
- *delete relationships*
- *examine relationships (by examining their attributes)*
- *modify relationships (by modifying their attributes)*

- *identify relationships (as specified in Section 4.3)*

Many of the attributes and relationships of entities important to software projects are associated with particular operations. Examples: the relationship between a source file and the entity file that it was transformed into; the date on which the entity file was created. Also, operations are generally done at the request of a particular person, or agent, whose identity is an important datum. The agent may have reasons for requesting the operation which are appropriate for capture.

4.4C Attribute Operations.

The EMS shall provide facilities to:

- *examine attributes*
- *modify attributes*

4.4D Exact Identity Operations.

The EMS shall provide facilities to:

- *pass exact identities between processes*
- *compare exact identities*

4.4E Uninterpreted Data Operations.

The EMS shall provide that use of the input-output facilities of the Ada language (as defined in Chapter 14 of ANSI/MIL-STD-1815A) results in reading/writing an uninterpreted data attribute of an entity. The facilities of Section 6 shall then apply.

4.4F Synchronisation.

The EMS shall provide dynamic access synchronization mechanisms to individual entities, relationships and attributes.

The EMS must be able to isolate sets of entities from each other, both logically and physically.

A software project will have many agents active simultaneously; to allow them to work productively and without mutual interference, it must be possible to isolate the collections of entities they're working with. This need is similar to that for configuration management, above.

There is a need to be able to prevent change operations on EMS items with a "temporary flavor."

An agent who has the access rights to change some particular part of the EMS data needs to be able to "lock out" changes by other agents with similar rights, so that s/he can perform a series of operations without interference. This is different from the change protection discussed above in that it must be legal for an agent who has only the right to change the data, not necessarily the right to change other agent's access rights to the data.

A lock operation must take as its operand an "area" or "neighborhood" of the EMS structure. For example, in a tree-structured EMS, it should be able to lock an entire subtree. If agents were forced to lock all of the components serially, deadlock would be unavoidable.

Locks are said to be "owned" by the agent that requests them; if the agent terminates, the locks must be cancelled.

4.5 Transaction.

The following definition, used in this subsection, pertains to all the rest of section 4 also:

TRANSACTION *A grouping of operations, including a designated sequence of operations, which requires that either all of the designated operations are applied or none are; e.g., a transaction is uninterruptible from the user's point of view.*

4.5A Transaction Mechanism.

The EMS shall support a transaction mechanism. The effect of running transactions concurrently shall be as if the concurrent transactions were run serially.

4.5B Transaction Control.

The EMS shall support facilities to start, end and abort transactions. When a transaction is aborted, all effects of the designated sequence of operations shall be as if the sequence were never started.

4.5C System Failure.

System failure while a transaction is in progress shall cause the effects of the designated sequence of operations to be as if the sequence were never started.

[Rationale TBD]

4.6 History.

The following definitions, used in this subsection, pertain to all the rest of section 4 also:

HISTORY *A recording of the manner in which entities, relationships and attribute values were produced and of all information which was relevant in the production of those entities, relationships or attribute values.*

The History mechanism discussed in this section is not necessarily different from a particular use of the

basic entity, attribute, and relationship facilities provided by the EMS. The history of entities that are important in some sense to the development project may be recorded in special relationships, attributes, and entities defined for that purpose. It is not intended that the recording of history be either mandatory or all-inclusive. The management of a particular project should be able to decide which entities will have their history recorded and whether that recording is to be mandatory.

4.6A History Mechanism.

The EMS shall support a mechanism for collecting and utilizing history. The history mechanism shall provide sufficient information to support comprehensive configuration control.

4.6B History Integrity.

The EMS shall support mechanisms for ensuring the fidelity of the history.

[Rationale TBD]

4.7 Robustness and Restoration.

The following definitions, used in this subsection, pertain to all the rest of section 4 also:

BACKUP *A redundant copy of some subset of the EMS-managed data. The subset is capable of restoration to active use by a EMS implementation, particularly in the event of a loss of completeness or integrity in the data in use by implementation.*

ARCHIVE *A subset of the EMS-managed data that has been relegated to backing storage media while retaining the integrity, consistency and availability of all information in the entity management system.*

4.7A Robustness and Restoration.

The EMS shall support facilities which ensure the robustness of and ability to restore EMS-managed

data. The facilities shall include at least those required to support the backup and archiving capabilities provided by modern operating systems.

Projects tend to have amounts of data that are large in terms of the capacity of existing storage units. Moreover, the need for reliability makes it necessary to keep several redundant copies of all data. The EMS must be able to divide the data that constitutes its entities among several different storage units. The economics of the cost of a storage unit vs. its speed make it necessary for the EMS to support several different kinds of storage if it is to provide reasonable access at acceptable cost, and to move entities easily from one medium to another.

Any but the smallest of software projects cannot be dependent on the continued operation of any single piece of equipment. In fact, most DoD projects are much too large to use a single host, and must be spread over several for capacity reasons as well as reliability. It cannot be expected that a project can be broken into isolated, single-host-sized pieces and run on individual EMSs on separate CPUs.

The CAIS shall provide a single, coherent Entity Management System that runs in distributed fashion on an arbitrary number of host machines. The capacity of an instance of the EMS shall be limited only by the capacity of the hardware available to it.

***** orphan text. needs a home *****

The EMS definition will probably include several elementary data types such as integer, string, and date, to be used for attribute values that are interpreted in various ways by EMS facilities.

If there are going to be several different APSEs, it will probably be necessary to define a single "transport format" of the primitive types, similar to the IGES standard for CAD/CAM or the NAPLPS standard for graphics. To move an entity from one APSE to another, it is necessary to be able to translate all data items from their internal format on one machine to that on another. If only the underlying data types defined by the EMS are used, only that relatively small set needs to be translated. To provide transportability of entities among some number of different APSEs, the CAIS must define a "Transport Format" for all underlying data types and each APSE must be able to translate its internal formats into and from the Transport Format. {Note: the "transport format" for tools and code packages is the Ada language!}

For example, one possible Transport Format for floating-point numbers is the IEEE standard. As more machines come to use it as their internal format, translation becomes less and less a burden. HOWEVER, if some of the machines that might be used as hosts cannot handle the full range or accuracy of the IEEE standard, either there will be restrictions on transport to those machines or the Transport Format must have a smaller range. If the range acceptable to all potential hosts is too small to be usable, the IEEE standard may not be usable.

The set of media to be used for transport of entities must also be considered by the CAIS, in order to fulfill the general I&T requirements. Most modern communications and storage systems will accept arbitrary bit strings, but there are still many in use that accept only subsets or variants of ASCII characters. If the CAIS is intended to support those, all Transport Formats must be specified in terms of the union of all subsets of ASCII.

Uninterpreted data, by definition, cannot necessarily be transported with only the facilities of the EMS. Any necessary translation will have to be provided by the user. It is conceivable that the CAIS could

define a few elementary types for uninterpreted data and provide automatic translation for attributes that are flagged somehow as being of those types. For example, the underlying type of "7-bit character" would support ASCII/EBCDIC translations when transporting between DEC and IBM machines.

.....

[The following is not explicit in the RAC:]

Type definition must be permitted on at least four levels:

CAIS-specified: to insure transportability of the contents of an EMS from one APSE implementation to another, there must be a set of "primitive" types defined as part of the CAIS. A new APSE must be able to translate values of these types from their representations on all previous APSEs to its representations.

APSE-specified: a particular APSE (implementation of the CAIS) will have its own types for its own purposes. These may be something as simple as the CAIS primitives with names in French rather than English. Another possibility is a graphics-oriented APSE that has various picture primitives, icons, and pointer values.

Installation-specified: each particular instance of an APSE (or EMS) will be used to implement a unique set of configuration control and management procedures, design and coding methodologies, testing and QA standards, and other aspects of the local "corporate culture." The installation must be able to define appropriate types to support this customizing.

User specified: each user, in setting up his/her own local environment, command procedures, and other tools, will want to define new types. It is conceivable that this would be prohibited, that there would be no mechanism for user type definition. However, because the user has full freedom to define new types in the application code s/he's producing, not being able to do so in command procedures would probably be seen as a severe restriction.

Type redefinition and overloading in Ada follow various scoping rules that do not necessarily apply to EMS types. For example, should a user be able to redefine a type defined by the installation? Are all installation and user type definitions kept in a single collection, thus insuring that there cannot be two different types having the same name? This would be a tremendous problem of implementation and administration. Are these concerns RAC or CAIS decisions?

A general philosophy of the APSE is that installations will be able to build an environment that suits their needs on the APSE. For example, a small software house in Southern California might feel it necessary to keep the astrological sign under which an entity is created.

5. PROGRAM EXECUTION FACILITIES

Access controls and security rights will apply to all CAIS facilities required by this section.

The following definitions pertain specifically to this section:

- PROCESS** *the CAIS facility used to represent the execution of any program.*
- PROGRAM** *a set of compilation units, one of which is a subprogram called the "main program." Execution of the program consists of execution of the main program, which may invoke subprograms declared in the compilation units of the program.*
- RESOURCE** *any capacity which must be scheduled, assigned, or controlled by the operating system to assure consistent and non-conflicting usage by programs under execution. Examples of resources include: CPU time, memory space (actuals and virtual), and shared facilities (variables, devices, spoolers, etc.).*
- ACTIVATE** *to create a CAIS process. The activation of a program binds that program to its execution environment, which are the resources required to support the process's execution, and includes the program to be executed. The activation of a process marks the earliest point in time which that process can be referenced as an entity within the CAIS environment.*
- TERMINATE** *to stop the execution of a process such that it cannot be resumed.*
- DEACTIVATE** *to remove a terminated process so that it may no longer be referenced within the CAIS environment.*
- SUSPEND** *to stop the execution of a process such that it can resumed. In the context of an Ada program being executed, this implies the suspension of all tasks, and the prevention of the activation of any task until the process is resumed. It specifically does not imply the release of any resources which a process has assigned to it, or which it has acquired, to support its execution.*
- RESUME** *to resume the execution of a suspended process.*
- TASK WAIT** *delay of the execution of a task within a process until a CAIS service requested by this task has been performed. Other tasks in the same process are not delayed.*

An Ada Programming Support Environment (APSE) provides the software engineering capabilities for developing Ada software. These capabilities are available to the user through a comprehensive set of transportable software development tools. The ability to extend the tool set by adding new tools, combining existing tools, and reusing tool components promises to lead to the cost effective automation of the life-cycle support for Ada software.

A fundamental requirement of the Common APSE Interface Set (CAIS) is to provide program execution facilities through which software development tools can be executed. These facilities support the composition, execution, and testing of tools that reuse executable programs in order to economically increase tool functionality. Furthermore, there is a desire to increase the longevity of APSEs through the

construction of tools that exploit execution environments that are hosted on distributed and heterogeneous systems.

The notion of building tools from smaller components is consistent with contemporary software engineering practice that advocates precise functional decomposition of complex programs into separate simpler programs. In addition, when the programs are independently compiled and linked, the separate program name space affords the opportunity for distributing the execution of the programs that comprise the tool. For example, the distribution of an APSE in an environment consisting of an interconnection of mainframes and workstations may be used to separate tool functionality to provide a more responsive user interface. In this instance, the composition of a tool would require interfaces that support the cooperation among code that executes on different computers. This composition may not be readily achieved without such interfaces.

The use of the Ada task model for tool composition may unnecessarily constrain the type of tool distribution useful to tool builders from both a practical and implementation perspective. The orientation of the model to support mission-critical applications in static execution environments can be too rigid for the program cooperation anticipated for the dynamic execution environment in which tools are hosted. Moreover, it is unlikely that the implementations of the Ada Run-Time System will support distributed execution of a program unless specifically mandated for a mission-critical target.

At a minimum, program execution facilities must enable one Ada program to call (invoke) the execution of another Ada program. This requirement is stipulated in the Stoneman document and is a requirement of the two Government sponsored Minimal APSEs, the ALS and ACS. The ability to concurrently execute another program provides greater utility to the tool builder than many existing interfaces that are used to accomplish tool composition through the concatenation of program executions.

The overall objective is to provide a minimal set of services that support synchronous and asynchronous modes of program execution. Furthermore, attaining this objective should not compromise the effective use of all processing resources in order to achieve parallel/distributed execution, or efficient serialized interleaved execution.

Finally, the ability to exploit tool synergism in order to compose complex tools from simpler tools has motivated that specific requirements for creating processes, controlling processes, and communicating among processes be specified in order to promote a comprehensive process execution model within an APSE. These requirements are the basis for services that can be provided within the CAIS to promote the straightforward construction of tools and toolsets beyond that provided by conventional command procedures. The requirements are specified to allow for maximum innovation in the CAIS design and to minimize the prejudicial influence of existing tool interface designs and implementations.

5.1 Activation of Program

In the context of the CAIS, and of program support environments generally, a program is a relatively static entity that may exist in various phases, from source to executable image, in the normal progression from inception to execution. It is implicit that program execution denotes any of the states that comprise a program's execution chronology, e.g., ready, blocked, waiting, etc. Therefore, the term process is used to denote an abstraction that represents the sequence of states that comprise the execution of a program.

Activation of a program may be viewed as an instantiation of an executable image of a program together with the establishment of the control information and resources that are necessary for its execution. Specific properties of a process are not elaborated, e.g., loci of control and address space, since this level of detail is superfluous to understanding the requirement. It is recognised, however, that for a distributed execution environment, an important capability is to normalise differences within the execution environment for the APSE and between the APSE and any target environment to which it is connected. Consequently, for most tools, the actual processing location of program execution is expected to be transparent in the denotation for a CAIS process and is consequently not explicitly required by the program execution facilities. This transparency enhances further APSE tool transportability over a variety of different computer architectures.

5.1A Activation. *The CAIS shall provide a facility for a process to create a process for a program that has been made ready for execution. This event is called activation.*

Activation refers to the creation of a process from an Ada main program that is in a format for execution. Following creation the process is ready for execution. The assumption is made that an executable image for the program has been prepared using the language processor tools either prior to or as a part of activation.

The principal rationale for program activation is derived from the requirement to provide a fundamental capability for one process to invoke the execution of another program, i.e., to create a process in a well-defined way that enhances the construction and transportability of APSE tool sets. This is consistent with the Stoneman statement that "the KAPSE shall provide a mechanism whereby it shall be possible for one APSE tool to invoke another APSE tool and supply the invoked tool with parameters". Thus, the potential transportability of the Command Language Processor tool among CAIS conforming APSEs is increased, allowing, for example, a compiler tool set to be constructed through a command procedure that invokes the various phases (programs) that comprise the compiler. Furthermore, it implies the existence of a relationship among concurrent processes and that, following activation, a process includes the notion of all resources explicitly and implicitly required to support the execution of the specified program including ancillary procedures or resources, such as those necessary to support program debugging and monitoring.

The requirements present the view of an Ada program as a single executable entity rather than a main program enclosing a collection of dependent tasks. Thus, when programs share processing resources, scheduling of the resource may be performed at the process level. The requirements, however, do not explicitly exclude a unified process/task execution model in those instances where it may be beneficial to coordinate the CAIS design with the Ada Run-Time System.

It is important to separate the requirement for program activation from requirements that may exist to support the fragmentation or distribution for a program's execution. The latter requirements are typically satisfied by the overlay and run-time functionality necessary to manage a program's use of the addressing and processing capacity of the execution environment. This should be transparent when writing source code for a transportable tool and is more properly considered to be a requirement for the language processor tools, viz., compiler and linker. Consequently, this is intentionally absent from the scope of the requirements for the CAIS.

5.1B Unambiguous Identification. *The CAIS shall provide facilities for the unambiguous identification of a process at any time between its activation and deactivation; one such capability shall be as an indivisible part of activation. This act of identification establishes a reference to that process. Once such a reference is established, that reference will refer to the same process until the reference is dissolved. A reference is always dissolved upon termination of the process that established the reference. A terminated process may not be deactivated while there are references to that process.*

After process creation, subsequent communication and synchronisation with the process depends upon the ability to reference it. The CAIS must, therefore, provide an unambiguous name through which each process can be referenced as the target of some process-related interface. This name refers to the same process until the name is rendered obsolete when the process becomes unreferenceable. Unambiguous, rather than unique, is used to avoid the connotation that the name of a process must be unique over space and time, i.e., it cannot be reused.

Three significant events are identified in the process life-cycle: activation, termination, and deactivation. Activation has been previously defined as the event that creates a process. Termination is defined as the event that precludes a process from becoming a candidate for further processing resources; the process ceases execution but remains a referenceable entity through the CAIS. Deactivation is defined as the event that erases all information about a process that is maintained by the CAIS, including the process name and termination related data. Consequently, the name of the process may be reused following deactivation providing that it does not compromise the integrity of the CAIS implementation.

The model assumed for referencing one process by another process is analogous to that used for referencing files in a contemporary file system. It is recommended that a compliant CAIS design provide a unified reference model for all CAIS objects. Implicit in this recommendation is the expectation that the process name provides efficient and secure reference to a process, and that it eliminates the need to revalidate access rights and privileges each time the process is referenced. Consequently, a process may not be deactivated until no further references are possible. This establishes the space/time boundary for process accessibility through the CAIS.

While there are no specific requirements on the way in which process identification is to be established, a scheme for process identification must be specified and its resultant impact on other CAIS facilities defined. One provision of a compliant scheme is that program activation and process identification are indivisible when program activation is completed successfully.

A CAIS design that guarantees uniqueness of names after deactivation, while not required, is compliant with the requirement.

5.1C Activation Data. *The CAIS shall provide a facility to make data available to a program upon its activation.*

The facility to make data available following activation of a program is similar to the presence of in-mode parameters that are included in the specification of an Ada main program. The requirement is intended to provide a straightforward capability for the activator of a program to optionally pass data to the process that results from the activation.

While it is expected that the rules for parameter conformance and passing will remain consistent with the rules in the Ada Reference Manual, restrictions on activation data are not prohibited in order to support a dynamic environment that is efficient and implementable. For example, a CAIS design may legitimately restrict activation data to a single subtype. It is likely that such restrictions would be compatible with any Appendix F parameter requirements for a compiler implementation; however, no additional functionality should be assumed from the Ada Run-Time System to support activation data. The rationale for the requirement is to ensure that a practical facility for parameterized activation of a program exists that is analogous to the call of an Ada procedure.

5.1D Dependent Activation. *The CAIS shall provide a facility for the activation of programs that depend upon the activating process for their existence.*

A typical requirement of many tools is for a process to have a well-defined relationship with the process that invoked its activation, i.e., the creating or parent process. There are two distinct motivations for requiring this relationship. First, there should be some notion of accountability within the CAIS. Since processes consume processing resources, there should not be uncontrolled access to them. Therefore, a capability is needed to authenticate the rights through which a process derives its processing resources. A dependency relation on the parent process provides a measure of accountability for such authentication, i.e., inherited rights. A second motivation is to simplify the termination of related processes. When a process is terminated explicitly by another process, it should optionally result in all processes created by the terminated process to become terminated. Thus, the process that requested the termination is not required to individually terminate all processes created by the process to be terminated. As a result of this, the requirement stipulates that a facility must be provided whereby the existence of a process depends upon the existence of its parent.

A straightforward dependency relationship for these requirements is one that implies a hierarchical process structure that originates from the process creating the processes executed by the tool or tool set. This permits the resources to be inherited and reclaimed in a well-established and accepted fashion. While a strict hierarchical relationship among processes is not mandated by the requirement, it should be possible for a parent process to optionally request, when activating a program, dependent process termination when it terminates. Consequently, a process should be provided a facility to create a dependent subordinate process, viz., a subprocess.

Originally, there was an explicit requirement for adherence to a strict dependency hierarchy for all process execution. This requirement is relaxed, since it is recognized that there is the practical need for a capability to create an independent process.

5.1E Independent Activation. *The CAIS shall provide a facility for the activation of programs that do not depend upon the activating process for their existence.*

Certain categories of APSE tools are expected to execute independently of the tool that requested program activation. Consequently, there is a specific requirement for a process to continue execution after its parent has terminated. For example, tools that execute in a background execution environment are usually independent of the invoking process, viz., spooler-type tools.

Independent activation does not preclude the need for accountability of a process. However, the means through which this accounting is achieved is not required to be specified by the CAIS since there is no apparent benefit that accrues to tool transportability.

5.2 Termination The rationale for the termination requirements is based upon the need to provide complementary capabilities to those specified for program activation, i.e., the CAIS must permit the removal of processes from the execution environment.

5.2A Termination. *The CAIS shall provide a facility for a process to terminate a process. There shall be two forms of termination; the voluntary termination of a process (termed completion) and the abnormal termination of a process. Completion of a process is always self-determined, whereas abnormal termination may be initiated by other processes.*

A process may terminate any process that it can reference subject to any access restrictions on this process. In particular, a process may terminate itself, i.e., self-determined termination.

An important aspect of self-determined termination is that it is both orderly and voluntary and is therefore called process completion. Conversely, termination of a process by another process is not synchronised with the process to be terminated and is not necessarily orderly; this is called abnormal termination.

An abnormal termination capability is required when aberrant process execution is detected or in the event that unhandled exceptions are visible outside the process in which they were raised. A useful distinction is recognised between completion and abnormal termination regarding the availability of information to other processes, vis., the parent process. When a process completes it is expected that the precise reason for termination will be provided by the terminated process. However, when a process is abnormally terminated this information may not be available. This distinction is used to guide the requirement stipulated in 5.2C for termination data.

5.2B Termination of Dependent Processes. *The CAIS shall support clear, consistent rules defining the termination behavior of processes dependent on a terminating process.*

The rationale for requiring clear and consistent rules that define the termination behaviour of processes that are dependent on the terminating process is based upon the need to safeguard against nonproductive process execution. In particular, when a parent process terminates abnormally, any dependent processes that are servicing the parent should be prevented from redundant execution.

The original requirement had specifically stipulated that when a parent process was terminated, either normally or abnormally, its dependent processes should be automatically terminated. This requirement is relaxed since it was argued that while a CAIS design might choose to enforce termination closure, it is not essential for tool transportability. However, what is essential for tool transportability are rules that govern the termination behaviour of dependent processes.

While it is not precluded by the requirement, a rule similar to that for normal Ada task termination, where a master task may not terminate until all dependent tasks have terminated, is unnecessary for normal process termination given the fundamental differences between a process and a task.

5.2C Termination Data. *The CAIS shall provide a facility for termination data to be made available. This data shall provide at least an indication of success or failure for processes that complete. For processes that terminate abnormally the termination data shall indicate abnormal termination.*

The rationale for the termination data requirement is motivated by the need to maintain logically consistent process execution among cooperating processes. For example, when a process is created to perform a service for another process, not necessarily the parent process, data indicating the success or failure of the service should be made available to accommodate asynchronous normal or abnormal termination. Furthermore, this data should remain accessible until the terminated process is deactivated.

The precise information that comprises termination data is not specified, other than it must, at a minimum, permit the recording of successful or failed execution. It is expected that a CAIS design will provide for additional information.

When a process terminates abnormally it cannot guarantee the accuracy of the termination data. Under these circumstances, the CAIS is required to ensure that the termination data includes an indication of abnormal termination.

5.3 Communication

The construction of tools that exploit the capability to distribute functionality as separate processes requires that a means for dynamic communication among processes be available. Supporting only communication that occurs as a result of process activation or termination is insufficient to provide the necessary synergism to implement cooperating processes within a tool. Consequently, there is a requirement for the exchange data among processes.

5.3A Data Exchange. *The CAIS shall provide a facility for the exchange of data among processes.*

Specific facilities must be provided for exchanging data among processes. The generality of the requirement is deliberate in order to avoid suggesting a particular design and to encourage an effective approach for interprocess communication.

For example, the following model is one potential approach that satisfies the requirement. In this approach, the exchange of data is specified using an abstract type resource that provides a means for passing data among processes having access to the resource. The resource is accessible through a symmetric interface that facilitates both the synchronous and asynchronous exchange of data. The identification and attributes of the resource is consistent with other CAIS objects and is independent of message synchronicity and the multiplicity of data exchanges.

The semantics of the interprocess communication facilities are expected to be functionally compatible with those specified for contemporary message passing models. While it is not required that the Ada language rendezvous semantics be specified for exchanging data, it is expected that the facilities are sufficiently comprehensive that the tool builder may construct a rendezvous style interprocess communication capability. The principal reasons for not requiring that interprocess communication be functionally equivalent to the task rendezvous style of communication are: a desire for consistency and orthogonality in the facilities that are available to the tool builder, and the fundamentally different execution contexts of a process and a task.

Requiring that interprocess communication provide identical features to the Ada rendezvous model

imposes the restriction that interprocess communication follow rules that may be unwarranted for communication among processes which are outside the semantics of the language. For example, the selective terminate requires that process termination adhere to the rules for Ada dependent tasks. This clearly compromises process termination orthogonality, for the sake of consistency of inter and intraprocess communication, since process termination should be independent of communication requirements. Conversely, relaxing the requirement for equivalent functionality compromises consistency by specifying similar communication facilities that are governed by different rules, thereby presenting a threat of confusing the tool builder. In addition, programs, rather than tasks, are likely to be choices for software distribution among separate computers. Consequently, the CAIS design should not be expected to overcome the complexities resulting from distributed rendezvous. For example, determining if a terminate alternative can be selected would require a substantial amount of computer coordination.

[TBS - Examples are required for this section to demonstrate how process communication may be effectively used by the tool builder. The proposed model is presented as basis for this demonstration of utility. The construction of an AI-like tool for this demonstration has been recommended.]

5.4 Synchronisation

A facility to synchronise process execution is required. Synchronisation is specified separately from the requirement for interprocess communication to emphasise the functional difference between communication and synchronisation.

A conforming CAIS design may include process synchronisation as a result of interprocess communication; however, specific facilities are expected to be available for process synchronisation. While it is mandatory that task execution semantics are not compromised by the process execution model, it is recognised that the strict separation of process and task management may incur unacceptable process execution behavior. Therefore, a coordinated execution model for process and task synchronisation may be appropriate but it should not require extensive increased support from the Ada Run-Time System.

5.4A Task Waiting. *The CAIS shall support task waiting.*

This requirement specifically stipulates that if a CAIS operation cannot be completed directly (as defined by the CAIS design), any resulting delay should be limited to the task enclosing the call to the CAIS operation.

A less precise requirement that stipulated that the process not be delayed was avoided since it would require that all CAIS calls complete directly. It is recognized that this requirement challenges the CAIS designer to achieve a delicate balance in satisfying the requirement while safeguarding the semantics of task execution. A corollary of this requirement is that all CAIS facilities should state explicitly when

execution blocking results from a call to a CAIS facility. Failure to specify this information may adversely effect tool transportability.

5.4B Parallel Execution. *The CAIS shall provide for the parallel execution of processes.*

This requirement specifically stipulates that processes within the CAIS may execute in parallel except when serialized execution is required by the semantics of the CAIS facility. A CAIS design that forces serialised process execution is unacceptable unless serialisation is necessitated by insufficient processing resources that require two or more processes to share a single instruction stream processor. In this event, the CAIS must provide interleaved (logically parallel) process execution that fairly allocates processing resources among processes.

5.4C Synchronisation. *The CAIS shall provide a facility for the synchronization of cooperating processes.*

A specific facility must be provided for synchronising the execution of processes. The generality of the requirement is deliberate in order to avoid suggesting a particular design and to encourage an innovative approach to process synchronisation.

Typically, the facility should provide for establishing synchronised execution points among cooperating processes. For example, a process may wish to coordinate its continued execution with some event; this event may occur as a result of some explicit action, provided through the CAIS, by another process.

5.4D Suspension. *The CAIS shall provide a facility for suspending a process.*

A specific facility must be provided for one process to stop the execution of another process such that execution of the stopped process may be continued. Activities currently being performed by the CAIS or the host OS on behalf of the process may be continued until intervention of the suspended process is required. Resources that may have been temporarily assigned, viz., processor(s) and physical memory may be released as a consequence of suspended execution.

In the context of an executing Ada program, this requires the suspension of all tasks that are executing; a facility not directly available in the Ada language. When the process is executing in a distributed environment tasks may be suspended individually. However, suspension of a process must not change the behavior of task execution within the process unless the change is a direct consequence of the task not being in an executable state, e.g, failure to honor an interrupt. Therefore, suspension of a process may result in failure of tasks that depend upon real-time interactions and delays.

5.4E Resumption. *The CAIS shall provide a facility to resume a process that has been suspended.*

A specific facility must be provided for a process to request that a suspended process resume the use of processing resources. As a result, all tasks suspended in a process are made available for execution simultaneously. It is acceptable to restrict resumption of a process to the process that suspended it.

The resumption of a process should not result in rescheduling of task execution within a process; i.e., the same set of tasks should continue execution following resumption of the process unless rescheduling is necessitated by the occurrence of a time-dependent event, e.g., completion of an external task blocking operation. Consequently, the semantics of the Ada task model are safeguarded by ensuring that rescheduling only occurs for events that would occur if the process had not been suspended and then resumed.

5.5 Monitoring [TBS - Introductory text.]

5.5A Identify Reference. *The CAIS shall provide a facility for a process to determine an unambiguous identity of a process and to reference that process using that identity.*

A specific facility must be provided for referencing a process. This requirement augments the requirement for the unambiguous identification of a process when it is created by requiring support for referencing the identification of a process. Therefore, once a process has been created, its identification may be used to support other program execution facilities, i.e., to designate the target process for an operation. A facility for a process to determine its own identity should be provided.

5.5B RTS Independence. *CAIS program execution facilities shall be designed to require no additional functionality in the Ada Run-Time System (RTS) from that provided by Ada semantics. Consequently, the implementation of the Ada RTS shall be independent of the CAIS.*

This requirement constrains the design of the CAIS to be implementable without support from the Ada Run-Time System (RTS). However, it is expected that the requirement may be overly optimistic when CAIS functionality cannot be implemented efficiently unless there is support available from the RTS, e.g., to support the Instrumentation interfaces of 5.5C.

The motivation for this requirement originates from a need to clearly delineate RTS interfaces for Ada program transportability from CAIS interfaces for tool transportability. Early requirements did not distinguish between the two levels of transportability and were a source of some confusion. When the distinction was clearly established, a recommendation was rejected that proposed the inclusion of program transportability interfaces to the RTS as CAIS requirements and design criteria. The reasons cited for excluding RTS interfaces included the concern that rehostability of CAIS interfaces might be compromised by RTS interfaces that were not readily retargetable, and that compiler developers would be unreceptive to RTS requirements that were not explicit in the Ada Reference Manual. Therefore, this requirement reduces dependencies upon potentially nonretargetable interfaces and unreasonable constraints upon compiler implementations.

It is recognised that there is an apparent conflict in the requirements when considering other interface requirements that seem to rely on additional RTS functionality, e.g., task blocking when required by a CAIS operation. This conflict can be expected to grow if a single program's execution is distributed on separate computers and the distribution is to remain transparent to the CAIS implementation. Consequently, future revisions to the requirements may ameliorate the conflict.

5.5C Instrumentation. *The CAIS shall provide a facility for a process to inspect and modify the execution environment of another process. This facility is intended to promote support for portable debuggers and other instrumentation tools.*

Specific facilities must be provided to support transportable tools for debugging and performance monitoring. In particular, a process must be able to inspect and modify the execution environment of a target process. It is desirable that this facility should not necessitate the addition of special code in the target process nor any additional compilation or linking directives.

It is important to recognise that the instrumentation requirements are specified to accommodate an

execution environment in which the target process is executing on a separate, and potentially different, computer from the process using the instrumentation interface. Consistent with the other CAIS interface requirements, the distribution of the execution environment is transparent to the tool builder.

6. INPUT/OUTPUT

Access controls and security rights will apply to all CAIS facilities required by this section.

The requirements specified in this section pertain to input/output between/among objects (e.g. processes, data entities, communication devices, and storage devices) unless otherwise stated. All facilities specified in the following requirements are to be available to non-privileged processes, unless otherwise specified.

The following definitions pertain specifically to this section:

BLOCK TERMINAL

a terminal that transmits/receives a block of data units at a time.

CONSUMER *an entity that is receiving data units via a datapath.*

DATA BLOCK *a sequence of one or more data units which is treated as an indivisible group by a transmission mechanism.*

DATA UNIT *a representation of a value of an Ada discrete type.*

DATAPATH *the mechanism by which data units are transmitted from a producer to a consumer.*

DATASTREAM *the data units flowing from a producer to a consumer (without regard to the implementing mechanism).*

HARDCOPY TERMINAL

a terminal which transmits/receives one data unit at a time and does not have an addressable cursor.

PAGE TERMINAL

a terminal which transmits/receives one data unit at a time and has an addressable cursor.

PRODUCER *an entity that is transmitting data units via a datapath.*

TERMINAL *an interactive input/output device.*

TYPE-AHEAD *the ability of a producer to transmit data units before the consumer requests the data units*

The level of these requirements is determined by a number of factors. There are too many terminal

types, and probably always will be, to list them all. It is undesirable to have a capabilities file accessed directly by a tool, since the tool would then need to change for each new terminal. It is likewise undesirable to have a high-level interface including prompting protocols, since this can be implemented on top of the CAIS, and can have a methodological impact. Portable tools cannot use all of the capabilities of powerful terminals, since they could then not port to other terminals. When the hardware is part of the tool, the unfiltered interfaces can be used to write a higher interface. Emerging technologies, such as color, bit-mapped graphics, and color can be handled in this way until standard interfaces evolve.

The word privilege is used in an informal sense to denote the absence of generally applied restrictions, and is not intended to require two classes of processes. It is almost universal that some processes operate with more capabilities than others. A non-privileged process is therefore one which does not have exceptional privileges. The privileged process in 6.3C is any process having the privilege to interrupt in this manner.

The devices listed in these requirements must be supported by the CAIS interfaces, but no implementation of the CAIS should be required to provide all of these devices. When such devices are added to a CAIS installation, the interfaces should be made available.

An alternative to the present approach is to provide generic device drivers, in the manner of UNIX and ioctl. This was felt to be inappropriate, since such devices are at too low a level to provide the clarity of the interfaces below. The CAIS may, however, specify such general interfaces and layer the specifics on top. If this is done, the layering must produce complete and portable interfaces as required, so that the same control mechanisms can be universally used.

The choice of a hardcopy terminal as the least common denominator is not intended to imply that such terminals are in standard use in software development. It merely provides an interface so that new or foreign terminals can talk to an APSE at some level of capability.

The cursor addressing requirement for the second class of terminals is necessary for the programs which use most terminal time in development: debuggers, editors, system status monitors, and databases. These programs are the core of the user interface.

The block terminals are little known in many circles, but are widely used, particularly where large numbers of users access a single computer.

The other devices are the external media for which standard interfaces can be based on current usage, and for which current Department of Defense need is established.

6.1 Virtual I/O Devices: Data Unit Transmission

6.1A Hardcopy Terminals. *The CAIS shall provide interfaces for the control of hardcopy terminals.*

Some terminals which are not truly hardcopy may be incapable of transmitting at the character level, deferring all transmission to lines or other units. Where these do not fit into the larger class of block terminal requirements, they can be treated as hardcopy terminals. Almost any terminal can be treated as a hardcopy terminal.

A cardreader and punch can be handled as a hardcopy terminal. PROM burners can be handled by this interface, although a higher level interface to a virtual PROM burner can be written on top of this.

6.1B Page Terminals. *The CAIS shall provide interfaces for the control of page terminals.*

This requirement is aimed at keystroke transmission and an addressable cursor. The term "page terminal" is not intended to imply page transmission, and is not intended to preclude scrolling.

The alternative to this is to ignore the keystrokes and deal with lines and blocks. This class of terminals is needed because of 6.4C, where further discussion can be found.

6.1C Printers. *The CAIS shall provide interfaces for the control of character-imaging printers and bit-map printers.*

This is meant to include all classes of printers.

6.1D Paper Tape Drives. *The CAIS shall provide interfaces for the control of paper tape drives.*

If this were absent, a low-technology medium which continues to be in wide use would not be supported. and an APSE might be incapable of supporting current ways of doing business. Paper tape is generally used for host/target communication.

The support of punched cards is in a similar category, but it is possible to use the hardcopy terminal interface to a cardreader/punch. Cards are generally punched with normal characters, while paper tape is more generally used for binary data and human-readable hole patterns, which cannot be accessed in this way.

6.1E Graphics Support. *The CAIS shall support the control of interactive graphical input/output devices.*

There is no existing standard that has universal acceptance as the definition of graphic capabilities. Consequently, the requirement for a specific interface is an issue requiring the sort of investigation and consideration expected of the CAIS designer. The imminent ISO/ANSI Graphics Standards for GKS and CG-VDI should be considered as candidates.

This requirement is not specific to terminals, but refers instead to all devices. The terminal interfaces described below are not intended to support graphics directly. The choice between interactive and batch graphics is not specifically specified, but interaction is the current standard access. The decision as to whether bit-mapped or line graphics are to be supported is let to the CAIS designer. "Graphical input device" refers to pointing/locating devices; raster imaging is not required.

6.1F Telecommunications Support. *The CAIS shall support a telecommunications interface for data transmission.*

Telecommunications support should be based on existing standards, but it is part of the CAIS design to decide which standards are appropriate. Since this is an external interface, the semantic specification of the telecommunications interface must completely describe the external interface.

6.2 Virtual I/O Devices: Data Block Transmission

6.2A Block Terminals. *The CAIS shall provide interfaces for the control of character-imaging block terminals.*

Block terminals behave as local editing stations for small blocks of data. The size of the block is a characteristic of the protocol, and is not necessarily constant.

The alternative to this requirement is to class such terminals as hardcopy devices. Ignoring the special features of these terminals would make these features inaccessible to portable APSE tools. Some of the features are essential to effective data entry systems.

This is an incomplete requirement deferred to the CAIS designer. Existing terminal capabilities should be considered in designing the interfaces, so that they can control the important features of the significant terminals in this class.

6.2B Tape Drives. *The CAIS shall provide interfaces for the control of magnetic tape drives.*

Magnetic tapes are expected to be the principal means of transporting data between APSES, and as such are external to an APSE. Many applications generate tapes which are not to be used by the APSE, but are specific magnetic patterns to be read by other systems (targets). It is thus necessary to view the tape drive as an I/O device. Tape communications between APSEs may be limited to the common external form.

There is an ANSI standard for tape formats, but there are also testimonies of problems in using ANSI tapes to communicate. This interface is external, so the semantics must include the exact format of the tape.

Hard disks, on the other hand, can be considered to be an implementation mechanism for the features required by Section 4. They need not be considered I/O devices, since they neither enter nor leave an APSE.

Floppy diskettes are more likely to be used as a transmission device, and deserve their own requirement. However, no accepted standard exists, and any interface would be arbitrary.

6.3 Datapath Control *The requirements and criteria in this section pertain to both data unit transmission and block transmission.*

6.3A Interface Level. *The datapath control facilities of the CAIS shall be provided at a level comparable to that of Ada Reference Manual's File I/O. That is, control of datapaths shall be provided via subprogram calls rather than via the data units transmitted to the device.*

Many devices permit escape sequences in their datastream to control their behavior.

It is possible to specify these escape sequences and their consequences, selecting CONTROL_L to create a new page, ESCAPE-[-w to change the color of a terminal, etc.

Since devices vary, it is unwise to take this approach. It would become necessary for the CAIS implementation to filter every transmission character and find the escape sequences, replacing them with the ones required by the device. By having the functions called as functions, it is possible to let the text go through unfiltered and still keep control of the device.

This does not prevent an implementation from filtering some characters, as described in 6.4E. Nor does it prohibit the transmission of escape sequences which control device. It only provides access to these behaviors in device-independent ways.

6.3B Timeout. *The CAIS shall provide facilities to permit timeout on input and output operations.*

Many input/output operations involve waiting for some completion or acknowledgement. When this is not forthcoming, a program can wait forever.

This requirement can be bypassed by writing a task which waits some time interval and checks for the completion of the basic action. This mechanism may be inefficient, and can usually be replaced with an efficient one if the driver is aware of the desire to time out.

This requirement affects all input/output interfaces, in that they must provide some mechanism for the control of timeout situations. If 5.4A cannot be met by an implementation of the CAIS, the interfaces for timeout are necessary to achieve some desirable effects.

6.3C Exclusive Access. *The CAIS shall provide facilities to obtain exclusive access to a producer/consumer; such exclusive access does not prevent a privileged process from transmitting to the consumer.*

A program should be able to own a device, to prevent interleaving by other processes. System shutdown messages should not be prevented from reaching such a device, as they do not represent normal interference. An example of this is the ownership of the printer by the print spooler/symbiont/daemon to prevent interleaved output under normal circumstances. This is also of use in implementing a windowing terminal interface.

The concept of a privileged process is discussed at the beginning of the rationale for section 6.

6.3D Datastream Redirection. *The CAIS shall provide facilities to associate at execution time the producer/consumer of each input/output datastream with a specific device, data entity, or process.*

This requires the CAIS to provide a means to allocate to a Virtual Device (essentially to a device type) a specific hardware device to implement an instance of the required functionality without requiring a program to include actual device and datapath values.

6.3E Datapath Buffer Size. *The CAIS shall provide facilities for the specification of the sizes of input/output data path buffers during process execution.*

There are interactive systems which buffer hundreds of characters for output and dump them at intervals sometimes measured in minutes. Without this control, it becomes necessary to force the processing of these buffers explicitly at many points in a program. In other instances, the default buffer size may be less than the size of a frequent large message, resulting in shortsighted buffer overflows. This also provides a means of permitting the programmer to shrink buffers when he can enhance efficiency by knowing details about his data.

The buffer which cannot hold one block may lose data. The buffer which is too large may hoard needed data, interfering with timeliness. Any incorrectly sized buffer can interfere with efficiency.

6.3F Datapath Flushing. *The CAIS shall provide facilities for the removal of all buffered data from an input/output datapath.*

In a reset after an exception, pending data often becomes irrelevant. A user of an APSE tool may request the suppression of the output he is seeing (e.g. with control-O), and it becomes necessary to prevent the output of what may be an immense buffer. A consumer may refuse to consume, and the buffer must be eliminated to permit the process to terminate. The disposition of the data in the buffer is of no consequence.

Note the difference between flushing (6.3F) and processing (6.3G) a buffer.

6.3G Output Datapath Processing. *The CAIS shall provide facilities to force the output of all data in an output datapath.*

The word flushing can be used in two senses. One sense is the concept used here: to force the processing of pending data units gathered in a buffer. The second sense is used in 6.3E: to discard the contents of such a buffer.

The data is sent despite the buffer not being full. This facility is often useful in debugging, when a paused process has done output which cannot yet be examined. It can also be used when it becomes clear that the buffer will not be soon filled. This may also be used when a datastream is redirected.

6.3H Input/Output Sequencing. *The CAIS shall provide facilities to ensure the servicing of input/output requests in the order of their invocation.*

It seems apparent to a programmer that if he calls two subroutines and the first outputs an A, the second a B, the A should come out first. The Ada language does not define input/output at a sufficient level of detail to ensure this. It is particularly important that prompts come out before the response is read. This has been a major area of difficulty in the Pascal language.

There is debate over the level at which such sequencing should be required. Within a single datapath, it is most likely to be compromised when separate tasks use that datapath. Sequencing could also be required between datapaths accessing the same device or different devices, or between datapaths used by different processes, even different processors. The appropriate level of control is left to the CAIS specifiers.

Most of this functionality can be provided by setting the buffer size to zero, but this can have terrible

ence consequences. In the absence of a buffer, output must happen when its command is . Compiler optimisations are still able to change the order of execution of output commands.

t/output of
consumer(s)
pe sequence,

requirement places constraints on the runtime support of the Ada language, and thus on the is of the language itself. This constraint is in conflict with requirement 5.5B. The constraint is nsequence of a particular CAIS design, but is rather a characteristic of a reasonable development

uce with the

manner on
at interfaces
t in systems
editors will
rocessor and

Data Unit Transmission

Data Unit Size. The CAIS shall provide input/output facilities for communication with equiring 5-bit, 7-bit, and 8-bit data units, minimally.

are the normal ASCII transmission widths, as suggested by the use of ASCII in the Ada language. width of the Baudot teletype code.

cluding the
le facilities
ses of units

n may allow
as separate
ured.

Raw Input/Output. The CAIS shall provide the ability to transmit/receive data units and s of units without modification. (Examples of modification are transformation of units, of units, and removal of units).

drivers suppress nulls, check line lengths, and otherwise alter the data they pass. This requires that ration be suppressable.

devices consider every byte to be data, and can receive values as the ordinal positions of bytes in ie sequences (a null byte means zero). In order to talk to these devices it is necessary to send them

cluding the
de facilities
ses of units

ntrol certain
very unit of

6.4F Modification. *The CAIS shall specify the set of modifications that can occur to data units in an input/output datastream (e.g., mapping from lower case to upper case). The CAIS shall provide facilities permitting a process to select/query at execution time the subset of modifications that may occur (including the null set).*

Many devices do not support lower case, and must have the transformation performed outside to operate correctly. Others need every unit of information as transmitted, and may not treat it as a character.

6.4G Input Sampling. *The CAIS shall provide facilities to sample an input datapath for available data without having to wait if data are not available.*

A debugger may run the target processor until the user types a command. The program can set up a task to read the command and run the processor until the command is read, safe in the guarantee of 6.7A. A communications program may wait for input from a message source, updating some factor periodically if no message is received. This can also be done with a pair of tasks. In both of these cases, the tasking model is conceptually different from the model of non-waiting polling. This requirement permits the design of tools to be flexible in choosing either model.

6.4H Transmission Characteristics. *The CAIS shall support control at execution time of host transmission characteristics (e.g., rates, parity, number of bits, half/full duplex).*

Many terminals support a large number of characteristics, including baud rate, duplex, parity, number of bits, handshake protocols. Explicit control of these permits the user of a tool to control how his terminal interacts with the APSE. It also permits the tool to access devices which do not conform to the system standard.

This allows a tool builder to deal with a Virtual Device of the appropriate type and enhance portability by deferring these device specific characteristics to an instance of tool execution.

6.4I Type-Ahead. *The CAIS shall provide facilities to disable/enable type-ahead. The CAIS shall provide facilities to indicate whether type-ahead is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable type-ahead in those implementations that do not support type-ahead (e.g., null-effect or exception raised).*

Type-ahead is similar to buffer control, but is generally only required to be present or absent. If keystrokes can be saved for the future, a user can get ahead of a slow machine. Type-ahead can be confusing, however, when the terminal is used for non-conversational modes, as in modelling process control.

Type-ahead enhances the interface to an interactive terminal by enabling continuation of input when a system is insufficiently responsive.

6.4J Echoing. *The CAIS shall provide facilities to disable/enable echoing of data units to their source. The CAIS shall provide facilities to indicate whether echo-suppression is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable echoing in those implementations that do not support echo-suppression (e.g., null effect or exception raised).*

Passwords should not generally be echoed. Commands should. When keystrokes have screen-oriented semantics, their echoing is useless and confusing. If typing C moves the cursor forward one character on the screen, the echo of C would overwrite the current character. This control is also essential for windowing, so that the appropriate window may be addressed before echoing is simulated.

6.4K Control Input Datastream. *The CAIS shall provide facilities to designate an input datastream as a control input datastream.*

Control datastreams are special streams in which certain units or sequences cause things to happen. The standard example is that a CONTROL_C from the user's terminal aborts a process. This is a special property of the user's terminal, not a general property of the bit pattern 0000011.

6.4L Control Input Trap. *The CAIS shall provide the ability to abort a process by means of trapping a specific data unit or data block in a control input datastream of that process.*

This permits urgent control over a process, even an unwilling process. The most common trap sequences are CONTROL _C and BREAK (which is not a unit sequence).

6.4M Trap Sequence. *The CAIS shall provide facilities to specify/query the data unit or data block that may be trapped. The CAIS shall provide facilities to disable/enable this facility at execution time.*

Some programs should not be interruptable, such as login and logout. They can disable trapping. Other programs would make good use of catching the character themselves and terminating smoothly.

6.4N Data Link Control. *The CAIS shall support facilities for the dynamic control of data links, including, at least, self-test, automatic dialing, hang-up, and broken-link handling.*

Data links will be important means of communication between APSES, and perhaps within APSEs. This sort of control, at the functional level as specified by 6.3A, must be available. Some control sequences are subject to timing constraints (such as the intercept sequence to intelligent modems), and cannot be accessed by normal output operations, as these do not guarantee the correct timing.

6.5 Data Block Transmission

6.5A Data Block Size. *The CAIS shall provide facilities for the specification of the size of a sequence of units during program execution.*

The flexibility of data blocks is needed to support general protocols, and also to enable communication with devices which were not specified at the writing of the program. This permits low level communication with devices known only to the user of the program, controlled by the data used. The tool builder can deal with a virtual device, independent of the actual characteristics of particular devices.

6.6 Data Entity Transfer If portability of tools between APSEs, even identical ones, is to occur, it is necessary to have the transported data exist outside of the APSE. When the APSEs differ, even in version, any internal format of data is not guaranteed to match. Thus an APSE reading an internal form is not assured of understanding what is written. This is expected to be the delivery format of programs, as well as a way of recording for posterity.

The requirement does not explain the effect of converting an entity into the external form when it has relationships to entities not being converted. Clearly, the external form cannot meaningfully convey a relationship to something inside the APSE, since the reading APSE will generally not have access to the writing APSE. It is thus necessary to refer only to those relationships referencing entities included in the subset of the APSE which is being converted to the external form. This sort of dangling reference is related to the consequence of deleting an entity in section 4.

There is also a potential ambiguity in moving files (untyped data) from one APSE to another. A file of digits 20, for example, may be meaningless on a new host, and even if it could be semantically represented, the bit patterns and effective precision may be different. Even the treatment of control characters in ordinary text files may differ between APSEs. Consequently, the CAIS must specify which data types in files can be put in the common external form without loss of semantics.

6.6A Common External Form. *The CAIS shall specify a representation on physical media of a set of related data entities (referred to as the Common External Form).*

6.6B Transfer. *The CAIS shall provide facilities using the Common External Form to support the transfer among CAIS implementations of sets of related data entities such that attributes and relationships are preserved.*

6.7 General Input/Output The requirement for task waiting, 5.4A, is essential to the correct behavior of input/output. That requirement was originally included here, but was removed as redundant. Some discussion of the importance of this to input/output is appropriate.

The concept of task waiting has been a surprise to many readers of the Ada Reference Manual. Some interpret the standard to require that the waiting of one task in a program cannot interfere with another task, while others see no such restriction. Some validated implementations do wait all tasks. This is a characteristic of the scheduler, which is part of the implementation of the CAIS and the language, and this specification governs the semantics of the scheduler.

In chapter 9 of the Reference Manual for the Ada language, paragraph 2 says "Tasks are entities whose executions proceed in parallel in the following sense. Each task can be considered to be executed by a logical processor of its own. Different tasks (different logical processors) proceed independently, except at points where they synchronize." Independence of tasks requires that one task should not be delayed by input/output wait in another. Execution is what proceeds, and execution is the process by which a statement achieves its action (chapter 5, paragraph 1). Nothing achieves its action by waiting on another task with which it is not in rendezvous.

The alternative is to leave this open, as the Ada language is officially interpreted to do.

Some programs need to talk to the user when, and often because one of their tasks is suspended awaiting a resource or a request. When this happens, it is desirable that the communication with the user does not wait for the other event, since it may never happen.

This requirement has an impact not only on the packages which implement the CAIS, but also on the implementation of the language itself.

It is argued that this is in contradiction with 5.5B.

6.7A Waiting. *The CAIS shall cause only the task requesting a synchronous input/output operation to await completion.*

6.7B Unsupported Features. *The CAIS should provide facilities to control the consequences when the physical device does not have all of the features of the virtual device.*

If the device does not have cursor control, it may be desirable to let the escape sequences echo on the screen for debugging purposes (i.e. to treat the device as another device), to do nothing, or to raise an exception. The control over this behavior is not required in all interfaces, but an explanation should be offered as to why the selections were made. The exception response is technically general in that the calling program can handle the exception with whatever action is desired, but this may raise high costs in complexity and runtime efficiency.

RAC RATIONALE Comment Form

!section:

!RAC version 13 Sept 1985

!submitter:

!date

!1-line topic/subject:

!extended comment or recommendation:

!rationale for recommendation:

!disposition by RACWG:

[Send via ARPA/MILNET to RAC-Comment@Ada20, or
via U.S. Mail to "Patricia Oberndorf/Hans Mumm,
Code 423, NOSC, San Diego, CA 92152"]

DRAFT
DRAFT
DRAFT
DRAFT
DRAFT
DRAFT

Guidelines and Conventions
Working Group

Ada Tool
Transportability Guide

DRAFT
DRAFT
DRAFT
DRAFT
DRAFT
DRAFT

CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	WHY THIS GUIDE WAS WRITTEN	1-1
1.2	DEFINITION: TRANSPORTABILITY	1-1
1.2.1	Things That Can Hinder Transportability	1-3
1.3	THE SIDE-EFFECTS OF STRIVING FOR TRANSPORTABILITY	1-3
1.3.1	The Least-Common-Denominator Problem	1-4
1.3.2	The Project Proposal	1-4
1.3.3	Project Performance	1-4
1.4	LEVELS OF TRANSPORTABILITY	1-5
1.5	THE CONTENTS OF THE GUIDE	1-7
1.6	THE GUIDE'S SPECIFIC USES	1-8
CHAPTER 2	REFERENCES	
CHAPTER 3	ADA LANGUAGE CONSIDERATIONS	
3.1	INTRODUCTION	3-1
3.2	PRAGMATIC LANGUAGE RECOMMENDATIONS	3-2
3.3	GUIDELINES SYNOPSIS	3-4
3.4	STANDARDS SYNOPSIS	3-7
CHAPTER 4	PROGRAM DESIGN AND STRUCTURE	
4.1	INTRODUCTION	4-1
4.2	ENCAPSULATION AND ISOLATION	4-1
4.3	SOFTWARE COMPONENTS	4-2
4.3.1	Importing Components	4-2
4.3.2	Designing Components	4-4
4.3.3	Multiple Body Implementations	4-5
4.4	USING EXISTING AND PROPOSED STANDARDS	4-6
CHAPTER 5	APSE CONSIDERATIONS	
5.1	CONFORMANCE TO CAIS	5-1
5.2	MINIMIZING APSE DEPENDENCIES	5-3
5.3	NON-CAIS SUPPORTED TOOLS	5-6
5.3.1	Hybrid Environments	5-6
5.3.2	Non-CAIS Supported Tools	5-10
CHAPTER 6	STYLE CONSIDERATIONS	
6.1	INTRODUCTION	6-1
6.2	PROLOGUE DOCUMENTATION	6-1
6.3	NAMING CONVENTIONS	6-2

CHAPTER 1

INTRODUCTION

1.1 WHY THIS GUIDE WAS WRITTEN

The Ada Programming Support Environment (APSE) Interoperability and Transportability (IT) Management Plan [NOS83] defines a set of objectives for the APSE IT effort. These objectives are:

1. to develop requirements for APSE IT,
2. to develop guidelines, conventions and standards to be used to achieve IT of APSEs,
3. to develop APSE IT tools to be integrated into both the AIE and ALS,
4. to monitor AIE and ALS development efforts with respect to APSE IT,
5. to provide initiative and give a focal point with respect to APSE IT,
6. to develop and implement procedures to determine compliance of APSE developments with APSE IT requirements, guidelines, conventions, and standards.

This guide addresses point 2 above in the area of tool transportability. As this guide develops and is presented in public forums, it will evolve into a form that is both realistic and useful for the APSE tool writer.

1.2 DEFINITION: TRANSPORTABILITY

Let us first look at some well established definitions of "portable" and "transportable" software [COW76].

software

software

Completely transportable software tends to be widely applicable, extremely well defined, and completely standard. While completely non-transportable software tends to be specific in nature, either in regard to the target, or host, or both, narrowly applicable, and completely non-standard. With standardization the software that falls in the middle of the continuum can be made to skew to the left. If even a small degree of transportability can be gained, both the software rehost engineer and the software maintenance engineer will be grateful. Money and time will have been saved.

1.2.1 Things That Can Hinder Transportability

Many factors can hinder transportable software. A representative list include:

1. Language subsets and supersets,
2. Language ambiguities,
3. Timing differences between APSEs,
4. Memory size and type across APSEs,
5. Differences in I/O equipment,
6. Machine arithmetic differences,
7. Representational differences,
8. Implicit use of system facilities (e.g. large address space),
9. Implicit use of KAPSE facilities,
10. The structure of the database,
11. Asynchronous events,
12. non-standard and differing communication protocols.

1.3 THE SIDE-EFFECTS OF STRIVING FOR TRANSPORTABILITY

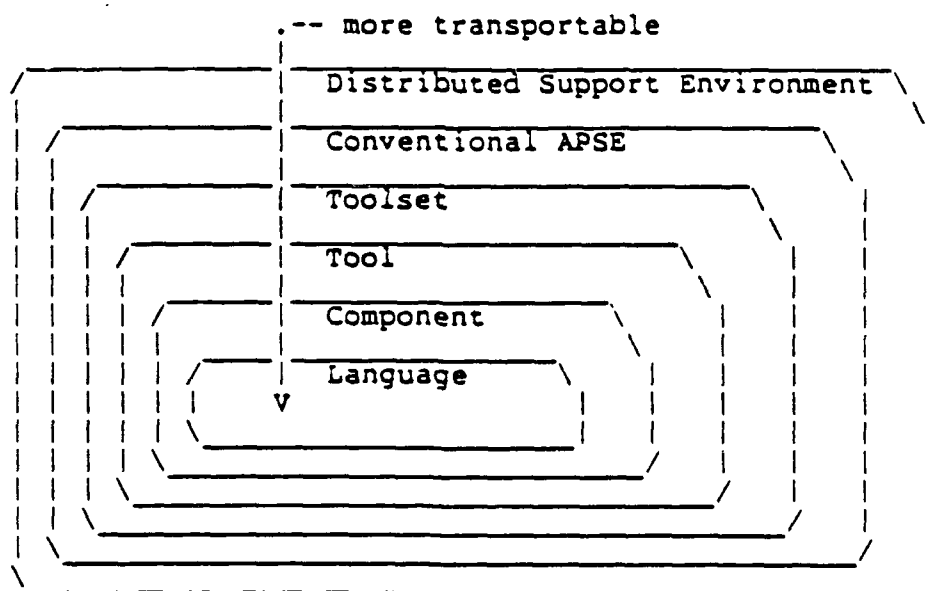
As a company develops a large base of in-house transportable Ada packages, its overall efficiency, productivity and precision increases. This can be used to significant advantage when bidding for a contract, allowing the company to bid lower with greater confidence.

Similarly, when a software proposal is under evaluation, the proposed reuse of proven software can

- * decrease the evaluation phase,
- * increase the degree of confidence in the proposed cost and time estimates.

1.4 LEVELS OF TRANSPORTABILITY

Below is a diagram showing an onion skin model of transportability. When transporting software many levels can exist each having a set of unique problems.



As we work our way in, the ability to transport increases (as the level of complexity decreases).

At the outermost level a Distributed Support Environment exists. This environment provides the systems developer with a uniform support environment that spans computer systems. To transport such a distributed support environment requires tremendous work. It is far beyond the scope of this document to address the transportability of

- * special communication subsystems (networks, etc).

Similar demands may exist for transporting a tool from one system to another. If a transportable design was implemented for the given tool, then transportation would entail the rehost of those software components that are system specific. And, of course, the bottom line is the programming language. If a standard language is used that is common across machines, then the transportability of the components is enhanced (thus increasing the transportability of the tools, toolsets, etc).

This document follows this pattern working out from the center. Standards and Guidelines exist for those areas that are well-defined. For those areas not well defined, issues are brought forth and discussed.

1.5 THE CONTENTS OF THE GUIDE

This guide contains 6 chapters and an appendix:

1. This introduction,
2. A list of references,
3. Ada language considerations,

This chapter will present guidelines and standards concerning the use of the Ada language. These items are presented in the same order as they appear in the Ada LRM.

4. Program Design and Structure,

This chapter presents the concepts of encapsulation and isolation, software components, existing and proposed standards.

5. APSE considerations,

This chapter presents transportability issues associated with APSEs.

6. Style considerations

Ada programming style guidelines and standards are presented with accompanying discussions of transportability.

An appendix presents a case study of transporting an Ada software tool from one host environment to another.

CHAPTER 2

REFERENCES

- [BOO84] Booch, G., "Software Engineering With Ada," Benjamin/Cummings Publishing Co, Inc. Menlo Park, CA, 1983.
- [COW76] Cowell, W. Ed., "Lecture Notes in Computer Science - Portability of Numerical Software," Workshop, Oak Brook, IL, June 21-23, 1976, Springer-Verlag, New York.
- [DOW82] Downward, J.G., "Software Migration from RSX to VMS: Programming for portability," DECUS Proceedings, Anaheim CA, December, 1982.
- [FOR78] Ford, B., "Parameterization of the Environment for Transportable Numerical Software," ACM Transactions on Mathematical Software, Vol.4 No.2, June 1978, pp 100-103.
- [HER81] Herman, E.A., "Transportability of Instructional Computer Programs: Issues and Examples," NECC 1981, pp 18-23.
- [JOH79] Johnson, A.G., "Transportability of Software," Automatic Testing Conference, 1979, Session 3.
- [KIT85] KAPSE Interface Team (Ada Joint Program Office), "Proposed Military Standard Common APSE Interface Set (CAIS)", January, 1985.
- [NIS84] Nissen, J. and Wallis, P., editors, "Portability and Style in Ada," Cambridge University Press, Cambridge, United Kingdom, May, 1984.
- [NOS82A] Naval Oceans System Center, Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report Volume I, NOSC Technical Document 509, Naval Ocean Systems Center, San Diego CA. 1-April-1982.

REFERENCES

- [TI83J] Texas Instruments, "APSE Interactive Monitor (AIM) Configuration Management Plan (CM)," Contract N66001-82-C-0440, 28 March 1983.
- [TI85A] Texas Instruments, "APSE Interactive Monitor (AIM) User's Manual (UM)," Contract N66001-82-C-0440, July 1985.
- [TI85B] Texas Instruments, "APSE Interactive Monitor (AIM) Program Design Specification (PDS)," Contract N66001-82-C-0440, July 1985.
- [TI85C] Texas Instruments, "APSE Interactive Monitor (AIM) System/Integration Test Procedures (SITPRO)," Contract N66001-82-C-0440, July 1985.
- [TI85D] Texas Instruments, "Installation and Maintenance Guide for the APSE Interactive Monitor (AIM)," Contract N66001-82-C-0440, July 1985.
- [TRE81] Treiber, A.E., "Interoperability Through Effective Information Exchange Standards," The Journal of Systems and Software 2, pp 337-350(1981).
- [ZOB75] Zobrist, D.W. et.al., "Software Standards and CAMAC ...a realtime demonstration," Instrumentation Technology, March 1975.

TI REFERENCES - TBD TBD TBD TBD TBD TBD TBD TBD TBD

CHAPTER 3

ADA LANGUAGE CONSIDERATIONS

3.1 INTRODUCTION

This chapter presents the Ada language considerations for enhanced transportability of Ada source code. This chapter contains two parts: pragmatic language recommendations and a set of standards and guidelines for programming in the Ada language. The standards (mandatory) and guidelines (recommended) presented here are copied from [NIS84]. For a more complete treatment and rationale presentation please refer to the referenced material.

By Expression:

Number of operators	100
Number of objects	100
Number of functions	100
Depth of parenthesis nesting	50

By subprogram:

Number of declarations in a subprogram	100
Number of formal parameters	100

By Package:

Number of declarations (visible) in a package	500
Number of declarations (private) in a package	500

By task:

Number of accepts	100
Number of entries	100
Number of select alternatives	100
Number of delays	100

bits for the model numbers should be avoided. (32 bits is the largest accuracy likely to be supported on small machines)."

Ada LRM Section : 3.5.9

GUIDELINE: "Constrained types should be used for integer discrete ranges, especially if a large range is required."

Ada LRM Section : 3.6.1

GUIDELINE: "The collection size should be specified if possible by using 't'SORAGE_SIZE expressed in terms of the size of the object of the access type, divided by system.storage unit. However, due allowance should be made for the space required by the allocator for tables and links, etc (see LRM 13.2)."

Ada LRM Section : 4.8

GUIDELINE: "Renaming should not be used to change the name of any entity from the package standard (see LRM Appendix C), or machine-dependent values from the package system. This ensures that the use of any such machine-dependent values in the user's program remains obvious."

Ada LRM Section : 8.5

GUIDELINE: "Guards should only depend upon local variables of the task in order to avoid the possibility of side-effects."

Ada LRM Section : 9.7.1

GUIDELINE: "No assumptions about the range of values that can be specified in the priority pragma should be made. Similarly, no assumption about the number of different priority levels should be made."

Ada LRM Section : 9.8

GUIDELINE: "The pragma shared should not be used."

Ada LRM Section : 9.11

3.4 STANDARDS SYNOPSIS

STANDARD: "No control character (known in the LRM as non-graphic characters) should be used except newline or the equivalent pair carriage-return/line-feed."

Ada LRM Section : 2.2

STANDARD: "Implementation defined pragmas must not be used."

Ada LRM Section : 2.8

STANDARD: "The evaluation of default expressions in an object declaration must not have side-effects on values in other expressions in the same declaration."

Ada LRM Section : 3.2.1

STANDARD: "No assumption must be made about the order of evaluation of the bounds in a range constraint."

Ada LRM Section : 3.5

STANDARD: "Values of type INTEGER must lie within the range -32767..32767 with corresponding ranges for NATURAL and POSITIVE. Integer ranges outside the range -32767..32767 must not be used except to define new integer types."

Ada LRM Section : 3.5.4

STANDARD: "The type names SHORT_INTEGER and LONG_INTEGER must not be used explicitly."

Ada LRM Section : 3.5.4

STANDARD: "Values of the type outside the range of safe numbers should not be assumed."

Ada LRM Section : 3.5.6

Ada LRM Section : 4.1.1

STANDARD: "A function in an expression of an indexed component must not change the prefix on another index of the same component as a side-effect."

Ada LRM Section : 4.1.1

STANDARD: "No assumption must be made about the order of evaluation of the name, prefix and discrete range of a slice."

Ada LRM Section : 4.1.2

STANDARD: "Evaluation of expressions in component associations in aggregates must not have side-effects affecting variables appearing in other components of the same aggregate."

Ada LRM Section : 4.3

STANDARD: "The evaluation of operands (except operands of short circuit control forms) must not have side-effects affecting other operands in the same expression."

Ada LRM Section : 4.5

STANDARD: "Unless the numbers correspond exactly to model numbers, then the result of equality or inequality between floating point numbers should not be relied upon."

Ada LRM Section : 4.5.7

STANDARD: "It must not be assumed that a real static expression is evaluated with greater accuracy than that of the target machine."

Ada LRM Section : 4.9

STANDARD: "Assignments in which the evaluation of the right hand expression can change the evaluation on the left or vice versa must be avoided."

Ada LRM Section : 5.2

Ada LRM Section : 9.6

STANDARD: "Evaluation of guards must not have side-effects changing the state of other guards in the same selective unit."

Ada LRM Section : 9.7.1

STANDARD: "The program should not rely on the algorithm used to choose between several open alternatives in a selective wait."

Ada LRM Section : 9.7.1

STANDARD: "A program must not rely upon the use of priorities to enforce synchronization."

Ada LRM Section : 9.8

STANDARD: "No assumption should be made about the moment at which the task being aborted becomes terminated."

Ada LRM Section : 9.10

STANDARD: "The main program must be a parameterless procedure."

Ada LRM Section : 10.1

STANDARD: "No assumption should be made as to the precise condition under which CONSTRAINT_ERROR, NUMERIC_ERROR, PROGRAM_ERROR, TASKING_ERROR, or STORAGE_ERROR exceptions are raised."

Ada LRM Section : 11.1

STANDARD: "The suppress pragma must not be used to achieve any change in the meaning of a program."

Ada LRM Section : 11.7

STANDARD: "Programs must not depend on the order of evaluation of actual generic parameters."

Ada LRM Section : 12.3

CHAPTER 4

PROGRAM DESIGN AND STRUCTURE

4.1 INTRODUCTION

This chapter presents guidelines for program design and program structure that can enhance transportability.

4.2 ENCAPSULATION AND ISOLATION

Encapsulation means placing procedures, functions, exceptions, types, etc. that all pertain to the same object into one package. Isolation means hiding a particular implementation in the package body. Typically isolated packages do not depend on other packages (they stand alone). Encapsulation and isolation promote:

- * Transportable programs,
- * Transportable packages,
- * Ease of modification,
- * Ease of understanding,
- * Potentially inefficient implementations.

Encapsulation and isolation guidelines:

- * Machine dependent facilities must be isolated (and thus hidden) in packages and accessed only through the package interfaces (procedures and functions).
- * A package should represent one unique object (or collection of identical objects) and the operations upon this object.

components for purchase, it is possible to obtain software to perform a variety of tasks. The risks that must be recognized include:

- * The software description seems to meet part of the need, however it may require some (minor or major) modification to achieve the complete functionality required,
- * The software documentation is poor or non-existent resulting in increasing the cost and time of understanding the software itself,
- * the software does not work,
- * the software was developed on a non-ANSI compiler (yes, these things exist, and much software is being written for them),
- * the software uses host OS facilities that may or may not be available on another system,
- * the software was not designed or implemented to be transportable,
- * local compiler and/or run-time bugs may make the software fail.

There are enough risks listed above to warrant the following statement:

Do not plan on using untested and/or unfamiliar software components for a time or cost constrained project unless the component has been thoroughly checked out ahead of time.

The desire to go out and grab public domain software to embed in a developing product can be irresistible (but typically only the first time).

- * the semantic meaning of each interface, type, and exception,
- * the recompilation order,
- * the implicit exceptions that can occur (constraint_error, tasking_error, etc),
- * any system dependencies, where they are located, exactly what they do, and how they do it,
- * any compiler/run-time work-arounds that were needed to get the component to compile and run on the development machine,

4.3.3 Multiple Body Implementations

The nature of encapsulation and isolation supported by Ada (and other languages) can allow multiple implementation of the same specification.

Typically, a package is designed by first producing a specification, either in PDL or more often, in Ada itself. From here the spec can be handed off to someone else to implement. Various implementations are possible, and each has a set of advantages:

1. the simple stub. A stub can be generated that simply returns when called. Functions return fake values, as do procedures with OUT parameters.
2. statistics gathering/debugging stub. This is a stub that could serve at least three functions:
 - gather statistics regarding number of times the unit is called,
 - produce debugging information at each call,
 - produce statistically correct results. This could mean waiting an appropriate length of time (or consuming an appropriate amount of resources) based either on simple tables or perhaps some random variate generator(s). Of course the semantic meaning of the contents may be inaccurate or meaningless.
3. a transportable Ada implementation,

CHAPTER 5

APSE CONSIDERATIONS

The Common APSE Interface SET (CAIS) [KIT85] defines a set of standard interfaces that should promote APSE and tool transportability. These interfaces can be used to create information structures, to run and communicate with processes and to interact with hardware devices. Even though one standardizes on these interfaces, there are some CAIS related issues that need to be addressed to improve the transportability of APSE tools. Some of these issues that need to be addressed include:

- * improved tool protocols,
- * conventions on how the CAIS is used, and,
- * how tools are designed.

Also, we must recognize that APSEs will make use of existing tool sets to support the development of applications.

This chapter discusses different strategies on how one can further improve transportability of tools developed on top of the CAIS. Even though the use of the CAIS is a significant factor for improving tool transportability, it is not sufficient to guarantee that tools which conform to the CAIS, are transportable.

5.1 CONFORMANCE TO CAIS

A tool is considered to conform to the CAIS if all of the tool's interactions with the environment are performed through the facilities provided by the CAIS. Figure 1 illustrates the difference between a tool set that conforms to the CAIS and a tool set that does not conform.

control is absolutely necessary to enforce access control and to allow the APSE to control and manage all the data objects created using the APSE. The key to providing control over the APSE database lies in the ability to enforce access controls and to ensure that the data structuring rules defined by the CAIS are followed. Only if all the tools conform to the CAIS interfaces can we assume that there is some control over how data is created and accessed. For this reason one should avoid instances where the tools bypass the CAIS in order to access data managed by the APSE. In reality, requiring that all APSE tools conform to the CAIS is somewhat idealistic for the following reasons:

- * There exists a large base of tools that have been developed before the CAIS standard is(was) established. These tools will continue to be used to support existing applications as well as new applications.
- * The CAIS does not address all the possible tool needs.
- * Many specialized applications will require the use of commercially available tools that may never conform to the CAIS.

We need to plan for a transition period during which time tool will migrate to conform to the CAIS.

5.2 MINIMIZING APSE DEPENDENCIES

Although conformance to the CAIS will aid in tool transportability, it is not sufficient to ensure that the cost of transporting tools will be low. Tools can be tightly coupled to the environment by building into the tool intimate knowledge about the way data is structured and managed by an environment. A tool's knowledge of how an environment structures and manages data will affect the transportability of tools to other APSEs that structure and manage the same data in a different manner. There are at least two strategies that can be used to further improve tool transportability. First, the tool design must isolate the APSE dependent code so that it can easily be modified to interface to a different APSE. Second, the tool can assume a simple interface and place the burden on the environment to provide the data that it needs in this simpler form. The following are examples of how one can improve tool transportability by applying these strategies.

Tools (even if they use the CAIS) need to be developed to minimize dependence on other tools in the toolset and the way a particular APSE structures it's data. For example, one should be free to select between different compilers because they meet desired code efficiency requirements and to readily integrate these compilers

the environment structures its data and makes use of this knowledge to locate the data that it needs. The dotted lines show that the tool knows about the entire data structure and uses the knowledge to locate the data that it needs. Because this knowledge is built into the tool it will take more effort to transport the tool to a new environment that structures its data differently. Figure 2(b) illustrates a case where the environment knows what information the tools needs and presents to the tool the exact information needed by the tool. This is indicated by the fact that the environment directs the tool to operate on exactly the data objects that it needs. This information can be presented to the tool in a simple standard format. Assuming that this strategy is adopted, the cost of transporting tools to different environments will involve the modification of an existing part of the environment to present the information to the new tool. The information presented to the tool is then accessed using CAIS facilities.

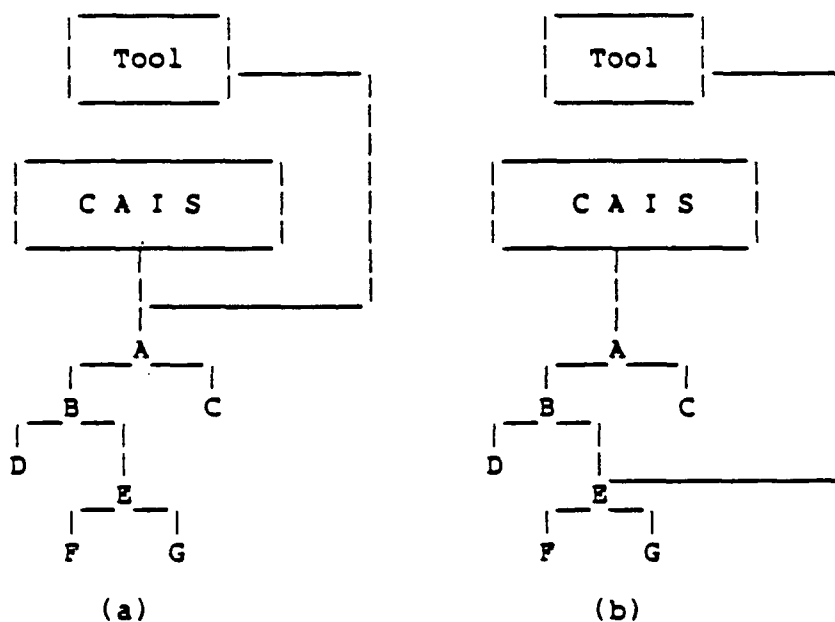


Figure 2

Ada compilers provide a good example to illustrate this point. It is highly desirable to incorporate many different compilers into an APSE to address different development needs (degree of optimization, support for different targets). If a compiler is intimately aware of how an APSE structures its Ada libraries and how the library objects are managed (configuration management) then the compiler will be tightly coupled to that environment. A

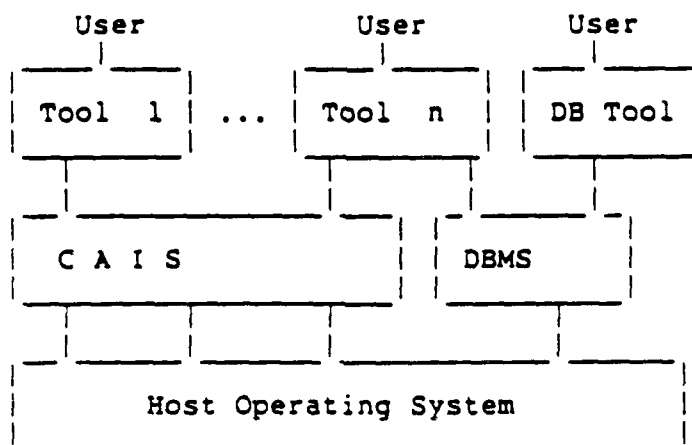


Figure 3

In such development environments there may be tools that talk to both systems. The following discussion will use the example of incorporating a database management system as significant tools system that is to co-function with the APSE. Other development systems may be substituted for the DBMS in this example.

A long term solution would be to provide portable databases implemented on top of the CAIS as illustrated in Figure 4. In this case, all the data is managed by the APSE through the CAIS providing central control over access to the data. All the tools, including the DBMS, are transportable.

some applications there is a need to integrate the use of the existing DBMS and its data. The question is how to integrate this DBMS with its tools and data into an APSE tool that attempts to conform to the CAIS.

- Treat DBMS as a CAIS process where all interactions with the DBMS are through process messages
 - possible efficiency considerations
 - possible problems with user interfaces
 - how to maintain the necessary relationships between the DBMS data definitions and the corresponding definitions included in Ada programs. The problem is that definitions defined and maintained by the DBMS are not controlled by the APSE. This provides a real consistency maintenance problem
- Tools developed to interface between the DBMS and the APSE would use the process communication protocols to communicate with the DBMS

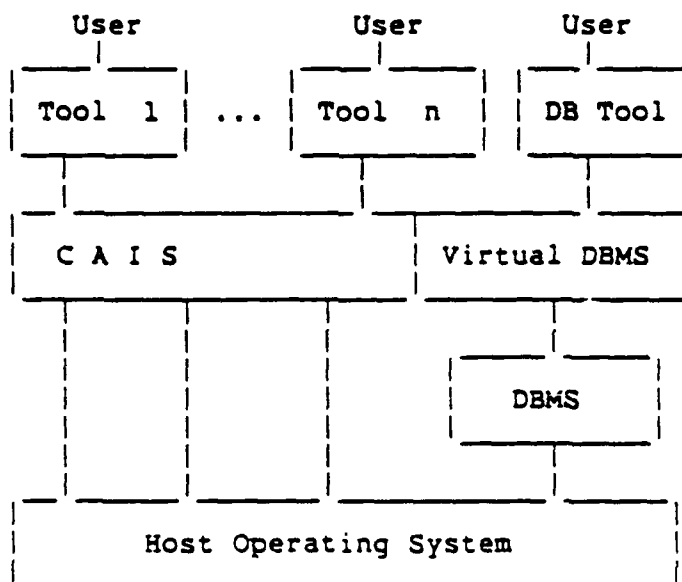


Figure 6

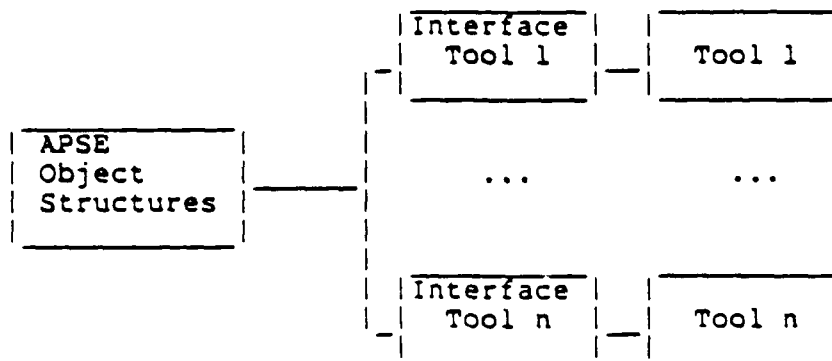


Figure 7

- Interface tool has knowledge of APSE object structures, how these structures are managed (CM). They may be used for both CAIS conforming and non-conforming tools.
- Interface tool also knows how to present data to each tool
- Inefficiency because of additional interface layer
- What is an interface tool:
 - An interface tool may be a CAIS process where data is passed between the tool and the environment via the process message passing facilities. This facility might be used to support the use of interactive tools. Especially if tools dynamically create objects and relationships in the APSE database
 - Some commercial off-the-shelf (COTS) tools will not conform to the CAIS. In this case the interface tool will convert the CAIS data structures to those expected by the tool (i.e. use existing host file system directly).
- Advantages:
 - Can add new tools quickly

CHAPTER 6

STYLE CONSIDERATIONS

6.1 INTRODUCTION

Transporting a software tools usually involves taking an existing tool at the source level, and attempting to recompile, relink, and run it on a new system. It is common that the group of persons doing the transport do not know the system. Transportability can be enhanced if these persons can grasp the meanings of the various program units and are able to recognize the Ada constructs and program structure easily and quickly. Standards and guidelines on coding practices is given in this section.

The coding practices are specified as either "standards" or "guidelines." "Standards" are mandatory requirements while "guidelines" denote preferred practices.

6.2 PROLOGUE DOCUMENTATION

This section specifies the minimum information to be included in the prologue of a program unit. The prologue provides for program version identification to facilitate configuration management.

STANDARD: Each separately compiled program unit shall have an associated prologue directly preceding the program unit.

STANDARD: All prologue information for each program unit must be present before the code is placed under configuration management control.

6.4 DECLARATIONS USAGE

6.4.1 COMMENTING DECLARATIONS

GUIDELINE: Each constant, type, and object/variable identifier declared should be accompanied by a brief COMMENT if the identifier is not self-explanatory. The comments should be aligned as closely as possible for enhanced readability. This guideline is illustrated in Example 1.

6.4.2 DECLARATION FORMATTING

STANDARD: All declarations shall be aligned and each one shall be on a separate line. Identifier lists shall not be used. This standard is illustrated in Example 1.

GUIDELINE: The grouping of declarations should be consistent and follow one of the methods described below. The decision as to which grouping method will be enforced shall be made by the project manager. The grouping methods are :

1. All declarations possessing the same characteristics should be grouped together and commented accordingly. For example, all CONSTANTS shall be together, all TYPES together, etc. Additionally, SUBTYPES should immediately follow their respective TYPES. A blank line should be used to separate the declaration groups. This guideline is illustrated in Example 1 and Example 1.
2. All declarations concerning the same usage should be grouped together and commented accordingly. A blank line should be used to separate the declaration groups. This guideline is illustrated in Example 1.

```

-- day types

type DAY is (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,
             FRIDAY,SATURDAY,SUNDAY);

TODAY : DAY;
TOMORROW : DAY;

-- card types

type SUIT is (CLUBS,DIAMONDS,HEARTS,SPADES);
type VALUE is ('2','3'...'10',JACK,QUEEN,KING,ACE);
type PLAYER is (NORTH,WEST,SOUTH,EAST);
type CARD is
  record
    SUIT_NAME : SUIT;
    NUMBER_VALUE : VALUE;
  end record;
type HAND is array(1..13) of CARD;
type TABLE is array(PLAYER) of HAND;

TRUMPS      : SUIT;
FIRST_TABLE : TABLE;
SECOND_TABLE : TABLE;

```

Example 1. ADA VARIABLE DECLARATIONS

6.4.3 USE OF CONSTANTS

STANDARD: Each character literal or string literal shall be located in the constant declarations. This standard is illustrated in Example 1.

STANDARD: Do not embed "MAGIC NUMBER" constants in your code. Such constants lack significance and are difficult to maintain. This standard is illustrated in Example 1.

6.5 CODING CONVENTIONS

This section specifies coding practices which apply to Ada statements.

6.5.1 ATTRIBUTES

GUIDELINE: The use of attributes is recommended (except as noted in the chapter Language Considerations, above) for maintainability. If you see a "MAGIC NUMBER", try to use an attribute or declare a constant.

6.5.2 UPPER/LOWER CASE USAGE

STANDARD: All Ada reserved words shall be one case and all other names may be mixed case. (reserved words upper case, user defined words lower case, etc.) Comments may be mixed case except where the syntax of the English language requires an upper case letter (i.e. starting of a complete sentence, abbreviations, etc.) This standard is illustrated in all examples.

6.5.3 STATEMENT FORMATTING FOR READABILITY

STANDARD: Each statement must begin on a separate line. Multiple statements per line are not allowed. This standard is illustrated in Example 1 and Example 1.

STANDARD: At least one space must appear before and after all relational and arithmetic operators. This standard is illustrated in all examples.

STANDARD: Indentation is approached with "comb" structures. Examples of proper indentation techniques are shown in Example 1 and Example 1.

```

case STRING_LENGTH is
  when 0      => STRING_ERROR;
  when 1..80  => COPY_STRING;
  when 81..160 => COPY_STRING;
                STRING_LENGTH := STRING_LENGTH - MAX_STRING;
  when 161 | 162 => DELETE_STRING;
  when 163      => null;
  when others   => raise SYSTEM_ERROR;
end case;

```

```

task SEQUENCER is
  entry PHASE_1;
  entry PHASE_2;
  entry PHASE_3;
end SEQUENCER;

```

```

task body SEQUENCER is
begin
  accept PHASE_1;
  accept PHASE_2;
  accept PHASE_3 DO
    INITIATE LAUNCH;
  end PHASE_3;
end SEQUENCER;

```

```

loop
  -- program unit statements
  select
    accept MAKE_DEPOSIT(ID      : in ACCOUNT_TYPE;
                        AMOUNT : in CASH_TYPE) do
      -- program unit statements
    end MAKE_DEPOSIT;
  or
    accept MAKE_DRIVE_UP_DEPOSIT(ID      : in ACCOUNT_TYPE;
                                (AMOUNT : in CASH_TYPE) do
      -- program unit statements
    end MAKE_DRIVE_UP_DEPOSIT;
  else
    DO_FILING;
  end select;
  -- program unit statements
end loop;

```

Example 1. STATEMENT ALIGNMENT AND INDENTATION

```

with SAMPLE;
procedure ANALYZE_SENSOR_VALUES is
  ACTUAL_DATA : SAMPLE.VALUES;
  FITTED_DATA : SAMPLE.VALUES;

  procedure GET_SAMPLES (DATA : out    SAMPLE.VALUES) is separate;
  procedure LIMIT_CHECK (DATA : in out SAMPLE.VALUES) is separate;
  procedure CURVE_FIT   (DATA : in     SAMPLE.VALUES;
                        LIMIT: in     SAMPLE.VALUES;
                        FIT  : out    SAMPLE.VALUES) is separate;

  procedure REPORT      (DATA : in     SAMPLE.VALUES) is separate;

begin
  GET_SAMPLES (ACTUAL_DATA);
  LIMIT_CHECK (ACTUAL_DATA);
  CURVE_FIT (ACTUAL_DATA, LIMIT_DATA, FIT => FITTED_DATA);
  REPORT (FITTED_DATA);
end ANALYZE_SENSOR_VALUES;

function IS_ODD(VALUE : in INTEGER) return BOOLEAN is
begin
  return ((VALUE rem 2) /= 0);          [BOO84 ]
end IS_ODD;

```

Example 1. PROCEDURE AND FUNCTION FORMATTING

GUIDELINE: The use of the WITH statement without the USE statement is recommended for understandability and clarity of locations of program units. This guideline is illustrated in Example 1. [BOO84]

GUIDELINE: Each subprogram or function should contain only one RETURN statement and it must be the statement immediately preceding the END, unless additional RETURNS enhance the clarity of code. This guideline is illustrated in Example 1.

GUIDELINE: Tasks should be used for concurrent actions, routing messages, controlling resources, and interrupts. [BOO84]

GUIDELINE: Generic program units should be used for factoring the properties of a class of program units, and passing types as parameters to program units. [BOO84]

GUIDELINE: Overloading should only be used to name an equivalent action for different types. [BOO84]

GUIDELINE: Packages should be named with noun phrases summarizing the package contents. [BOO84] For example:

```
-- MATH_FUNCTIONS, EARTH_CONSTANTS
```

GUIDELINE: Tasks should be named with noun phrases, usually denoting some action. [BOO84] For example:

```
-- TIMER, MESSAGE_ROUTER, LIST_SEARCHER
```

6.5.6 COMPLEX EXPRESSIONS

GUIDELINE: Complex expressions should be parenthesised for readability and understandability. This guideline is illustrated in Example 1.

```
-- poor
```

```
GAMMA_VALUE = BETA_VALUE + ALPHA_VALUE / 2 ** 5 * 6;
```

```
-- better
```

```
GAMMA_VALUE = BETA_VALUE + ((ALPHA_VALUE / (2 ** 5)) * 6);
```

Example 1. COMPLEX EXPRESSIONS

6.5.7 LABELS AND GOTOS

STANDARD: The use of LABELs and GOTO statements shall require project manager approval on a case by case basis.

6.5.8 TASK TERMINATION

GUIDELINE: ABORT should not be used. [BOO84]

GUIDELINE: Every server task should have a TERMINATE statement to achieve regular termination with its parent unless an ENTRY is provided for

APPENDIX A

TRANSPORTING AN ADA SOFTWARE TOOL : A CASE STUDY

A.1 INTRODUCTION

This Appendix presents a case study of transporting an Ada software tool from one environment, the Data General AOS/VS Ada Development Environment (ADE)(tm), into another environment, the Digital Equipment Corporation VAX/VMS(tm) environment (with the DEC(tm) Ada compiler and tools).

The APSE Interactive Monitor (AIM) was developed in Ada in the DG ADE. The AIM was transported to the VAX/VMS in 2.4 man-months. The transport turned up many issues including:

- * how to deal with compiler bugs,
- * problems with run-time storage allocation schemes,
- * problems with scheduling and task blocking schemes,
- * how inappropriate assumptions can be made with regard to low-level models of the operating system functions,
- * how inappropriate reliance can exist on operating system services,
- * the positive and negative aspects of techniques that improve transportability,
- * problems relating to debugging a transported tool.

The use of Ada can promote the transportability of source code. This case study shows that, with appropriate transportability guidelines, and attention paid to the details, a software tool written totally in Ada can be moved from one system to another with minor difficulties.

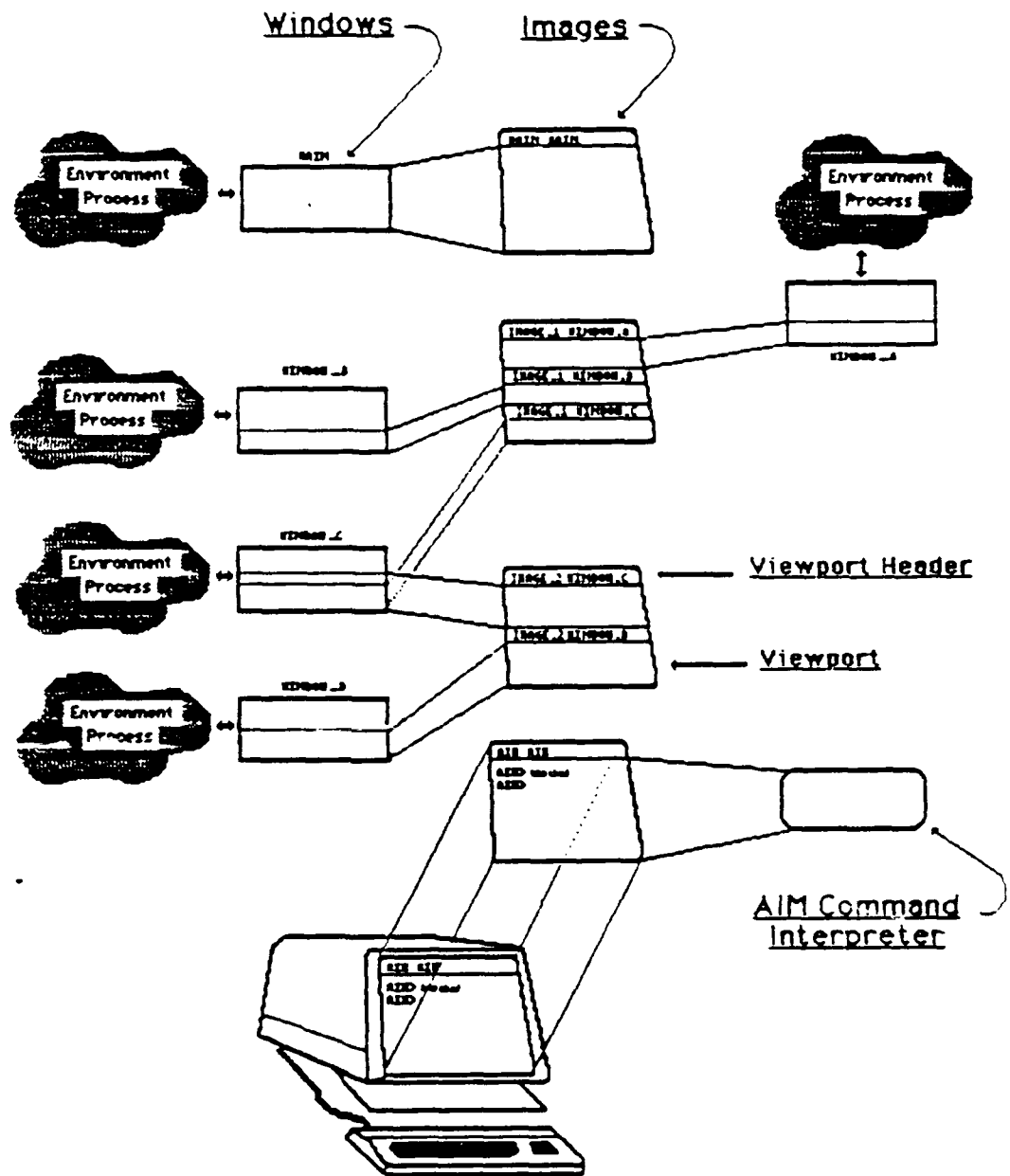


Figure 1. The Elements of the AIM

A.1.2 Design For Transportability

A great effort was made to increase the transportability of the source code. The following techniques were used:

- * Isolation - the system dependent parts of the AIM were minimized and placed in packages. A model was developed for each system dependency, and interfaces developed (the package specification). The implementation of each interface would need to be rewritten when moving from one system to another.
- * Encapsulation - Object oriented design was used in an attempt to identify the objects, their attributes, and the operations that were to be performed on the objects. These objects were placed in packages. As each package was developed, particular emphasis was placed on minimizing inter-package dependencies.
- * ANSI Ada was used. Chapter 13 issues (optional items) were isolated into the system dependent packages as much as possible.

A.1.3 The Environments

The DG Ada Development Environment (ADE) [DAT84] is a complete environment that is entered from the standard DG AOS/VS operating system. Within the environment all of the AOS/VS command interpreter commands are available as well as specific ADE commands to support Ada program development.

The DEC VAX/VMS environment has integrated the Ada compiler into the standard VAX/VMS program support environment. The compiler is run like all other integrated compilers on the system. A support system for automatic recompilation, program library management, and other support functions is provided through the Ada Compilation System (ACS).

Both environments include:

- * full ANSI Ada compiler,
- * Ada linker,
- * Ada program librarian,
- * Ada source level debugger,
- * project management capabilities,

- * computer terminal control and communications,
- * process control and communications,
- * environment variables,
- * Ada "length" representation clause.

These operating system dependent interfaces were developed in VMS over a period of about one man-month. A rather complete description of these is presented to promote

greater understanding when the problems and issues are discussed in section 3.

A.2.2.1 Computer Terminal Control And Communications

The AIM computer terminal control and communications package is known as SYSDEP. This package provides very elementary interfaces for controlling and communicating with the computer terminal. The interfaces are:

- * Open the computer terminal - When the terminal is open all characters written to it will be sent directly to the terminal immediately (no buffering), and there will be no translation performed by the host operating system. If the computer terminal cannot be opened, then an exception is raised.
- * Close the computer terminal - Reset the computer terminal back to the characteristics it had before OPEN was called. If there are any outstanding I/O requests pending on the terminal, they will be dequeued immediately.
- * Read data from the computer terminal's keyboard - At least one character is read at a time. There is no translation done on the characters before they are passed back to the calling program. No echo is performed before passing the characters back to the calling program (that responsibility is held by the calling program). This "no echo" characteristic can be setup in the OPEN on some systems. Also, a call on READ must not block the entire process that contains both the reader and the SYSDEP package, only the task calling the read should be blocked. In this manner, tasks can be fired up to infinitely loop reading data, rendezvousing with a buffer task and passing the characters on, eventually to be read by another task.
- * Write data to the computer terminal - A call on WRITE causes a string to immediately be sent to the computer terminal. When the call returns the string either must be on the screen or queued to the screen (via host operating system services).

- * getting the terminal name,
- * getting the name of the terminal capabilities file (TCF),
- * getting initial script file name,
- * getting the parse table initialization file,
- * getting the help file.

On the Data General these are implemented using files. There must be files (or links to files) named TERM, TCF, AIM_INIT_SCRIPT_FILE, and AIM_HELP_FILE on the user's search path.

On the VAX, "logical names" are used for these environment variables. A logical name TERM must exist and contain the name of the terminal. Also, the logical names TCF, AIM_INIT_SCRIPT_FILE, and AIM_HELP_FILE must exist and point to valid filenames.

From the point of view of the calling program there is no difference. The implementation is hidden in the packages SYSDEP and DATABASE_SUPPORT.

A.2.2.4 Ada Length Representation Clause

To get the tasks to run on the Data General it was necessary to expand the task memory size using the Ada representation length clause (see the Ada LRM section 13.2):

```
for TASK_NAME'SORAGE_SIZE use TASK_SIZE;
```

This clause can have different meaning on different systems. Also, this clause had to be placed in a specific place in the source code.

A.2.3 Order Of Integration During Rehost

The AIM was a debugged, running system, when the rehost was attempted. After the AIM system dependent parts were developed and debugged, the simplest technique for debugging the AIM was simply to compile it all, link it, and run it. It did NOT run the first time. A technique had to be developed to debug the AIM. However, the debugging information had been removed much earlier in the module testing phase. Also, the module tests that DID exist were developed to test the separate modules as they were being coded and developed. When the modules were integrated into the whole the module tests became obsolete (more on this in section 2.5 and 3.4).

- * Problems with the terminal model (see section 2.2.4), and
- * Problems with task storage allocation (see section 3.3.2).

A.3 REHOST PROBLEMS

This section will discuss the problems encountered during the rehost effort. It is important to note that the VAX/VMS compiler that was used was a field test version. However, it was validated.

Specific details of the problems will be omitted, to conform to the field test agreement.

A.3.1 Code Work-Arounds

With a language as complex as Ada, problems can be expected when moving a program from one compiler implementation to another. Ada validation helps, but does not completely guarantee that bugs will not appear.

The AIM uncovered a variety of bugs in both compilers. Since the AIM was developed in the ADE, these were immediately fixed during module development and testing. During the rehost, however, bug fixing worked differently. The code work-arounds were developed starting with the entire system as a whole, fixing modules, recompiling the separate units, then re-linking the whole AIM and continuing the tests.

The eventual working program was entirely ANSI Ada, however the code work-arounds had two negative effects:

- * It changed the code. Although this is required due to the nature of the problem, it can increase risks when moving the code back to the original system. For maintainability purposes, it was desirable to have exactly the same code running on both systems except for the system dependent parts (right down to the blank lines and comments).
- * Recompiling the separate units was inconsequential due to the fact that the AIM was broken down into such small compilable units. However, linking was extremely resource intensive. Easily 70% of the rehost debugging time was spent relinking the AIM.

By doing this, we diverged from the original implementation. Luckily, a Data General systems programmer gave us the answer to the dequeuing an I/O request. By ABORTing the tasks that were doing the operating system I/O service calls, the queued requests would become dequeued.

By careful encapsulation and isolation, the ABORT statements were embedded in the system dependent code.

A.3.3 System Dependencies

There was a variety of problems that surfaced in the area of system dependencies. Without performing a detailed analysis of the various systems that you will be rehosting to, you cannot be sure that your model will work in all the systems.

A.3.3.1 Enforcing A Model (and Making Assumptions About It)

The design of the AIM left nebulous the exact details of the models for terminal control and communications and for process control and communication. During implementation these models firmed up and took shape. The models were implemented as described in section A.2.2.

As it turned out, these were all very reasonable design criteria, and almost every one had problems. They will be addressed in the next two sections in order.

For communications with the terminal the following capabilities will be discussed:

- * read every character from the terminal with no translation,
- * read at least one character at a time,
- * exclusive access to the terminal.

For control and communication with processes, the following capabilities will be discussed:

- * Spawn a son and pass it standard input and standard output files
 - All terminal directed output would be intercepted by the AIM through the process' standard output file. All process directed input (that would normally come from the terminal) would be supplied by the AIM through the process' standard input.
- * Deleting a process would shut it down immediately regardless of what it was doing -

has a special character that means "ATTENTION" and allows a user to log the data to a file, or exit the program. So, the path is:

User terminal <-> Novation modem <-> LAN <-> VAX <-> Hayes modem
<-> TAC <-> ARPAnet host

The characters that the user must concern themselves with are:

- * XON/XOFF (CONTROL-S and CONTROL-Q),
- * percent,
- * tilde,
- * at sign '@',
- * modem attention characters (which typically are settable).

The problem can be severe. One solution (which the AIM uses) is to define an interface to the terminal system dependent package which allows a calling program to query about the characters that are important. Passing a CONTROL-S into the function will return a boolean identifying whether you can expect to see that key coming from the keyboard, or be able to send that key to the display screen. This solution is not graceful because it cannot be determined completely which characters are valid at a given instant in time due to the variability of the communication medium that is being used.

- * Read at least one character at a time - In the Data General AOS/VS operating system, characters from the computer terminal can be read only one character at a time. Because of this restriction, code written to support this aspect of the model, that is, handling more characters than one at a time, was not tested until the rehost. The VAX supports reading multiple characters at a time, and the code was checked out on the VAX, then eventually moved back to the DG.
- * Exclusive access to the terminal - This was possible on both systems, but was found to be not desirable. System messages such as "Going Down in 5 Minutes" would never "break through" the AIM to the terminal and could not be intercepted by the AIM on the DG or the VAX if exclusive access were enabled. By simply adding a repaint screen procedure, this requirement was eliminated.

A subtle problem was discovered late in the rehost. On the VAX an infinite loop task that makes the operating system I/O call will rendezvous with a buffer task that takes the information and places it into a buffer, from which another task can extract it (at another

System engineers at Data General studied the problem and told us that that functionality was not supported and would never be supported. They said that it was a design decision made when the operating system was being developed. On the VAX this worked just fine.

- * Deleting a sub-process would shut it down immediately regardless of what it was doing - This is NOT the suicide call described in A.3.2. The system service on the DG "?TERM" is supposed to be able to do this. However, it could not be made to work. A similar facility on the VAX worked just fine. So, in this example, there exists a semantically different meaning for the interface. Contrary to the terminal communication termination problem, where the choice was made to make the semantic meaning the same to preserve the model, here the choice was made to correctly implement the model on the VAX even though it made the two models different. In this case it is preferred that the interface react correctly rather than being consistent and acting incorrectly.
- * Detect and control any son processes of a son process (grandson processes) - This was possible to do on both systems. However, when attempting the implementation an interesting problem arose. The Data General AOS/VS operating system uses processes much more extensively to perform operations than the VAX does. Whenever a command is issued to the command interpreter to invoke a program it spawns a new process in which to run the program. These subprocesses then have the option of spawning new ones (and typically in an unpredictable manner) and thus treeing down. Not all commands to the command interpreter cause a subprocess to be spawned, but most do. Users do not see the spawning of the subprocesses (unless they look for it). Typically there is no limit on the number of subprocesses that a user can spawn. They are resources that are not that precious.

On the VAX, processes are used much less. When a command is issued to the command interpreter, it either does the command or replaces itself in the current process with the program to be run to perform the command. When the command is completed, the command interpreter is brought back into the same process. It is expensive and slow to spawn a subprocess on the VAX. Typically, a subprocess quota of 2 or 3 is sufficient for most users.

This is an inherent and subtle difference in the models presented by the two operating systems to the programmer. The difference involves not the presentation of the processes, but their availability and cost.

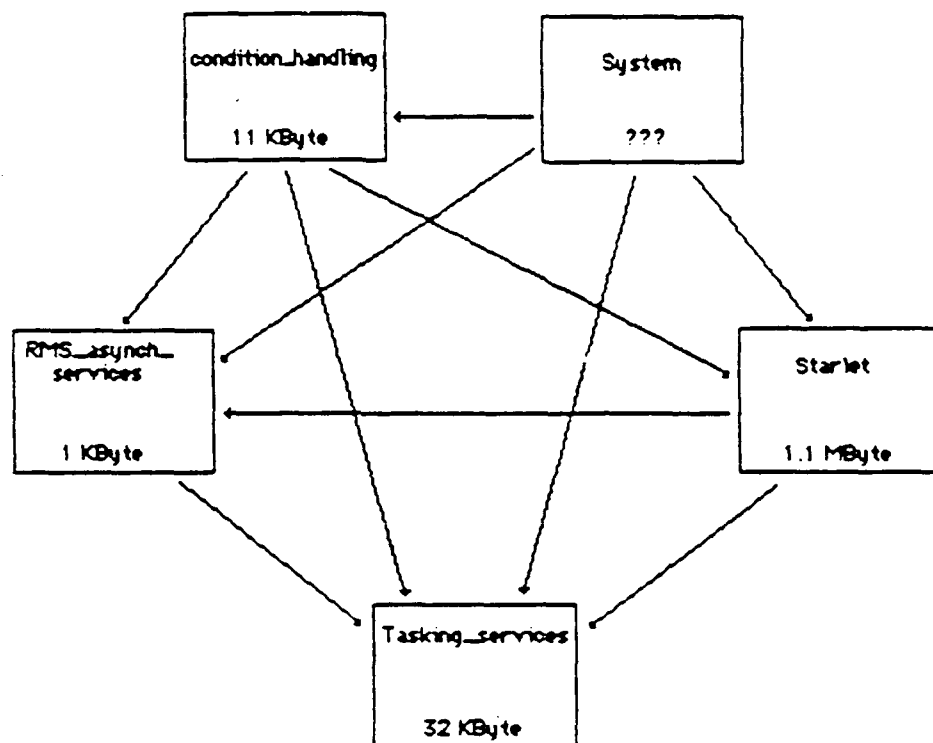


Figure 2. VAX/VMS System Service Package Structure

A.3.4 Module Testing Baggage

An analogy between module testing and a rocket taking off can be effective in demonstrating this issue. A multi-stage rocket has a variety of lower stages that are used to boost the payload into orbit. The goal is to obtain orbit. As the lower stages perform their function they are discarded. In some more modern systems the lower stages are recovered, completely refurbished and eventually (if all goes well) re-used.

Module development and testing works similarly. As the modules are debugged and eventually integrated into the whole, the testing baggage becomes obsolete. If a system is to be rehosted, the question arises: Can the modules testing code itself be re-used? When rehosting, one works backwards from the original development. The working system exists and runs correctly on one host. On the new host, however, it may not work. As outlined above, should the testing involve debugging the system as a whole, using a debugger, making changes to the individual modules, then recompiling and re-linking? Or, should an attempt be made to adapt the old module tests? This has some interesting issues associated with it:

- * Can the modules be moved with the source code onto the new host?
- * Should consideration be given to the transportability of the developed module tests?
- * Should the design and documentation of the system reflect the requirements and methods of rehosting the module tests and the data required for these tests?

Lastly, another approach is possible. After it has been determined that the system does not work on the new host break it down, stubbing out low level modules and developing brand new module tests. This implies a whole new technique for testing which is out of the scope of this document.

A.4 CONCLUSIONS

The AIM moved from the DG ADE into the VAX/VMS environment in 2.4 man-months. Most of the problems were due to:

- * compiler bugs,
- * inappropriate assumptions made with regard to the low-level models of terminal and process, control and communications,

INDEX

- Abort, 3-11
- Access control, 5-2, 5-7
- Accuracy, 3-4, 3-9
- Ada application developers, 1-8
- Ada compiler, A-5
- Ada language considerations, 2-3
- Ada librarian, A-5
- Ada linker, A-5
- Ada program developers, 1-8
- Ada source level debugger, A-5
- Ada-sw, 4-2
- Ade, A-6
- Aim, A-1
 - elements of the, A-3
- Ajpo, A-2
- Allocator, 3-5
- Alternatives, 3-11
- Apse, 1-1
 - conventional, 1-5
- Apse considerations, 4-6
- Apse dependencies
 - minimizing, 5-3
- Apse interactive monitor, A-1
- Archival systems, 1-6
- Arpanet, 4-2
- Assumptions
 - accuracy, 3-9
 - conditions under which
 - exceptions are raised, 3-11
 - during exception processing, 3-10
 - low-level models, A-21
 - low-level os models, A-1, A-13
 - number of priority levels, 3-5
 - order of activation of task objects, 3-10
 - order of evaluation in slices, 3-9
 - order of evaluation of bounds in a range constraint, 3-7
 - order of evaluation of each range of an index constraint, 3-8
 - order of processing of calls in a queue, 3-10
 - priority pragma, 3-5
 - safe numbers, 3-7
 - task aborting, 3-11
- Asynchronous
 - ipc files, A-16
- Asynchronous data, A-4
- Attributes
 - delta, 3-8
 - large, 3-8
 - machine, 3-6
 - mantissa, 3-8
 - representation, 3-12
 - size, 3-6
 - small, 3-8
- Bugs
 - compiler, 4-2, A-1, A-11, A-21
- Cais
 - bypassing the, 5-2 to 5-3
 - conformance, 5-1
 - definition, 5-1
 - issues, 5-1
 - transition to, 5-3
 - use conventions, 5-1
- Check, 3-6
- Colon, 3-4
- Communication
 - inter-process, 5-9
 - inter-tool, 5-2
- Compilation order, 4-5
- Compiler
 - ada, A-5
 - non-ansi, 4-3
- Compiler bugs, 4-2
- Compiler/linker/debugger, 1-6
- Complexity, 1-5
- Component, 1-5
- Components, 4-2
 - designing, 4-4
 - for purchase, 4-2
 - importing, 4-2
 - in software repositories, 4-4
 - risks associated with, 4-2
 - untested and unfamiliar, 4-3
- Computer language, 1-5
- Configuration management tools, 1-6, A-6
- Constrained type, 3-5
- Constraint
 - discriminant, 3-8
 - floating point, 3-8
 - index, 3-8
 - range, 3-4
- Constraint_error, 3-11, 4-5
- Control character, 3-7
- Conventional apse, 1-5

- what it promotes, 4-1
- Kit/kitia, 1-2, A-2
- Last, 3-4
- Length of
 - attributes, 3-2
 - expanded names, 3-2
 - identifiers, 3-2
 - labels, 3-2
 - qualified identifiers, 3-2
 - strings, 3-2
- Librarian
 - ada, A-5
- Linker
 - ada, A-5
- Long_float, 3-8
- Long_integer, 3-7
- Machine attributes, 3-6
- Machine code insertion, 3-6
- Machine dependent, 4-1
- Machine-dependent, 3-5, 3-12
- Main program, 3-11
- Memory_size, 3-12
- Milnet, 4-2
- Model numbers, 3-5, 3-9
 - ltiple body implementations, 4-5
- Natural, 3-7
- New_line, 3-12
- New_page, 3-12
- Non-graphic characters, 3-7
- Number of
 - attributes, 3-2
 - case alternatives, 3-2
 - compilation units, 3-2
 - declarations, 3-2
 - declarations on a block, 3-2
 - elements in an array, 3-2
 - elsif alternatives, 3-2
 - enumeration values, 3-2
 - explicit exceptions, 3-2
 - functions, 3-2
 - identifiers, 3-2
 - library units, 3-2
 - literals, 3-2
 - objects, 3-2
 - operators, 3-2
 - package names in a use clause, 3-2
 - package names in a with clause, 3-2
 - priorities, 3-2
 - record components, 3-2
 - statements, 3-2
 - statements in
 - sequence-of-statements, 3-2
 - type declarations, 3-2
 - variant parts, 3-2
- Number sign, 3-4
- Numeric_error, 3-4, 3-11
- Onion skin model of
 - transportability, 1-5
- Operating system, 5-1, A-6
- Operating system dependencies, A-7
- Operating system facilities, 4-3
- Operating system functions, A-1
- Operating system interfaces, A-2
- Operating system services, A-1
- Pad, A-2
- Parameter, 3-10 to 3-11
- Pdl, 4-5
- Performance analysis tools, 1-6
- Physical rehost, A-6
- Positive, 3-7
- Pragma
 - implementation defined, 3-7
 - interface, 3-6
 - priority, 3-5
 - shared, 3-5
 - suppress, 3-11
- Pragmatic language
 - recommendations, 3-2
- Priorities, 3-11
- Priority, 3-5
- Process
 - cais, 5-9, 5-11
 - communication protocols, 5-9
 - host operating system, 5-1
- Program design, 3-12
- Program structure, 3-12
- Program_error, 3-6, 3-11
- Project management tools, 1-6, A-5
- Project performance, 1-4
- Project proposals, 1-4
- Protocol
 - configuration management, 5-4
 - process communication, 5-9
 - tool, 5-1
- Rehost procedures, A-6
- Renaming, 3-5
- Repository, 4-2, 4-4
- Representation attributes, 3-12

techniques, A-1
things that can hinder, 1-3
tool, 5-1
tools, A-2
tools that enhance, A-6
who can benefit from, 1-8
Unchecked_conversion, 3-12

Uses of the guide, 1-7

Vertical bar, 3-4
Viewport, A-2
Vms, A-6

Window, A-2
Work-arounds, 4-5