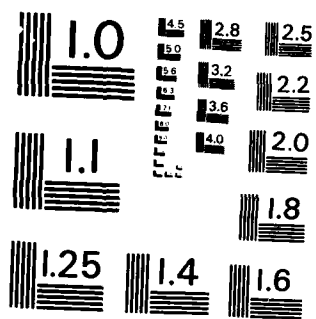


1/6

NL

[illegible]



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

12

NOSC TD 552

AD A 123 136

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM: PUBLIC REPORT

VOLUME II

Patricia Oberndorf, KIT Chairman
Naval Ocean Systems Center
San Diego CA 92152

28 October 1982

Interim Report for 1 April 1982-28 October 1982

Prepared for
ADA JOINT PROGRAM OFFICE
801 N Randolph St
Ballston Towers 2, Suite 1210
Arlington VA 22209

DTIC
ELITE
JAN 5 1983
A

Approved for public release; distribution unlimited

83 01 05 008

DTIC FILE COPY



NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

JM PATTON, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

This report is the second in a series consisting of inputs from the KAPSE Interface Team and its auxiliary industry/academia team. Readers of Volume II will note that a new cover design has been employed to replace the standard Naval Ocean Systems Center cover of Volume I. This change was made to reflect the DOD-level nature of the KAPSE Interface Team and its activities. The work was sponsored by the Ada Joint Program Office under program element RDAF, project CS22, sponsor order AF0038AJPO-82-2. The contributions are reproduced here exactly as received.

Released by
R. A. Wasilausky, Head
C³I Support Systems
Engineering Division

Under authority of
V. J. Monteleon, Head
Command, Control, Communications
and Intelligence Systems Department

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC Technical Document 552 (TD 552)	2. GOVT ACCESSION NO. ADA 123136	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE) INTERFACE TEAM: PUBLIC REPORT Volume II	5. TYPE OF REPORT & PERIOD COVERED 1 April - 28 October 1982	
	6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) KAPSE Interface Team Patricia A. Oberndorf (NOSC), Chairman	8. CONTRACT OR GRANT NUMBER(s)	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Ocean Systems Center San Diego CA 92152	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS RDAF, CS22, sponsor order AF0038AJPO-82-2	
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 801 N. Randolph St Ballston Towers 2, Suite 1210 Arlington VA 22209	12. REPORT DATE 28 October 1982	
	13. NUMBER OF PAGES 265	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Computer language Ada Interface standards Programming support systems		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The continuing activities of the Kernel Ada Programming Support Environments (KAPSE) interface team and its industry/academia auxiliary are reported. (Ada is a recent, DOD-developed programming language.) The Ada Joint Program Office (AJPO)-sponsored effort will ensure the interoperability and transportability of tools and data bases among different KAPSE implementations. The effort is the result of a Memorandum of Agreement (MOA) among the three services directing the establishment of an evaluation team, chaired by the Navy, to identify and establish KAPSE interface standards. As with previous ADA-related developments, the widest possible participation is being encouraged to create a broad base of experience and acceptance in industry, academia, and the DOD.		

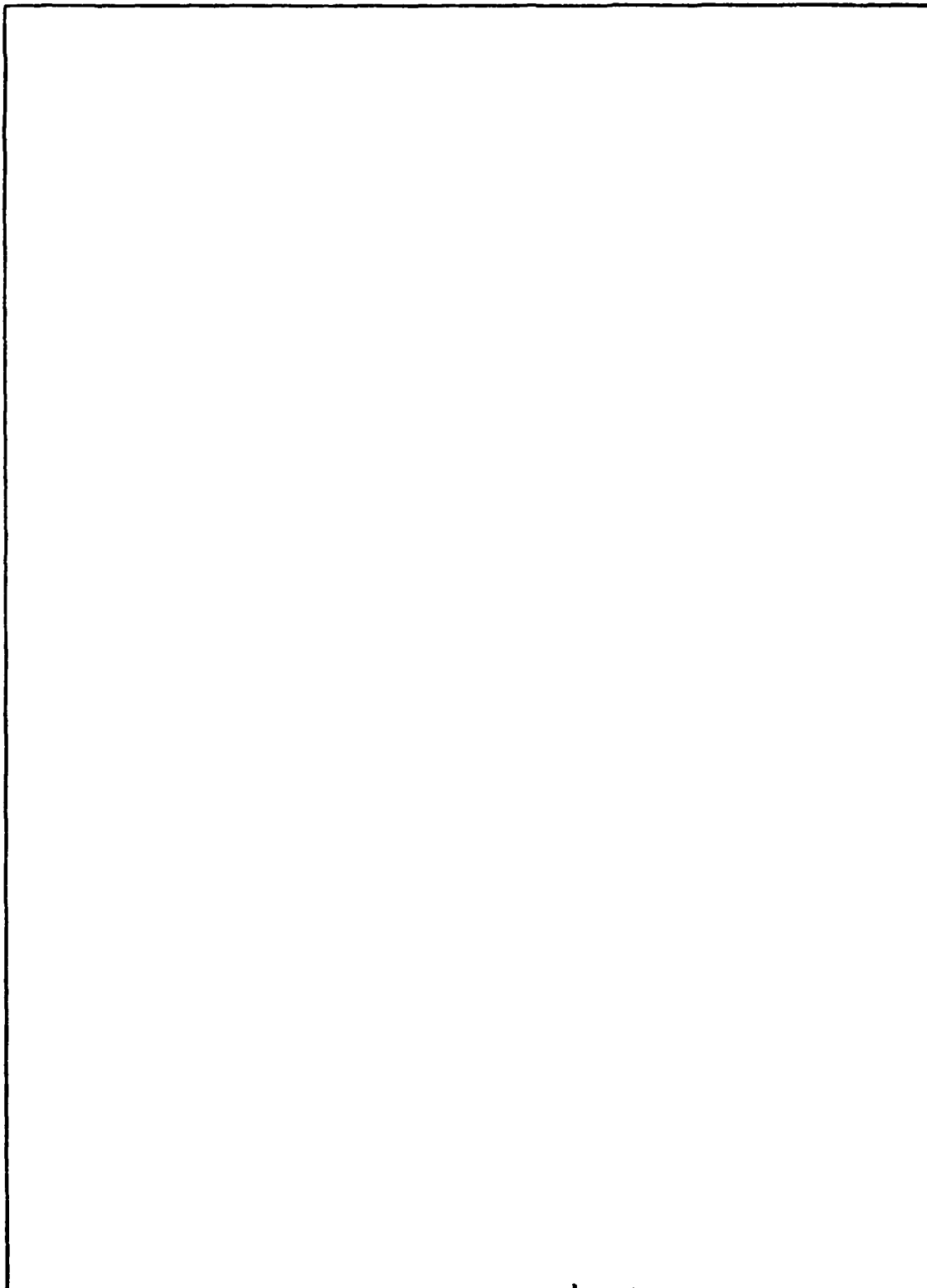
DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 68 IS OBSOLETE
S N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



S N 0102- LF- 014- 6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

CONTENTS

I.	INTRODUCTION	1-1
II.	TEAM PROCEEDINGS	
A.	KIT Minutes 20-21 April 1982	2A-1
B.	KIT Minutes 27-29 July 1982	2B-1
C.	KIT Meeting Proceedings 4-5 October 1982	2C-1
D.	KITIA Minutes 28-30 April 1982	2D-1
E.	KITIA Minutes 18-20 June 1982	2E-1
F.	KITIA Meeting Proceedings 4-5 October 1982	2F-1
G.	Minutes of Meeting KITIA Working Group One 10-11 August 1982 . . .	2G-1
H.	Reference Message	2H-1
III.	KIT/KITIA DOCUMENTATION	
A.	Memorandum of Agreement	3A-1
B.	Ada Programming Support Environment Interoperability and Transportability Plan	3B-1
C.	Original I & T Schedule	3C-1
D.	Revised I & T Schedule	3D-1
E.	Terms	3E-1
F.	KAPSE Interface Worksheet	3F-1
G.	Draft Requirements and Criteria	3G-1
H.	Notes on Stoneman Refinement	3H-1
I.	Interface Analysis of the Ada Integrated Environment and the Ada Language System	3I-1
J.	KITIA Charters	3J-1
K.	Towards a KAPSE Interface Standard	3K-1
L.	Program Invocation and Control	3L-1
M.	Specifying KAPSE Interface Semantics	3M-1
N.	A Machine Architecture for Ada	3N-1
O.	Validation in Ada Programming Support Environments	3O-1
P.	Overview of Current Work of the Database Interface Working Group	3P-1
Q.	KITIA Data Base Group (#2) Time Line Analysis of KAPSE Interfaces During a Compilation	3Q-1
R.	The Need for a Revision of the Stoneman KAPSE Concept	3R-1
S.	KAPSE Semantics and the Layered KAPSE	3S-1
T.	Introduction (KITIA GP 3)	3T-1

CONTENTS (CONTINUED)

U.	The Adaptation of VAX VMS System Services to KAPSE to Accommodate Transportability of Tools	3U-1
V.	KAPSE Model of VMS System Services	3V-1
W.	Outline for Ada Interoperability and Transportability Guidelines	3W-1
X.	Ada/APSE Portability	3X-1
Y.	New Category - - Extensibility	3Y-1
Z.	KAPSE Interface Category G.1 Debugger Support (RTS)	3Z-1
AA.	The Consequences of Multiple DoD KAPSE Efforts	3AA-1
BB.	KAPSE Services - Expanded Outline	3BB-1
CC.	An Executive Summary of the TOPS - 20 Operating System Calls	3CC-1
DD.	KAPSE Interface Team Industry and Academia (KITIA GP 4)	3DD-1
EE.	APSE Support for Targets	3EE-1
FF.	APSE Recovery Mechanisms	3FF-1
GG.	APSE Extensibility	3GG-1
HH.	Ada/APSE Program Policy Issues	3HH-1
II.	Group IV Future Plans	3II-1
JJ.	Computer Simulation and the Ada Environment	3JJ-1
APPENDIX A		A-1

SECTION I

INTRODUCTION



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Avail and/or	
Dist	Special
A	

I. INTRODUCTION

This report is the second in a series that is being published by the KAPSE Interface Team (KIT). The first was published as a Naval Ocean Systems Center (NOSC) report, TD-509, dated April 1982 and is now available through the National Technical Information Service (NTIS) for \$19.50 hardcopy or \$4.00 microfiche; ask for order number AD A115 590.

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. These reports are issued approximately every six months. 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 point in time will be included. By the same token, they also represent evolving ideas, so the contents should not be taken as fixed or final.

1. The KIT and KITIA

As the first report conveyed, the KIT was organized in three services and the higher levels of the DoD. This MOA is included in the first appendix of this report and serves as a charter for the KIT. The KITIA was subsequently organized to provide a broad range of industry and academia expertise which could help guide the KAPSE interface standardization effort. Since its first meeting in February, the KITIA has established its own charter, which is also included in this report. In addition, both teams are organized into four working groups, and the KITIA working groups have developed individual charters which are included following the general KITIA charter.

The KIT is a DoD team with members from Navy laboratories and agencies, the Army, the Air Force and the National Security Agency. Its membership is shown in the second appendix to this report.

The KITIA is a wide-ranging team whose members come from all across the United States and even from Europe. Its membership and geographical distribution are also given in the first appendix to this report.

The KITIA is establishing operating rules and procedures which will be published in the next report. Important aspects of these rules are:

- . the membership is limited to 30 organizations
- . organizations, not individuals, hold membership, so if an individual leaves a participating company or university, it is that organization's responsibility to replace him/her with another qualified representative
- . if a member organization defaults on its membership, the KITIA and the AJPO will work together to determine a qualified replacement
- . the KITIA elects its own officers (chairman, vice-chairman and working group chairmen); it relies on the KIT for secretarial support
- . KITIA meetings are not open to the general public; any requests to attend must be made to the KITIA chairman.

The KIT has not formally established operating rules, but it operates in a manner not unlike other DoD teams.

Any questions concerning membership on these teams or information about meeting attendance should be directed to

the chairmen: Tricia Oberndorf, NOSC, for the KIT or Edgar Sibley, Alpha Omega Group, for the KITIA.

2. Meetings

Both the KIT and the KITIA have now held four meetings. The KIT meetings were in January, April, July and October of 1982; the KITIA meetings were in February, April, June and October. This schedule included the first joint meeting of the two teams on 5 October 1982. This joint meeting was found to be very useful, and more joint meetings are planned for the future.

The approved minutes from most of these meetings are included in this report. Since the teams will not approve the October minutes until their next meetings, only summaries of the October meetings are included here. In addition to meetings of the full teams, working groups hold separate meetings as they are needed; this is especially true of the KITIA working groups. The minutes from one such meeting - of KITIA Working Group I in Blacksburg, Virginia - are also included here.

The next planned KIT meeting will take place in January in San Diego; the next KITIA meeting will be in February, also in San Diego, to coincide with the February AdaTEC meeting.

3. ADATEC Sessions

These public reports are only one way in which the KIT and KITIA will be conveying their work to the public. Another avenue is to make presentations at AdaTEC and other similar conferences. The first such session was held in October; it was an AdaTEC panel session consisting of members of both the KIT and KITIA. We reported on general status and progress, the results of some preliminary analyses of similarities and differences between the Army's Ada Language System (ALS) and the Air Force's Ada Integrated

Environment (AIE) and some of the policy issues which have been discussed during team meetings. The reports were followed by a general question and answer session. It is the intent that such sessions, together with these reports, serve to expose the KIT and KITIA efforts to public scrutiny and that they result in feedback from all those who wish to share ideas and opinions with the members of the teams. We strongly urge anyone who has feedback to convey it to either of the team chairmen or to any team member.

4. The APSE Interoperability and Transportability (IT) Plan

The original APSE IT Plan was issued in December 1981. Since then changes in thinking and schedule have occurred which warrant a revision to that plan. This is being pursued now, and a new plan will be released in December and included in the April 1983 Public Report. The original plan is included here, along with anticipated revisions in the schedule for 1983 through 1985.

5. Definitions and Categories

The first public report included some initial definitions which the KIT and KITIA had established. Subsequently, more relevant terms were defined, and they are included in this report.

The first report also included an initial list of interface categories and some explanations of them. These categories have since undergone considerable revision, and the latest KAPSE Interface Worksheets (KIWs) are included here. These changes have not resulted in any changes of the working groups of the two teams, which are organized according to the grouping of the categories shown in this report.

The revisions include some new categories, the removal of some others and changes in scope of others. The most notable of these are discussed below.

A. SMAPSE

The idea of a Standard or Subset MAPSE (SMAPSE) emerged from a meeting held last July in Bernreid, W. Germany. This meeting brought together a large number of people with interest in the various interfaces which will be found in an APSE. A similar conference in 1981 resulted in the creation of DIANA; this conference expanded the horizon beyond just the intermediate language interface and included other interfaces of interest to the KIT and KITIA. The SMAPSE idea may be critical to the achievement of the KIT/KITIA goals of interoperability and transportability between APSEs, so it is included now as a category. The SMAPSE idea is further developed in the paper by Erhard Ploedereder, included here with the KITIA papers. In response to the SMAPSE idea and Erhard's paper, the KIT has begun consideration of what changes need to be made to STONEMAN in light of the experiences of the last two years. This is discussed in the paper by D. Milton.

B. Extensibility

It has been a consistent notion that a MAPSE would evolve into an APSE and that APSEs could grow almost indefinitely to support system development. Therefore it was decided that one critical area of KIT/KITIA concern is how that notion of extensibility affects the establishment of KAPSE and other interface standards. There are at least two aspects to this issue, as can be seen from the Extensibility KIW and the papers by T. Standish and J. Ruby. One is how one provides in a standard interface for the evolution and extension of the standard interface set itself. The other is how one provides for the orderly and consistent addition of new tools as the MAPSE evolves into an APSE and that APSE grows.

C. Pragmas and Other Tool Controls

As the categories in Group II evolved, it became clear

that concerns about Pragmas and other similar means for one tool to control another more properly belonged in the data interfaces group. Therefore, this category was eliminated and all issues concerning it should now appear in the Group II KIWs.

D. Group II Categories

The categories in Group II have been revised. All of the issues should still be covered, but their placement has been reorganized.

E. Debugging

A new category for Debugging was considered. It was decided that, although debugging does place a number of unique requirements on the KAPSE interfaces, it is not appropriate to create a new category for each such new capability. Instead, all of the concerns raised by debugging issues should be covered under the categories which are affected. These appear primarily in Groups I and III.

F. Binding

Considerable debate has arisen from the inclusion of a Binding category. This seems in many ways to be a category that is somehow different from most of the others. The results of this debate will be reflected in future revisions to the Binding KIW.

The KIWs included in this report are not yet complete, nor have they been completely considered by members of both teams. They do represent a developing convergence, however, and those parts which are complete are probably nearing stability.

6. AIE/ALS Analysis

The study of the current DoD MAPSE designs, as well as those of the United Kingdom (UK), West Germany and the European Economic Community (EEC), is of great interest to the KIT and KITIA. They not only provide preliminary guidelines for what interfaces must be included, but they are also invaluable sources of experience with the interface decisions their designers have made. The DoD MAPSE designs are particularly important because it is desirable that the new standard interface set not differ from them arbitrarily, as that would increase the expense to the DoD. Therefore, an attempt has been in progress to analyze the various interfaces and features of the AIE and the ALS in order to determine the ways in which they are similar and the ways in which they differ. This analysis has been conducted both through documentation review and through visits with the contractors. It must be mentioned here that the cooperation of the Army and the Air Force and particularly of the two contractors (SofTech and Intermetrics, respectively) has been outstanding and without it the KIT and KITIA would not have been able to make a fraction of the progress that has been realized.

This desire to study the AIE and ALS was given an added impetus when the AJPO requested that an early attempt be made to determine ways in which the two MAPSEs could be brought into greater agreement. The result of this request is a series of meetings between members of the KIT Executive Committee (KITEC) and representatives of SofTech and Intermetrics. The meetings to date have been extremely productive and more are expected to take place throughout the winter. A first publication of potential standard interfaces on which the AIE and the ALS could agree is expected in the spring or summer of 1983.

The struggles and results of the analysis effort to date are given in a paper in this report.

7. Requirements and Criteria

This summer work was initiated to establish requirements and criteria which could be used in deciding on which interfaces to standardize and how to accomplish that standardization. The report included here is the very first tentative collection of such requirements and criteria. The format and content were modelled after those found in a series of requirements documents issued by the ANSI committee X3H1 on Operating System Command and Response Language; they are document numbers X3H1/05-SD, X3H1/06-SD and X3H1/07-SD. It is the intent of the KIT and KITIA to similarly take advantage of other such standards work whenever that is appropriate.

The requirements and criteria found here originated from the KITEC. The initial attempts were circulated to the KIT and KITIA and the feedback received resulted in the versions considered at the KIT meeting in October. The KIT spent less than one full day reviewing and discussing these, and the KITIA has not yet had a chance for thorough review. Therefore, what appears here is very preliminary, particularly section 6.4 which has not been reviewed by either team. These requirements and criteria are included here not because they represent any KIT and/or KITIA consensus, but because the teams felt it was important to solicit early feedback concerning the direction which these requirements and criteria should take. Most of the issues involving these requirements and criteria are yet to be resolved, so feedback from the reader is strongly solicited.

8. Policy Discussions

Recommendations concerning AJPO policy and possible approaches to the KIT and KITIA standardization task have been a topic at virtually every meeting of the two teams. These topics were first raised at the February KITIA meeting, at which T. Lyons proposed the existence of seven basic alternatives for the AJPO. These are included in this report, along with the comments by D. Cornhill of the KITIA and H. Hart of the KIT. In addition D. Wrege and S.

Glaseman of the KITIA have been very concerned about this topic and have also generated papers which appear in this report. The KIT took up the issue by considering Lyon's original write-up and trying to expand it and revise it to become a more thorough analysis of the available options. At the October meeting, the KIT decided, however, that instead of trying to expand another's paper we should simply try to articulate the factors which affect the policy decision, to analyze those factors for pros and cons and to construct from that analysis one alternative which could be recommended by the KIT to the AJPO. No reflection of that work is contained in this report, but it is expected to be ready for publication in the April report.

In addition, the October KITIA meeting resulted in another form of a policy recommendation for the AJPO. This recommendation is reflected in the proposal by E. Sibley which is included in this report. The KIT and KITIA recommendations do not necessarily conflict nor are they particularly redundant, since they deal with different aspects of the problem. As they develop, their status with respect to one another will be clarified.

9. KITIA Papers

As any reader of the first Public Report knows, the KITIA has made substantial original contributions to the IT effort. Much of this contribution has taken the form of working papers and point papers which are used to clarify issues and suggested approaches and solutions. A large number are included in this report.

The KITIA has taken the initiative in two very important areas: policy (as reflected by the papers by Lyons, Wrege, Sibley and Glaseman) and KAPSE validation (as reflected by the paper by Lindquist, et al.). In addition, KITIA members have made considerable contributions in the areas of KAPSE interface semantics (see Freedman's paper), KAPSE security (see papers by Willman and Glaseman) and the

AIE/ALS analysis (see Fischer's Time-Line Analysis paper). Also included here are the minutes from a very productive KITIA Working Group I meeting in Blacksburg.

10. Other KIT/KITIA Activities

There are several KIT/KITIA activities which are not represented in this report by any papers.

A. IT Tools

As will be seen by reading the MOA, it is part of the KIT charter to sponsor the development of three or more tools which will run in the AIE and the ALS. The intent of these tools is that their development process will help the KIT and KITIA to better understand the issues and interfaces which affect IT, both in general and as they apply specifically to the two DoD MAPSE developments. The first of these tools will be a Configuration Management System by CSC. That work has produced a Program Performance Specification (PPS) and will proceed soon to tool design. The second contract for tools was initiated on 4 October 1982 and will result in the development of an APSE Interactive Monitor (AIM) by Texas Instruments. The "one or more" other tools will result from a competitive procurement. Development of the Request For Proposal (RFP) is underway, but no schedules have been established. An announcement will be made in the Commerce Business Daily when the RFP is ready.

B. ALS and AIE Public Reviews

It is the intention of the AJPO and the Army and the Air Force to continue the sequence of public reviews of the AIE and ALS. The KIT is responsible for coordinating these. As can be seen from the IT Plan, the goal is to conduct one review approximately every six months, alternating between the AIE and the ALS. The next scheduled review will be of the AIE. It will be a review of the new B-5 specifications

which are being produced by Intermetrics. It is expected that review copies will be available by mid-January to those who wish to participate. Reviewers will be asked to return their comments to Warren Loper at NOSC by mid-March so that a report on the review results can be released in May. Anyone wishing to participate in this review process should contact Warren (WLOPER@ECLB on the ARPANET or (714)225-2743).

Another review of ALS progress will be held approximately six months following the AIE review, depending on the availability of appropriate products for review. More details will be announced in the April report.

C. UK Study Reports and Other Efforts

The KIT and KITIA have acquired several copies of the UK Study Reports which document that team's thinking concerning development of APSEs. It is the desire of the teams to avail themselves of all possible expertise, so these study reports and the presence on the KITIA of representatives from the UK and W. Germany are most welcome. In addition, members from the EEC involved in that development attended the October KITIA meeting.

11. 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 subsequent reports are encouraged. They should be addressed to:

Patricia Oberndorf
Code 8322
NOSC
San Diego, CA 92152

or sent via ARPANET to POBERNDORF@ECLB.

I would also like to extend my appreciation to the many DoD, academic and commercial activities whose continued support make this effort possible. The sense of teamwork and cooperation displayed by all members of these two teams are outstanding and will mean the success of what we have undertaken.

SECTION II

TEAM PROCEEDINGS

KIT Minutes
Meeting of 20-21 April 1982
Naval Avionics Center
Indianapolis, Indiana

Attendees : See Appendix A

Bibliography of Handouts : See Appendix B

20 April 1982

1. Opening Remarks

The KIT chairman, Tricia Oberndorf, brought the meeting to order. Members introduced themselves and new members were asked to participate in the KAPSE Interface working group of their choice. The revised working groups are shown in Appendix C.

Tricia announced the first public report was in printing and would be made available through the National Technical Information Service (NTIS). The Ada Joint Program Office (Ada is a registered trademark of the Ada Joint Program Office (AJPO) of the U.S. Government) program review meeting indicated the following current milestones :

- Army APSE, the initial release of the ALS will be in January 1983.
- Air Force APSE, the initial release of the AIE will be in November 1984
- Navy APSE, the RFPI is due July 1982

A schedule of the public review of the ALS and AIE systems has been set. The reviews will be at six month intervals and will alternate between systems. The first report will be on the ALS system and will be published in June. The first public review of the AIE system will be published in December 1982.

The Industrial/Academia Team has decided to organize itself along the same lines as the KIT. They will be charged with producing the IT requirements for use by the KIT.

2. Discussion of Mailing and KIT Objectives

Because of the changes in the ALS documentation, all members of the KIT will receive updated issues of the ALS documentation. The AIE documentation will also be distributed. Several members of the KIT expressed their frustration with the ARPANET and voiced their need for a Primer to accelerate use of the system. Tricia will distribute a copy of the ARPANET Primer to the KIT members who request it.

Tricia went over the APSE IT plan and KIT objectives. Multiple APSE tools will be developed to test Interoperability and Transportability between the AIE and ALS systems. The KIT will monitor the AIE and ALS development effort with respect to APSE Interoperability and Transportability. The KIT will be the arena to address issues of the APSE IT brought to its attention. The KIT will develop procedures to determine compliance of APSE developments with APSE IT requirements, guidelines, conventions, and standards.

3. Definitions

The KIT developed the definition of the following terms as they apply to IT.

- * Specifications
- * Requirements Specification
- * Guidelines
- * Conventions
- * Standards

The strawman definitions used to initiate the discussion were distributed before the discussion.

4. AIE/ALS Interface Matrix

Charlie Forrest (TRW) presented a comparison of the AIE and ALS system interfaces derived from the available documentation. The Softech and Intermetrics representatives strongly objected to the model used in the comparison. Also the differences in documentation purposes - i.e., one as a sales tool (AIE) and the other to document a developing system (ALS) - prevented a clear comparison. It was determined a meeting between Softech, Intermetrics and Tricia and TRW would be required to provide the most appropriate model for the comparison effort.

5. Guidelines, Conventions and Standards Documentation Format

Tricia and George Robertson (TRW) discussed the alternative formats the KIT may use in documenting the guidelines, conventions, and standards for IT. Further action on the KIT documentation format was deferred until the July KIT meeting.

6. Policy

Tricia led a discussion of the KIT's policy in the development of APSE IT standards. The options developed by Tim Lyons of the Industry/ Academia team were discussed. These options are listed in Appendix E. The consensus of the KIT was that an insufficient understanding of IT requirements and the unknown cost impact of the alternatives prevented a decision from being made.

Adjourned for the day.

21 April 1982

7. Working Group Strategy

Tricia opened the second day with a discussion of the direction the working groups should take in their individual meetings. The following items should be looked at by each group to fill in the interface category work sheets.

- * Review the existing Interface Category worksheet for more detailed description
- * Review the OSCRL requirements for items that have an application to the Interface Category
- * Review the AIE/ALS interface matrix for items that have an application to the Interface Category
- * Expand key issues section to explain the issues of IT that apply to the Interface Category

Working groups then met to work on expanding worksheets.

8. Closing Discussions

Approval of the minutes was accomplished with specific changes to be added before release. The San Diego KIT meeting was tentatively scheduled for 27-29 July 1982. Working groups were to continue discussions after the close of the KIT meeting.

Meeting was closed.

Appendix A - Attendees

Richard Baldwin	U.S. Army CECCM
Jinny Castor	U.S.A.F./AFWAL
Bob Converse	NAVSEA/PME-408
Edward Dudash	NSWC/DL
John Foreman	Texas Instruments
Charlie Forrest	TRW
Hal Hart	TRW
Ron House	NUSC
Larry Johnston	NADC
Larry Lindley	NAC
Warren Loper	NOSC
Donn Milton	CSC
Tricia Oberndorf	NOSC
Shirley Peele	FCDSSA
Lee Purrier	FCDSSA
George Robertson	TRW
Mike Ryer	Intermetrics, Inc.
Barbara Santanelli	U.S. Army CECCM
S. Tucker Taft	Intermetrics, Inc.

Guy Taylor

FCDSSA

Richard Thall

SofTech, Inc.

Elizabeth Wald

NRL

Chuck Waltrip

Johns Hopkins University

Douglas White

RADC/COES

Appendix B

Bibliography of Handouts

1. ALS/AIE Comparison Viewgraphs and References
2. Definitions of Requirement, Requirements Specification, Guideline, Convention, and Standard
3. KIT Members Address List
4. KITIA Options or End Product - Tim Lyons
5. Minutes of 19-20 January KIT Meeting
6. Proposed Session for National AdaTEC in October

Appendix C

Revised Working Groups Assignments

Group I KAPSE User Support

Group Leader: John Foreman

Members: Jack Kramer
Larry Johnston

Group II Data Interfaces

Group Leader: Donn Milton

Members: Elizabeth Wald
Ed Dudash
Lee Purrier
Chuck Waltrip

Group III KAPSE Service Interfaces

Group Leader: Charlie Forrest

Members: Warren Loper
Ron House
Larry Lindley

Group IV Miscellaneous

Group Leader: Hal Hart

Members: Guy Taylor
Bob Converse
Jinny Castor
Shirley Peele

KIT Minutes
Meeting 27-29 July 1982
Naval Ocean Systems Center
San Diego, California

Attendees: See Appendix A

Bibliography of Handouts: See Appendix B

27 July 1982

1. Opening Remarks

The KIT Chairman, Tricia Oberndorf, brought the meeting to order. Members introduced themselves and new members were asked to participate in the KAPSE Interface Working Groups of their choice. The revised Working Groups are shown in Appendix C.

KITIA - Representatives from the KITIA met with the AIE and ALS developers to gather additional data on these implementations.

CSC SOW - The Statement of Work for the Configuration Management tool has been sent out. Appendices A and B are available by request (due to size).

German and British status - A briefing was made to the KITIA on the status of the German system. The British studies have been ordered and distribution is planned for each Group leader after arrival.

AJPO status - The new Ada Language Reference Manual has been distributed. ANSI will submit Ada for ISO approval. European Economic Community is strongly supporting Ada. NATO is reviewing Ada as their standard for NATO sponsored Command and Control developments. A Tri-Services Program Review is scheduled for August. The education work is in progress.

AIE/ALS Analysis - the analysis of the interfaces of the AIE and ALS systems is continuing. A coordination meeting with the developers is planned to occur during this KIT meeting.

2. April Minutes

The minutes of the April meeting were approved as submitted.

3. Agenda/Objectives Discussion

The planned agenda was reviewed with a discussion of the objectives of this meeting defined. The agenda will be modified to support the needs of the team.

4. Break for Working Group Meetings

The working groups met to discuss the KAPSE Interface Worksheets and to suggest revisions to the same.

5. The working groups requested additional time to focus more attention on the KIWs. Additional working sessions were held.

28 July 1982

6. General Discussion of KIW's and Standards Development

A general discussion of the methodology for KIWs was held. It was suggested that the KAPSE B5 specs from the AIE and ALS developments may be a good source of material for consideration. This was suggested as a starting point and not as a definition of the only capabilities to be considered.

7. The status of the ALS documentation was discussed. KIT copies delayed but should be forthcoming.

8. The results of the poll for consideration of new categories for KIWs resulted in addition of Command Language, Debugger, and Extensibility with assignments of the first two to Group 1 and Extensibility to Group 4.

9. KIW Working Group Meetings

The Groups broke into cross-group meetings to discuss impacts of their various categories on other group categories. The result of these meetings was that each group would require additional time to consider other data in the finalization of their specific categories.

10. AIE Review Schedule

The schedule of the AIE documentation review was discussed. The documents would probably be available for public review in the late September time frame. This would call for responses to NOSC by the KIT by 1 Dec. with a report from NOSC about 1 Feb 1983. It was pointed out that the government should solicit industry support since their management was reluctant to fund this work if not recognized. A discussion of the value of previous reviews followed.

11. Intermetrics - ALS Tool transfer to AIE (AJPO Request)

The results of the Intermetrics response to AJPO is still in progress. This is still under review by the technical personnel. This could provide a valuable contribution to the work of the KIT.

12. Donn Milton reviewed the events of the German conference. These included point-by-point reviews of the STONEMAN document which included:

- o database management in the KAPSE
- o levels of transportability are not defined
- o KAPSE as a minimum operating system versus a portability interface
- o relationship of object and reconstruction
- o definition of the KAPSE database
- o confusion in run-time support area
- o pretty printer in MAPSE versus tool
- o host loader in the KAPSE versus MAPSE
- o SMAPSE

Jack Kramer of the AJPO indicated the KIT should worry about the KAPSE interfaces that are necessary to support extension. The SMAPSE interfaces could be considered within the scope of the KIT but the front end tools are not specified as yet. This may fall under the role of the Software Initiative now in discussion.

13. C. Forrest presented a review of the policy issues that the KIT could consider. These are to be published in the October Public Report. The option to review the current AIE/ALS systems to identify those interfaces that could be standardized now followed by a definition of what is missing with a later revision and unification effort is the current KIT course of action.

29 July 1982

14. Address List Corrections

Correction for the KIT address list were solicited for inclusion in the master list.

15. Category Rationales

A general discussion to review the rationales for the various categories and responsible groups was conducted.

16. DoD Session

A session of the AJPO, KIT chairman, and Group Leaders was held to discuss plans for expansion of the KIWs and generation of the Requirements and Criteria document. The KIW vice-chairmen met with their groups to discuss coordination of inputs to the next revision of the KIWs. A coordination meeting with the AIE and ALS developers was held to define terminology differences between the two implementations.

17. Requirements and Criteria Session

A general discussion of the Requirements and Criteria document was held. The intention is to have an initial draft by January 1983. H. Hart presented material from the Operating System Command and Response Language (OSCRL) documentation was presented for consideration. G. Robertson presented some ideas for the form the document may take.

18. The KIT Plans for October were discussed including:

- Public Report
- KIWs
- Policy write-up
- Req & Criteria write-up
- RFP for Third Tool
- Define January Kit Meeting

19. Closing Discussions

The Washington KIT meeting was tentatively scheduled for 4-5 October.
This is the week of the National AdaTEC meeting in D.C..

APPENDIX A ATTENDEES

Richard Baldwin	U.S. Army CECOM
Jinny Castor	U.S.A.F. AFWAL/AAAF
Edward Dudash	NSWC DL
Jack Foidl	TRW
John Foreman	Texas Instruments
Charlie Forrest	TRW
Hal Hart	TRW
Ron House	NOSC

Larry Johnston	NADC
Elizabeth Kean	RADC COES
Richard Kubischta	TRW
Jack Kramer	AJPO
Larry Lindley	NAC
Warren Loper	NOSC
Jim Moloney	Intermetrics
Gilbert Meyers	NOSC
Donn Milton	CSC
Eldred Nelson	TRW
Tricia Oberndorf	NOSC
George Robertson	FCDSSA SD
Carl Russ	FCDSSA DN
Barbara Santanelli	U.S. Army CECOM
Maurice Stein	NSWC DL
Tucker Taft	Intermetrics
Guy Taylor	FCDSSA DN
Richard Thall	SofTech

Elizabeth Wald

NRL

Chuck Waltrip

John Hopkins University

Douglas White

RADC COES

APPENDIX B
MEETING HANDOUTS

1. A map of the locations of the 30 KITIA members.
2. Viewgraphs on "Activities of IAT KAPSE Services Group (Grp 3)".
3. "Some Initial Thoughts on APSE Security" by Dr. S. Glaseman.
4. Minutes from the February KITIA meeting.
5. "KAPSE Semantics and the Layered KAPSE" by D. E. Wrege.
6. A memorandum from KITIA Group 3 Chairperson Sabina Saib on KAPSE Services Group Responsibilities and Meeting.
7. The KITIA Group 4 Charter.
8. A KITIA Group 4 paper on "Computer Simulation and the Ada Environment".
9. A KITIA Group 1 paper on "A Machine Architecture for Ada" by Pekka Lahtinen.
10. "Validation Methods for the KAPSE Interface" by Dr. T. E. Lindquist, et al.

11. Viewgraphs on "Security-Classification" by Herb Willman.
12. "Draft Specification for APSE Security-Classification-Attribute" by Herb Willman.
13. "APSE Formal Definition" by R. S. Freedman.
14. A KITIA Working Group 1 interim technical note (WG.1-A002) on Program Invocation and Control (the KITIA logo on the cover has been made into a sticker and is available).
15. A draft WG.1 charter from H. R. ("Dit") Morse.
16. An ARPANET message from KERNER presenting the KITIA Data Interfaces Working Group (Grp 2) Charter.
17. Minutes from the KITIA April meeting.
18. The KITIA Charter dated 4-30-82.
19. "Ada Package Specifications for Ada Language System KAPSE" dated 6-23-82.

KIT MEETING PROCEEDINGS

4-5 OCTOBER 1982

Hyatt Crystal City

MONDAY, 4 October

0830 Tricia Oberndorf (NOSC)
Brought the meeting to order.

General announcements were made concerning:

- o introduction of new members
- o plans for the AdaTEC panel session later in the week
- o KITEC progress since the July meeting
- o the meeting the previous week with SofTech and Intermetrics
- o progress on the IT tools (CSC, TI, the RFP)
- o requests for the Ada tutorial materials
- o STONEMAN revisions (Donn Milton)
- o receipt of the UK study reports
- o anticipated contents of the October Public Report
- o work on a KIT logo
- o the new AIE review schedule

The July meeting minutes were approved as corrected.

0915 Working Group meetings

The progress on the KIWs was explained and particular areas needing the attention of the groups were pointed out; the groups then met separately.

1145 LUNCH

1300 Working Group meetings (continued)

1400 KIT reconvened

Reports were given by the four group chairmen on progress during the group meetings.

Requirements and Criteria Session

Hal Hart's and the KITEC's work to generate IT Requirements and Criteria using the OSCRL papers as a model; copies were handed out and discussion was opened to item-by-item consideration; numerous revisions were made.

1715

Adjourned for the day

TUESDAY, 5 October

0830 Requirements and Criteria Session (continued)

1045 Policy Rewrite Session

Discussion ensued concerning the KIT's attempts to further develop the 7 policy alternatives originally generated by Tim Lyons of the KITIA (see paper XXX in this report). It was decided that the KIT should extract the essential parameters from our work so far in this area and, using trade-offs concerning these, construct the KIT's recommended alternative.

1145 LUNCH

1300 Joint meeting of KIT and KITIA

Several announcements of interest to both teams were given by T. Oberndorf. Edgar Sibley, KITIA chairman, presented KITIA work on a proposal which the KITIA will make to the AJPO; discussion followed. LCOL. Larry Druffel of the AJPO addressed the joint meeting and answered questions. The teams broke down into joint meetings of the working groups. Upon reconvening, J. Foidl of TRW presented current progress on the AIE/ALS analysis (see paper YYY in this report); discussion followed.

1700 ADJOURNED

KITIA Minutes
Meeting of 28-30 April 1982
Washington, D.C.

Attendees : Appendix A
Bibliography of Handouts : Appendix B
Auxiliary Assignments : Appendix C

28 April 1982

1. Opening Remarks

Edgar Sibley, KITIA chairman, brought the meeting to order and introduced the keynote speaker - Larry Druffel of the Ada* Joint Program Office (AJPO). (Ada is a Registered Trademark of the Ada Joint Program Office - U.S. Government)

2. Keynote presentation

Larry Druffel discussed the importance of the KITIA in the development of standards for interoperability and transportability (IT). As users and implementers, the KITIA will play an important part in the review of the current APSE development and in the development of standards for IT.

An interim report comparing the Ada Language System (ALS) and Ada Integrated Environment (AIE) is being developed by Tucker Taft of Intermetrics. The interim report will cover conventions which will make it possible to transport tools to the AIE.

The charters of the KIT and KITIA were discussed. Larry Druffel stated the KIT charter is driven by the Memorandum of Agreement (MOA) signed by the assistant secretaries of the three services. A common set of tools in use by all three services is the goal of the MOA. The KIT is to define the interfaces so a common set of tools can be realized.

A serious discussion of the strategy for conforming to IT standards occurred. Larry Druffel explained what was being proposed to reach conformity and why. Several objections were raised to the development of more than one APSE. Edgar Sibley explained his reason for opening the meeting with an address by Larry Druffel was to establish the arena for development of the KITIA charter and the working group charters.

3. General Business Session

Several items of general business were discussed. These are encapsulated below.

- The name selected for the Industry and Academia team was KAPSE Interface Team - Industry and Academia (KITIA).
- Reservations for a meeting room for the October KITIA meeting will be made by Donn Milton. Edgar Sibley will make connection with Donn for this purpose.
- The United Kingdom (U.K.) reports were discussed. The AJPO will procure several copies and make these available to interested KITIA members on a temporary basis.
- The KIT will be making a panel presentation at the National AdaTEC meeting in October. Copies of the session proposal were distributed. Individuals were requested to make comments or requests for changes to the proposed agenda.

4. KIT Review

Tricia Oberndorf presented a review of the KIT meeting of 20-21 April 1982 in Indianapolis, Indiana. The KIT has developed a set of definitions for specification, requirements specification, guideline, convention and standard. The definition of requirement was tabled at the KIT meeting. A set of definitions for review will be sent on the ARPANET.

The form for the interface standards was discussed. The KIT is reviewing several examples of current forms of standards to determine the form the KIT standards will take.

Tricia Oberndorf stated a review schedule has been determined for review of the AIE and ALS systems. The next review will be of the AIE A-Spec and B-5 documentation to be released in the summer or fall of this year. The review schedule will be at six month intervals with the review alternating between the ALS and AIE systems. The discussion shifted to the documentation for both the AIE and ALS. A complete set of ALS documentation will be sent to all members of the KIT and KITIA. The current AIE documents will be sent only to the individuals who request them. The AIE documentation of the summer or fall of this year will be distributed whenever available to all members of the KIT and KITIA.

The KIT has started a comparison of the AIE and ALS to determine where the interfaces exist and to develop a model for comparison. TRW presented a first look at the comparison of the two APSE's at the Indianapolis KIT meeting. Based on the reaction of the two contractors, Tricia Oberndorf and Charlie Forrest will meet with Softech and Intermetrics in May to develop a model for comparison of the two systems.

5. Presentation of Working Group Charters

The charters for the working groups were presented by H.R. Morse (group 1), Judy Kerner (group 2), Ron Johnson (group 3), and Steve Glaseman (group 4). The efforts of each of the groups were covered including specific assignments of each individual within the working group.

6. Presentation of Potential Charters for KITIA

Tim Lyons and Tricia Oberndorf presented possible charters for the KITIA. A general discussion opened on the purpose of the KITIA in the process of developing standards for IT and the relationship of the KITIA to the KIT. The KIT is chartered to produce the formal standards for IT, and the KITIA is to support the work of the KIT by the means it feels is most appropriate (e.g., development of requirements for IT, development of the initial draft of the standards, review of the KIT effort, etc.).

7. General Information for KITIA

Tricia Oberndorf covered a potpourri of topics of general KITIA interest. These topics were :

Use of the ARPANET

Public Report

Letter of Appreciation

Navy APSE Development

Adjourned for the day.

29 April 1982

8. Presentation of Point Papers

Doug Wrege gave a presentation of the effects of multiple DOD environments. The presentation covered the problem of multiple APSE's, the results and consequences of having multiple APSE's, and several possible solutions to the problem. The discussion that followed developed more results and consequences of having multiple APSE's, and several more possible solutions.

Tim Lyons gave a presentation of a comparison of database facilities of APSE's. The presentation covered the use of a data model of the development process, the mapping of the data model onto the underlying system database, and a comparison of the current APSE databases. The current APSE databases were compared by use of a simple database storage illustration which showed the database relations. The presentation showed what part of the database the four APSE's have put at the KAPSE level.

Ann Reedy gave a presentation of an introduction to DIANA. The presentation covered the general case for an intermediate language, special reasons for using DIANA as the intermediate language, DIANA design principles, and an example of a simple program and its DIANA representation. Several points of controversy in the use of an intermediate language were also covered.

9. ARPANET Tutorial

Tricia Oberndorf gave a brief tutorial on the operation and use of ARPANET. ARPANET will be an important communication tool for use by the KIT and KITIA.

10. Working Group Sessions

A general discussion on the KITIA charter preceeded the working group meetings. The general discussion produced an outline for the KITIA charter to be filled in during the cross-group meetings.

The KITIA adjourned and successive cross-group meetings were held to determine the KITIA charter. The working groups adjourned and the general KITIA meeting resumed. It was decided a smaller group of members of the working groups would meet during the evening to develop a single KITIA charter. The group is listed below:

Herman Fischer
Steve Glaseman
Judy Kerner
Tim Lindquist
H. R. Morse
Tricia Oberndorf
Herb Willman

Adjourned for the day.

30 April 1982

11. General Session

General business items were discussed with the following results.

- The minutes of the meeting of 17-19 February 1982 were unanimously approved as corrected.
- Future meetings of the KITIA were set. The local host of the meetings will insure sufficient facilities and information are available to support the KITIA meetings.

Boston	18-20 June 1982
Washington, D.C.	4-6 October 1982
San Diego	12-14 January 1983

- The location of the KITIA meetings in principle will be variable with no specific sites assigned.
- Steve Glaseman was elected vice-chairman of the KITIA.
- The proposed KITIA charter was read and distributed. It was decided the working groups would review the KITIA charter and the comments would be returned by the working group chairman.

12. Working Group Session

The general session adjourned and the working groups met.

13. General Session

The working groups adjourned and the general group received a report on the work of the working groups. A discussion on the material to be covered with the ALS and AIE contractors in the June meeting produced an agenda. The agenda will be two four hour sessions - one session with each contractor. Within a session, there will be four working group sessions when only the designated working group will be allowed questions. Potential questions will be produced by the work-

ing groups. The questions will be sent to Edgar Sibley and LCDR Kramer by the working group chairmen.

The auxiliary assignments of the KITIA members were discussed. A list of the assignments is contained in Appendix C.

The KITIA expressed its appreciation to General Electric for the use of the facilities and thanked Dave McGonagle for his efforts to achieve a successful meeting.

Meeting adjourned.

Appendix A
Attendees
KITIA Meeting
28-30 April 1982

KITIA Members :

BAKER, Nicolas	McDonnell Douglas Astronautics (substitute for Eric Griesheimer)
CORNHILL, Dennis	Honeywell
COX, Fred	Georgia Institute of Technology
FISCHER, Herman	Litton Data Systems
FREEDMAN, Roy	Hazeltine Corporation
GARGARO, Anthony	Computer Sciences Corporation
GLASEMAN, Steve	Teledyne Systems Corporation
JOHNSON, Ron	Boeing Aerospace Corporation
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missles and Space
KRISHNASWAMY, R.	Ford Aerospace and (substitute for Larry Yelowitz) Communication
LAHTINEN, Pekka	Oy Softplan Ab (Finland)
LAMB, Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
LOCKE, Doug	IBM

LYONS, Tim	U.K. Ada Consortium
McGONAGLE, Dave	General Electric
MOONEY, Charles	Grumman Aerospace
MORSE, H. R.	Frey Federal Systems
NEAL, Larry	General Research Corporation (substitute for Sabina Saib)
REEDY, Ann	Planning Research Corporation
RUBY, Jim	Hughes Aircraft Corporation
SIBLEY, Edgar	Alpha Omega Group, Inc.
WESTERMANN, Rob	TNO-IBBC (the Netherlands)
WILLMAN, Herb	Raytheon
WREGE, Doug	Control Data Corporation

KITIA Members unable to attend :

FELLOWS, Jon	System Development Corporation
LOVEMAN, Dave	Massachusetts Computer Associates
PLOEDEREDER, Erhard	IABG (Germany)
STANDISH, Thomas	University of California at Irvine

Other Attendees :

DRUFFEL, Larry	AJPO
FORREST, Charles	TRW

KRAMER, Jack LCDR AJPO

OBERNDORF, Tricia NOSC

Appendix B
Bibliography of Handouts

I. Working Group Charters

- Group 1 (Draft)
- Group 2 (Draft)
- Group 3 (Draft)
- Group 4 (Draft)

II. Group 3 Activities Report

- A. A Review of VAX VMS System Services - Ron Johnson
- B. Debugger Support - Doug Wrege
- C. Expansion of KAPSE Services Outline From KIT - Herb Willman

III. Ada Development Methodologies - Peter Freeman

IV. Point Papers

- A. Effects of a Multiple DOD Environment - Doug Wrege
- B. An Informal Introduction to DIANA - Ann Reedy
- C. A Machine Architecture for Ada - Pekka Lahtinen

V. ARPANET Tutorial

VI. KITIA Charter

VII. KITIA Member List

VIII. KIT Member List

IX. First KITIA Meeting Minutes

X. KIT Presentation Agenda for October AdaTEC

Appendix C
Auxiliary Assignments

1. Create a model of existing OS's -

VAX/VMS		Ron Johnson
		Brian Koerble (NOSC)
MULTICS		Dennis Cornhill
UNIX		
TOPS-20		
IBM		
CDC		

2. Policy recommendations Douge Wrege
Steve Glaseman

3. Seven- item spectrum of alternatives Tim Lyons (complete)

4. STONEMAN evolution

5. Expression medium for standards

6. Tools/packages for porting

7. Universal vs. application-specific partitions

8. General architectural models

9. Pragmatic limitations ,..... Herb Willman

10. Development of the layered concept Tim Lindquist

KITIA Minutes
Meeting of 18-20 June 1982
Boston, Mass.

Attendees:	Appendix A
Bibliography of Handouts:	Appendix B
Meeting with Intermetrics:	Appendix C
Meeting with SofTech:	Appendix D

18 June 1982

1. Meeting with Intermetrics

The KITIA met with Intermetrics to discuss AIE design concerns of the working groups. Each working group covered the prepared questions (Appendix B, section I, A) during their specified time period. Tucker Taft of Intermetrics presented the AIE design to the working groups. (See Appendix C.)

2. Meeting with SofTech

The KITIA met with SofTech to discuss ALS design concerns of the working groups. Each working group covered the prepared questions (Appendix B, section I, A) during their specified time period. Rich Thall of SofTech led the SofTech group in a presentation of the ALS design to the working groups. (See Appendix D.)

19 June 1982

3. Individual Presentation

Herman Fischer presented a comparison of the databases in the ALS and AIE systems. The presentation covered the definition of database and file (object) names, the storage of objects in the ALS and AIE, ALS database and container operations, AIE database operations, comparison of ALS and AIE database operations, and the application of database requirements.

4. Review of SofTech and Intermetrics Meetings

The working groups presented a review of the meetings the previous day with SofTech and Intermetrics.

Working Group I - Harrison Morse
Working Group II - Judy Kerner
Working Group III - Eli Lamb
Working Group IV - Steve Glaseman

5. Working Group Sessions

The general group adjourned and the working groups met.

6. Working Group Reports

The general group met and the working groups made a progress report on their efforts to date.

Working Group I - Harrison Morse
Working Group II - Judy Kerner
Working Group III - Sabina Saib
Working Group IV - Steve Glaseman

Anthony Gargaro presented a KITIA logo and introduced a Working Group I interim paper. (See Appendix B, section II,C.)

7. General Session

A general discussion of the aim of the KITIA ensued. The discussion covered the need for a model of the KAPSE and the policy questions Tim Lyons has sent out via the Arpanet.

The agenda was modified to include several more presentations. The agenda development for future meeting was also discussed. The initial draft of the agenda from the chairman will be sent to the working group chairmen. The final agenda will be organized by the vice-chairman from the initial draft and the comments of the working group chairmen.

8. Individual Presentations

Dennis Cornhill gave a presentation on an approach to specification of the KAPSE interfaces for interoperability and transportability. The presentation covered the need to achieve a degree of IT and a degree of implementation freedom, the use of denotational semantics in the Ada language specification and the implications for the KAPSE interface specification, and examples of static and dynamic semantics in the Ada language specification.

Roy Freedman gave a presentation on an APSE Formal Definition. The presentation covered the description of interfaces and their definitions, KAPSE interface semantics(KIS), a denotational approach to interface description, the problems of interface description when multiple APSE's exist, and tool semantics.

Steve Glaseman gave a presentation of some initial thoughts on APSE Security. The presentation covered a review of data security, VAX/VMS security features, the ALS access control, and the AIE access control. Also covered in the presentation were the hazards associated with the APSE concept and fundamental questions regarding APSE security.

Herb Willman gave a presentation on Security-Classification. The presentation began with the three attributes Stoizman requires for database objects - history, category, and access rights. The presentation proposed a fourth attribute to cover the immediate problem of security, the security classification attribute. The presentation further developed the types and use of security information(i.e., internal and external types and use) and the uses of the proposed security classification. Included in the presentation was a draft specification for an APSE Security Classification attribute.

Tim Lindquist gave a presentation on a preliminary study of KAPSE interface validation. The presentation covered the need for a KAPSE and APSE validation technique, a proposed approach to a validation method, and the expected results of the approach. It was proposed to review the different techniques available for validation, the specific needs of an APSE and a KAPSE, and alternative approaches to KAPSE evolution. This review would produce a strawman for KAPSE validation which would be applied to an existing APSE. Finally the strawman would be used in the development of the KAPSE.

Erhard Ploedereder gave a presentation on the development of a programming support environment in Germany. The German Ministry of Defense is financing a project, Standardized Program Development System for the Armament Sector (SPERBER), which is developing a programming support environment. The programming support environment will support Ada and the German computer language PERL. The system will be hosted on a German machine and will be targetted to several German machines.

9. General Session

Tricia Oberndorf covered several topics concerning the KIT and KITIA. A synopsis is given below.

- KIT Public Report has been published
- KIT Working Groups
 1. Assignment of DOD personnel as chairmen
 2. Expansion of interface category work sheets

3. Addition of new categories(i.e., Command Language, Debugging, Extensibility)

- KIT definitions for requirements, guidelines, conventions, standards, etc.
- ALS/AIE meeting to develop a model for comparison
- Availability of APSE(ALS/AIE/UK Study) documents

Edgar Sibley introduced a draft set of rules for the KITIA. The officers of the KITIA will consist of a chairman and vice-chairman. The secretary will be provided by the KIT. The draft rules will be provided as a set of rules(section A and B of the draft) and a set of procedures(section C and D). The membership section in the draft will be rewritten by LCDR Jack Kramer based on the discussion in the meeting.

The minutes of the April KITIA meeting were unanimously approved. The structure of the KITIA minutes was retained.

10. Working Group Session

The general session adjourned for the day and the working groups met.

20 June 1982

11. General Session

Several items were discussed.

- January meeting of KITIA was changed to February to be held in conjunction with the CMS-2 User Group Conference.
- A May meeting is planned to be held in conjunction with National Computer Conference(NCC) May 16-19 in Anaheim, California.
- A general outline of the KITIA agenda was discussed. The first half day would be spent in working group meetings, then the general session(including point papers) would be held, and the last half day would be spent in working group sessions.
- Steve Glaseman, vice-chairman, will coordinate the scheduling of future meetings of the KITIA.
- New categories for the working groups was discussed (i.e., group I - Command Language; group III - Extensibility and Debugger; and group IV - Extensibility).
- The joint KIT and KITIA meeting will be a half day with the following schedule: 1) a one hour general session, 2) a two hour working group session, and 3) a one hour general session.

12. Individual Presentation

Doug Wrege gave a presentation on KAPSE Semantics and the Layered KAPSE. The presentation covered the need to separate the KAPSE semantics from a particular implementation, and the approach of layering the KAPSE interfaces from a minimum host interface layer toward higher and higher interface levels. The minimum host interface layer would be a layer upon which the remainder of the KAPSE could be implemented in a portable manner. The successive higher and higher layers would be important in the development of a KAPSE semantic definition.

13. General Session

The working groups reported to the general group the deliverables their members plan to have by October. The list is presented by working groups below.

Working Group I

Draft IT requirements for program invocation, control, and security; KAPSE requirements for command languages; KAPSE interface semantics; and a point paper for KITIA policy

Working Group II

Time line of the KAPSE interfaces utilized in the compilation of an Ada program in the ALS and AIE system. Draft requirements for IT and a rationale behind those requirements.

Working Group III

Draft IT requirements, appendices for Stoneman which address the issues of 5E6(Ada executing program) and 5E7(Ada symbol table) of Stoneman. Recommendations for pragmatics for Ada.

Working Group IV

Group IV will meet before October to determine the areas in which a set of draft IT requirements can be developed for delivery in October.

The KITIA expressed its appreciation to Raytheon for the facilities provided and thanked Herb Willman for his efforts to achieve a successful meeting.

Meeting Adjourned.

Appendix A

Attendees

KITIA Members:

BAKER, Nicolas (substitute for Eric Griesheimer)	McDonnell Douglas Astronautics
BARDIN, Bryce (substitute for Jim Ruby)	Hughes Aircraft
CORNHILL, Dennis	Honeywell
COX, Fred (also alternate Larry Gallaher)	Georgia Institute of Technology
DRAKE, Richard	IBM
FELLOWS, Jon	System Development Corp
FISCHER, Herman	Litton Data Systems
FREEDMAN, Roy	Hazeltine Corp
GARGARO, Anthony	Computer Sciences Corp
GLASEMAN, Steve	Teledyne Systems Co.
JOHNSON, Ron	Boeing Aerospace Co.
KERNER, Judy	Norden Systems
KOTLER, Reed	Lockheed Missiles & Space
LAMB, Eli	Bell Labs
LINDQUIST, Tim	Virginia Polytechnic Institute and

State University

LOCKE, Doug	IBM
LOVEMAN, Dave	Massachusetts Computer Associates, Inc.
LYONS, Tim	Software Sciences Ltd.
McGONAGLE, Dave	General Electric
MOONEY, Charles	Grumman Aerospace
MORSE, Harrison	Frey Federal Systems
PLOEDEREDER, Erhard	IABG
REEDY, Ann	PRC
SAIB, Sabina	General Research Corp.
SIBLEY, Edgar	Alpha Omega Group
STANDISH, Thomas	University of California at Irvine
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communication Corp.

Other Attendees:

FORREST, Charles	TRW
KRAMER, Jack	Ada Joint Program Office
LOPER, Warren	NOSC
OBERNDORF, Tricia	NOSC

Appendix B

Bibliography of Handouts

To obtain copies of handouts, the members of the KITIA should request from Tricia Oberndorf or TRWKIT the required handouts. These will be sent in the KITIA mailing prior to the next meeting.

I. ALS/AIE Meetings

- A. KITIA Questions for Softech and Intermetrics
- B. Working Group I Additional Questions
- C. Working Group II Example Program
- D. Intermetrics Guide to AIE Interfaces
- E. Softech's Ada Package Specifications
- F. Softech's Break-In Command Specification

II. Point Papers

- A. ALS and AIE Databases Compared and Made Simple - Herman Fischer
- B. APSE Formal Definition - Roy Freedman
- C. KITIA Interim Technical Note, Program Invocation and Control - Anthony Gargaro
- D. Some Initial Thoughts on APSE Security - Steve Glaseman
- E. A Preliminary Study of KAPSE Interface Validation - Tim Lindquist
- F. Security Classification - Herb Willman
- G. KAPSE Semantics and the Layered KAPSE - Doug Wrege

III. KAPSE Interface Working Sheet

- A. Bindings and Their Effect on Tools

IV. Draft Rules of KITIA

V. Map of KITIA Members Locations

Appendix C
Meeting with Intermetrics
18 June 1982

Tucker Taft of Intermetrics presented an AIE system overview before the Working Group question and answer sessions. Specific topics covered were:

- KAPSE objects(simple, composite, and window);
- KAPSE "agent" which handles all requests from running programs;
- Difference between KAPSE/Tool interfaces and KAPSE/Host interfaces;
- "Physical" and "logical" structures during program execution;

Working Group I Session

Harrison Morse requested the AIE Logon/Logoff, Program Invocation, and Command Processor be covered. Tucker responded with the following:

During login, a terminal manager(to be fully defined later in the design) is a significant part of login and establishes the user program context and windows. For each running program, there exists a program context which includes file handles, disk temporary buffers, and the program windows. Program windows are the means of controlling access and state what is able to pass through the window. Login should use composite object and program invocation package specifications.

During Program Invocation, the Command Language Processor(CLP), via its program context, searches the tool directory for the specified program (e.g., "PRINT"). The linked program context of the specified program and the program context of the CLP is used to establish the sub-program context of the specified program ("PRINT"). The program code and initial memory data of the specified program is then copied into memory.

The command language processor (CLP) in the AIE adds prefixes to the names of specified programs in order to identify the tools within the tool directory. Though the user can define the program context of the invoked tool, the default is to establish the program context using the invoking program context and the linked program context. This program context is used at program invocation time to establish the initial running program context.

In order to utilize a user supplied command language processor, the tool must adhere to the program invocation and control interface. There are no CLP support services supplied by the host operating system. The terminal manager in the AIE has been designed as a virtual terminal interface by making the input/output files for each program disk files.

Working Group II Session

Ann Reedy of Group II presented a simple Ada program for use in analyzing the AIE KAPSE interfaces required to compile the Ada source program. (Ada is a Registered Trademark of the Ada Joint Program Office - U.S. Government). Tucker Taft was asked to walk through the compilation process showing the specific KAPSE interfaces in a time series. A write-up of this time series analysis will be provided to the KITIA by working group II.

The differences between the history attributes of each data object and the separate attributes maintained by the compiler were discussed. Every object has one history attribute which reflects the last program to change its contents. The compiler attributes are used by the compiler to store/remember/record relational information about objects in the database specifically used by the compiler.

Working Group III Session

Eli Lamb requested Tucker Taft to continue with the simple program presented by working group II and show how the executable module is loaded, executed, interrupted, and debugged. Within the AIE system, the command language processor invokes the loader to load both the executing program and the debugger. When the user requests to suspend a program which is being tested using the debugger, the suspend request is sent to the running program. The running program sends the suspend request to the debugger via the inter-program communication of the KAPSE. The debugger support task looks like a running program and is treated as such by the scheduler. In the original design of the AIE debugger, hooks would be used to access the symbol tables. This is not being designed or implemented in the current AIE system. In the AIE design, there is no dynamic binding.

Tucker Taft read the list of names of Packages that will be included in the KAPSE portion of the AIE. The following is the list as used.

- Simple Object
- Composite Object
- Window Object
- Category Operations
- Access Control
- History Control
- Program Invocation and Control
- Debugging
- Inter-Program Communication
- Parameter Passing
- Text I/O
- Direct I/O

Working Group IV Session

Steve Glaseman requested the issue of security violation during recovery be covered. The concept of a terminal manager outside the KAPSE was perceived as a potential security problem.

In the AIE design, nothing is accomplished by a user directly. The user requested operations are always done by a program on the users behalf. Also the user access controls are implied by the programs the user is running. The concept of windows in the AIE design is significant to the issue of security.

Tucker Taft explained the AIE use of windows and their operations. During login the objects of the database are viewed to select the individual user's initial program context. Contained within the program context are the windows of access. Windows in the AIE design are analogous to capabilities in capability based systems, and windows define what can pass through the window. Initially the system is delivered with one user called system who has infinite capabilities (i.e., a window on everything). To prevent access problems, windows are created by the subsetting of predecessor windows.

The KITIA thanked Intermetrics and specifically Tucker Taft for their efforts in a very productive meeting.

Appendix D
Meeting with SofTech
18 June 1982

Larry Weissman of SofTech introduced ALS designers and turned the floor over to Rich Thall who gave an introduction to the ALS system. Specifically, Rich Thall covered the package specifications for the ALS KAPSE interfaces.

Working Group I Session

Harrison Morse presented the questions to be addressed by SofTech which effect the Logon/Logoff, Program Invocation, and Command Processor. Bill Wilder of SofTech covered the startup and shutdown of an ALS session. The ALS user enters an ALS session via the VMS operating system. VMS invokes a process called Job Init which calls the file management, within the KAPSE, to verify the system access. Following successful access verification, Hierarchy Management is called to determine the initial working directory. The Program Call routine is executed to invoke the initial user process which is typically the command language processor (CLP). To exit an ALS session, the CLP sends a return command to the Break_In Monitor which initiates the Job Termination program to delete the user process.

Invoking a program and the operations of the CLP were covered next. During Program Invocation, the Hierarchy Management routine is called to verify the specified executable image exists and the caller has access rights to the image. The program is added to the users call tree and VMS is called to produce a process, from the database file. The question of accessing characters from the terminal one at a time was raised. The ALS does not currently provide access to the individual

characters from or to the terminal. The ALS terminal interface is a string of characters.

Working Group II Session

Judy Kerner requested SofTech to go through the process of compiling the simple program presented by the working group. A time series analysis of the ALS KAPSE interfaces utilized will be provided to the KITIA by working group II. When a "with" clause is used in an Ada program, the latest compiled container of the object specified is used. To use previous versions of containers during compilation, the user must modify the program library to insure use of containers that are older than the current version.

Working Group III Session

SofTech was requested to go through a debugging session indicating the KAPSE services required. In the ALS, the command language processor interfaces to the debugger. The debugger interfaces to the executing program via a communication segment in the executing program and in the debugger. The interfaces are only a few (e.g., peak, poke, etc.) and are part of the ALS KAPSE interfaces. The debugger is part of the MAPSE and a part of the KAPSE.

The performance monitoring in the ALS consists of some MAPSE tools called the timing analyzer and the frequency analyzer. Miscellaneous KAPSE interfaces also contain some performance monitoring interfaces.

Working Group IV Session

SofTech was requested to review the security mechanisms the ALS contains. Basic to an understanding of the design of the ALS security are two principles - 1) all users are non-malicious and 2) no assumption is made about host security being provided. Rich Thall covered the ALS access control mechanism which is used to

protect against illegal access. The concept of keys and locks was described with the keys being the user identification, team identification, and tool identification. The locks were a set of attributes (NO_ACCESS, READ, APPEND, WRITE, ATTR_CHANGE, EXECUTE, VIA) which were lists of individuals who could or could not operate on the node of the database. The NO_ACCESS list contains user identifiers. The presence of these identifiers on this list prevents those users from further access to the database object. The rest of the attributes, except VIA, operate as the mnemonic implies. The VIA attribute precludes access except through the specified tool(s) in the list.

To retarget software, specific software must be modified. A list of the software requiring modification is listed below.

- Code Generator(20%)
- Runtime Support Library
- Debugger back end and Kernel
- Timing and Frequency Kernel
- Linker, Exporter, Assembler, and Loader

The KITIA thanked SofTech and specifically Rich Thall for their effort in a very productive meeting.

Adjourned for the day.

KITIA MEETING PROCEEDINGS

4-5 OCTOBER 1982

Hyatt Crystal City

MONDAY, 4 October

0800 Edgar Sibley, KITIA Chairman, brought the meeting to order.

A general discussion of the KITIA rules regarding visitors to the meetings was held regarding attendees from the European Economic Committee and MITRE Corp. Members were reminded that visitors must be approved prior to the meeting and that visitors may only obtain the floor at KITIA meeting through their sponsors.

A discussion of the memorandum "The Need for a Standard KAPSE" and the KITIA proposal was held to refine the KITIA position. Concern was centered in the definition of the scope of the work defining the interface specification versus defining the design of the KAPSE. Additional work would be completed on the proposal prior to KITIA submission to AJPO.

1025 Working Group Meetings

The progress of the four working groups was presented. The highlights are as follows:

Group 1

- held a working group meeting at Blacksburg, Virginia where the Command Language was a central topic. They are looking at various Command Language features to assist definition of the interface requirements. Concern was also expressed regarding the target environment.

Group 2

- working on a draft of interoperability and Transportability requirements. A presentation of a Time Line Analysis will be presented later in the meeting.

Group 3

- work is progressing in several areas:
 - o expansions on STONEMAN
 - o symbol table and debug requirements
 - o operating system capabilities for KAPSE requirements
 - o paper presentations on Extensibility and Pragmatics

Group 4

- conducting internal critiques on papers regarding
 - o Security
 - o Extensibility
 - o Recovery Mechanisms

- 1115 Consideration of items to be addressed at the joint KIT/KITIA meeting such as the Standard Interface Specification and the revised KAPSE Interface Worksheets was requested.
- 1130 LUNCH
- 1300 There were no objections to the minutes of the June KITIA meeting. The next KITIA meeting is scheduled for February in San Diego.
- 1315 Herm Fischer presented a paper "Time Line Analysis of KAPSE Interfaces During a Compilation".
- 1330 The KITIA secretary was designated as the distribution point for papers to be routed within the KITIA. A lengthy discussion on the number and form of future KITIA meetings followed.
- 1500 The following papers were presented:
 - T. Standish "New Category-- Extensibility"
 - H. Willman "Ada/APSE Portability, A Recommendation for Pragmatic Limitations".
- 1600 Break for Working Group meetings.
- 1700 Adjourned for the day.

TUESDAY, 5 October

0800 Refinements to the previous days proposal resulting from late night discussions were presented to the KITIA.

0830 Items for the joint KIT/KITIA meeting were discussed including:

- presentation of the KITIA proposal
- status and involvement of KITIA in AIE/ALS analysis
- KITIA participation in KIWs
- identification of material for the October Public Report (to be formulated by the Group chairman)

0900 Break into Working Groups.

1115 LUNCH

1300 Joint meeting of KIT and KITIA (see KIT Proceedings).

1700 ADJOURNED

READERS NOTE:

The following minutes refer to message traffic that may be found in this report at the following locations. The alternatives list by Tim Lyons may be reviewed at page 3K-16. The message from Dennis Cornhill is at page 3K-20. The message from DIT Morse follows the Blacksburg minutes.

MINUTES OF MEETING

KITIA WORKING GROUP ONE (WG.1)
AUGUST 10-11, 1982
VIRGINIA POLYTECHNIC INSTITUTE
BLACKSBURG, VA

ATTENDEES:

F. COX, R. FREEDMAN, A. GARGARO, T. LINDQUIST,
C. MOONEY, H. MORSE

AGENDA:

1. MEETING MINUTES
2. KAPSE ALTERNATIVES OR OPTIONS
3. IT REQUIREMENTS
4. COMMAND LANGUAGE
5. RESPONSE TO J. KRAMER MSG ON BINDING
6. DEBUGGER
7. PROGRAM INVOCATION AND CONTROL
8. INFORMAL DISCUSSION ON LAYERED KAPSE
9. KITIA MEETING SCHEDULE DISCUSSION
10. OCTOBER DELIVERABLES

1. MEETING MINUTES

C. Mooney offered to document this and future WG.1 meetings for the sake of preserving and formalizing the technical issues that are raised and the resolutions adopted. Agreed to by WG.1. Also resolved that a draft be distributed to WG.1 members via the net for review and comment, and that, following suitable corrections and updates, the minutes be sent to E. Sibley and T. Oberndorf.

2. KAPSE ALTERNATIVES OR OPTIONS

Discussion was held on the alternatives list introduced by Tim Lyons at the KITIA kickoff meeting in San Diego and readdressed by Dennis Cornhill in a net message of 7 July 82. The discussion focused not on individual alternatives or options but on the set of alternatives as an entity and the timeliness of addressing specific options. The discussion may be summarized as follows:

- a. Neither the ALS or the AIE suffice as a model for a KAPSE Interface Standard for the the purposes of I&T.
- b. The KIT and KITIA can use the ALS, AIE and, potentially, the UK effort as an information base to consider when building a standard.
- c. It is more appropriate to address IT issues not fully addressed in Stoneman as it now stands and to update Stoneman accordingly. (It is WG.1s understanding that Donn Milton of CSCKIT is currently performing an assessment of Stoneman from the viewpoint of what sections need such update).
- d. WG.1 agrees with the last paragraph (less the last sentence) of the Dennis Cornhill message of 7 July. The last sentence may need clarification.
- e. WG.1 recommends that the KITIA concentrate on establishing IT requirements as a matter of priority and defer discussion of the alternatives until some later date.

ACTION ITEM: Dit Morse to convey the above five points to Tricia Oberndorf et al in a net message.

3. IT REQUIREMENTS/SCOPE OF KITIA WG.1 EFFORTS

Discussion was held on what constitutes the IT requirements in terms of the detailed WG.1 areas of investigation. The following outline was derived to provide a framework for developing these requirements for the WG.1 areas.

I. OVERVIEW

High Level Requirements
Decomposition of Requirements
Components

II. HIGH LEVEL REQUIREMENTS

1. Tool set must be transportable.
 - o definition of a tool
 - o criteria for determining what is a tool
 - o requirements for tool to use only KAPSE services for hardware/host OS interactions
2. Application sets, application support and application specific tools must be:
 - o transportable
 - o interoperable
 - a. Data
 - o tool "data sets"
 - o applications data bases
 - b. Functionality
 - o build
 - o test
 - o instrumentation
3. If an application set tests on a host, then it must so test on a target or on another host. BUT things that test on a target need not always test on a host (due to absense of specialized hardware, etc.).
4. It is a requirement to avoid interface definitions which necessitate awkward or in-efficient implementations within the KAPSE. (The same concern given to implementability for the language should now be given to the KAPSE).
5. All MAPSE/KAPSE access to O/S and machine facilities shall be through the KAPSE interface.

6. Not a requirement to apply rigorous host KAPSE standards on target.

The above high level requirements are seen to be high level drivers to be considered in terms of the detailed WG.1 areas of investigation.

ACTION ITEM: C. Mooney to put above outline onto VAX (Frey) for review and updates by WG.1 membership. Protocol originally proposed for KIT review to be used for this process.

4. COMMAND LANGUAGE (CL)

This item occupied no small amount of meeting time, and, at times, the discussion wandered into other WG.1 areas. The following serves to highlight the discussion:

- a. Tim Lindquist will prepare a paper on CL. The planned outline of this paper follows:

I. Introduction

Background - CLs and the Apse with respect to IT

Statement of Needs - perhaps a simple CL for interactive use and a more complex one for CL files.

II. Command Language Structures

- control
- data (primitive objects - files, directories, etc.)

III. Format of Command Entry

IV. Units of Communication between the User and the KAPSE

V. Implications on KAPSE Support

VI. Future Concerns - Consistent Tool to User
Interface

- b. A discussion took place of programming languages, binding characteristics, typing, etc. as these topics relate to CL requirements. This discussion may be shown tabularly as follows:

Languages

CYBOS

LISP

MUMPS

PASCAL PL/1

Ada

Characteristics of Language Products

Fast painless construction

Unpredictable

Difficult to verify

Slow painful construction

Extremely reliable

Validatable

Predictable

Binding:

Latest, Weakest

Earliest. Strongest

It is a belief that it is a requirement for reliable Ada software, the products of CL procedures require dynamic but very strong binding.

The following characterizes the groups feelings on this topic:

- CL is a topic deserving of lengthy discussion, and worth a separate meeting devoted to it alone.
- CL procedures need strong typing.

- CL procedures have many of the needs of programs written in standard languages, i.e., readability and reliability.
- CL has inherent in it strong ties to file names (it was noted that within a file name, frequently file attributes are embedded - where the file is stored, who "owns" it, what revision level it is at, etc.).

Dit Morse suggested the following, which was agreed to by WG.1:

Schedule and hold a three day meeting on CL. Required at this meeting are D. Morse, T. Lindquist, F. Cox. The other members of WG.1 are invited to attend, if available. Each of the other WGs will be asked to send a delegate, Also, a representative of ANSI X3H1 would be invited. Dit Morse will endeavor to contact X3H1 for this purpose.

In summary, the following conclusions were reached during the CL part of the meeting:

- o Command language support is the objective.
- o Designing a standard CL is outside the scope of the current activity.
- o The activity is that of identifying the required functionality of command languages that require KAPSE support.
- o Command language procedures have a critical effect on IT.

ACTION ITEM: Tim Lindquist to develop a scope statement and distribute it over Arpanet.

5. RESPONSE TO J. KRAMER MESSAGE ON BINDING

The subject message was received over Arpanet during the course of the meeting. After suitable discussion. Anthony Gargaro drafted a response and sent it over the net. This response is included as Attachment A of these minutes.

6. DEBUGGER

Discussion was started with a review of the net msg of 13 July 82 from Dit Morse. The following resulted from this discussion:

- o Major point 4 of the 13 July msg should include the topic of security.
- o The message needs the added dimension of debugging on the target.
- o The separate category proposed in the message should be forgotten.
- o It is WG.1's belief that this item should be folded into the Program Invocation and Control category.
- o It is also believed that this item falls under WG.1 purview. Both since it is contained in Program Invocation and Control and because of strong interrelationships with Device Interactions and Support. Also a WG.1 topic. (Consider a target connected to an APSE as a device with specific KAPSE requirements.)

WG.1 will attempt to provide a more elaborated position on this topic. in the post October meeting time frame. Using the Fairley schematics as a frame of reference.

ACTION ITEM: Dit Morse to generate and distribute a net msg stating WG.1 position and plans on the debugger issue.

7. PROGRAM INVOCATION AND CONTROL

Anthony Gargaro distributed an updated draft of the KITIA Interim Technical Note -- Program Invocation and Control. He requested that all WG.1 members review this Technical note prior to the October meeting.

8. INFORMAL DISCUSSION OF LAYERED KAPSE

An informal discussion was held on the layered approach from a formal

definition viewpoint. A rigorous conclusion was not obtained at that time.

9. KITIA MEETING SCHEDULE

Discussion was held on the topic of how often meetings should be scheduled. This topic was motivated by the clause in the proposed KITIA bylaws that there be five KITIA meetings per year.

It was agreed that this represents far too many meetings of the entire team, and that many WG.n meetings, interspersed with a few KITIA coordinating sessions would be a far more productive approach. Also, meetings composed of the WG.n chairs and the KITIA executive were proposed as a means of providing additional coordination.

A poll was held to determine membership feelings. The following was the average of the individual votes:

- a. Entire KITIA meeting as a group - twice per annum
- b. KITIA Executive plus WG.n Chairs - 3 per annum
- c. WG.n - as required. For WG.1, approximately six per annum.

ACTION ITEM: Dit Morse to send the above meeting recommendations to Tricia Oberndorf and Ed Sibley.

10. OCTOBER DELIVERABLES

The topic of deliverables for the October meeting was addressed several times during the two days. The following summarizes this topic:

1. Point Paper on IT Requirements - Dit Morse
2. CL Position Paper - Tim Lindquist

3. Program Invocation and Control Paper - Anthony Gargaro

4. Formal Definition Paper - Roy Freedman

MEETING ADJOURNED

Charles Mooney

(Attachment A)

11-Aug-82 06:23:27-PDT,1269;000000000001

Date: 11 Aug 1982 0623-PDT

From: GARGARO at USC-ECLB

Subject: BINDING CATEGORY

To: KITIA-EXEC:

cc: KITIA-WG1:, KITIA-WG3:

JACK-

I DO NOT BELIEVE THIS IS A RUN-TIME ISSUE. YOUR CONCERN RELATING TO INTERPROGRAM FUNCTIONALITY IS A KAPSE ISSUE FOR THE PROGRAM INVOCATION AND CONTROL INTERFACE TO PROVIDE INTERPROGRAM COMMUNICATION THAT IS MORE COMPREHENSIVE THAN THE EXCHANGING OF PARAMETERS AT PROGRAM INVOCATION AND TERMINATION. AS A CONSEQUENCE, THE KAPSE IDENTITY OF AN INVOKED PROGRAM NOW BECOMES SIGNIFICANT TO THE INTERFACE.

WHETHER OR NOT IT IS A DISTRIBUTED ENVIRONMENT IS IMMATERIAL, IF THE INTERFACE SEMANTICS ARE CAPACITY TRANSPARENT. HOWEVER, THE RATIONALE FOR MORE ELABORATE COMMUNICATION IS MOTIVATED BY THE OPPORTUNITIES FOR PHYSICAL DISTRIBUTION (REFER, CLASS-3, WG.1-A002).

DIT MORSE BELIEVES THIS ISSUE IS A SUBSET OF THE GENERAL PROBLEM OF NAMING RESOURCES (ACCESSIBLE ENTITIES IN A SYSTEM) FOR PURPOSES OF COMMUNICATION. THE GENERAL PROBLEM INCLUDES LOGICAL NAME RELATIONSHIPS IN NETWORK ENVIRONMENTS, AND RESOURCES INCLUDE PROCESSES, DATA OBJECTS, AND DEVICE/CHANNELS WHICH ARE DATA SOURCES AND SINKS.

ANTHONY.

79 -- *****

Mail-From: USC-ECLB

Received-Date: 13-Jul-82 0939-PDT

Date: 13 Jul 1982 0939-PDT

Sender: MORSE at USC-ECLB

Subject: CL/"Debugger" poll (DRAFT)

From: MORSE at USC-ECLB

To: poberndorf at USC-ECLB,

To: sibley at USC-ECLB

Cc: fcox at USC-ECLB, freedman at USC-ECLB, gargaro at USC-ECLB,

Cc: mooney at USC-ECLB, lindquist at USC-ECLB,

Cc: morse at USC-ECLB

Message-ID: <[USC-ECLB]13-Jul-82 09:39:51.MORSE>

Tricia, et al -

[This is a draft of my reply to your poll.

I am sending the draft to you and my committee for review, prior to sending it to the working group Chairs and the KITs, for two reasons:

1. You may wish to address the issue of ad hoc committees in some other way.
2. Although this represents my own view, comments by the members of WG.1 would be appreciated.

Thanks all, dit]

Embedded in this message on Command Languages and Debuggers are my votes on the poll.

Please consider that some problems of area placement into working groups are indicators that the working group structure is not suitable to address all issues.

In particular, problems arise in those areas for which we do not have a consensus for the need to address, nor the approach to take when addressing.

I recommend for these issues we feel free to form small short-lived ad hoc committees to prepare a report on positions, options, and recommendations for the full committees to discuss.

These reports should present pros and cons of all issues rather than come to definitive conclusions, unless so directed.

Command Languages:

The purpose of including an area is to address that area relative to issues of I&T. In the KITIA, at least, we still have some groundwork to do vis-a-vis the goals of I&T, in particular, the extent to which I&T requirements and implications should be addressed relative to:

- o Tools (totally of course?)
- o Operational programs and collections of same (=Systems)
- o Programmers (without retraining)
- o Hosts and Targets (Tested programs still work?)
- o Distributed Systems
- o (More to come)

Aside from that, the inclusion of an area such as Command Languages or Command Language Support does not, de facto, imply a requirement to develop a standard, nor does the development of a standard for a Command Language imply that there be only one Command Language, or CL, standard. It does imply the need to address the relation of CLs, I&T, and KAPSE requirements for support.

KITIA Group 1 has taken the position that the implications of configuration control on I&T require the commands to build systems be transportable, and have the same semantic meaning (build the same system) on the destination system. This single requirement, if commonly accepted, is sufficient to warrant addressing the general issues surrounding command languages and command language support.

Transportability can be achieved through standardization at a number of levels (e.g. - the CL itself, the KAPSE support required for a specific CL processor at least), and also by formal transformation of a CL from a source systems to another CL on the destination system, maintaining semantic "integrity", as Roy Freedman is addressing. If the latter approach is adopted, KAPSE support must be defined to insure a transportable "semantic base" exists.

Our position is that the issue must be addressed. WG.1 will prepare a paper on the issues and the options, as we see them. In parallel, we will develop a list of KAPSE support capabilities which we feel must be addressed in the context of command language requirements.

After all the dust has settled, it may come down as Donn Milton suggest, that CL support is just another tool with standards to be addressed at the MAPSE or APSE level, but I would dare not assume so without raising the issues and addressing them explicitly.

Vote on Item #1:

I opt for the title of "Command Languages and Support Requirements", assigned to Group 1.

The issue of "DEBUGGERS":

It is with great difficulty that I restrain myself from climbing upon my "Debuggers" soapbox here. I will try to restrain myself to brief and hopefully cogent comments.

I include debugging facilities in a category which I will call here:

"Ancillary Control and Data Manipulation Facilities"

which includes as main topics:

- o Debugging Facilities
- o Performance Measurement
- o Validation Support
- o Default Exception/Abnormality Handling

The support requirements include:

- o Command Facilities
Specify what is to be done (trace, trap, record, etc.) under what conditions.
- o Control Mechanisms
Mechanisms to "implant" traps to capture control flow (Program calls, procedure calls, task invocations) and data "flows" (data creation, modification, and access).
- o Data Facilities
Ability to collect, record, and present data in a variety of forms at a variety of destinations (such as files, terminals, printers).
- o Ancillary Information
Access to structural information and language level names to permit (?all) interactions to take place at the Ada language level.

To the largest extent possible, these facilities must be transportable, with implications on I&T and KAPSE facilities as well as implications on language processors, linkers, etc.

My major points are:

- o the user facilities for debugging, performance measurement, interface validation, etc. are different, but the control mechanisms are the same.
- o Like other tools, these facilities should be transportable, which means KAPSE Interface requirements must be addressed, and standards developed.
- o The requirements of such a comprehensive set of facilities affect many aspects of a system, and must be considered in the whole rather than piecemeal.
- o The requirements are almost always considered as an afterthought, with the consequential effects on consistency, coherence, generality, performance, and usefulness.

Vote on Item #2:

A new category be created entitled:

"Ancillary Control and Data Manipulation Facilities".

Vote on Item #3:

This area should be assigned to an ad hoc committee and vigorously pursued across all KIT/KITIA working groups.

Vote on Item #4:

That we assign Extensibility requirements to an ad hoc committee to prepare a position on the relationship with other areas.

Dit

SECTION III

KIT/KITIA DOCUMENTATION

MEMORANDUM OF AGREEMENT AMONG
DEPUTY UNDER SECRETARY (AM)
ASSISTANT SECRETARY OF THE ARMY (RD&A)
ASSISTANT SECRETARY OF THE NAVY (RE&S)
AND
ASSISTANT SECRETARY OF THE AIR FORCE RD&L)

Subject: Ada Programming Support Environment (APSE) Tool
Transportability

Reference: Requirements definition for Ada Programming Support
Environment - STONEMAN

1. Purpose

This memorandum is to establish the procedures and working relationships within which the Army, Navy and Air Force will cooperate to converge on a set of Ada Programming Support Environment (APSE) interface standards to permit the sharing of tools and other software between DoD supported APSEs.

2. Objective

The objective of this effort is to establish the necessary interface conventions for APSE tools, users and data bases to permit the consistent introduction of new tools into the software development and maintenance environment and to permit the portability of tools among different implementations of the Kernel Ada Program Support Environment (KAPSE).

3. Background

Numerous studies have predicted that the cost of DoD software will continue to escalate in the 1980s and that the availability of qualified software personnel will be a critical factor in the development and maintenance of weapon systems. The Ada Program will make the goal of a common language within DoD a reality. The high level of cooperation among the Military Departments and agencies required to establish this program has generated a unique opportunity for the DoD to adopt modern software and management practices and to develop support tools to improve productivity.

4. Agreement

We recognize that to realize the full potential of this opportunity, the DoD must focus its limited resources, including funding and talent, on the development of an Ada Programming Support Environment (APSE) which can be shared by all three Military Departments, so that software tools may be readily transported among systems and across Service applications. The STONEMAN requirements document defines the concept of a KAPSE. We agree with the concept of standard tool interfaces to the KAPSE, and a standard for all other aspects of the KAPSE which are visible to the tools. Although it may be desirable

for the DoD to support different KAPSE designs to reduce risk in the early phases of the Ada Program, the long term goal is to establish the necessary interface conventions so that multiple efforts may converge to a single set of interface standards in the 1985 time frame.

The current KAPSE designs, namely the Army supported Ada Language System and the Air Force supported Ada Integrated Environment and any other KAPSEs which the DoD may support in the future will be closely monitored by the Ada Joint Program Office (AJPO) and a joint Service evaluation team to identify and establish interface conventions. The evaluation team will be chaired by the Navy. All APSE tools procured by the DoD will adhere to these conventions. In the event that, for schedule or contractual reasons, one KAPSE design violates these conventions, or if conventions are established which are not supported by a previous design decision, that KAPSE will be evolved to conform to these conventions. This agreement will be implemented through a set of procedures developed by the AJPO and coordinated by NAVMAT, DARCOM, and AFSC.

5. Duration

The provisions of this memorandum will commence when signed and will remain in effect until formally rescinded.

Paul Epstein 4 Dec 81

Assistant Secretary of Army
(Research, Development and
Acquisition)

William A. Long 19 Dec 81

Deputy Under Secretary of Defense
for Research and Engineering
(Acquisition Management)

W. M. H. H. H.

Assistant Secretary of Navy
(Research, Engineering and
Systems)

14 Dec 81

Mark Chen 5 Nov 81

Assistant Secretary of Air Force
(Research, Development and
Logistics)

Ada Programming Support Environment
(APSE)
Interoperability and Transportability

Plan

Contract # N00123-80-D-0242
Delivery Order # 7N45
Task # T-051
CDRL Item A001

for
NAVAL OCEAN SYSTEMS CENTER
271 Catalina Boulevard
San Diego, California 92152

Prepared by
TRW
SYSTEMS ENGINEERING APPLICATIONS DIVISION
DEFENSE SYSTEMS GROUP
3420 Kenyon Street, Suite 202
San Diego, California 92110

TABLE OF CONTENTS

Paragraph		<u>Page</u>
	1. INTRODUCTION	1-1
	1.1 Background	1-1
	1.2 APSE Concept and Definitions	1-3
	1.3 Objectives of the APSE IT Effort	1-6
	1.4 Document Organization	1-9
	2. ORGANIZATION	2-1
	2.1 AJPO	2-1
	2.2 Army, Air Force and Navy	2-1
	2.3 KAPSE Interface Team (KIT)	2-1
	2.4 Industry/Adademia Team (IAT)	2-3
	2.5 Support Groups	2-5
	2.6 User Groups and Professional Societies	2-5
	2.7 Standards Organizations	2-5
	3. FUNCTIONS OF THE KITEC	3-1
	3.1 Work Breakdown Structure	3-1
	3.2 Schedules/Milestones/Deliverables	3-8
	3.3 Provisions for Special Needs	3-10
	APPENDIX A Glossary of Terms	A-1
	APPENDIX B Applicable Documents	B-1

LIST OF FIGURES

		<u>Page</u>
Figure	1 APSE Structure (STONEMAN)	1-4
	2 APSE IT Participants	2-2
	3 APSE IT Program Work Breakdown Structure	3-2

LIST OF TABLES

Table	1 APSE IT Schedule (1982 CY to 1985 CY)	3-11

1. INTRODUCTION

The Ada Programming Support Environment (APSE) Interoperability and Transportability (IT) Plan is presented in this document. The IT activities associated with APSE developments are described. Schedule and milestones for these activities are presented as well as a Work Breakdown Structure (WBS) to accomplish these activities.

These IT activities are directed by the Kernel APSE Interface Team Executive Committee (KITEC).

The plan additionally describes the constitution and the major responsibilities of the KITEC. These responsibilities are:

- a. APSE IT Requirements Development and Analysis
- b. APSE IT Conventions and Standards Development and Implementation
- c. APSE IT Meetings and Working Groups for APSE IT Requirements, Conventions and Standards Development.
- d. APSE IT Specification Compliance
- e. APSE IT Tool Development and Test Application
- f. Monitoring of the Ada Integrated Environment (AIE) and the Ada Language System (ALS) development efforts with respect to APSE IT.

1.1 BACKGROUND

In 1975 the High Order Language Working Group (HOLWG) was formed under the auspices of the U.S. Department of Defense (DoD). It consisted of representatives from the Army, Air Force, Navy, Marines and other defense

agencies with the goal of establishing a single high order language for new DoD Embedded Computer Systems (ECS). The technical requirements for the common language were finalized in the Steelman report of June 1978. International competition was used to select the new common language design. In 1979 after review by roughly eighty teams representing DoD organizations, industry, academia and NATO countries, and after intensive analysis by the three services and other defense agencies, the DoD selected the design developed by Jean Ichbiah and his colleagues at CII-Honeywell Bull. The language was named Ada in honor of Augusta Ada Byron (1816-1851), the daughter of Lord Byron and the first computer programmer.

It was realized early in the development process that acceptance of a common language and the benefits derived from a common language could be increased substantially by the development of an integrated system of software development and maintenance tools. The requirements for such an Ada programming environment were stated in the STONEMAN document. The STONEMAN paints a broad picture of the needs and identifies the relationships of the parts of an integrated APSE. STONEMAN identifies the APSE as support for "the development and maintenance of Ada application software throughout its life cycle". The APSE is to provide a well-coordinated set of tools with uniform interfaces to support a programming project throughout its life cycle.

The Army and Air Force have begun separate developments of APSEs. The Initial Operational Capabilities (IOCs) are called Minimal Ada Programming Support Environments (MAPSEs).

The Army APSE has been designated the ALS and that of the Air Force, the AIE. The Navy will review both APSE developments and identify critical aspects of the designs where conventions or standard interfaces and specifications are needed to insure compatibility, paying particular attention to the Kernel Ada Programming Support Environment (KAPSE) interface and the Descriptive Intermediate Attributed Notation for Ada (DIANA) user interface. The Navy plans to make maximum use of Army/Air Force products that meet Navy requirements, and develop only the additional components required for Navy-unique applications.

AD-A123 136

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM: PUBLIC REPORT VOLUME II(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P A OBERNDORF 28 OCT 82

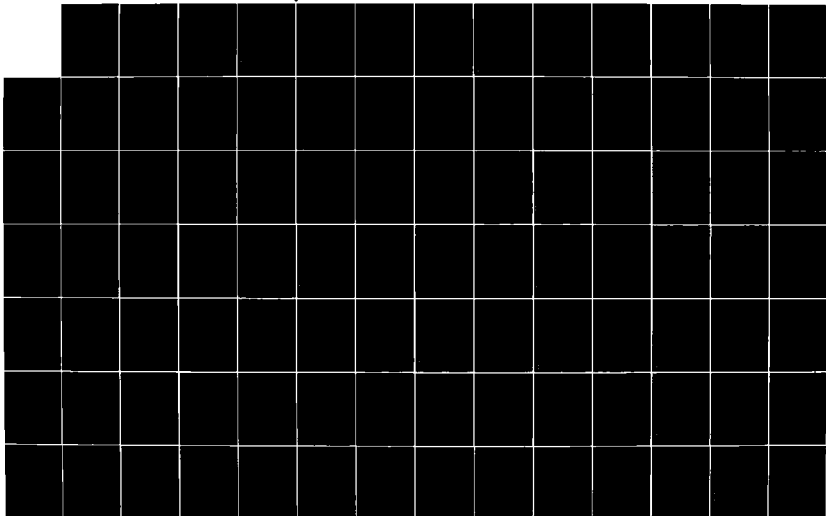
26

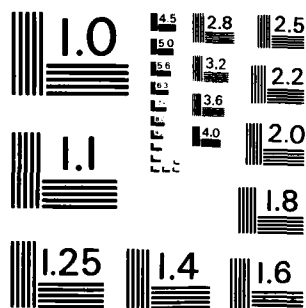
UNCLASSIFIED

NOSC/TD-682

F/G 9/2

NL





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

The Ada Joint Program Office (AJPO) was formed in December 1980. It is the principal DoD agent for development, support and distribution of tools, common libraries, and environments for Ada. The AJPO will coordinate all Ada efforts within DoD to ensure their compatibility with the requirements of other Services and DoD Agencies, to avoid duplicative efforts and to maximize sharing of resources.

1.2 APSE CONCEPT AND DEFINITIONS

The APSE provides a virtual interface between the user of the APSE and the particular host system upon which the APSE is installed. This interface is designed to be machine and operating system independent; in effect, it defines an Ada virtual machine whose features are available on all actual host machines. The purpose of the APSE is to provide an environment for the design, development, documentation, testing, management, and maintenance of embedded computer software, written principally in the Ada programming language.

Figure 1 depicts the APSE structure. The STONEMAN shows the APSE as a structure with the following layers or levels:

- level 0: Hardware and host software as appropriate
- level 1: Kernel Ada Program Support Environment (KAPSE), which provides database, communication and run-time support functions to enable the execution of an Ada program (including a MAPSE tool) and which presents a machine-independent portability interface.
- level 2: Minimal Ada Program Support Environment (MAPSE) which provides a minimal set of tools, written in Ada and supported by the KAPSE, which are both necessary and sufficient for the development and continuing support of Ada programs.

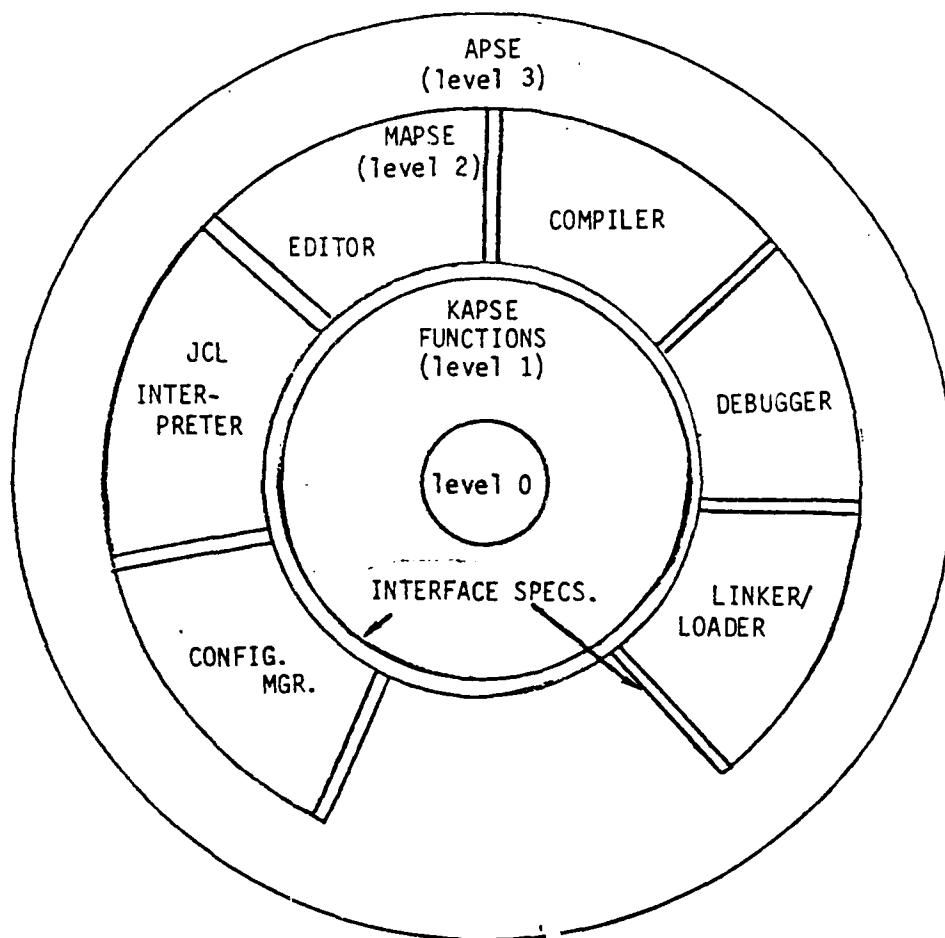


Figure 1 APSE Structure (STONEMAN)

level 3: Ada Program Support Environments (APSEs) which are constructed by extensions of the MAPSE to provide fuller support of particular applications or methodologies.

For the purpose of this plan and subsequent activities of the KITEC, the following definitions are provided in the context of APSEs:

INTEROPERABILITY: Interoperability is the degree to which APSEs can exchange data base objects and their relationships in usable forms without conversion.

TRANSPORTABILITY: Transportability of an APSE tool is the degree to which it can be installed on a different APSE without reprogramming; the tool must perform with the same functionality in both APSEs. Synonyms commonly used are portability and transferability.

REUSABILITY: Reusability is the degree to which a program unit is employable, without reprogramming, in the construction of new programs.

HOST: A host is a computer system upon which an APSE resides, executes and supports Ada software development and maintenance. Examples are IBM 360/370, VAX 11/780, CDC 6000/7000, DEC 20 and UNIVAC 1110 and any of their operating systems.

TARGET: A target is a computer system upon which Ada programs execute.

Hosts are, in fact, also targets, in as much as the APSE is written in Ada. A target might not be capable of supporting an APSE. An embedded target is a target which is used in tactical applications. Examples of embedded target computer systems are AN/AYK-14, AN/UYK-43 and computer systems conforming to MIL-STD-1750A and MIL-STD-1862 (NEBULA).

REHOSTABILITY: Rehostability of an APSE is the degree to which it can be

installed on a different HOST with needed re-programming localized to the KAPSE. Assessment of rehostability includes any needed changes to non-KAPSE components of the APSE, in addition to the changes to the KAPSE.

RETARGETABILITY: Retargetability is the degree to which an APSE tool does not have to be modified to accomplish the same function with respect to another target. Not all tools will have target specific functions.

1.3 OBJECTIVES OF THE APSE IT EFFORT

The objectives of the APSE IT effort are:

- a. To develop requirements for APSE IT.

The STONEMAN document paints a broad picture of the needs and relationships of the parts of an integrated APSE. Although the STONEMAN is being used as the primary reference document for APSE development efforts, it does not give sufficient detailed requirements to assure IT between APSEs. APSEs built with a sound set of IT requirements will insure cost savings in the development of tools. The cost of re-programming tools for different APSEs will be significantly reduced.

- b. To develop guidelines, conventions and standards (GCS) to be used to achieve IT of APSEs.

GCS describes the process or means by which requirements can be satisfied. It is realized that it is premature to develop steadfast standards during the early part of this APSE IT effort. There is no precedent for IT between programming support environments of this magnitude to give guidance in the development of these GCS. The GCS that are developed by this APSE IT effort will evolve over a four year period 1982 to 1985. The APSE IT schedule (see Table 1) calls for a new draft of GCS to be developed each year. Presentation of these GCS will be given in public forums to insure that they are sound and realistic.

The AIE and ALS APSE development efforts have different schedules. This makes it impossible to bring these two efforts into perfect compliance knowing their contractual and schedule status. Hard and fast standards that satisfy both AIE and ALS development efforts would be unrealistic at this stage of their development. An appraisal of the AIE and ALS development effort will be made to see where GCS can be used to achieve IT of their respective APSEs.

- c. To develop APSE IT tools to be integrated into both the AIE and ALS.

This APSE IT effort provides for the development of three or more tools to be integrated into both the AIE and the ALS. This tool development effort will help surface interfaces and interface problems associated with tool IT between different APSEs. IT should also show how close the GCS developed by this APSE IT effort reflect the reality of the AIE and ALS efforts. The tools developed by this APSE IT effort will not be limited to only a test function. They will be well documented tools designed to help our understanding of interfaces and interface conflicts between different APSEs, and they will be tools developed for user applications (e.g. configuration management tool).

Two of the tools chosen will be selected from contractors with whom the Air Force is not negotiating. The selection of these tools will be based upon designs that were developed and were well received by the Ada community in the six month competition period for the Air Force contract for the AIE. In addition an RFP will be issued for one or more additional tools.

- d. To monitor the AIE and ALS development efforts with respect to APSE IT.

This APSE IT effort provides for the monitoring of the AIE and ALS development efforts. The monitoring will be at a level of making recommendations for resolution of situations in which the AIE and/or the ALS differ from one another on the evolving APSE IT conventions and standards. Interface problem areas which would inhibit IT between the AIE and ALS will be identified.

AIE and ALS document review, analysis and recommendations will be made. When questions arise that need resolution and/or clarification with regard to the ALS and AIE development efforts the KIT (see Section 2.3) has Army and Air Force representatives on it who are knowledgeable of these efforts.

- e. To provide initiative and give a focal point with respect to APSE IT.

There is a need for a focal point to be provided to APSE developers and users with regard to information about IT between APSEs. APSE IT questions arise frequently within professional societies and user groups. A forum is needed in which APSE IT questions can be addressed and discussed, and in which APSE IT information can be disseminated throughout the Ada community.

The KIT and the IAT (see Section 2.3 and 2.4) will provide focal points for the Ada community. Public reports on the results of this APSE IT plan will be given through professional societies such as AdaTEC every six months. This is in keeping with the AJPO philosophy of public exposure. This effort is the lead DoD effort in regard to APSE IT. In this respect, the KITEC will participate in, and help when

possible, other programs connected with APSE IT (these include international development efforts).

- f. To develop and implement procedures to determine compliance of APSE development with APSE IT requirements, guidelines, conventions and standards.

There is a need to establish procedures by which the recommendations that are developed by this APSE IT effort will be reviewed and implemented by the AJPO. The procedures that are to be followed should apply not only to the AIE and ALS development efforts, but, should also apply to other APSE development efforts. Compliance to AJPO approved recommendations should include review and analysis of test reports.

1.4 DOCUMENT ORGANIZATION

(TBD)

2. ORGANIZATION

Figure 2 depicts the participants in the APSE IT effort. The following sections provide a brief description of these organizations and the relationships to the KITEC.

2.1 AJPO

The KITEC is an agent of the AJPO. The KITEC supports the AJPO by performing the activities outlined in this plan, and provides recommendations and information to the AJPO. The AJPO makes final decisions in the areas of requirements, policy, procedures and funding.

2.2 ARMY, AIR FORCE AND NAVY

Currently the Army and Air Force have begun separate developments of APSEs. In the development of its APSE, the Navy plans to make maximum use of Army/Air Force products that meet Navy requirements. The Navy will review both APSE developments and identify critical aspects of the designs where conventions or standard interfaces and specifications are needed to insure compatibility. It will be the role of the KITEC to interact with these services and the respective APSE contractors for information-exchange and consultation. The contractor for the Army's ALS is SofTech; the Air Force contractor for the AIE is yet to be determined.

2.3 KAPSE INTERFACE TEAM (KIT)

The Navy has the responsibility for the KIT. The membership of the KIT is composed of the following representatives:

- Naval Ocean Systems Center (NOSC)
- Project Manager Ships (PMS)
- Naval Underwater Systems Center (NUSC)

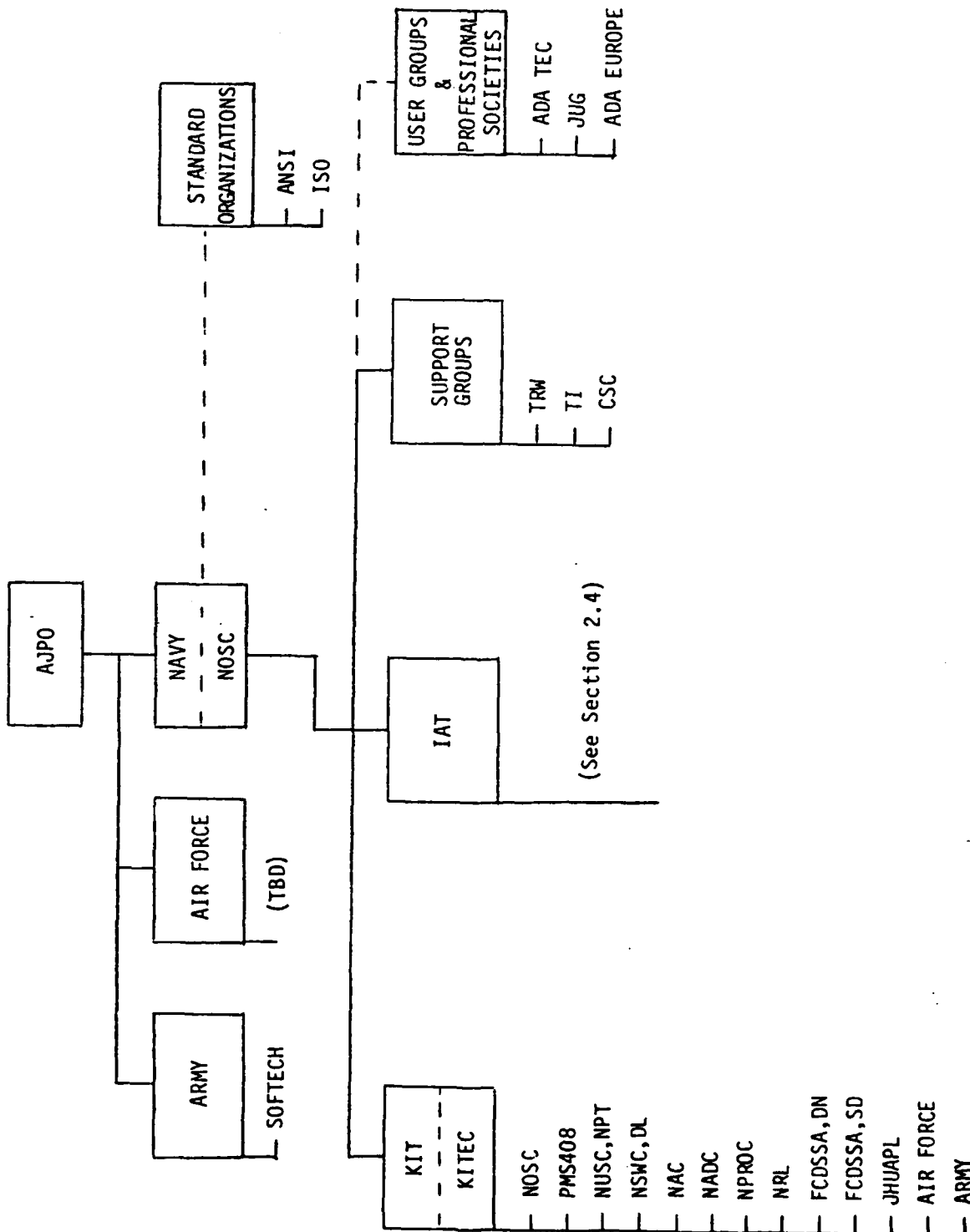


Figure 2 APSE IT Participants

- Naval Surface Weapons Center (NSWC)
- Naval Avionics Center (NAC)
- Naval Air Development Center (NADC)
- Naval Personnel Research & Development Center (NPRDC)
- Naval Research Laboratory (NRL)
- Fleet Combat Direction System Support Activity (FCDSSA) -
Dam Neck
- Fleet Combat Direction System Support Activity (FCDSSA) -
San Diego
- Johns Hopkins University Applied Physics Laboratory (JHUAPL)
- Air Force
- Army

NOSC is the lead laboratory in the APSE IT effort. The KITEC is part of the KIT and is responsible to the AJPO. The objectives of the KIT are the objectives of the APSE IT effort (see Section 1.3).

2.4 INDUSTRY/ACADEMIA TEAM (IAT)

The IAT will supplement the activities of the KIT. IT will assure broad inputs from software experts and eventual users of APSE's. The KITEC selected a team of qualified representatives from industry and academia, who agreed to commit long-term, part-time availability to interact with the KIT, as sounding board, as proposer of APSE IT requirements, guidelines, conventions and standards, as consultants concerning implementation

approaches and impacts.

Below are listed the selections for membership to the IAT:

Alpha-Omega Group (Dr. Sibley)	IBM (Doug Locke)
Bell Laboratories (Eli Lamb)	Litton (Herman Fischer)
Boeing Aerospace (Roger Arnold)	Lockheed (Reed Kotler)
Computer Sciences Corp. (Anthony Gargaro)	McDonnell Douglas (Eric Griesheimer)
Control Data Corp. (Dr. Wrege)	Norden (Judy Kerner)
Ford Aerospace (Dr. Yelowitz)	Oy Softplan Ab (Pekka Lahtinen)
General Electric (David McGonagle)	PRC (Dr. Reedy)
General Research (Dr. Saib)	Raytheon (Herb Willman)
Georgia Inst. of Tech. (Fred Cox)	SDC (Jon Fellows)
Grumman Aerospace (Charles Mooney)	Teledyne (Dr. Glaseman)
Hazeltine (Dr. Freedman)	TNO (Rob Westermann)
Honeywell (Dennis Cornhill)	UK Ada Consortium (Tim Lyons)
Hughes Aircraft (Dr. Ruby)	Virginia Polytechnic (Tim Lindquist)

In addition, the following have been asked to be special associated members of the team:

Frey Federal Systems (Harrison Morse)
IABG (Germany) (Erhard Ploedereder)
MCA (Dave Loveman)
UC Irvine (Dr. Standish)

2.5 SUPPORT GROUPS

Currently there are three contractors working with the KITEC: TRW is providing system engineering support; TI and CSC are developing APSE IT tools. One or more additional contractors will develop additional APSE IT tools.

2.6 USER GROUPS AND PROFESSIONAL SOCIETIES

It is anticipated that AdaTEC, JOVIAL-Ada Users Group (JUG), and Ada Europe will provide valuable contributions to the Ada IT effort. The KITEC has no formal relationship with these groups; however, the KITEC will use some or all of these groups as regular forums for the presentation of reports and technical results of the APSE IT effort, and will solicit inputs from members.

2.7 STANDARDS ORGANIZATIONS

The ANSI and the ISO are standards organizations which are already involved in establishing the Ada programming language as a broadly recognized, enforceable standard. It is possible that the eventual results of KITEC activity will be submitted for such canonization, to effect the commonality of APSE's deemed necessary to achieve DoD's life-cycle objectives. The KITEC initially will become familiar with the organization's standardization procedures so that future standardization actions can be planned and accomplished with minimum difficulty. This will include the study of existing standards which may interact with, or guide development of, APSE IT standards.

3. FUNCTIONS OF THE KITEC

Figure 3 shows the areas of KITEC responsibility. They are:

- IT Management - 1000
- IT Analysis - 2000
- IT Standards & Compliance - 3000
- IT Tools - 4000
- IT Support - 5000

3.1 WORK BREAKDOWN STRUCTURE

A discussion of each element in the WBS is presented below.

1000 APSE Interoperability & Transportability (IT) Management

1100 APSE IT Systems Management

This WBS element provides for management of the APSE IT program. It further provides for a Technical Report to be prepared every six months. The Technical Report will cover the technical accomplishments of the APSE IT program for the period in question and will be suitable for distribution in hard copy or for presentation at a symposium.

1200 APSE IT Planning

This WBS element provides for the planning necessary to follow through and complete the APSE IT program. It further provides for the updating of the APSE IT plan on a yearly basis.

1300 APSE IT Administrative Support

This WBS element provides the administrative support necessary in the implementation of the APSE IT program.

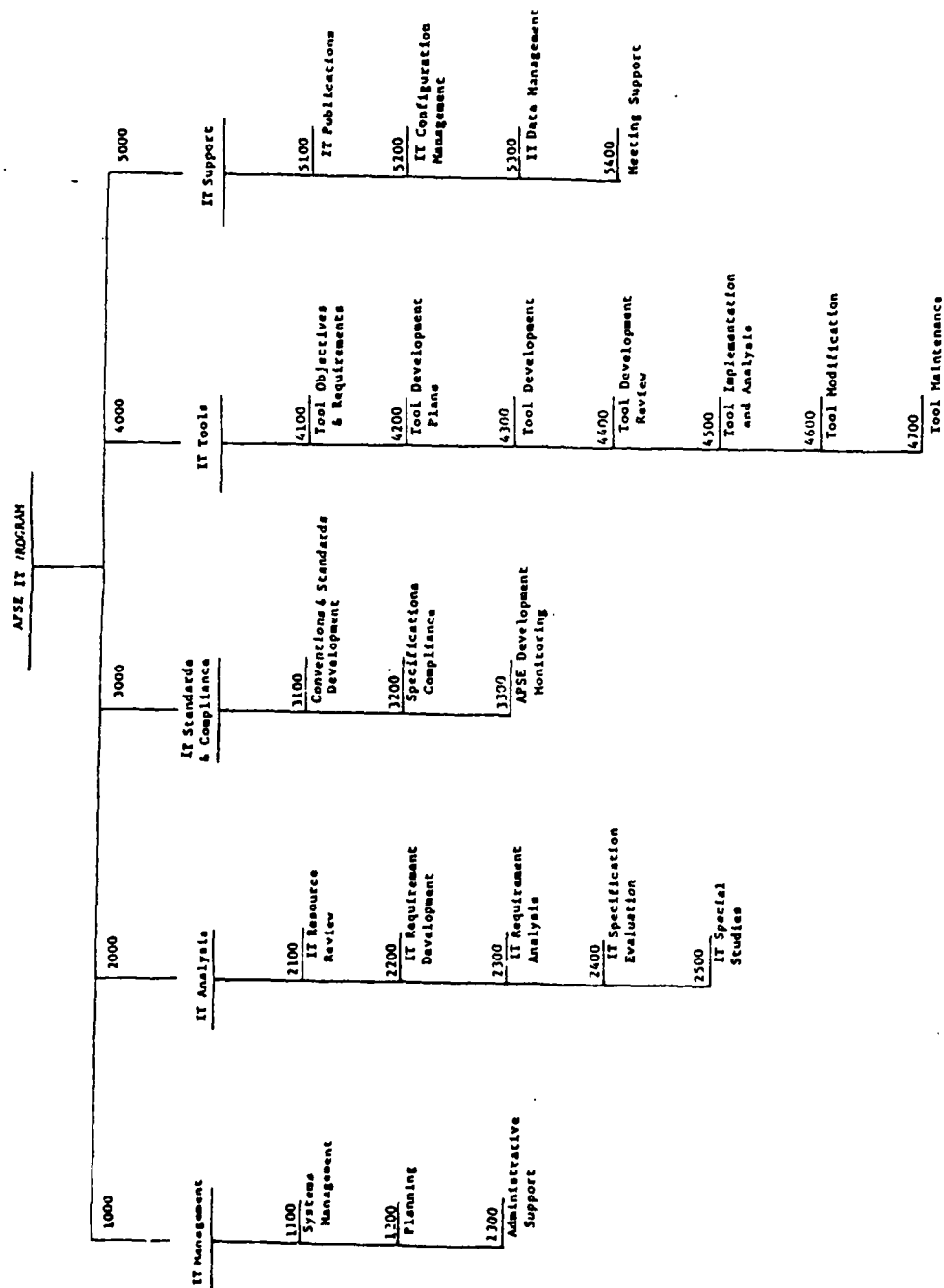


Figure 3 APSE IT Program Work Breakdown Structure

2000 APSE IT Analysis

2100 APSE IT Resource Review

This WBS element provides for the review of literature and documentation applicable to APSE IT requirements. Such literature and documentation will include subjects such as APSE requirements, specifications, conventions and guidelines.

2200 APSE IT Requirements Development

This WBS element provides for the development of requirements for APSE IT.

2300 APSE IT Requirements Analysis

This WBS element provides for the analysis of APSE IT Requirements developed under WBS element 2200. This analysis will be conducted to determine completeness, traceability, testability, consistency and feasibility.

2400 APSE IT Specification Evaluation

This WBS element provides for the analysis of AIE and ALS MAPSE tools and KAPSE's and the interface with respect to APSE IT specifications. The results of the analysis will include problem identification with proposed solution and issue identification with proposed resolution.

2500 APSE IT Special Studies

This WBS element provides for any technical analysis or study not mentioned elsewhere. Specifically included are studies resulting in methods for assessing the risk associated with achieving levels of APSE IT, and cost benefit analysis that will provide a quantitative means to assist in making recommendations and decisions concerning implementation.

3000 APSE IT Standards and Compliance

3100 APSE IT Conventions and Standards Development

This WBS Element provides for the development of guidelines, conventions and standards to be used to achieve IT of APSE's. These guidelines, conventions and standards are to be developed from APSE IT Requirements.

3200 APSE IT Specification Compliance

This WBS Element provides for: the development of procedures to determine compliance of APSE development with APSE IT specification of requirements, conventions and standards; the carrying out of these procedures to determine compliance including review and analysis of test reports.

3300 APSE IT Development Monitoring

This WBS Element provides for the continuing review of the AIE and ALS development efforts with respect to APSE IT. The monitoring is at a detailed level and seeks to discover all interfaces which affect IT. The purpose of the review is to identify issues and problems, to analyze the issues and problems, and to report the results.

4000 APSE IT Tools

4100 APSE IT Tool Objectives and Requirements

This WBS Element provides for the identification of objectives, criteria and requirements to be used for the selection of approximately three APSE IT Tools to be integrated into both the ALS and AIE. These tools will be used as test tools for APSE IT.

4200 APSE IT Tool Development Plans

This WBS Element provides for the analyses necessary to recommend the three specific APSE IT Tools to be developed. It further provides for making the recommendation, and developing plans for the development and acquisition of these three tools.

4300 APSE IT Tool Development

This WBS Element provides for the development and acquisition of the three APSE IT Tools to be integrated into the ALS and AIE.

4400 APSE IT Tool Development Review

This WBS Element provides for the monitoring of the APSE IT Tool development and participation in the APSE IT Tool Development review process. It further provides for the reporting of the results of monitoring and reviews.

4500 APSE IT Tool Test Application and Analysis

This WBS Element provides for the overseeing of the test application of the three or more APSE IT tools. It further provides for the development of guidelines for tool test applications and analysis.

4600 APSE IT Tool Maintenance

This WBS Element provides for the maintenance of the APSE IT Tools, after they are developed.

4700 APSE IT Tool Modification

This WBS Element provides for the modification of the APSE IT Tools which may be required, for example, to correct inadequacies in the first development or to respond to changing requirements.

5000 APSE IT Support

5100 APSE IT Publications

This WBS Element provides for the publication and distribution of APSE IT requirements, guidelines, conventions and standards.

5200 APSE IT Configuration Management

This WBS Element provides for the Configuration Management of all APSE IT documents generated and all tools developed in the APSE IT program.

5300 APSE IT Data Management

This WBS Element provides for the maintenance, storage and updating of all documentation and data in the APSE IT program. It further provides for the distribution of all data in the APSE IT program.

5400 APSE IT Meeting Support

This WBS Element provides for the technical support required in planning, preparing, conducting and reporting on formal APSE IT meetings. These meetings are held for the purpose of establishing APSE IT requirements, guidelines, conventions and standards.

3.2 SCHEDULES/MILESTONES/DELIVERABLES

It is anticipated that the following milestones will be achieved in the calendar year indicated:

1982

- The KIT and IAT groups selected and working
- Documentation and Design Reviews of AIE and ALS APSE efforts
- Requirements developed for three APSE IT Tools
- Contract let for three APSE IT Tools
- First Draft of APSE IT Requirements
- Second Draft of APSE IT Requirements
- Rough Draft of Conventions and Standards
- Draft Procedures Determination of APSE IT Compliance
- APSE IT Configuration Management Plan
- One Technical Report concerning AJPO IT efforts

1983

- Updated APSE IT Plan
- Documentation, Design and Test Reviews of AIE and ALS APSE Efforts
- Tool Development and Tool Development Reviews

- Three APSE IT Tools Delivered and Integrated into the AIE and ALS APSEs
- Third Draft of APSE IT Requirements
- Second Draft of Conventions and Standards
- Second Draft of Procedures for Determination of APSE IT Compliance
- First Draft Interface Specifications
- Two Technical Reports concerning AJPO IT Efforts

1984

- Updated APSE IT Plan
- Documentation, Design and Test Reviews of AIE and ALS Efforts
- Testing and Modification of 3 APSE Tools complete
- Final Draft of APSE Requirements
- Third Draft of Conventions and Standards
- Final Draft of Procedures for Determination of APSE IT Compliance
- Second Draft Interface Specifications
- Two Technical Reports concerning AJPO IT Efforts

1985

- Documentation, Design and Test Reviews of AIE and ALS APSE Efforts
- Final Draft of Conventions and Standards
- Final Draft of Interface Specifications
- Two Technical Reports concerning AJPO IT Efforts

Table 1 shows a further breakdown of these goals into deliverables, by quarters.

3.3 PROVISIONS FOR SPECIAL NEEDS

This APSE IT Plan emphasizes the development of requirements, conventions and standards. It is unusual in that it is written for a programming language and programming support environment that are also in the development states. At this point in development it is essential for the KITEC to provide a forum and act as a focal point for the Ada community, APSE developers and the DoD. This will provide broad input to the KIT from which a complete realistic set of requirements, guidelines, conventions and standards will be developed that account for ongoing APSE development and long term APSE needs.

Normally to achieve APSE IT the APSE itself would be written in Ada. However, STONEMAN recognizes that "in cases where there is a large current investment in software projects, written originally in other languages" provisions and guidelines must be developed that account for cost effective transitions to Ada environments. In the development of APSE IT requirements, conventions and standards the KITEC should provide cost benefit analysis with respect to their recommendations and decision concerning implementation.

Table 1 APSE IT Schedule (1982 CY to 1985 CY)

	1982				1983				1984				1985			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
APSE IT Mgmt		• APSE IT Effort	• APSE IT Effort	• Update IT Plan for 83		• APSE IT Effort	• APSE IT Effort	• Update IT Plan for 84		• APSE IT Effort	• APSE IT Effort	• Update IT Plan for 85		• APSE IT Effort	• APSE IT Effort	• APSE IT Effort
Requirement Development • APSE IT Rmt. Doc.		1st Draft		2nd Draft				3rd Draft				Final				
Conventions & Standards Development								2nd Draft				3rd Draft			Final	
Interface Specification Development								1st Draft				2nd Draft			Final	
Specification Compliance Development								2nd Draft				Final				
AIE & ALS Review								Tech Report				Tech Report			Tech Report	
Tool Objectives & Requirement Development	Final															
• Rmts for 3 APSE IT Tools																
Tool Plans Development	Final															
• Plans and Procedures for Tool Development																
Tool Development Review								Final								
Tool Test Application and Analysis								Report on Tool Integration into AIE & ALS				Tool Test Application Report				
Configuration Management								1st Draft, Final								
Plan Meeting Support	• Jan KIT, • Apr KIT, • Jul KIT, • Oct KIT, • Feb IAT, • Jun IAT, • Jan IAT															

During the initial phase of carrying out this APSE IT plan the KITEC will be studying and contributing to the IT aspects of APSE developments by the Army and Air Force (ALS and AIE). When the Navy begins its development of an APSE the KITEC will also concern itself with the IT aspects of its design. The KITEC will develop requirements, conventions and standards that can be used for validation testing. In addition APSE development by the private sector and international development should be addressed. Criteria for validation testing for all APSE development efforts should be established. In the future a central agent can perform IT validation testing on each APSE. The model for a strong central validation is the Ada Compiler Validation Facility.

APPENDIX A: GLOSSARY OF TERMS

(TBD)

APPENDIX B: APPLICABLE DOCUMENTS

(TBD)

ORIGINAL I & T SCHEDULE (SEE REVISION)

	1982				1983				1984				1985			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
APSC IT Mgmt		APSC IT Effort	APSC IT Effort	APSC IT Effort		APSC IT Effort	APSC IT Effort	APSC IT Effort		APSC IT Effort	APSC IT Effort	APSC IT Effort		APSC IT Effort	APSC IT Effort	APSC IT Effort
Environment Development		1st Draft	2nd Draft	3rd Draft		1st Draft	2nd Draft	3rd Draft		1st Draft	2nd Draft	3rd Draft		1st Draft	2nd Draft	3rd Draft
Conventions & Standards Development																
Interface Specification Development																
Specification Compliance Development																
ALC & ALS Review																
Tool Objectives & Requirement Development																
Tool Plans Development																
Plans and Procedures for Tool Development																
Tool Development Review																
Tool Test Application and Analysis																
Configuration Management																
Plan Meeting Support																

REVISED I & T SCHEDULE

	1902				1903				1904				1905				
	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
APSE IT Mgmt	Public Report • Update IT Plan for 83		Public Report		Public Report • Update IT Plan for 84		Public Report		Public Report • Update IT Plan for 85		Public Report		Public Report		Public Report		Public Report
Requirement Development • APSE IT Rpt., Doc.	2nd Draft				3rd Draft				Final								
Guidelines & Conventions Development		1st Draft			2nd Draft				3rd Draft								Final
Standards (Including Interface Specifications) Development	Rough Draft		1st Draft				2nd Draft								3rd Draft		Final
Specification Compliance Development	1st Draft				2nd Draft				Final								
AIE & ALS Review		Tech Report			Tech Report				Tech Report					Tech Report			
Tool Development Review			1st Tool reviewed	2nd Tool reviewed			3rd Tool reviewed										
Tool Test Application and Analysis					Report on 1st Tool Integration into AIE & ALS		Report on 2nd Tool Integration into AIE & ALS		Tool Test Application Report								
Configuration Management Plan		1st Draft	2nd Draft		Final												
Meeting Support	Oct KIT & KITIA Jan KIT Feb KITIA	Jan KIT Apr KIT Jun KITIA	Jul KIT Oct KIT & KITIA		Jan KIT Apr KIT Jul KIT Oct KIT									Jan KIT Apr KIT Jul KIT Oct KIT			

TERMS

CATEGORIES:

- . the aspects of an APSE which will potentially lead us to those interfaces which need to be standardized; also provide some criteria for deciding if standardization is necessary in a given area.

REQUIREMENT:

- . a condition or capability, either stated or implied, that must be met or provided by a system or system component. (IEEE)
- . An IT requirement addresses a high-level condition or capability which is relevant to IT and which shall be delineated by one or more conventions and/or standards. Thus IT requirements document those aspects which are determined to affect IT. By implication, any condition or capability which is NOT addressed by an IT requirement has been determined NOT to affect IT.
- . Each IT requirement will be stated, followed by alternatives and/or a rationale for its inclusion.

SPECIFICATION:

- . a document that defines requirements, details a design, or describes a product. (IEEE)
- .. For IT purposes, this term will not be used alone, but will always be qualified by the type of specification; e.g., IT requirements specification.

GUIDELINES:

- . recommendations that, if followed, will aid in achieving IT; for example, they may convey techniques for satisfying the standards or conventions. There will be recommendations for the tool writer as well as the KAPSE designer and implementor.
- . This document will cover those ideas which do not fit well under conventions or standards or which can only be encouraged (as opposed to enforced) or which do not bear strictly on IT.

CONVENTIONS:

- . a set of interfaces, services and practices whose use is recognized as contributing to IT and is to be voluntarily agreed to. Conventions are largely potential IT standards (see definition below); if they prove effective and practical, they may be adopted as IT standards.
- . IT conventions will not be subject to enforcement. The conventions will primarily take the form of "preliminary IT standards" which are being put forward as candidates for IT standardization. As such they are likely to initially form a larger set than the IT standards. In fact, several alternatives that address the same point may be part of the IT conventions at the same time. They can be viewed mainly as constituting a catalogue of potential future standards, and the document containing them becomes a holding bin while they are under consideration.

STANDARDS:

- . a set of interfaces, services and practices whose use has been established for achieving APSE IT. Tools and data bases which are to be ported between conforming APSEs can depend on these interfaces/services/practices only. It is a goal to be able to validate whether or not a given APSE meets the IT standards. Interfaces, services, and practices in this set are formally documented and approved for use by the AJPO in the construction of DoD-compatible tools, data bases, and KAPSEs.

INTERFACE: (deferred pending STONEMAN modification)

TOOL: (deferred pending STONEMAN modification)

(IEEE): Definitions attributed to the IEEE are taken from a draft of an IEEE Glossary of Software Engineering Terms, dated 21 January 1982; Russell J. Abbott, Glossary Consistency Coordinator.

KAPSE INTERFACE WORKSHEET

=====

Date: 19 Aug 82

KAPSE Interface Group: 1

KAPSE INTERFACE CATEGORY: 1A. Program Invocation and Control

1. EXPLANATION:

Program invocation and control consists of a minimum set of facilities that must be provided by the KAPSE to invoke and control the execution of an Ada program. The category is concerned with the methods by which execution is initiated, suspended, resumed, terminated, or interrupted; the methods used to specify parameters and options; and the methods to used to display or report on the status of an execution.

This category distinguishes between a program, which is the executable image, and a process which is the executing image of a program.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

Standard means of invoking and controlling programs are required for transportability of the Command Language Processor and the Debugger.

3. STANDARDIZATION PROBLEMS:

Program invocation :

- o The requirements for program invocation and control raise issues of the minimum capabilities that a machine/operating system combination must provide to support a KAPSE.
- o On some operating systems it is impossible for programs to invoke another program or itself.

Process Control :

- o Facilities must be available to interrupt a running program. Interrupts that can be processed should include both hardware and software interrupts. In particular, a user should be able to suspend, resume, or terminate execution.
- o Identify if facilities must be provided for running a program with a monitored context for debugging.

4. PRIORITY:

5. RELATED CATAGORIES:

Category List

- a. 1D
- b. 4A
4B
4C
4E

6. APPROACHES:

- a. ALS:
- b. AIE:
- c. Existing Standards:
- d. Other:

- Various wait options should be availabe after a subprocess is invoked.

- Calling task of caller waits
- Caller waits.
- No one waits.

7. RISK:

8. ACTIONS TO TAKE:

K A P S E I N T E R F A C E W O R K S H E E T

=====

Date: 19 Aug 82

KAPSE Interface Group: 1

KAPSE INTERFACE CATEGORY: 1B. Initiate/Maintain User Environment

1. EXPLANATION :

This category is concerned with the orderly provision of system access and resources for recognized users and the orderly termination of these when no longer required.

This category includes granting system access to a user based on access codes and accounting verification. This allows the user to establish a personalized user working environment. This environment may include but not be limited to the connection to a pre-established collection of data base objects and command procedures, the execution of pre-defined command procedures at login time, and restrictions on tool or data base object access and usage (security). This also includes the resource allocation and billing for system resources, the collection and disposition of system generated messages. Within this category is the alteration of personalized environments both permanently and temporarily during a session along with the ability to save and recall a pre-existing environment.

Termination of a user environment includes the execution of pre-established command procedures at logoff, the disposing of programs and data base objects, and the passing of a user environment to a subordinate task.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

3. STANDARDIZATION PROBLEMS:

- o It is not clear if standard logon/logoff facilities will be provided.
- o It is not clear if standard interfaces will be defined for tools that provide logon/logoff services.
- o It is not clear how system security will be impacted from logon/logoff services in an APSE environment.

4. PRIORITY:

5. RELATED CATEGORIES:

6. APPROACHES:

- a. ALS:
- b. AIE:
- c. Existing Standards:
- d. Other:

7. RISK:

8. ACTION TO TAKE:

KAPSE INTERFACE WORKSHEET

=====

Date: 24 Aug 82

KAPSE Interface Group: 1

KAPSE INTERFACE CATEGORY: 1C. Device Interactions

1. EXPLANATION:

Device interaction consists of facilities that the KAPSE must provide to control the set of devices on the host machine. The category includes terminal input output, disk and tape input output, and communications to other computers and special (one of a kind) devices.

Device interactions include transmission of data and control information between programs or processes and all forms of input/output devices. This category also covers character sets, communication protocols, device service routines in the KAPSE and host system, and high-level and low-level input/output interfaces in the Ada language.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

3. STANDARDIZATION ISSUES:

Terminal type devices:

- o If the hardware prevents transmitting all character codes then mechanisms must be provided for the disposition of characters that cannot be transmitted.

Physical input/output considerations:

- o The KAPSE may need to permit Ada low-level input/output. This facility would be useful when one-of-a-kind devices are added to a host machine.

- o Physical input/output is a major security problem. Programs with the ability to do physical input/output can gain control of the host.
- o If low-level input/output is provided, the user must be able to access the data and control channels on physical devices.

Logical Device Considerations:

- o If no physical input/output is provided then device drivers must be provided for a wide variety of devices and it must be easy to add new devices to the set of supported devices. Generic or parameterized device facilities may be used for this.

4. PRIORITY:

5. RELATED CATEGORIES:

- a. 4A
- b. 4B
- c. 4C

6. APPROACHES:

- a. ALS:
- b. AIE:
- c. Existing Standards:
- d. Other:

7. RISK:

8. ACTIONS TO TAKE:

KAPSE INTERFACE WORKSHEET

=====

Date: 19 Aug 82

KAPSE Interface Group: 1

KAPSE INTERFACE CATEGORY: 1D. The Subset MAPSE

1. EXPLANATION:

The Subset MAPSE (SMAPSE) consists of a minimum set of tools that runs on a KAPSE and provides support to both the human user and other MAPSE programs. This category includes candidates for a minimum set of tools necessary to transport scripts and programs between APSEs, such as the Ada Compiler, the Linker, a Command Language Processor, an Editor, and a minimum file system.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

It will become necessary to move entire software projects, including the programmers between Ada Programming Support Environments (APSEs). A project includes:

- developed software
- Ada support programs
- programmers
- minimum scripts.

For the purpose of this discussion a "Minimum Script" is the minimum set of commands needed to install and test an Ada program on a new APSE. A Minimum Script may therefore use some minimum set of tools and could

include editor and linker commands. The minimum tool set required for IT does not necessarily mean that tools should have minimum capabilities. This may be an appropriate place to define a very capable editor or command language processor.

3. STANDARDIZATION PROBLEMS:

Command Language :

- o The degree to which the command language should be Ada-like is a standardization issue.
- o It is not clear if the command language will limit the number of available variables.

Linker :

- o The linker may require its own command language to properly process overlays.
- o The linker may require its own command language to properly process different configurations of the same software.

File system :

- o It is not clear if direct disk access may be required to support certain data base designs.

4. PRIORITY:

5. RELATED CATEGORIES:

Category List

6. APPROACHES:

a. ALS:

b. AIE:

c. Existing Standards:

d. Other:

7. RISK:

8. ACTIONS TO TAKE:

KAPSE INTERFACE WORKSHEET

=====

Date: 19 August 1982

KAPSE Interface Group: 2

KAPSE INTERFACE CATEGORY: 2A. Basic I/O Interfaces

1. EXPLANATION:

The Basic I/O Interfaces consist of those KAPSE services required to support primitive I/O to files and devices. These services include the lowest level of I/O services to be made available to Ada programs running on the KAPSE.

The Basic I/O Interfaces directly support the following packages described in the Ada Language Reference Manual (July 1982):

SEQUENTIAL_IO
DIRECT_IO
TEXT_IO

In addition, some form of SCREEN_IO and keyed access I/O may also be considered as candidates for standard packages.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

The Basic I/O Interfaces are intrinsic to IT inasmuch as they require the KAPSE to support I/O as defined in the Ada Language Reference Manual. The standardization of higher-level I/O interfaces will provide enhanced IT

3. STANDARDIZATION PROBLEMS:

- o It is not clear whether the packages as defined in the LRM are the lowest level of I/O to be made visible by the KAPSE, or whether these packages are to be supported by a set of more primitive (yet still visible) functions.
- o It is not clear whether files must be "typed", whether such typing is restricted to "categories", or whether such typing is within the purview of Basic I/O Interfaces or Data Management Interfaces.
- o The package LOW_LEVEL_IO has not been included in the list of packages for which support is required. It must be determined whether low-level i/o is a valid concept for a KAPSE to support.
- o The level at which Basic I/O is to be supported by the KAPSE is directly linked to the level at which the KAPSE supports database management functions. In particular, it is unclear which facets of database management (and I/O) beyond those required for security and integrity should be included in the KAPSE.
- o SCREEN_IO and keyed access I/O (as well as other special purpose I/O for non-standard files and devices) may more appropriately be the target for guidelines and conventions rather than standards. Or they might be made standard but optional.

4. PRIORITY:

High. Since I/O is the primary interface of all Ada programs, and since I/O is a relatively well-understood area within which support levels may be identified, Basic I/O should be one of the first candidates for standardization.

5. RELATED CATEGORIES:

Category List

a. 1C

b. 2B

6. APPROACHES:

a. ALS:

b. AIE:

c. Existing Standards:

d. Other:

7. RISK:

8. ACTIONS TO TAKE:

K A P S E I N T E R F A C E W O R K S H E E T

Date: 19 August 1982

KAPSE Interface Group: 2

KAPSE INTERFACE CATEGORY 2B. -- Database Management and Control

1. EXPLANATION:

The Database Management and Control Interfaces consist of those services required to support APSE programs in the following areas:

- o Basic database object operations -- open, close, delete, create; basic database objects are atomic objects that, from the point of view of database management, are unstructured.
- o Attribute manipulation and query operations -- create attribute, delete attribute, modify attribute, read attribute, write attribute, and list object with specified attributes; in addition, the system-defined attributes must be defined along with their semantics, and provision for user-defined attributes must be defined.
- o Category operations -- create and delete categories; definition of categories and associated semantics, provision for standard system-defined categories as well as user-defined categories.
- o Structural operations -- open, close, create, delete, read, and write directories, partitions, variations (variation headers), revisions (revision headers), and other composite objects; all of these forms of composite or structured objects must be defined along with their semantics.

- o Access control operations -- grant, deny, revoke, and request access to database objects and/or their attributes; in addition, the syntax and semantics of the access control attribute must be defined.
- o History maintenance operations -- create, record, update, and delete derivation histories, record derivation dependencies, and creation/modification time-date stamps; the syntax and semantics must be specified for each of these history components.
- o Archive and back-up operations -- full and incremental.
- o Recovery operations -- system-initiated and user-initiated.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

These interfaces are, for the most part, irrelevant to those tools that transform database objects independently of database structure. Such tools will receive database object names as arguments and will do little else than basic I/O on these names. On the other hand, there are tools (such as the Command Language Interpreter and Configuration Manager) that operate on and manipulate the structure and attributes of the database. The IT of these latter tools directly depends on the above interfaces.

3. STANDARDIZATION PROBLEMS:

- o A major issue is whether the functionality supplied by these interfaces should reside totally in the KAPSE or whether the functionality should be split between the KAPSE and the MAPSE. If standard MAPSE packages are to be provided, then this issue is largely irrelevant to IT, but highly relevant to KAPSE rehostability.

- o Related to the above issue is the question of the degree of freedom that is to be given to MAPSE programs. That is, is it valid to allow MAPSE programs to circumvent database management and to use Basic I/O? Presumably, basic I/O would be more efficient and free of the encumbrances entailed by history management.
- o The pervasive problem that affects all aspects of data management and control standardization is the lack of a standard terminology.

4. PRIORITY:

Moderate. As indicated in the relevance subsection, there are many tools that can be easily transported without standardization in this area. Moreover, this is an area with considerable divergence in proposed approaches. On the other hand, an essential tool from the point of view of "user" transportability is the command language interpreter, which is highly dependent on standards in this area.

5. RELATED CATEGORIES:

Category List

- a. 1A
- b. 2A

6. APPROACHES:

a. ALS:

b. AIE:

c. Existing Standards:

d. Other:

7. RISK:

8. ACTION TO TAKE:

KAPSE INTERFACE WORKSHEET

=====

Date: 19 Aug 1982

KAPSE Interface Group: 2

KAPSE INTERFACE CATEGORY 2C. Inter-Tool Data Interfaces

1. EXPLANATION:

The Inter-program Data Interfaces include the data formats and intermediate languages required to support data communication between APSE programs.

An Ada Intermediate Language is required in order to retain and transmit Ada program information for:

- o separate compiler phases
- o optimizers
- o static and dynamic analyzers
- o debuggers
- o test tools
- o pretty-printers
- o syntax oriented editors

Two other data interfaces, program library format and object file format, are used by the following tools:

- o compiler
- o linker
- o loader
- o debugger

Data interfaces for the following sets of tools are also included:

- o documentation tools
- o requirements and specification tools
- o project management tools
- o configuration management tools

The last major data interface is the implementation-dependent aspect of the Ada language, namely implementation-defined pragmas and representations. In particular, and in accordance with the Language Reference Manual, the following aspects must be considered:

- o The form, allowed places, and effect of every implementation dependent pragma.
- o The name and the type of every implementation dependent attribute.
- o The specification of the package SYSTEM.
- o The list of all restrictions on representation clauses.
- o The conventions used for any system generated name denoting system dependent components.
- o The interpretations of expressions that appear in address clauses, including those for interrupts.
- o Any restriction on unchecked conversions.
- o Any implementation dependent characteristics of the input-output packages.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

The implementation-dependent aspects of the language have a pervasive effect on IT. While they are not allowed to impact the "legality" of a program, the implementation-defined pragmas can affect the portability of an Ada program to a given APSE as well as the performance of a program on an APSE.

The other data interfaces affect IT with respect to relatively small subsets of tools. Standardization is necessary in order to permit transportability of individual tools without requiring transporting an entire tool subset.

3. STANDARDIZATION PROBLEMS:

It must be recognized that for the implementation-dependent language aspects, guidelines, conventions, and standards imposed for tool portability will not affect the language used for embedded applications.

The other data interfaces lie outside of the KAPSE (except, perhaps, the object file format), and standardization will thus place requirements on APSE programs and libraries rather than on the KAPSE. The major problem is to design interfaces that are efficient yet do not overly constrain innovation in tool design.

A final consideration is whether this category should be merged with category 1D.

4. PRIORITY:

High for the implementation-dependent language aspects. Good programming practice mandates the minimization of dependence on implementation-specific features. Standards in this area could enforce this mandate.

Low for the other aspects of inter-tool data interfaces. Without standards in these areas, transportability of individual tools will be limited but transportability of tool sets will be largely unaffected. An exception is the object file format, which may be directly interpreted by the KAPSE itself. It would be highly desirable to standardize a self-describing object file format that could serve a wide variety of architectures.

5. RELATED CATEGORIES:

Category List

a. 3A

6. APPROACHES:

a. ALS:

b. AIE:

c. Existing Standards:

d. Other:

7. RISK:

8. ACTIONS TO TAKE:

KAPSE INTERFACE WORKSHEET

=====

Date: 19 August 1982

KAPSE Interface Group: 3

KAPSE INTERFACE CATEGORY: 3A. Ada Program Run-Time System (RTS)

1. EXPLANATION:

The Ada Program RTS provides the basic run-time support facilities that are required to support Ada semantics within the APSE. These may include, but are not limited to, closed-routine (non-inline code) provision for multitasking, task/program memory management exception handling certain string operations, program initialization/termination and standard package INPUT_OUTPUT capabilities.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

The design and development of tools including man-machine interface and data base access segments that are heavily dependent on particular computer architecture system calls may have significant impact on the interoperability and/or transportability of the tool.

3. STANDARIZATION PROBLEMS:

- o Identify the relationship between command-language input form of expression and tool interface to KAPSE.
- o Other considerations:

- Care must be taken so that proposed approaches do not limit extension of APSE's to distributed systems (See category issues).
- Care must be taken so that if proposed approaches do not limit design and implementation in support of future system Security requirements.
- o Identify how KAPSE RTS basic services and linked I/O functions interact when tasks are activated within the Ada environment.
- o Determine how amenable to extensions is the KAPSE RTS implementation for new device drivers.
- o Identify the impact distributed tasks have on the RTS.
- o Identify the impact synchronous/asynchronous I/O requirements have on the RTS.

4. PRIORITY:

5. RELATED CATEGORIES:

Category List

- a. 1C

6. APPROACHES:

- a. ALS:
- b. AIE:
- c. Existing Standards:
- d. Other:

7. RISK:

Med/High - may vary dependent upon implementation within/without a particular family of processors. This may be related to word size, procedure calls, etc of the implementations.

8. ACTIONS TO TAKE:

- o Identify the specification of the run-time support system.
- o Identify the relationship between command-language input form of expression and tool interface to KAPSE.
- o Identify privileges that should be specificable, grantable, delegatable, or revocable with respect to RTS accessibility as a function of user, project, module, process, etc. at the command language level. Define if enacted by tools directly or by program.

KAPSE INTERFACE WORKSHEET

Date: 19 August 1982

KAPSE Interface Group: 3

KAPSE INTERFACE CATEGORY: 3B. Bindings and Their Effect on Tools

1. EXPLANATION:

Binding is the association of a value or referent to an identifier. This category is concerned with link-or execution-time assignment of concrete subprograms, devices, and data-base objects to identifiers in Ada programs.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

3. STANDARDIZATION PROBLEMS:

- o Decide on whether the binding must be static or dynamic or both.
- o Identify the facilities that exist for deferring commitment with respect to binding and exploiting the deferral of commitment for debugging, exploration, evaluation, withdrawal, and change of commitment.
- o Identify facility that should exist for measuring performance benefits of alternative forms of binding.
- o Identify how the KAPSE handles conflicting RTS service linking assignments, e.g., segmented architectvie memory mapping services when detected during task execution within Ada environment.

o Other considerations (Group 4 cross-category issues):

- Proposed approaches do not limit the capability of the KAPSE, or APSE tools, to effect APSE system recovery.
- Proposed approaches do not limit extension of APSE's to distributed systems.
- Proposed approaches do not limit design and implementation in support of future system Security requirements.
- Proposed approaches do not limit extension of APSE capabilities for Target-Dependent functionality.

4. PRIORITY:

5. RELATED CATEGORIES:

6. APPROACHES:

- a. ALS
- b. AIE
- c. Existing Standards
- d. Other

7. RISKS:

8. ACTIONS TO TAKE:

- o Identify the operations that must be supported by the KAPSE regarding diagnostic, interpretive, and full-speed execution of programs incorporating static and dynamic binding to subprograms and to data.
- o Identify the facilities for making and exploiting a commitment to static binding.

KAPSE INTERFACE WORKSHEET

=====

Date: 19 August 1982

KAPSE Interface Group: 3

KAPSE INTERFACE CATEGORY: 3C. Performance Measurement

1. EXPLANATION:

Performance measurement is the selection, collection, and recording of static and dynamic information about APSE activity and resource utilization. The KAPSE must be capable of capturing such information regarding all operating system activities and be able to associate such information to various levels of identity, e.g., task, processor, user, project, etc.

Performance measurement may be divided into three classes for consideration:

- Class 1 - KAPSE host internal for the collection of performance data related to the host system itself.
- Class 2 - KAPSE host-to-target for the collection of performance data related to the target system.
- Class 3 - KAPSE tool frequency usage for the collection of data related to frequency of tool/tool sets used in the APSE.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

The design and development of tools including man-machine interface and data base access segments that are heavily dependent on particular computer architecture system calls may be significant impact on the interoperability and/or transportability of the tool.

3. STANDARDIZATION PROBLEMS:

- o Determine what access should exist to clocks and hardware locations, memory control, logical and physical device configuration, and networks, etc.
- o Identify the mechanisms that should exist to support high-level event recording. Examples of high-level events are:
 - Logon/Logoff
 - File Open/Close
 - History of Tool invocation
- o Identify differing operational objectives that can be identified and how easy is it to activate/deactivate the needed KAPSE process activity that supports such data collection.
- o Other considerations (Group 4 cross-category issues):
 - Proposed approaches do not limit the capability of the KAPSE, or APSE tools, to effect ASPE system recovery.
 - Proposed approaches do not limit extension of APSE's to distributed system.
 - Proposed approaches do not limit design and implementation in support of future system Security requirements.
 - Proposed approaches do not limit extension of APSE.
 - Transaction recording is covered in Recovery.

4. PRIORITY:

5. RELATED CATEGORIES:

a. 3A

b. 2B

6. APPROACHES:

a. ALS:

b. AIE:

c. Existing Standards:

d. Other:

7. RISK:

8. ACTIONS TO TAKE:

- o Identify KAPSE support mechanisms that should exist to support selection, collection and recording of dynamic information on process resource utilization as a function of time. Identify KAPSE support mechanisms that exist for tuning KAPSE characteristics based on evaluation of such information.

KAPSE INTERFACE WORKSHEET

=====

Date: 19 Aug. 1982

KAPSE Interface Group: 4

Kapse Interface Category: 4A. Recovery Mechanisms

1. EXPLANATION:

Recovery mechanisms are the means by which an APSE becomes a highly robust system that can protect itself from user and system errors, that can recover from unforeseen situations and that can provide meaningful diagnostic information to its users.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

This topic cannot add to the standardization and cost saving objectives of a KAPSE but without consideration of this area and effective implementation, a successful tool standard implementation would not be possible.

3. STANDARDIZATION PROBLEMS:

- o A hardware failure must be recovered from and data protected. Examples of hardware failures include: parity errors, power failures, loss of communication link from a remote user, device failures, etc. The recovery mechanism would be host dependent and some of the backup must be external to the KAPSE such as backup tapes. Also consideration must be taken to assure that database or files are not contaminated due to hardware problems.
- o Once major system failures are discovered there should some standard way to determine at what point the system must be recovered from.

4. PRIORITY:

5. RELATED CATEGORIES:

Category_List

a. 1A.

b. 1C.

c. 4B.

d. 2A.

e. 4C.

6. APPROACHES:

a. ALS:

b. AIE:

c. Existing Standards:

d. Other:

7. RISK:

Failing to define standardization in this area would result in tool interfaces not being compatible and uniformly defined. Although the KAPSE interface and recovery mechanism are probably not consistent between the AIE and ALS, this requirement must be met at any cost. Over standardization is bad in any instance and tends to limit creativity and hinder getting the job done. At this point it is hard to distinguish where standardization ends and over-standardization begins.

8. ACTIONS TO TAKE:

Review issues by committee and decide whether recovery issues should be addressed or incorporated with other topics. If key issues, start research of how recovery mechanisms are implemented in AIE and ALS. If their specifications handle it differently, the KIT should decide on the proper approach.

KAPSE INTERFACE WORKSHEET

=====

Date: 11 Oct 82

KAPSE INTERFACE GROUP: 4

KAPSE INTERFACE CATEGORY: 4B. Distributed APSE

1. EXPLANATION:

An APSE is distributed if its KAPSE resides on or executes on a configuration including more than one host computer system, or if the KAPSE supports invocations of tools residing on other hosts (than the KAPSE's host), or if the KAPSE-encapsulated data base physically resides (in part or whole) on a host other than the KAPSE's host. It is possible that equipment-interface requirements will be inconsistent with the KAPSE interface mechanisms prescribed for the non-distributed MAPSEs of the early 1980's. Examples include:

- o utilization of "intelligent terminals" whereby some computations (e.g., editing) by the APSE user are off-loaded from the APSE's central CPU (possibly with local storage in such terminals),
- o systems similar to the National Software Works (NSW) whereby the APSE shields the user from knowledge about what networked host executes each user step (giving the user the illusion of a monolithic host).
- o data bases that are stored remotely or according to distributed data base schemes (either with or without user cognizance).

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTIBILITY:

This category has medium or future relevance as regards Interoperability and Transportability. In general, we believe that successful KAPSE interfaces and tool-writing conventions should shield tools and users from knowledge about whether the KAPSE implementation is distributed or not. However, the inevitability of "smart terminals", data base machines, and other innovative equipment which will be interfaced to futures APSEs presents potential incompatibilities in interface methods (with each other and with existing host-software methods) that threaten effective Interoperability and Transportability.

3. STANDARDIZATION PROBLEMS:

- o A key problem is whether distributed APSE's (according to the above explanation) can be designed which are consistent with Stoneman requirements. In the case of tools executing on separate CPU's from the KAPSE, either tools execute low-level host services directly (a violation of KAPSE encapsulation of host dependencies, affecting tool portability), or an inconceivably inefficient implementation results with each tool networking to KAPSE for such services (with network call itself being host dependent), or a KAPSE is partially or redundantly implemented on each of multiple processors (introducing extra dimensions of synchronization/communication and control problems). In the case of distributed data bases, it is clear that access mechanisms from the KAPSE must be replicated on each machine which hosts part of the data base; implementation schemes must provide required Stoneman capabilities to achieve configuration control and maintain all possible inter-object relationship links. Is there

some narrow discipline for distributing the data base initially (required to achieve Stoneman data base requirements), and does it cover dynamic expansion of the data base as will be typical in project evolution?

- o It is not completely resolved that the standard interface set should always completely shield tools and users from knowledge about quantity of processing units which implement an APSE. Although there is a general presumption that this shielding is good, a few exceptions have been cited (not universally agreed upon), e.g., the need for user cognizance of user and data-base path names over multiple hosts, and user interfaces specific to special equipment such as smart terminals.
- o Given the inevitability of smart terminals, certain tool portability is meaningless. There currently exist no guidelines for KAPSEs to provide non-standard interfaces to accommodate such devices, nor demonstrably usable approaches for layering such interfaces atop a minimal standard interface set. In addition to smart terminals, implementations of certain other tool functionality (some of which will be desirable to use in APSEs) in hardware or other "self-contained" devices will likely be introduced in this decade. This will further stress the interface standards and threaten at least Transportability considerations. Whether or not the standard KAPSE interface specification will limit the usage of such devices (to a portable subset of their functionality) or provide a controlled "escape" allowing full exploitation of such functionality is an issue to be resolved.
- o Considering distributing the data base of an APSE, there will be particular challenges in defining standard interfaces for access mechanisms which completely shield the nature of data base distribution. Under current distributed data base schemes, there may be certain parameters of tools' data-base interfaces which affect

choice of machine or storage device containing the manipulated data base object; this may imply both a user-cognizant philosophy and a need to standardize in the KIT effort. User-name-oriented control schemes represent one such current approach.

- o Expanding the concern to general distributed systems (networked instances of APSEs, which may each be implemented by traditional single-host methods -- nominally out of the scope of this category), there is a need for standard interfaces to networks. A uniform approach to the degree of networking considerations visible to or controllable by individual tools and users (versus communication logic wholly encapsulated by the KAPSE) must be agreed upon early because the implementation of the approach may widely affect categories of interfaces.
- o An example of a problem likely to confront future large projects is the desire to use separate, networked APSEs for software development of subsystems, but with the need for complete configuration management (including control of the name space of software components) across the entire project. This problem is alleviated if a single APSE distributed across multiple hosts is achieved, but the acceptability of that practice from a resource utilization or company practice point of view will be an issue. Therefore, the previous issue of separate, networked APSEs needing standard communication interfaces may be only the predecessor of a harder problem: distributing aspects of project management and control between APSEs.
- o In general, KAPSE designers need to continually ask if it is possible to define an interface set which interferes with a possible distributed implementation. When a situation is identified which answers that affirmatively, reconsideration must be given to the penalties so involved and the approaches/costs of alleviating the situation.

4. PRIORITY:

MEDIUM/LOW. While the early 1980's ALS and AIE have no requirements for distribution, future APSEs will. There are several issues concerning whether a standard KAPSE interface specification is possible which shields users and tools from information about whether the APSE implementation is distributed, thereby achieving acceptable degrees of interoperability and transportability.

5. RELATED CATEGORIES:

- a. 1A
- b. 1B
- c. 1C
- d. 2A
- e. 3A
- f. 3B
- g. 3C

6. APPROACHES:

- a. ALS
- b. AIE
- c. Existing Standards
- d. Other

7. RISK:

There appears to be some risk that de-emphasizing Distributed APSE considerations will jeopardize Interoperability and Transportability. For the near term, KIT believes an interface set can be achieved (almost without concentrating on Distributed APSE considerations) which adequately shields tools and users from knowledge of implementation distribution (not because failure of the standard interface set to accommodate distributed implementation would hurt little -- presumably such failure could extract heavy prices given the future of micros and distributed systems). [Also, this risk assessment does not address difficulties in achieving distributed APSE implementations; that is presumed to be a state-of-the-art challenge, but definition of a sufficient "shielding" interface standard may not be.]

8. ACTIONS TO TAKE:

KIT is not completely satisfied that germane IT concerns arising from the Distributed APSE perspective have already been factored into the other related categories, even though many aspects of these issues are now reflected on other KIWs. While this KIW may sit dormant during the process of drafting interface specifications and conventions, the issues raised in it will be used during review of those (and successive) draft(s). Part of that future review process will be to stay abreast of advances in special devices and networking technology, to provide concrete interfaces to be reviewed for compatibility with the proposed standards and conventions.

KAPSE INTERFACE WORKSHEET

=====

Date: 19 August 1982

KAPSE Interface Group: 4

KAPSE INTERFACE CATEGORY: 4C. Security

1. EXPLANATION:

Computer security includes issues of proper isolation of separate users (and their APSE objects) from each other, and issues of multilevel DoD security classification schemes when an APSE is to contain and handle classified data and tools. In addition, computer security includes issues of controlled sharing of computer resources, and notions of discretionary access. Discretionary access is a protection mechanism in addition to any DoD security mechanism which may be in effect during a given session.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

The importance of security to the DoD community is highlighted by the newly established Computer Security Center within the DoD. Requirements for security within the context of an APSE are demonstrated by the recent RFP from the U.S. Army (CECOM, Fort Monmouth) calling for a secure operating system for the Military Computer Family. Although this RFP is predicated on the ALS, we anticipate that future RFPs will be written based on the AIE also.

To avoid having such efforts be start-from-scratch in each case, it is vital to identify a common set of primitives to be provided by any given KAPSE that will suffice to implement a desired security policy. Such a set of primitives will also lead to IT of tools and data base objects within the context of a given security policy.

3. STANDARDIZATION PROBLEMS:

- o Present approaches should not limit the design and implementation of future system security requirements.

DISCUSSION: Achieving this goal requires a KAPSE to make primitives available, which are applicable to a variety of security models. Primitives would be of the following sort:

- What are the entities recognized by the system? There should be primitives for defining objects such as files, segments, database objects, interprocess communication objects (e.g., buffers, signals), subjects (e.g., processes), etc.
- There should be primitives for marking these types of entities with a given security level, and for passing this marking to tools, processes, and other entities which require the marking.
- There should be primitives for defining an object which represents the desired security policy. In some models this object is called an "access matrix", and shows allowable types of access of a subject to an object. In other models, (e.g., Bell and LaPadula), the desired security policy is represented by axioms, and thus there is no need to explicitly embody all allowable accesses. Nevertheless, such an object is necessary to maintain passwords, audit trails, etc.
- There should be primitives for creating trusted processes, which are allowed to violate a given security policy to perform needed functions such as login, change access level, etc. These trusted processes should only be able to execute under controlled circumstances, and their privileges should not be disseminable to other processes.

- As mentioned briefly above, there should be primitives for maintaining an audit log, to be monitored on an ongoing basis, as well as analyzed subsequent to security-related events.
- There should be primitives for creating type managers for the various kinds of entities recognized by the system. These managers would provide resource encapsulation for the entities. There should also be primitives by which the type managers can solve the object migration problem, i.e., if a given object migrates to an environment no longer under the control of the type manager, and then returns to the original environment, how can the type manager ascertain that the object is in a legitimate state?
- Etc.
- o The KAPSE should make available primitives that restrict all operational aspects of system execution to allowable state changes with respect to a given security model.

DISCUSSION: These primitives deal with issues such as system startup, shutdown, checkpoint, restart, down line loading, exporting data base objects, color changes, running in dedicated mode, running in multilevel mode, providing appropriate markings on all hard copy output, managing passwords, login ids, dial-up phone lines, etc.

- o The KAPSE should make available primitives to implement a variety of desired discretionary security policies.
- o The KAPSE should make available primitives to allow for continuous protection of all system objects, and provide for accountability.

DISCUSSION: It is imperative that system objects embodying security policy information (e.g., access

matrices, security markings) be protected from unwarranted modification, and that all security checks are continually enforced.

- o Tools and data base objects should be written in a modular fashion, partitioned into security-relevant and security-independent components.

DISCUSSION: This would allow the security-relevant components to be stripped away in contexts where security is not required, leading to more efficient operations.

- o Misc. Issues: The above issues deal with traditional types of security concerns. In this miscellaneous category, we note the following:
 - Security models will have to cover networks, databases, host/target environments, workstations, in addition to operating systems.
 - In addition to the software/modeling aspects of security dealt with above, there are the following types of security concerns: physical security, administrative security, personnel security, communications security, emanations security, and hardware/software security interfaces.
 - It may be a problem to provide the required security primitives if a KAPSE encapsulates a host OS which was designed without security issues in mind.

4. PRIORITY:

HIGH, with respect to the importance of this issue for KAPSEs that begin to appear after the first iterations of the current ALS and AIE efforts.

5. RELATED CATEGORIES:

- a. 1A
- b. 1B
- c. 1C
- d. 2A
- e. 2B
- f. 3A
- g. 3B
- h. 3C
- i. 4A
- j. 4B
- k. 4D

6. APPROACHES:

- a. ALS
- b. AIE
- c. Existing standards
- d. Other:

1. Rushby model (cf KITIA Working Group 1 Interim Tech Note, "Program Invocation and Control").

2. Kernel models.

3. Capability models.

4. MCFOS.

5. Ongoing research results as they become public, e.g.,
secure network protocol handlers.

7. RISK:

The current ALS and AIE efforts will not achieve these security features, since security has not been part of the design from the outset. Building on the experience of these current efforts, coupled with ongoing research in security, there is only a MODERATE risk that future KAPSE efforts will fail to achieve required security features, provided that such features are part of the effort from the beginning of the requirements phase.

8. Actions To Take:

- o Determine precisely a variety of sets of primitives to be provided by KAPSEs to implement different security models. This determination will require parallel research into security models.
- o Determine how well these primitives are provided in the current ALS and AIE efforts. Answers may be of the following types:
 - Primitive is directly provided by the host OS and percolated up by the KAPSE.
 - Primitive is provided by the KAPSE in terms of lower level host OS objects.
 - Primitive could be provided (with some difficulty) in terms of existing KAPSE objects.

- Primitive is virtually impossible to provide; it is too inefficient, or it conflicts with existing design.
- o Perform cost/benefit analysis of existing security techniques.
- o Monitor the ongoing work of the Computer Security Evaluation Center and relevant projects (e.g. MCFOS) and feed these results into subsequent KITIA documents.

KAPSE INTERFACE WORKSHEET

=====

Date: 19 Aug 82
KAPSE Interface Group: 4

KAPSE INTERFACE CATEGORY: 4D. Support for Targets

1. EXPLANATION:

Support for Targets is concerned with the interfaces involved in providing means tools and capabilities specific to the operational target computer. This includes loading and executing, debugging, testing, evaluating, and maintaining host-target configurations of varying modes, including (but not limited to) target simulators (as APSE tools), physical environment simulations (APSE tools or connected microprocessors), coupled target emulations or in-circuit emulations, and/or physical instances of the target processors (at varying degrees of coupling) connected to the APSE host.

2. RELEVANCE TO INTEROPERABILITY AND TRANSPORTABILITY:

This category does not address specific KAPSE interfaces. However, there do exist numerous concerns associated with target support which are relevant to other KAPSE Interface Categories, thereby having direct relevance to IT. For each of the Standardization Problems delineated in Section 3, a cross-reference is provided to other appropriate categories. The purpose of this KIW is to provide useful areas of concern to be considered by these related categories.

3. STANDARDIZATION PROBLEMS:

- In order to promote user transportability, uniformity of protocol in the communication between users and tools should be considered, not only for a single target system, but for different target systems as well. [In a single target system the calling convention for one tool should be consistent with the calling conventions for all remaining tools. For example, user access to a linker should be similar to user access to a debugger.] Likewise, user access to a linker for target system "A" should be similar to user access to a linker for target system "B".

- Providing support tool I/O compatibility (i.e., uniform code format among assemblers, compilers, linkers, loaders, data collectors, dynamic analyzers, etc.) is relevant to tool transportability.
- The format (representation form) of the code being transferred from the host machine to the target machine should be considered for uniformity among various targets. For example, the transferred code format could be designed to be target independent, host independent, and/or communications hardware independent, thereby promoting tool transportability.
- User and tool utilization of common conventions across various target systems for load time functions would assist both user and tool transportability. Load time functions to be addressed should include: (1) down-line loading of programs to a target system; (2) exporting of programs to a stand-alone target system followed by direct loading of the programs; and (3) loading of programs to a simulated/emulated target system.
- Target dependent portions of tools should be isolated to allow for commonality of tool design/utilization for different targets. If target dependent portions are clearly delineated, modifications to support tools would be modular, thereby extending the APSE capability for additional targets in a straightforward manner.
- In addition to the run time support functions of scheduling, storage management, I/O handling and exception handling, additional support must be provided to accommodate the REAL TIME application requirements of embedded computers as target systems.
For example:
 - physical memory size constraints
 - time constraints (minimal run time overhead)
 - interface with specialized peripheral equipment
 - access to real time clock
 - cyclic (periodic) execution of applications
 - interrupt servicing
 - hardware monitoring
 - conversion and display of large amounts of data
 - error recovery
 - processing of classified data

Such support may not be restricted solely to the run time system and may require coordinated efforts of several support tools for a particular target system.

- Requirements may exist for the interface of "foreign code" with Ada programs for execution on the target system. Such requirements may be made on the assumption that utilization of some existing code in tandem with new Ada code will help transition software systems into the Ada arena.

- In order to provide facilities for the dynamic analysis/testing and postmortem analysis of software executing on a target system, various issues must be addressed. For a target system directly linked to the host, the communications software handler dependencies (for the communications link) should be clearly and cleanly separated from the functions performed by the dynamic analysis tools. If the hardware communications link is modified, only the software required for the communications handler should be affected.

4. PRIORITY:

5. RELATED CATEGORIES:

Category_List

- a. 1A.
- b. 1C.
- c. 2A
- d. 2B.
- e. 3A.
- f. 3B.
- g. 3C.
- h. 3D.
- i. 4A.
- j. 4C.
- k. 4E.

6. APPROACHES:

- a. ALS:
- b. AIE:
- c. Existing Standards:
- d. Other:

7. RISK:

8. ACTIONS TO TAKE:

The above issues have been examined to determine those categories in which the issues would be more appropriately addressed as concerns when determining their relevance to KAPSE interface implementation and/or APSE tool interface implementation.

DRAFT REQUIREMENTS AND CRITERIA DOCUMENT

Introduction

These requirements and criteria are very tentative. The KIT spent less than one full day reviewing and discussing them, and the KITIA has not yet had a chance for thorough review. Therefore, what appears here is very preliminary, particularly section 6.4 which is so far mostly a collection of ideas which occurred during other discussions and which has not been reviewed by either team. These requirements and criteria are included here not because they represent any KIT and/or KITIA consensus, but because the teams felt it was important to solicit early feedback concerning the direction which these requirements and criteria should take. Most of the issues involving these requirements and criteria are yet to be resolved, so feedback from the reader is strongly encouraged.

WORKING PAPER

Ada Programming Support Environment (APSE)

Requirements for Interoperability and Transportability

and

Design Criteria for Standard Interface Specifications

29-Oct-1982

NOT APPROVED

TABLE OF CONTENTS

- 1.0 INTRODUCTION
 - 1.1 Scope
 - 1.2 Background
- 2.0 IT DEFINITIONS
- 3.0 BACKGROUND
- 4.0 EXISTING SOURCES OF APSE INTERFACE SPECIFICATIONS
 - 4.1 Stoneman
 - 4.2 AIE
 - 4.3 ALS
- 5.0 THE APPROACH TO REQUIREMENT/CRITERIA DEVELOPMENT
- 6.0 REQUIREMENTS AND CRITERIA
 - 6.1 The Standard Interface Specification (S_I_S) and the Outside World
 - 6.1.1 Scope
 - 6.1.2 Subsets
 - 6.1.3 Extensibility and Supersets
 - 6.1.4 Existing Operating System Compatibility
 - 6.1.5 Language and Technology Compatibility
 - 6.1.6 System Dependencies
 - 6.1.7 Efficiency
 - 6.1.8 Security
 - 6.2 The S_I_S and the APSE Tool
 - 6.2.1 General Criteria
 - 6.2.2 Versatility and Flexibility
 - 6.2.3 Consistency
 - 6.3 The Syntax and Semantics of the S_I_S
 - 6.3.1 Criteria for Syntax
 - 6.3.2 Criteria for Semantics
 - 6.3.3 Criteria for Responses, Return Values and Exceptions
 - 6.4 Category-Specific Requirements
 - 6.4.1 User Services
 - 6.4.1A Program Invocation and Control

6.4.1B Initiate/Maintain the User Environment
6.4.1C Device Interactions
6.4.1D The Subset MAPSE

6.4.2 Data Interfaces

6.4.2A Basic I/O Interfaces
6.4.2B Database Management and Control
6.4.2C Inter-Tool Data Interfaces

6.4.3 KAPSE Services

6.4.3A Ada Program Run-Time System (RTS)
6.4.3B Bindings and Their Effect on Tools
6.4.3C Performance Measurement

6.4.4 Miscellaneous and Non-Categories

6.4.4A Recovery Mechanisms
6.4.4B Distributed APSE
6.4.4C Security
6.4.4D Support for Targets
6.4.4E Extensibility

6.5 Interoperability Criteria

6.6 Transportability Considerations

7.0 FUTURE PLANS AND RECOMMENDATIONS

APPENDIX A -- Unresolved Requirements/Criteria Issues

APPENDIX B -- Deleted Requirements

1.0 INTRODUCTION

1.1 Scope

This document provides preliminary requirements and design criteria for the future development of a standard interface specification (S I S) for Kernel Ada Programming Support Environments (KAPSEs). This version of the document is NOT APPROVED and is circulated for review and comment; it has been partially reviewed by KIT and KITIA, but is currently incomplete and has not received final approval. Please address all comments to "NOSC, Code 8322, San Diego, CA 92152"; also, comments via ARPANET to "POberndorf@ECLB" are encouraged.

The S I S is for KAPSE developers who are serious about conforming to government standards derived from the DoD economic objectives of interoperability (I) of APSE data bases and transportability (T) of APSE tools. Section 2 of this document provides numerous definitions of terms and phrases used in this document; these definitions provide a context for understanding the requirements and design criteria in terms of the government's KIT/IT activities. Hence, they should be read by serious reviewers before reading the rest of the document.

The first objective of this document is to be complete in requiring that the S I S contains all interfaces germane to moving data bases and tools (I & T, or simply "IT"). Preliminary analysis indicates that IT considerations impact all potential KAPSE interfaces; therefore the KIT currently assumes that this IT completeness objective means that a conforming KAPSE will provide all the functionality needed by any APSE tool.

If in fact a conforming KAPSE provides additional interfaces beyond the S I S, it must be the case that strict adherence to the S I S by serious tools writers will be sufficient for transportability of tools between conforming KAPSEs.

In addition to facilitating the transportability of individual tools and the development of new tools which interact with existing tools, the scope of the KIT/IT effort includes facilitating the en suite transportability of highly interdependent tool sets. [Note that a tool set may contain one tool.]

The KIT currently operates under the presumption that

conformance to the S_I_S in KAPSE development will not, by itself, be sufficient to guarantee the economic payoffs of IT. Minimally, guidelines for the development of APSE tools, standardizing aspects of tool set construction beyond the issues of KAPSE interfaces, will be necessary for tools writers who are serious about IT. While the development of those guidelines is in the purview of the KIT, such guidelines are beyond the scope of this S_I_S requirements and design criteria document.

Additionally, economic trade-offs of tool transportability versus future APSE adaptation to new technology are barely understood. This document attempts to not bias or limit the S_I_S and APSEs against the evolution toward utilization of future technologies which may have life-cycle IT pay-offs.

This version of the document may present an inconsistent distinction between requirements and design criteria. Recommendations for clarifying the distinction and restructuring the document (especially the current Section 6) are welcome.

1.2 Background

[TBD - brief background on Ada, APSE/Stoneman, IT, & KIT. Section 3 (also titled "Background") will be eliminated in the next revision of this document. The current document numbering structure is retained in this version to facilitate comparison (especially of Section 6) with the previous 26-Sept-1982 version.]]

2.0 IT DEFINITIONS

For the purposes of this document and KIT/IT work in general, the following terms and phrases should be interpreted according to the narrow definitions listed below:

- a. Category: The KIT has partitioned the totality of KAPSE functionality into narrow categories of related services or considerations. Each such category is described by a KAPSE Interface Worksheet (KIW). In this document, the requirements on the functionality (6.4) to be provided by the S_I_S are organized according to the same partitioning scheme as are the KIWs.
- b. Conforming KAPSE: A conforming KAPSE is a KAPSE which

provides and implements all components (or entities) of the S_I_S.

- c. Conventions: Conventions are sets of interfaces, services, and/or practices whose use is recognized as contributing to IT and is to be voluntarily agreed to. Conventions are potential (or "proposed" or "preliminary") IT standards. Conventions are not subject to enforcement; but, if they prove effective and practical, they may be adopted as IT standards.
- d. Criteria: [see Design Criteria]
- e. Design Criteria: In this document, design criteria are desirable properties of the S I S. They are goals to be achieved by those who develop the S I S. [See also the definition of "Requirement."] They are weaker than requirements in that if full conformance to every design criterion can not be achieved (perhaps due to conflicts or contradictions), one or more criterion may be compromised based on a trade-off analysis of IT loss. Design criteria should be identified in this version of the document by usage of the verb "should", and they are almost exclusively the contents of sections 6.1, 6.2, and 6.3. [Additionally, the document contains some entries which are of the nature of "The S I S is not required to ..." or "The S I S is not constrained to ..."; these are technically only clarifications, and are neither criteria nor requirements in that they do not establish goals to be achieved by the S_I_S.]
- f. Guidelines: Guidelines are recommendations that, if followed, will aid in achieving IT; for example, they may convey techniques for satisfying the standards or conventions.
- g. Host: [from IT Plan]
- h. Interface: An interface is a shared boundary, e.g., between two separately developed, interacting pieces of software. The methods for accessing all KAPSE services and objects will be provided by an interface specification. "Interface" and "interface specification" are used interchangeably in this document; a KAPSE interface specification is expected to be provided by an Ada package specification (providing "syntax") and an accompanying description ("semantics," which may be embedded in comments in the Ada package specification) of each entity in the Ada package. The

implementation of the functions and objects provided by the KAPSE interface is beyond the scope of this document (other than feasibility-type requirements).

- i. Interface Specification: [see Interface]
- j. Interoperability: [from IT Plan]
- k. Requirement: An IT requirement addresses a condition or capability which is relevant to IT and which shall be addressed by one or more conventions and/or standards. Thus IT requirements document those functional capability aspects which are determined to affect IT. [See also the definition of "Design Criteria."] By implication, any condition or capability which is NOT addressed by an IT requirement or criterion has been determined NOT to affect IT. [However, see comment in Scope (1.1) stating current KIT assumption that IT impacts permeate all KAPSE interfaces.] In this version of the document, requirements are goals which must be met by the S_I_S, and they are largely confined to section 6.4 ("Category-Specific Requirements"); they should be identified by usage of the verb "shall".
- l. Semantics: The "semantics" of the S_I_S refers to the meaning or functionality of each entity in an interface specification.
- m. S_I_S: [see Standard Interface Specification]
- n. Specification: This document concerns itself with KAPSE interface specifications. KAPSE interface specifications are expected to provide tools writers all information needed to use all facilities provided by a KAPSE. It is anticipated that KAPSE interface specifications will be provided by Ada package specifications (providing the "syntax") and accompanying descriptions of the meaning ("semantics," which may be embedded in Ada comments in the Ada package specification) of each entity in the packages.
- o. Standard Interface Specification (S_I_S): The Standard Interface Specification (S_I_S) for KAPSEs is the interface specification [see definition of "Interface Specification"] which is developed in accordance with these requirements and design criteria, and which is approved by KIT and AJPO as the government standard for KAPSE interface specifications. Development of the S_I_S is a future activity of the KIT; proposed candidates for the S_I_S will have the status of "Conventions" [see definition above] until a decision is

made to select one candidate for incorporation into a government Standard.

- p. Standards: A possible future Standards document may consist of interfaces, services, and/or practices whose use has been established for achieving APSE IT [see also definition of Interface]. Tools and data bases which are to be ported between conforming APSEs can depend on these interfaces/services/practices only. It can be verified whether or not a given APSE meets the IT standards. Interfaces, services, and practices in the Standard are formally documented and approved for use by the AJPO in the construction of DoD-compatible tools, data bases, and KAPSEs.
- q. Syntax: The "syntax" of the S_I_S narrowly refers to the allowable identifiers or values for subprogram names, parameters, types, data objects (global data), operators, exceptions, etc. which are provided in the Ada packages which are the interface specifications and which are usable by APSE tools. Of course, for APSE tools, all references to components of the S_I_S are implemented in accordance with the syntax of the Ada programming language.
- r. Tool: A tool is any Ada program or command language script which runs in the context of a KAPSE and is dependent upon it for normal support services. A routine resident in a library which is linked into a program or script which runs in an APSE is also a tool, in as much as it depends on KAPSE services instead of the host operating system.
- a. Transportability: [from IT Plan]

3.0 BACKGROUND

[Section 3 will be eliminated in the next revision of this document; "Background" is now the new Section 1.2. The current document numbering structure is retained in this version to facilitate comparison (especially of Section 6) with the previous 26-Sept-1982 version.]

4.0 EXISTING SOURCES OF APSE INTERFACE SPECIFICATIONS

Listed below are documents which provide information about

existing requirements, criteria, and specifications for (K)APSE interfaces. These references are regarded as inputs for consideration, but not necessarily proposed candidates, in the development of these requirements and criteria and in the subsequent development of proposed standard interface specifications which fulfill these requirements and criteria.

4.1 Stoneman

REQUIREMENTS FOR Ada PROGRAMMING SUPPORT ENVIRONMENTS
("STONEMAN"), Department of Defense, February 1980.

4.2 AIE

TBD

4.3 ALS

TBD

5.0 THE APPROACH TO REQUIREMENT/CRITERIA DEVELOPMENT

[Section 5 will be eliminated in the next revision of this document; its previous contents have been subsumed by the new Section 1.1 titled "Scope". The current document numbering structure is retained in this version to facilitate comparison (especially of Section 6) with the previous 26-Sept-1982 version.]

6.0 REQUIREMENTS AND CRITERIA

6.1. The Standard Interface Specification (S_I_S) and the Outside World

6.1.1 Scope of the S_I_S

- o The S_I_S should be the same everywhere and for everyone.
- o The S_I_S should be implementable on a single-processor system, a multi-processor system, and network systems.
- o The S_I_S is not required to provide all general operating system or language system capabilities.
- o The S_I_S is not required to provide execution capability for tools and programs not source-coded in Ada.

6.1.2 Subsets

- o A conforming KAPSE shall implement all of the S_I_S (i.e., no subsets).
- o The S_I_S should be designed such that a user (e.g., tool writer) can employ a subset of the standard interfaces without being aware of the full S_I_S.

6.1.3 Extensibility and Supersets

- o The S_I_S should be designed to impose no constraints (e.g., a maximum number of interface entry points) that limit future extension of the S_I_S.
- o The S_I_S should be designed to prohibit extending the capability of any specified S_I_S interface (or entry point).

6.1.4 Existing Operating Systems

- o The S_I_S should allow flexibility in the implementations of conforming KAPSEs.
- o The S_I_S should allow implementation of a conforming KAPSE that coexists with (if not employs) any one of a variety of file systems, including hierarchical file systems and non-hierarchical file systems.

- o The S_I_S should allow implementation of a conforming KAPSE that runs under any one of a variety of existing operating systems. This should not be interpreted as requiring S_I_S compatibility with existing operating system interfaces.
- o The S_I_S should allow implementation of a conforming KAPSE that coexists with (meaning "runs along side," not "runs under") and operates with the characteristics of any one of a variety of existing operating systems. This should not be interpreted as requiring S_I_S compatibility with existing operating system interfaces.
- o Design of the S_I_S is not constrained to be compatible with the functionality and interfaces of existing systems.
- o The S_I_S should be based on the functionality found in many existing hosts and the functionality found in the Ada language reference manual.
- o Design of the S_I_S is not constrained by a goal that implementations need no modification to any existing operating system.
- o The S_I_S should control direct access to underlying system services (i.e., no "by-passing" the KAPSE is allowed; it is regarded as non-portable circumvention if a KAPSE allows user access to underlying system services). In particular, configuration management, security, and integrity of data bases should be protected.

6.1.5 Language and Technology Compatibility

- o The nomenclature used in the S_I_S should be consistent with the Ada programming language.
- o The S_I_S should avoid requiring interface mechanisms which have not been tested in existing KAPSEs, operating systems, kernels, or command processors.

6.1.6 System Dependencies

- o The S_I_S should promote the isolation of operating system, machine, and device-type dependencies; and

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM: PUBLIC REPORT VOLUME II(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P A OBERNDORF 28 OCT 82

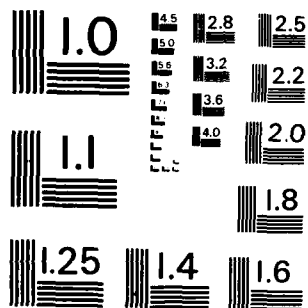
316

NO5C/TD-582

F/G 8/2

NL

A 10x10 grid of squares, with the top-left square missing.



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

thereby, the S_I_S should promote simple rules for guaranteeing tooT transportability.

- o The S_I_S should accommodate device characteristics in recognized standards, e.g., ANSI, ISO, IEEE, and DoD standards.

6.1.7 Efficiency

- o The S_I_S should be implementable on or with or without any one of a wide variety of operating systems.
- o The S_I_S should be implementable by conforming KAPSEs that do not significantly constrain resources available to user programs (APSE and non-APSE).
- o The S_I_S should be structured to allow its implementation, as well as the functions it controls, to be reasonably distributed over either locally or remotely connected networks.

6.1.8 Security

- o The S_I_S should allow implementations of conforming KAPSEs that coexist with (without compromising) and operate within a variety of system security mechanisms, including the lack of such a mechanism.

6.2. The S_I_S and the APSE Tool

6.2.1 General Criteria

- o The S_I_S should allow expression of any reasonable action in a straightforward and reasonable manner.
- o The S_I_S should allow that simple actions be invoked by simple commands or calls.
- o The S_I_S should define standard defaults whenever an appropriate default exists. This includes defaults for, as a minimum, calling parameters, standard global packages, assumed USE constructs, and other compilation environment standards.

6.2.2 Versatility and Flexibility

- o If a choice has to be made, the basic syntax and semantics of the S_I_S should be targeted toward basic

services frequently needed by common tools; then features can be added which support unique special-purpose tool needs.

6.2.3 Consistency

- o S_I_S interface components applicable to more than one interface will be designed to be optimal and consistent across the set of interface to which they apply, rather than be optimized on a interface-by-interface basis.
- o The S_I_S should avoid facilities used solely by the system operator, system administrator, or privileged user when there are equivalent facilities defined for non-privileged users.
- o The aspects of the S_I_S that are applicable in a network environment and the aspects that are applicable in a local environment should be identical when they overlap.

6.3. The Syntax and Semantics of the S_I_S

6.3.1 Criteria of Syntax

- o The syntax for each S_I_S call shall be Ada.
- o Aspects of syntax (e.g., limits on name lengths, abbreviation styles, other naming conventions, relative ordering of input and output parameters, etc.) should be designed in uniform and integrated manner for the whole S_I_S, not as independent issues. This goal should not be pursued so vigorously as to force an excess of artificiality on the S_I_S syntax.
- o The S_I_S should have no unnecessarily restrictive rules, constraints, or anomalies.
- o There will be no unreasonable restrictions on length of names.
- o The S_I_S will contain no specially coined words (literals or identifiers), and will use no words in an unconventional sense.
- o Standard defined names should be natural language words or industry accepted terms whenever possible.

- o Standard defined names should not be trademarks or derivatives of trademarks.
- o No two standard defined names in the S_I_S shall be only a transposition away from identity.

6.3.2 Criteria for Semantics

- o A complete, precise definition (meaning the precise meaning of every interface) should be given for the S_I_S, suitable for use as a KAPSE implementation guide, as part of a tool implementation guide, and for KAPSE validation criteria.
- o It must be possible in the definition of the S_I_S to specify, in their entirety, the results of using each interface in the S_I_S.
- o The specification of semantics should be both precise and understandable.
- o The semantic specification of each S_I_S call should include precise statement of assumptions and effects on global data and packages.
- o The S_I_S should preclude unnamed exception propagation.
- o The semantic specification of each S_I_S call should include precise definition of Ada exceptions (if any) to be handled and raised by the implementation.
- o Each standard defined S_I_S call should have only one function.
- o Use of a S_I_S call should require provision of only relevant and necessary information and parameters. For example, input and output are relevant and necessary for a file copy but not for a suspension.
- o Opportunity to produce side effects in S_I_S calls should be minimized.
- o Natural language synonyms or near synonyms will not be given different meanings. Only one word, and not its possible synonyms, will be used when the same generic meaning is intended.

6.3.3 Criteria for Responses, Return Values and Exceptions

- o All standard defined responses to S_I_S calls should be implementation independent.
- o Each S_I_S call should have a unique, non-null response (return value or exception) for each case of unsuccessful completion.

6.4. Category-Specific Requirements

WARNING/DISCLAIMER: The requirements listed in this section 6.4 are totally unreviewed by the KIT. Each is the recommendation of one or a few members of the KIT or KITIA, and will be placed on the agenda for future deliberations by the KIT. It is understood that many subsections here in 6.4 are incomplete, and it should be recognized that this section is NOT APPROVED by KIT.

6.4.1 User Services

6.4.1A Program Invocation and Control

6.4.1A.1 Program Invocation

- o Facilities shall be provided for a program to invoke another program. These facilities shall not prevent a program from invoking itself as a process.
- o Program invocation and control facilities shall not be specially privileged or constrained.
- o Facilities shall be provided for a user to direct or redirect program input and output.
- o Facilities shall be provided for simultaneous execution of several programs for one user.
- o Various wait options shall be available after a subprocess is invoked:
 - Calling task of caller waits.
 - Caller waits.
 - No one waits.

- o Facilities shall be provided to allocate, deallocate, or share resources among processes.

6.4.1A.2 Process Control

- o Facilities for interprocess communication shall be available.
- o Facilities shall be available to interrupt a running program. Interrupts that can be processed shall include both hardware and software interrupts. Facilities shall be available for processes to either respond to an interrupt or to lock out interrupts that they do not desire to process. In particular, a user shall be able to suspend, resume, or terminate execution.
- o Facilities shall be provided for running a program with a monitored context for debugging.

6.4.1A.3 Process Termination

- o Facilities shall be provided for the return of values or status by a process. It shall also be possible to test these status values and make decisions based on their values.

6.4.1A.4 Process Monitoring

- o Facilities shall be provided for a user to determine the hierarchy of program invocation and control on a status display. It shall also be possible for a running process to determine the "name" of another running process.
- o Facilities for the collection and display of system- and program-generated error messages shall be provided.
- o Facilities shall be provided to determine the status of a process.
- o Facilities shall be provided for query of, or the setting of, system environment and process parameters (e.g., date, time).

6.4.1B Initiate/Maintain the User Environment

- o The S_I_S shall include some aspects of controlling a user's logical access to an APSE, such as "sign on" and "sign off."
- o The logon processor shall start the standard command language interpreter (or command language processor).
- o Facilities shall be provided to maintain the user's terminal environment after a line drop.

6.4.1C Device Interactions

6.4.1C.1 Terminal Type Devices

- o A virtual device interface shall be provided.
- o All 256 8-bit character codes shall be able to be transmitted between devices and processes. The standard shall describe the disposition of characters that cannot be transmitted in certain implementations.
- o Facilities shall be provided to deal with accidental disconnection of devices. This includes the interruption and resumption of data flow between processes.
- o A conforming KAPSE should be able to coexist with and operate with a variety of system mechanisms for immediate control operations (e.g., character or word or line rub-out, or command completion) on interactive entry, including the lack of such a mechanism.

6.4.1D The Subset MAPSE

6.4.1D.1 Command Language

- o The method for referencing parameters shall not restrict the length of parameters or the number of parameters that can be passed to a program.
- o No special KAPSE facilities shall be required by the command language interpreter.
- o Command language restrictions on data flow between tools must be minimized.

- o The command language shall not limit the number of available variables.
- o Facilities for creation and use of stored command language scripts must be provided.
- o Command language scripts shall be ASCII text files.

6.4.1D.2 Editor

- o The minimum set of commands in a useable editor must be identified. Candidates for this set include: add, change, delete, and find.
- o The editor shall reference both character strings and line numbers.

6.4.1D.3 Linker

- o The linker shall have a standard command language.

6.4.1D.4 File System Issues

- o The S I S shall provide the minimum file system capabilities required to build the ALS or AIE type data base system.
- o Access control must be provided.

6.4.2 Data Interfaces

6.4.2A Basic I/O Interfaces

6.4.2B Database Management and Control

- o The DBMS and the memory manager must be inside the KAPSE (or SMAPSE?).

6.4.2C Inter-Tool Data Interfaces

- o All KAPSE data interface calls and inter-tool data interface calls shall be made to abstract data types.
- o Both the KAPSE primitive operations (Stoneman

5.E.2.i) and the operations on the abstract data types (Stoneman 5.E.2.ii) must be standardized.

- o The KAPSE interfaces must be modifiable to allow for the easy addition or modification of the abstract data types.

6.4.3 KAPSE Services

6.4.3A Ada Program Run-Time System (RTS)

6.4.3B Bindings and Their Effect on Tools

6.4.3C Performance Measurement

6.4.4 Miscellaneous and Non-Categories

6.4.4A Recovery Mechanisms

- o The KAPSE shall be immune from abortions/crashes due to tool errors (e.g., checksum error, integrity checks, invalid input/output, etc.).
- o The KAPSE shall have a recovery procedure which allows system programmers to terminate hung terminals, programs, and tools.
- o The S_I_S shall support "predefined escape mechanisms" (planned errors) for abnormal exits (e.g., Ada exceptions) from tools. But, no tool or program shall be allowed to circumvent or bypass the KAPSE.
- o The S_I_S shall provide standardized ways to report errors to users and/or log them to some audit/recovery information file.
- o Standard recovery shall be provided from deliberate/accidental attempts to destroy portions of the database.
- o The S_I_S shall provide a standard way to determine at what point the system must be recovered from major system failures.

6.4.4B Distributed APSE

6.4.4C Security

6.4.4D Support for Targets

6.4.4E Extensibility

[See 6.1.3.]

6.5. Interoperability Criteria

- o The S I S should facilitate inter-APSE exchange of types of data structures such that the size of projects which can be feasibly moved is maximized.
- o The S I S should facilitate movement of types of data structures such that the disruption of projects moving to new hosts is minimized.
- o The S_I_S should define standard file transfer protocols.

6.6. Transportability Considerations

- o Each interface in the S I S is selected because it significantly increases the number of useful tools which are easily portable.
- o Each interface in the S I S is selected because it significantly decreases the size or complexity of tool fragments which can be moved independently.

WORKING PAPER
IT Requirements/S_I_S Design Criteria

NOT APPROVED
29-Oct-1982

7.0 FUTURE PLANS AND RECOMMENDATIONS

TBD

APPENDIX A -- Unresolved Requirements/Criteria Issues

[This appendix serves as a visible repository for open issues (concerning this document) which have been identified but not resolved by the KIT.]

- o The precise distinction between "requirements" and "criteria" is not resolved. This version of the document largely presumes that functional requirements (6.4) are requirements and everything else (6.1, 6.2, 6.3) is criteria.
- o The consistent use of "shall" versus "should" and their relationship to requirements versus criteria is unresolved. However, this version of the document presumes that "shall" sentences give requirements and "should" sentences give criteria.
- o Whether any minimum access from tools, directly or even under KAPSE control, to underlying system services is acceptable is unresolved (see related criterion in 6.1.4). An instance of concern is that the ALS specifies that the command language processor (via the KAPSE) can invoke certain VAX VMS tools, and this may lead to tool operation which does not preserve the integrity of the configuration management system or a data base.
- o It is unresolved whether allowable S I S "extensibility" should be constrained to only specifying new interfaces of functions which can be simulated by combinations of existing S I S interfaces, or whether a less restrictive paradigm is economically justified (outweighing possible tool transportability losses) to take advantage of unique special-device interfaces and future technology advances. KIT has deliberated this significantly and leans toward the latter, freer notion currently; this version of the document reflects that.

APPENDIX B -- Deleted Requirements

[This appendix serves as a container for requirements and criteria which have been deleted or substantially changed from previous versions of this document. An attempt is made [IN SQUARE BRACKETS] to identify the original section under which a now-deleted entry was located and to provide some rationale for the deletion.]

- o A conforming KAPSE implements a superset of the S_I_S only by using extensibility features in the S_I_S which are reproducible in any other conforming KAPSE, or by providing interfaces that do not impact Interoperability or Transportability. [FROM 6.1.3: THIS IS DEEMED TOO STRONG AND PROBABLY INHIBITING USAGE OF SOME FUTURE TECHNOLOGY ADVANCES.]
- o Those aspects of the S_I_S which support a network environment should be biased toward neither session-based nor transaction-based inter-process communication. [FROM 6.1.5: PRESUMED TO BE AN OVERLY DETAILED, NARROW STATEMENT, WHERE THE LAST BULLET OF 6.1.7 IS MORE APPROPRIATE.]
- o It should be easy to implement the S_I_S on a wide variety of operating systems. [FROM 6.1.7: REPLACED BY 1ST BULLET OF 6.1.7, REPLACING "easy" WITH "possible" AND EXPANDING RELATIONSHIPS TO OSs.]
- o No cute tricks are allowed; the S_I_S must say what it means and mean what it says. [FROM 6.2.1: THIS WAS HARD TO INTERPRET CONCRETELY, AND IS COVERED BY OTHER ASPECTS OF 6.2 AND 6.3.]
- o The syntax of the S_I_S must be unambiguous. [FROM 6.3.1: UNNECESSARY -- ARTISES FROM Ada's BEING THE SYNTAX.]
- o S_I_S literals and subprogram names should be as short as practical (e.g., minimize noise words and employ recognized abbreviations) without forfeiting readability and understandability. [FROM 6.3.1: THIS GOAL WAS DEEMED UNDESIRABLE; IT MIGHT BE CONSTRUED TO DISCOURAGE NATURAL NAMES.]
- o The set of graphics used in the S_I_S, excluding system

dependent character strings, should come from those characters in the international use positions of ANSI X3.4-1977, ASCII. [FROM 6.3.1: THIS IS COVERED IN THE Ada LANGUAGE REFERENCE MANUAL, WHICH PROVIDES THE SYNTAX OF THE S_I_S.]

- o All standard defined text must be insensitive to case. [FROM 6.3.1: THIS TOPIC IS SPECIFIED BY THE Ada LANGUAGE REFERENCE MANUAL.]
- o The design of the S_I_S should include consideration of potential user and tool calling errors, especially in those cases where permanent (irreversible) actions are concerned. [FROM 6.3.1: HANDLING USER ERRORS IS THE TOOL-WRITER'S RESPONSIBILITY, NOT PRIMARILY THE KAPSE'S. TOOL CALLING ERRORS SHOULD BE VIRTUALLY IMPOSSIBLE USING Ada, EVEN MORE SO FOR TOOLS CHECKED OUT SUFFICIENTLY TO WARRANT INSTALLATION INTO AN APSE.]
- o In calls and in responses, numbers in standard-defined uses (if any) should be expressed as decimal numbers, not as binary, octal, or hexadecimal numbers. [FROM 6.3.1: Ada's SYNTAX ALREADY MAKES USAGE OF ANYTHING OTHER THAN DECIMAL NUMBERS VERY CUMBERSOME.]
- o The function of each S_I_S call should be specified in a manner independent of any specific syntax used to invoke the command. [FROM 6.3.2: UNCLEAR AND OF DUBIOUS MERIT.]
- o The semantic specification of each S_I_S call should include precise definitions of parameter defaults (if any). [FROM 6.3.2: THIS IS ALREADY REQUIRED BY Ada.]
- o Each S_I_S call should have available a non-null response to its successful completion. [FROM 6.3.3: DEEMED UNNECESSARY, GIVEN Ada AND THE REQUIREMENT (STILL IN 6.3.3) FOR A NON-NULL RESPONSES FOR ALL UNSUCCESSFUL CALLS.]
- o The S_I_S should be designed such that users (e.g., tool writers and tools) can be informed of the presence of errors and anomalies as early as is practical. [FROM 6.3.3: THIS IS HARDLY THE RESPONSIBILITY OF THE S_I_S, BUT RATHER Ada COMPILERS, TESTING METHODS, OTHER TOOLS, ETC.]

Notes on Stoneman Refinement

Donn Milton
COMPUTER SCIENCES CORPORATION

1.C "Tool" must be defined. There are two possibilities:

- a) A tool is complete Ada program used to support program development.
- b) A tool is an Ada program, subprogram, package, command language script, editor script, etc. used to support program development.

The first definition has some problems for systems that support dynamic linking. The second definition is a user view that is so general that it becomes useless for describing APSE structure.

1.D There is no useful distinction that can be made between user programs and software tools. In light of the controversy on tool definition and its relationship to portability, distinction should be made, throughout Stoneman, between:

Ada programs
Scripts
Data formats
(others?)

since each of these places different requirements on portability.

1.E "Database" is a loaded word that is perhaps improperly used in the description of level 1 (KAPSE). There is a serious question as to whether the KAPSE database should be anything more than a flat file system.

The notion of "portability interface" requires careful definition. It appears that this interface cannot be confined to the KAPSE as suggested here.

The notion of MAPSE, as expanded later in the document, includes the toolset as well as libraries.

1.J The generality of Stoneman may be its undoing. We must define the role that a refined Stoneman is to play with respect to standardization and portability.

2.B.2 It should be explicit that a host is also a target.

2.B.4 "Object" should be defined. There are too many different ideas as to the granularity of an object.

2.B.10 "Low level portability interface" is equated to the KAPSE interface. This is probably correct, but there are clearly other portability interfaces that must be examined.

2.B.12 It is not generally agreed that declarations of intermediate language abstract data types should be provided by the KAPSE.

4.A.4 Both versions and revisions should be required.

4.A.5 "Configuration" and "partition" need more definition. There is no general agreement as to what these terms mean.

4.A.6 This requirement has severe implications on the performance of the database. "Derived" and "basic" objects should be distinguished, and reconstructability might be considered a user (or project option).

4.A.7 See 4.A.5.

4.B.4 The relationship between object preservation and object reconstructability is very unclear.

4.C.8 What about breaks or interrupts (or other control keys)?

4.D.4 It is apparent that the notion of "virtual interface" is somewhat muddy. A clear distinction should be made between the interface provided by the KAPSE and that provided by the command language interpreter (look at 4.C.3, 4.C.4, 4.C.6, and 4.D.4 all together).

The third paragraph of 4.D.4 should be deleted.

The possibility of APSEs that support dynamic linking clouds all command language issues.

4.E.5 See notes on 4.D.4.

5.A This entire section is proposing a far more elaborate database than it is generally agreed that a KAPSE must support. Proposals have been made that the KAPSE should provide only a minimal file system, upon which MAPSE tools can build and support a database in the sense of 5.A.

5.A.5 The relationship between history and configuration control needs better definition. "Configuration control" itself needs to be defined with respect to the KAPSE.

5.A.6 The notion of "category" is closely related to that of "type". There is disagreement as to whether objects must or may be typed.

5.A.7 Requirements for access rights need considerable elaboration.

5.C.1 The sub-area of run-time support for Ada tasking has been assumed by many to be included in the KAPSE services. It is likely, however, that it will be difficult to define a machine-independent virtual interface for this support. Moreover, tasking services are not callable from Ada programs directly, as are other KAPSE services (there are some exceptions, e.g., a service to provide status information about extant tasks). This is an argument for relegating tasking support to level 0.

5.C.4 Most designs severely constrain the inter-program passing of parameters. There seems to be no good way of generalizing the Ada subprogram call mechanism for program invocation.

5.D.2 (a) Does "function" mean "function within an Ada program"? Also there may be more than one "current program".

5.E.2-9 There is considerable resistance to including these interface definitions as part of the KAPSE (with the possible exception of 5.E.6, the abstract definition of an executing program). It may be more generally acceptable to include these in a MAPSE library.

5.F.1 This section should be deleted.

5.F.2 As discussed in 5.A.1 and related to 5.E, it is not clear that library format is a KAPSE rather than a MAPSE issue.

6.A.1 The notion of standard interfaces to a set of standard MAPSE tools has not really been addressed. Nevertheless, this is a critical issue for many kinds of portability, and it is a non-KAPSE issue.

6.A.2 The requirement for a prettyprinter does not follow from the definition of a MAPSE in 2.B.14.

6.A.5 Off-line and down-time loader requirements also do not follow from 2.B.14. Since a host must be a target, a host loader is required. But by analogy with most existing systems, a host loader should be part of the KAPSE, not the MAPSE.

6.A.6-7 Same comment as for 6.A.2.

6.A.9 This is a strange area. with low-level I/O it is probably possible for these routines to exist in the MAPSE, but is that the best approach? Moreover, this requirement can be considered to conflict with the KAPSE definitions of I.E. However, there may be no placement of these routines that does not conflict with I.E.

6.A.12 Requirements for a configuration manager need considerable elaboration. Moreover, the 6.A.2 comment also applies here.

6.C MAPSE libraries are another essential area for standardization. However, is it appropriate to place any target-specific support in the MAPSE?

6.D.1 See comment above.

7.A.5 How does this relate to 6.A.12?

Interface Analysis of the Ada Integrated Environment and the Ada Language System

Jack Foidl
TRW

ABSTRACT

This paper presents a status report of the analysis of the functional interfaces of the Ada Language System (ALS) and the Ada Integrated Environment (AIE) in accordance with documentation available at the time of this writing. This analysis is the initial step in the definition of the Kernel Ada Programming Support Environment (KAPSE) Standard Interface Specification (S_I_S) by the KAPSE Interface Team (KIT) under direction of the Ada Joint Program Office (AJPO).

BACKGROUND

The goals of tool transportability as described in STONEMAN [1] are a serious concern for the Department of Defense (DoD) in the development of the Ada Programming Support Environment (APSE) that will be utilized in DoD-sponsored software development programs. In January of 1982 a Memorandum of Agreement [2] was signed by Undesecretaries of the Air Force, Army, and Navy, in which standardization of interfaces at the Kernel Ada Programming Support Environment (KAPSE) level was the joint agreement. The KAPSE may be viewed as the interface medium for tool-to-tool or tool-to-operating-system data and services interfaces (Figure 1). This standardization is to be

APSE

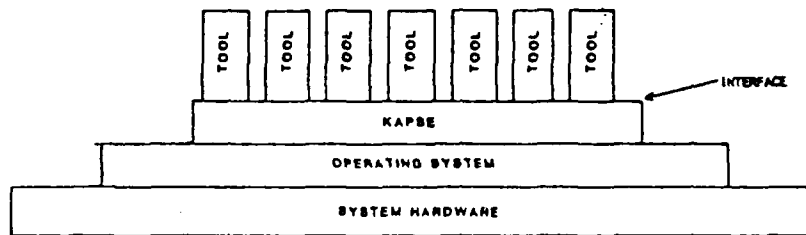


FIGURE 1
THE KAPSE IN THE ADA PROGRAMMING SUPPORT ENVIRONMENT (APSE)

documented in a Standard_Interface_Specification (S_I_S) which will be the basis for KAPSE interfaces for all tools developed for DoD-sponsored or-supplied APSEs. The standardization at the KAPSE level is intended to allow flexibility for tool implementation by various tool developers while supporting transportability for tools across standardized KAPSEs. This is intended to allow transportability of tools from APSE to APSE independent of the systems operating software or hardware processor. Although the AIE and ALS systems appear different, the definition of standard KAPSE interfaces would provide a standard socket, so to speak, into which tools could "plug" (Figure 2). The responsibility for definition of the standard KAPSE interface was assigned by the AJPO to the KIT.

The initial point of departure for the definition of this interface standard was an analysis of the KAPSE interfaces in the current AIE and ALS development efforts. The analysis was intended to identify the commonality, if any, between these implementations and to provide the initial identification of the Standard Interface Set (SIS) that would be included in the specification. The analysis was also intended to provide a model for evaluation that could be extended, as required, to a complete KAPSE interface specification. The plan was to define an initial interface model and identify the present interfaces of the AIE and ALS systems. This would be further analyzed for the inclusion of additional interfaces not present in these systems, followed by a review for internal consistency and completeness. The result would be the distribution of a draft Standard_Interface_Specification (S_I_S) for review and comment.

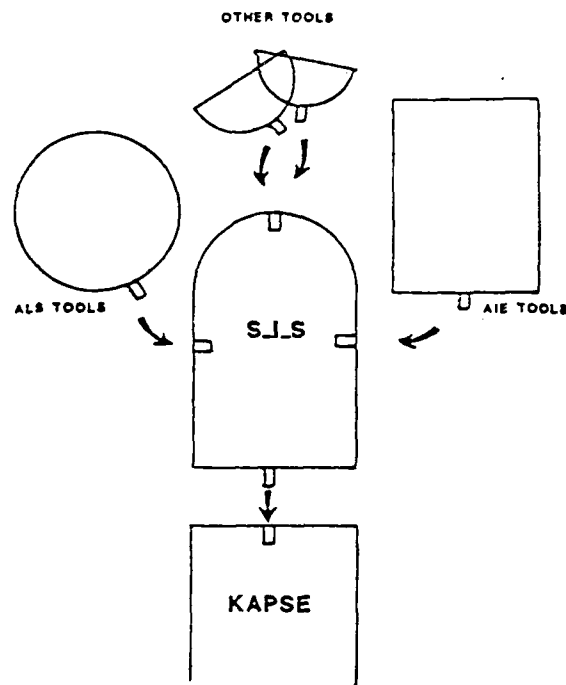


FIGURE 2. THE FUNCTIONALITY OF THE STANDARD INTERFACE SPECIFICATION

The analysis is not intended to redesign either the AIE or ALS; it provides an identification of the commonality that currently exists in these different implementations. This work could not have been performed without the complete support of the U.S. Air Force and U.S. Army and the excellent technical information provided by their implementation contractors, Intermetrics and SofTech.

INTERFACE ANALYSIS METHODOLOGY

The interface comparison effort presented herein is a preliminary analysis of the initial KAPSE implementations represented by the AIE and ALS

systems. The analysis was performed in an effort to determine the initial set of interfaces that are candidates for standardization in the future Standard Interface Set (SIS). A logical starting point for such an effort is the determination of the present interfaces which are common or similar in the current AIE and ALS systems. This provides a point of departure for future interface standardization. This analysis does not apply metric evaluation to the present AIE or ALS systems nor is it intended to be construed as an evaluation of the interface implementations. The mappings of the interfaces between the AIE and ALS and to the SIS model provide a point of standard comparison between these systems and do not suggest a "goodness of fit".

The analysis was intended to be performed by constructing a model KAPSE consistent with the STONEMAN document. Comparative models were to be constructed for the AIE and ALS systems. An attempt would then be made to formulate a SIS model that would be representative of the interfaces of the other three models as depicted in Figure 3. The model definition was intended to be a top-down functional decomposition and lateral partitioning to attempt to map the corresponding AIE and ALS functions and interfaces into the appropriate model location and to provide a logical reference point for comparison. The model skeleton and the initial decompositions of the AIE and ALS are shown in Figure 4. The hexagonal structures represent the interface points of the various models.

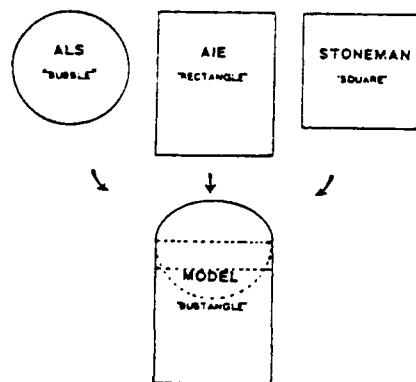
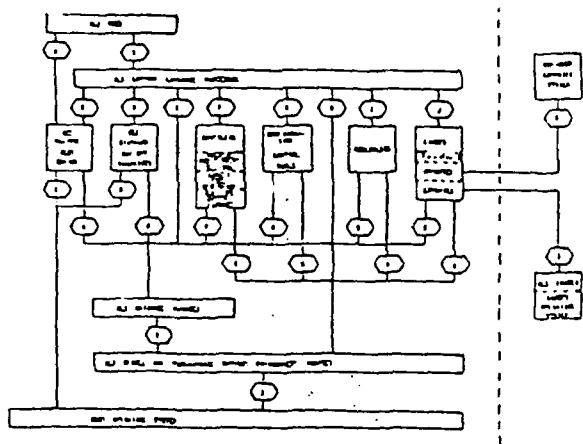
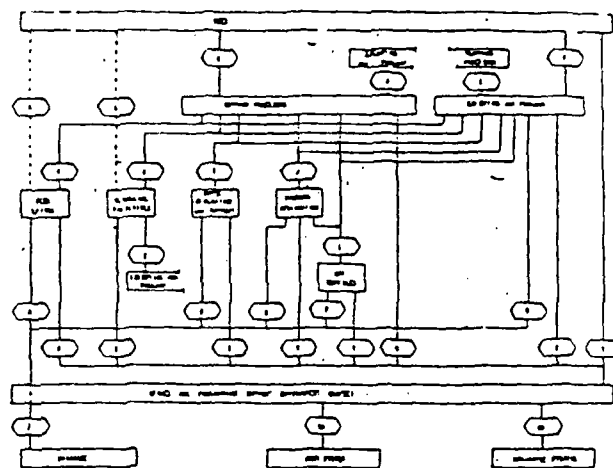


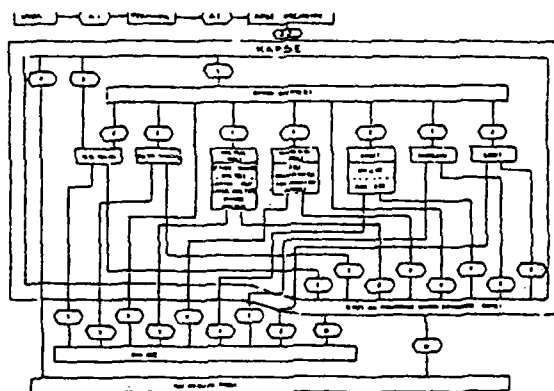
FIGURE 3. TOP-DOWN SIS MODEL FORMULATION



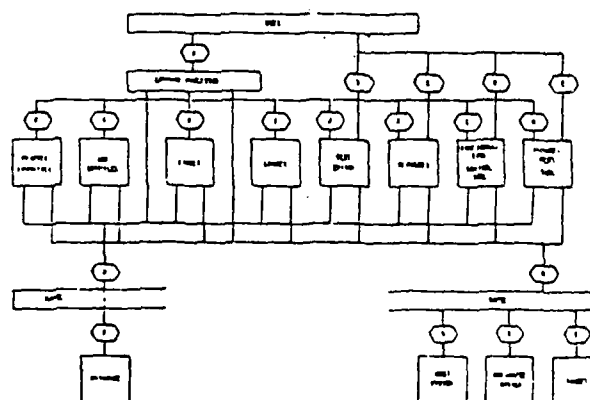
ALS



AIE



STONEMAN



MODEL

FIGURE 4. INITIAL MODEL FORMULATION

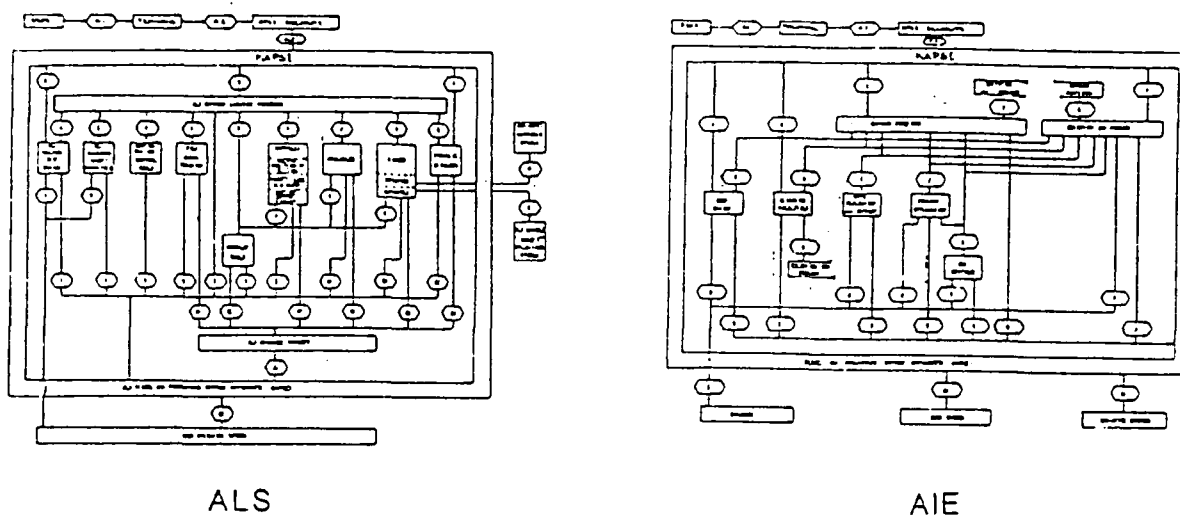


FIGURE 5. EXPANDED MODEL FORMULATION

The models were then refined as later documentation was available which resulted in the expansion of the models as depicted in Figure 5. The perspective of the KAPSE as the pipeline through which tools/toolsets interface resulted in an increased number of interface points. The differences in the AIE and ALS implementations regarding the database also increased the difficulty of defining a "standard" model. Since the models were constructed from the available documentation, the functional decomposition was based at times on the definition of the Computer Program Configuration Item description defined in the respective development contracts rather than the actual functionality of the system and the KAPSE interfaces. This was confirmed by a meeting with the AIE and ALS developers who presented their perspective of their respective systems as in Figure 6. Since the models were not representing the actual implementations the basic methodology was reviewed and evaluated as inappropriate. The point of the exercise was to determine the commonality of the AIE and ALS systems to assist in the definition of a Standard Interface Set. This would be attainable from a bottom-up approach [3]

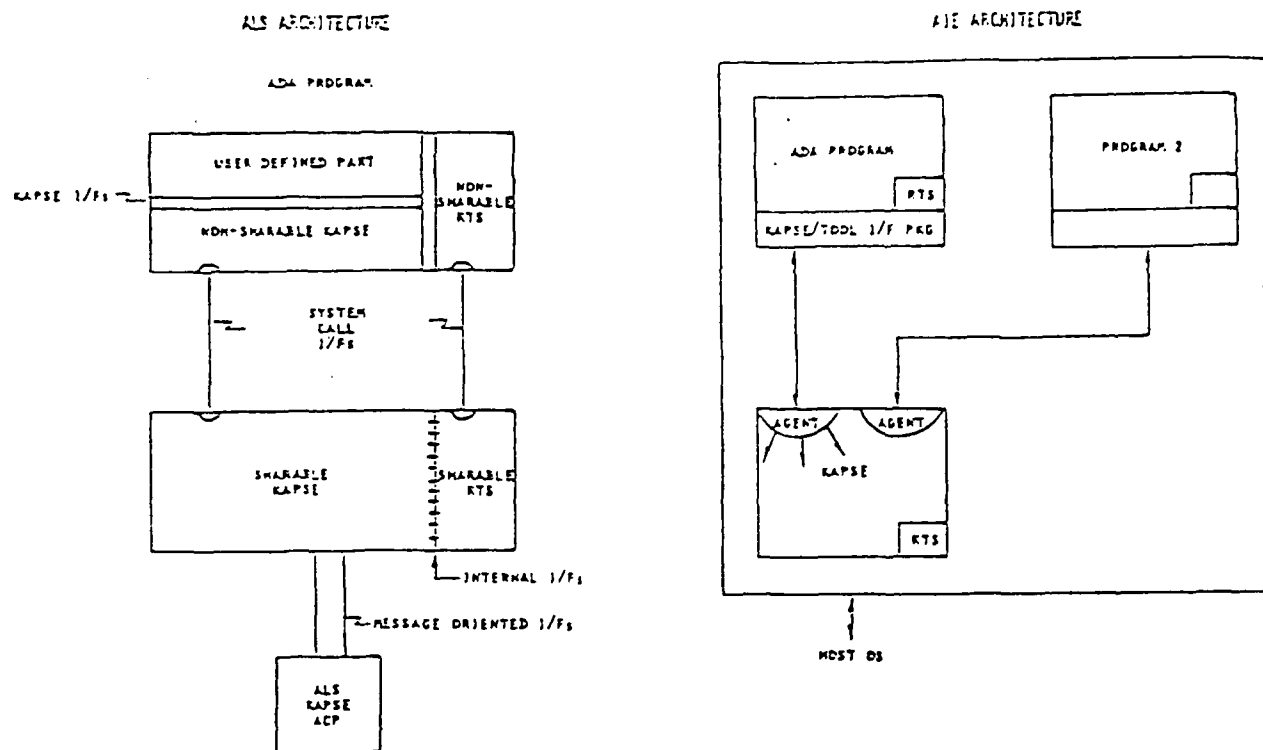


FIGURE 6. AIE/ALS DEVELOPERS PERSPECTIVE.

since the models of the systems did have features in common as shown in Figure 7. The interfaces for database management, run-time support and I/O, call, and interface support were the object of the analysis and not the graphical representation of theoretical functionalities.

The bottom-up approach was then designed to examine the existing AIE and ALS interface packages which were composed of specific interface procedures. At this point, interface commonalities could be identified. The different implementations would probably have varying degrees of commonality, so a basis for comparison was established to reflect the degree of commonality present. The basis for comparison for the interface procedures was defined as:

Ada Package	Defined in standard Ada
Common	Highly compatible
Minor	Some minor differences
Analogous	Similar capabilities but different implementations
AIE/ALS Specific	Specific to either the AIE or ALS systems

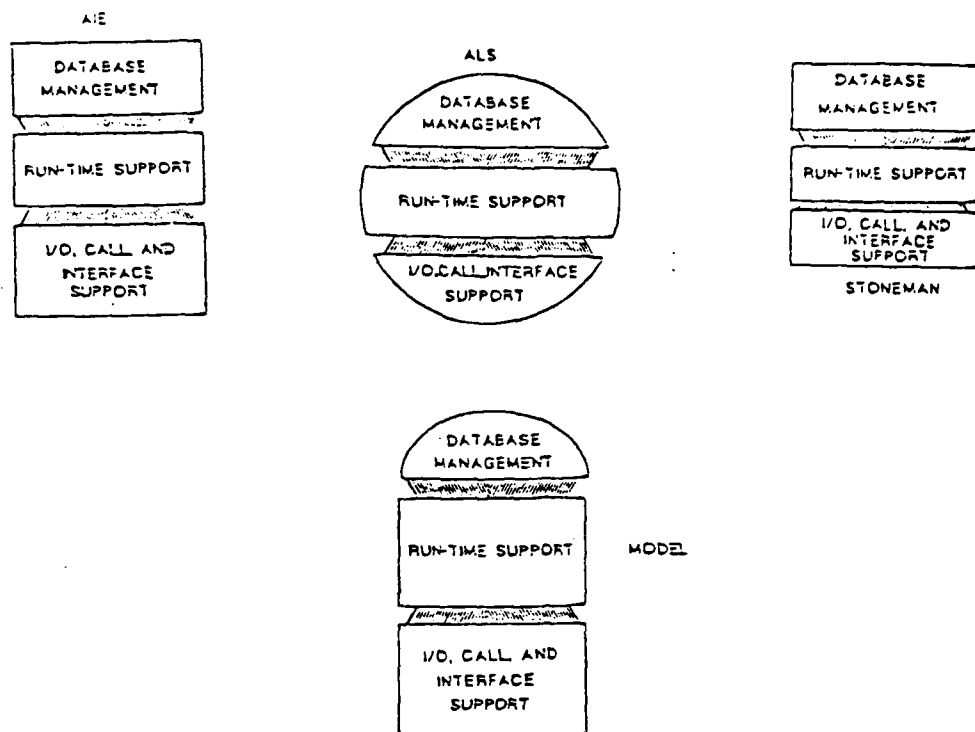


FIGURE 7. BOTTOM-UP FUNCTIONAL COMMONALITY

ANALYSIS RESULTS

In preparation for the joint meeting with the KIT members, the AIE and ALS developers prepared listings of the Ada packages and related procedures and functions that compose their present implementations. This allowed the analysis team to evaluate the commonality of the existing packages and procedures in accordance with the above criteria. A representative list of the compared KAPSE applicable packages is contained in Figure 8. There are two packages for the AIE and two packages for the ALS presented for demonstration purposes.

This analysis methodology was continued for all the KAPSE related packages except for the data base interfaces which are to be considered in a later phase of the analysis effort. The overall results of this analysis effort is presented in Table 1.

AIE		ALS	
PACKAGE	PROGRAM INVOCATION		RESULTS
PROCEDURES	CALL_PROGRAM		OPTION
	PROGRAM_SEARCH		MINOR
	INITIATE_PROGRAM		OPTION
	ANALYZE_PROGRAM		AIE SPEC
	SUBMIT_PROGRAM		OPTION
	RESTART_PROGRAM		OPTION
	CREATE_PROGRAM		1
	PICK_PATH		OPTION
	INVOKE_OPERATION		AIE SPEC

AIE		ALS	
PACKAGE	INTERACTIVE_IO		RESULTS
PROCEDURES	SET_EDD		ANA
	NO_EDD		ANA
	GET_OUTPUT		AIE SPEC
	SET_OUTPUT		AIE SPEC
	GET_INPUT		AIE SPEC
	SET_INPUT		AIE SPEC

ALS		AIE	
PACKAGE	BASIC_IO		RESULTS
PROCEDURES	MAKE_FILE		ADA
	DELETE_FILE		ADA
	OPEN_FILE		ADA
	CLOSE_FILE		ADA
	READ_FILE		ADA
	WRITE_FILE		ADA
	GET_POSITION		ADA
	SET_POSITION		ADA
	GET_INDX		ADA
	SET_INDX		ADA
	PERIOD		ADA
	GET_LENGTH		ADA
	TRUNCATE		OPTION
	SET_PROFT		OPTION
	REVISE_FILE		ALS
	PRINTER_FILE		ALS
	GET_FILE_NAME		ADA

ALS		AIE	
PACKAGE	AUX_IO		RESULTS
PROCEDURES	DEDUCE		MINOR
	MAKE_DIR		ANA
	MAKE_VAR		ANA
	DELETE_NODE		MINOR
	DELETE_TREE		OPTION
	GET_OFFSPRING		OPTION
	RENAME_NODE		ANA
	SHARE_NODE		ANA
	GET_WORKING_DIR		OPTION
	SET_WORKING_DIR		MINOR
	READ_ATTR		OPTION
	WRITE_ATTR		OPTION
	GET_ATTR_NAME		OPTION
	GET_ASSE_NAME		MINOR
	READ_PEP		MINOR
	ADD_PEP		MINOR
	DELETE_PEP		MINOR
	CHECK_ACCESS		MINOR

LEGEND:

ADA PACKAGE
 OPTION
 MINOR
 ANA TED
 AIE SPECIFIC
 ALS SPECIFIC
 *DEPENDENT ON FURTHER ANALYSIS

FIGURE 8. REPRESENTATIVE AIE/ALS COMMONALITY

Package Criteria	Present Commonality
Ada Package	30 %
Common Procedures	20 %
Minor Difficulty	20 %
TBD	25 %
Major Difficulty	5 %

TABLE 1 - AIE/ALS COMMONALITY STATUS

The identification of the commonality status as presented above indicated the present AIE and ALS systems would have approximately 70% of the analyzed packages in common. This could be attained with only minor modifications to the existing systems. The area designated as "TBD" in the preceeding table will be the target of future analysis.

The attainment of such a high degree of commonality allowed the analysis team to advance one step further. This was the definition of a straw family of interface relationships that would be common to the AIE and ALS and applicable to the S_I_S. The interface relationships are presented in Figure 9. These functional categories provide a baseline for the future analysis of the AIE and ALS interface points and compose the initial definition of a Standard Interface Set which is expected to evolve to the S_I_S.

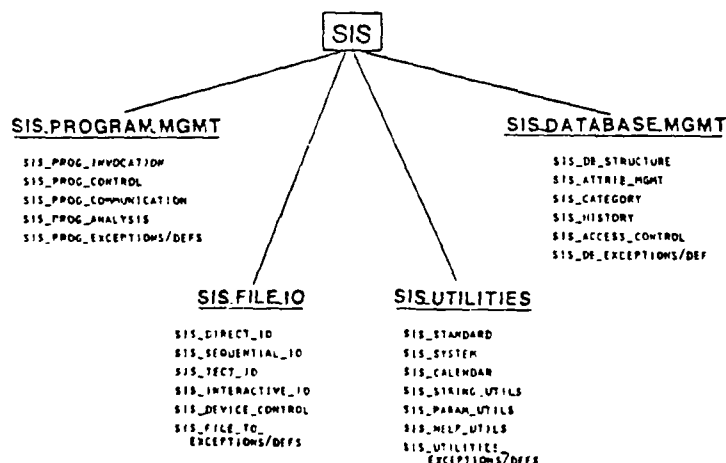


FIGURE 9. STRAW FAMILY OF INTERFACE RELATIONSHIPS

FUTURE PLANS

The results presented herein are preliminary and subject to further review and evaluation. They do provide the starting point for the definition of the S_I_S and will be expanded in future work in this area. Additional meetings will be held with the AIE and ALS developers to explore additional interfaces which are germane to the S_I_S. The interfaces thus defined will be reviewed for completeness and consistency and a draft Standard_Interface_Specification produced for review. Additional reports such as this paper will be published to reflect the present status of the interface analysis and definition status. The next report in this series is planned to contain a cross-reference listing of terms used in the AIE and ALS.

REFERENCES

- [1] Buxton, J.N., Requirements for Ada Programming Support Environments, "STONEMAN", U.S. Department of Defense, February, 1980.
- [2] Memorandum of Agreement Among Deputy Under Secretary (AM), Assistant Secretary of the Army (RD & A), Assistant Secretary of the Navy (RE & S), and Assistant Secretary of the Air Force; Subject: Ada Programming Support Environment (APSE) Tool Transportability, January, 1982.
- [3] Freedman, R.S., "Specifying KAPSE Interface Semantics", KITIA Paper, October, 1982

KITIA Charter

- The KAPSE Interface Team from Industry and Academia, KITIA, is concerned with issues affecting the portability of tools, programs, and databases among Ada Programming Support Environments.
- The Team's purpose is to identify critical issues for the effective implementation of Ada Programming Support Environments, to clarify these issues, and to pursue resolution of these issues. The team will communicate its recommendations and the results of its activities to the Government KAPSE Interface Team and the Ada Joint Program Office.
- The team's activities will include:
 - Definition and clarification of terminology;
 - Study of KAPSE requirements, further definition of "Stoneman";
 - Study of KAPSE interfaces and services;
 - Development of concepts useful for generating requirements, conventions, and standards for KAPSE interfaces;
 - Recommendation of approaches to design and implement KPASEs;
 - Review of major government-sponsored KAPSE programs (ALS and AIE);
 - Assistance and review for government agencies planning for and formulating new APSE-related projects;
 - Recommendation of policy to the Ada Joint Program Office for control of APSEs;
 - Examination of relationship between requirements for interoperability and trasportability and the functionality of a KAPSE and its interfaces;
 - Initiation and development of interoperability and trasportability requirements, methodology, and verification and validation technology.
- The team will provide the following products for October, 1982:
 - Draft documents describing results of selected activities; and
 - The plan for subsequent activity.

GROUP I - CHARTER

DRAFT

23 April 1982

Introduction

This document contains the draft charter of Working Group 1 of the "KAT".

1.0 Policy Issues

Identify issues, raise questions, and formulate postures on critical driving issues which must be addressed and resolved by either the KIT, "KAT", or individual working groups.

During the discussion of the Working Group's charter, many questions affecting the WGs activities were raised. Many of these can only be addressed by higher authorities, such as the entire committee, the KIT, or the JPD. Many of these questions reflect what we determined were policy issues requiring guidelines before we can proceed with other Wg.1 charter areas.

Consequently, we feel that a major responsibility of each working group, and of the committee as a whole, is to identify such issues, and raise them at the organizational level at which they are appropriately addressed.

These issues fall into three major categories:

(Please note that examples are included in the descriptions of the three major areas below for purposes of clarification. Any specific recommendations will be made in separate documents.)

a. Issues fully in the scope of WG.1.

Examples:

1. KAPSE support for LOGON/LOGOFF.

b. Issues of vital interest to WG.1, but outside the scope of WG.1 to resolve.

Examples:

1. Data Base:

Resolve whether a standard interface and standard storage techniques, or an accommodation to permit introduction of developing technology, is the accepted approach.

2. Command Language:

Problems with interoperability of transported systems has not be adequately addressed. Should a command language "approach" be defined by the "KAT".

c. Issues totally outside the scope of WG.1 areas.

Examples:

1. The "Hole"

The specification hole between the Stoneman and the KIWs needs to be filled. What level of specification, what level of detail, and who will address this issue.

2. Levels of Standardization:

Identify what levels of standardization are appropriate in which circumstances. This includes:

1. Standardization

Techniques include common Help Facilities, common functional capability, identical interactions at a terminal (Logon/Logoff).

2. Areas or Levels

Different approaches are appropriate for different components of an Ada support system. It is necessary to identify at what points what approach, if any, is appropriate to meet the (as yet not clearly identified) goals of transportability and interoperability.

2.0 Activities

The following activities constitute the current scope of work of WG.1.

The committee intends to expand the description of these activities, develop philosophies and approaches, and identify the results of the activities.

2.1 Definitions

The definitions of transportability, interoperability, etc., need clarification and refinement.

2.1.1 Goal

Provide "better" working definitions of goals of KAPSE/MAPSE/APSE development.

2.1.2 Activity

Elaborate on the provided definitions for Transportability, etc.

2.1.3 References

1. Definitions

Distributed at the initial "KAT" meeting.

2.2 Review

Review the current situation, considering ALS, AIE, and the Stoneman requirements, to develop an understanding of our starting point.

2.2.1 Goal

Understand current givens, identify current conflicts.

2.2.2 Activity

Perform a comparative review of AIE and ALS in the WG.1 areas. Identify areas in which:

1. Conflict (actual or potential) exists.
2. Inconsistency between the Stoneman and either AIE or ALS exists.
3. Areas in which the Stoneman provides insufficient guidance.

2.2.3 References

2.3 Requirements Development

A clear understanding of the requirements is a prerequisite of any activity developing proposed approaches.

2.3.1 Goal

Requirements, commonly understood, are a critical first step in any development activity.

2.3.2 Activities

1. Host Taxonomy
2. Stoneman Review
3. Tool Requirements
4. Host "Constraints"
5. Target Requirements

2.3.3 References

2.4 Develop KAPSE Design and Implementation Approaches

Approaches range from loosely stated development goals to precisely stated restrictive standards which are deemed inviolate.

Which are applicable in which circumstances must be determined, some as issues resolved by the committee ("KAT") or higher, some within the working group.

2.4.1 Goals

Develop "approaches" to guide Ada Environment development in WG 1 areas.

This is a many part effort, including:

1. Identify areas of consideration.
2. Identify appropriate approach.
3. Develop and define the approach in detail.

Some of the inputs to this activity come from other areas of endeavor, both within WG.1 and outside WG.1.

2.4.2 References

3.0 Products

This section will identify the "products" of WG.1, which will be produced for use by others.

1. Definitions Update
2. Stoneman:!"Hole"!!KIW - Detailed specifications (Stoneman Appendices) in WG.1 areas.
3. Update KIWs for Group 1 - Organization and content
4. Interface Package Specifications and Interface Package definitions for standard interfaces
5. Hit List - Propagate Identified Problems, mainly from Policy Issues.

ADDENDUM

User Support Working Group Charter

1952-04-07

1. Introduction

Seven candidate goals for achieving APSE interoperability and transportability have been proposed for consideration by the KIT/KITI [10]. These goals elaborate on discussions that occurred at the first KITI meeting, and are sufficiently diverse that the choice may have a significant impact on the orientation of each working group's activities. Therefore, a mandatory task for each working group must be to refine the overall charter so that individual charters can be prepared with a consistent understanding of the KIT/KITI's objective [11]. Preliminary evaluation of the proposed goals has indicated that only three options (nos. 2,3,6) retain the original rationale for the KIT/KITI teams. On this basis, the proposed user support working group charter has been formulated assuming that none of the four noncompliant goals (nos. 1,4,5,7) are adopted by the KIT. The interoperability and transportability requirements for user support rely upon the informal definitions developed by the KIT. Formalization of these definitions has been recommended as a necessary activity, since the existing definitions are subject to various interpretations. A preliminary approach, that is consistent with the Ada Formal Definition, has been proposed as a means by which rigorous definitions can be derived [12].

2. General

The charter of this working group (WG.1) is to provide technical input on the requirements that pertain to the interface topics defined as user (tool builder) support. These requirements must be defined in order to achieve the preferred objective of specifying a single APSE-KAPSE interface [1] that meets the perceived interoperability and transportability goals of current and future APSEs. In this context, user support is comprised of the facilities or services required by the tool builder, the Ada program implementor, and the casual user of the APSE. Each type of user may require, in addition to a suite of common services, specialized services that are unique to the user's role or interaction with the system. An understanding of a user's interaction with the APSE is deemed essential, because user transportability is an objective to be attained by rehosting of the APSE through a well-defined APSE-KAPSE interface.

The initial endeavours of the group should be confined to the topic functionality presented in the KAPSE Interface Worksheets (KIWs), so that consistency is maintained with the KIT and other working groups (WG.2 .. WG.4). However, it is realized that there exists a missing level of documentation between the seminal Stoneman Report and the KIWs. This omission may be rectified by suitable appendices to Stoneman, or by another lineage of documentation to which each Working Group might contribute (e.g., Ruddenman, Pondman, Lakeman, Oceanman).

The session objectives for the KIWs indicate a recommended set of immediate activities for all working groups. Therefore, deviations in conforming to this recommended approach should be avoided. Using this rationale, several areas become candidates for early investigation to acquire a better appreciation of the

KITI/WG.1-A001.2

1982-04-07

requirements, and to prompt improved definition and formalization of the KIWs. Five candidate topics for investigation are proposed and would include an analysis of the applicable sections of the AIE and ALS designs, since an urgent need to minimize their differences has been identified by the KIT.

3. Topics for Investigation

3.1 Tool Requirements

While it should be understood that tool functionality is beyond the scope of this working group, an understanding of APSE tool requirements is essential for developing the user support interface. Currently, scant attention has been expended on some aspects of tool functionality; for example, interaction with the target environment. The intended use of the APSE to implement ECS applications demands that this interaction receive significant consideration. Additional insight into tool requirements may be acquired from the tool taxonomy that is being prepared by the National Bureau of Standards.

The services supplied by the User Support interface may influence the user-tool interface and APSE tool writing standards (refer, KIW 1.C8). These issues have already been argued in early position papers [2,3] with respect to the Command Language Processor tool. A recommendation for the eventual standardization of a minimal command language has been proposed [13], however, it is recognized that this has implications outside of the topics assigned to this working group. Consequently, there needs to be an agreed position by the working group on these issues at an early date. In addition, understanding tool requirements may introduce gradual and systematic revisions to the topic breakdown of the KIWs. As an example, the Logon/Logoff services may be refined to a collection of more general services that are required by, but not limited to, the tool that handles terminal start-up. Some of these services may eventually be supplied outside of this interface group (refer, KIW 1.B1).

3.2 Interaction with Other KAPSE Interfaces

The boundaries of each grouping may overlap for some considerable period of time. An important objective of each group's charter is to refine their group topics to minimize this overlap. Identification of requirements by a group should include a rationale for which existing (or new) group should be delegated final responsibility for developing the associated KAPSE service interface. One example of this has been discussed during the recent KITI meetings: device interactions may be viewed as a part of the data base interface and user support interface, since services for storage addressable data carriers (refer, KIW 1.C1) are candidates for either grouping. The miscellaneous group (WG.4) encourages that each group's charter consider generic requirements that are "orthogonal" to more than one group. This was reaffirmed by WG.4's report at the KITI meeting on its perceived role that includes auditing each group's consideration of these generic requirements.

3.3 Host Capabilities

The need to understand the capabilities of the KAPSE environment has been stressed in early position papers [4,5], and is of particular relevance to user support. The working group's charter should address this aspect in order to ensure that transportability is not achieved by enforcing a restrictive KAPSE implementation. The notion of capacity transparent interfaces [6] has been advocated, and should be developed in terms of interface requirements. As a result, interfaces may then provide a means for specifying the semantics of tool requirements that can be implemented efficiently on different host system architectures [5]. Efficiency has been identified as a key concern[1]. The development of a taxonomy of contemporary host systems [6] may be considered as an implied responsibility of this activity, and should be included in the working group's charter. Special attention to distributed system architectures is recommended with respect to program control and device interactions.

3.4 KAPSE Organization

The KAPSE has been described as similar to a virtual operating system [1,5,6], and suggestions for its organization have been proposed [7,8]. This organization not only impacts the rehostability of the KAPSE [9], but is seen to influence the services offered. The concept of layered interfaces [1] has been raised, therefore, this working group needs to develop organizational requirements that relate to user support that are consistent with host capability requirements for capacity transparent interfaces.

3.5 Target KAPSE

The requirement for a KAPSE on target environments is not clearly understood. Currently, KAPSE services for the target are often factored into the ubiquitous run-time system. However, it is evident from the KIWs (viz., 4.M), that each working group must consider the functionality that might be encapsulated by a target KAPSE. Because user support includes services that must exist on target systems, the charter should stipulate a thorough study of target requirements that are most appropriately handled by the KAPSE. In some instances, the host and multiple targets may be configured in a distributed network of machines, thereby motivating the need for target KAPSEs for program control and communication protocol processing. These requirements should be formulated as a subset of the host KAPSE to maintain the relation that the host is a target. Study of this topic is closely allied with "bare machine" requirements.

~~References~~

- [1] Morser, H; Position Paper 28; 1982-02-18.
- [2] Cox, F; Position Paper 2; 1982-02-17.
- [3] Lindevist, T; Position Paper 14; 1982-02-17.
- [4] Cornhill, D; Position Paper 1; 1982-02-17.
- [5] Standish, T; Position Paper 29; 1982-02-18.
- [6] Gardner, A; Position Paper 6; 1982-02-17.
- [7] Fellows, J; Position Paper 3; 1982-02-17.
- [8] Wrege, D; Position Paper 25; 1982-02-18.
- [9] Lyons, T; Informal Proposal; 1982-02-18.
- [10] Lyons, T; Arpanet Mail from Stennino@ECLB; 1982-03-05.
- [11] Lyons, T; Arpanet Mail from Lyons@ECLB; 1982-03-25.
- [12] Freedman, R; APSE Semantic Domains; WG.1 Meeting; 1982-04-05.
- [13] Taft, T; Portability and Extensibility in the Kernel and Database of a Programming Support Environment; AdaTEC Meeting; 1982-04-01.

Group II - Charter

The Data Interfaces Working Group is concerned with the study of database services, inter-tool data interfaces, and inter-tool program representation (intermediate languages). The purpose of the Group is to identify and define the interfaces a KAPSE is to provide for programs. A KAPSE may contain host operating system interfaces, machine-specific coding, and code to implement support of the interfaces for programs which use it. A KAPSE provides the software to support the interfaces between transportable tools, Ada programs (systems, support, and applications), data, and the underlying host system.

The tasks proposed to accomplish this include the following:

- Fact gathering. Examine all major current efforts in KAPSE development from the point of view of identifying how their differences affect transportability (of tools) and interoperability (of data).
- Determine what control and functionality we would like to see in the data interfaces of a standard KAPSE interface.
- High level design. Design the underlying conceptual model for use in specifying the operations of the KAPSE and its database, and define the data interfaces of the standard KAPSE interface.
- Examine possible implementation routes. Determine whether (a) the current efforts contain within themselves the defined standard KAPSE interface (i.e. the standard turns out to be common to the current efforts) or (b) the current efforts can be added to or modified in some way to meet the standard KAPSE interface.
- Check consequences. Determine whether the proposed method of the implementation of the standard KAPSE interface (or indeed the establishment of the standard in itself) would contravene the security, integrity and design goals of the existing KAPSE designs and their intended method of use.
- Other issues. Determine whether other areas of the KAPSE are affected by the proposed data interfaces of the standard KAPSE interface. Consider the inter-tool program representation (i.e., the intermediate language).

Group IV - Charter

Background

Ada is a high-order language for programming computers. The language, together with a group of programming tools and data bases collectively called an environment, is currently being implemented in the Department of Defense under the guidance of the Ada Joint Program Office (AJPO). Developing requirements and evolving standards and conventions for tool and data base portability among current and future environments is the responsibility of the Kernel Ada Programming Support Environment (KAPSE) Interface Team (KIT). In support of the KIT, and Industry/Academia Team (IAT) has been formed to pursue questions and issues of specific relevance to tool portability and interoperability. The IAT will report results and make recommendations both to the KIT and to the AJPO, as the topic demands. The first meeting of the IAT was convened in San Diego, California on February 17-19, 1982. One result of that meeting was the formation of four IAT working groups corresponding to the KAPSE interface categories (see attachment) defined at an earlier KIT meeting. This document is a revised draft of the charter for Working Group IV.

Charter

Introduction. The topics allocated to Group IV demand special attention because their relevance extends across the boundaries of the other groups. In some cases group IV will act mainly as caretaker, guaranteeing visibility to a topic until it is resolved by other groups. In other cases a topic, though broadly relevant, might be addressed primarily by Group IV, and results communicated to other groups. In any event, inter-group coordination is one primary concern of Group IV.

Current Topics. Initially, Group IV will address six topics, all but one (Policy Issues) currently shown on the existing list. The added topic will be jointly addressed by Group IV and several other interested individuals.

Objective. The primary objective of the IAT is to support the KIT in developing a set of requirements for software tool portability and interoperability among different Ada programming support environments. Within this general context -- the details of which will emerge gradually -- Group IV will concentrate the efforts of its first year on proposing, developing, and setting forth explicit recommendations in each of its four areas.

Activities. Groups IV's recommendations will address technical and, where applicable, non-technical aspects of each of its topics. It is expected that the set of topics addressed will expand and contract as other groups develop new issues and resolve existing ones. Group IV personnel will, as the occasion demands, interact with and make recommendations to other IAT groups, the KIT, and the AJPO. The work of Group IV will be structured around the following specific activities:

1. Amplification of existing Group IV KAPSE interface categories, and the preparation of issue papers on each topic.
2. Regular contact with other IAT subgroups, and with the KIT, to identify topics to be added to, or deleted from Group IV's responsibilities.
3. Examination of the states-of-the-arts associated with each topic.
4. For each topic, determination of the characteristics most relevant to IAT objectives.
5. Development of portability and interoperability requirements for topics not explicitly treated by other groups.
6. Periodic review of this chapter.

TOWARDS A KAPSE INTERFACE STANDARD

A report on discussions during the meeting of the KITIA Oct. 4 and 5, 1982.

Introduction and Background

At the initial meeting of the KITIA in February, 1982, some concern was expressed at the fact that there were going to be more than two versions of the "KAPSE". There are, at present, two U.S. Government contractors, SofTech and Intermetrics, and in Europe three more, EEC, German, and British. The philosophies of the interfaces of a KAPSE are apparently different among these developers, though all proclaim to be consistent with the stated STONEMAN requirements. (There are even more possible efforts, commercially, in the U.S.)

The main problem of a set of inconsistent interfaces was seen as one of transportability of tools and data between KAPSEs of inconsistent interfaces. In fact, it seemed that this problem could have serious impact on the acceptance of the Ada concept as a whole. Some of the concerns were reported by a member of the KITIA in the paper that he gave at the AdaTEC meeting immediately following the KITIA meeting in October (see Wrege paper, 3S-1).

The secondary consequence of the inconsistent set of interfaces was seen to be that industry would tend to delay environment and tool investments until a standard settles out. This appears to be a more serious consequence than the inconsistency problem itself.

At our first meeting, Tim Lyons suggested that there were seven "actions" that could be taken by the KITIA to meet the problems that might accrue from two or more KAPSEs with incompatible interfaces. These are given in his paper that is included in this report as 3K-16. This set of seven alternatives spans a spectrum from "Do nothing, and expect to have many different KAPSEs in a few years" to "propose that the DoD cancels one contract so that there is only one today, and then insist that all future DoD KAPSEs conform with this one." In between these was the "marshmallow fluff" alternative; this would put a new interface layer over both current U.S. Government "KAPSEs" to hide the differences. The "fluff" interfaces could then become the standard interface for porting tools and data, etc.

The conclusion, at that time, was that we did not know enough about the two U.S. contractors' system designs and interfaces. As a result, a visit was made to each contractor (by the entire KITIA) with a view to determining enough about the problems to make reasonable proposals to AJPO.

Throughout these discussions there was no doubt about the commitment of the KITIA to the entire Ada concept; indeed, the general belief of the team was that there was a good basis for Ada today: it had a well defined language specification and the proposed standard was moving well forward. The problem was in providing a stable platform (in the sense of a consistent set of interfaces for a KAPSE). And yet it seemed that "KAPSE" had become a term used indiscriminately to mean whatever a particular developer wishes. The impact, as viewed by the computer industry representatives on the KITIA and those approached by KITIA members, is that the two DoD KAPSEs should be used as models for non-DoD KAPSEs. However, the investment to provide a commercial KAPSE, which could prove incompatible with a future standard DoD KAPSE interface definition, is more than most industries are willing to risk. Thus we see reluctance to implement anything associated with a KAPSE (from

commercial environments to Tools-for-Sale).

We were therefore faced with some deep questions:

1. Could Ada be successful without a definition of "standard KAPSE interfaces"?
2. Is it possible to port tools without the basis that a standard provides?
3. What are the hidden costs of a non standard KAPSE?
4. How about the costs of maintenance under non-standard environments?
5. Why isn't STONEMAN enough?
6. Is it too early to standardize, even if the standard is evolutionary?
7. Will AJPO have power to enforce the standard; will services accept software developed using standard-but-non-DoD-developed KAPSEs?

The answers to these questions are not all available yet, but the consensus has been that ultimately a set of KAPSE interface standards is essential for the future. However, in some ways this begs the question, because many of these problems are "today's issues" rather than ones that can be addressed tomorrow.

Prior to the October 1982 meeting, a memorandum and proposal were distributed to the KITIA (see Appendix 1).

In the ensuing discussion, many important points were made, and it was decided that the KITIA should press forward efforts to present a modified proposal to AJPO. The essence of this is included in the next sections. However, a few clarifying remarks are first in order.

The major concerns, as expressed in the meeting, were of three major categories:

1. If effort focuses only on the eventual future standard (e.g., no near-term version), what do the (industrial contractor) technical specialists recommend to their management for current efforts?

The obvious risks of loss by implementing a KAPSE which becomes incompatible, in the future, makes most management reluctant to do anything now, and this may well adversely affect the future acceptance and use of Ada Environments.

2. If we start to generate interface standards too soon, we may stifle good ideas in their implementation, or stop some potentially valuable experiments.

This concern is not just a restatement of the anti-standard position. It represents a feeling of malaise, because there have been very few "glorious experiments" in the realm of environments and the last thing that anyone wants is a poor (but standard) environment. It does seem that a standard could be developed later, maybe in a few years time when there has been some experience gained from the two contracted efforts, as well as from the European and some U.S. commercial ventures.

3. The effort of producing a standard specification is non-trivial and may require substantial resources and time; this suggests that the work will be difficult to staff and that a consortium of contractors may be the only method available. This raises a problem, because the firms would be asked to commit to the use of some of their best technologists for a considerable time, but without any promise of a "leveraged" return on this time.

The first two of these issues seem, at first, to be diametrically opposed, but the following proposal is intended to aid in reaching some middle ground.

Fortunately, AJPO has already initiated a program that will form a base for a future standard.

The realization that there were real concerns about the differences between the two contracted system environments led AJPO to start an effort aimed at finding some commonality of interfaces between the two. The first meeting has already suggested that there is a basic set of interfaces that will make it possible to write portable tools, given some enhancements to the interfaces of both systems (to provide compatibility in certain interfaces that are not compatible at present). However, this will not necessarily help the external contractor who wishes to develop a new version on their machine or preferred system.

The following set of proposed steps are partially intended to aid these organizations.

One final but crucial pair of issues must be raised:

Will AJPO have the power to enforce KAPSE standards after they have been developed?

Will the services be able to cooperate in allowing/mandating the use of the standard?

Obviously, the answers to these questions are crucial to the success of any future effort, and, though we know of the inter-service agreements that say that this power has been delegated, we know of many other efforts with similar backing that have still failed.

A Three Part Action Plan for an Interface Standard

The proposed effort to develop an interface standard is merely outlined here it will be further discussed in a paper scheduled to be delivered to AJPO in mid-November 1982.

The work is split into three major sections. These are to be achieved in the short, medium, and long term.

Stage 1. Work on a "stable platform"

Starting from the work now underway by AJPO to provide a homogeneous set of interfaces, the KITIA proposes the following:

(a) That this be considered to be the start of a standards effort that is evolving an early prototype standard. The importance of this statement to the industrial community cannot be overemphasized, as it will encourage them to expand efforts once they see stability in the interface definitions.

(b) The common interfaces must not just appear as a set of "gentleman agreements" but be written down as a set of good specifications so that others can use them as a requirement statement for their architecture of a KAPSE.

(c) The specifications need a "validation suite" (i.e., a set of procedures that may be used either by DoD in their validation of a vendor offering, or by the vendor in testing, or by "tool producers" in validating their design or implementation).

(d) One of the problems that must be addressed in this effort is the provision of a uniform resource naming convention (or standard) for program and tool names, command feature names, file names, user identifiers, etc.

(e) Software written for the initial standard KAPSE interface must be guaranteed to be supported by the future standard KAPSE interface.

The essence of this effort is speed. It is evident that many companies are awaiting developments, and therefore any help in clarifying policy on KAPSE interfaces will help all

interested parties. We feel that this stage should be completed by mid-1983, (at least in the definitional stage), though a validation suite will require time for its implementation and testing (as well as a few instances of KAPSEs on which to try it out).

Of course, the result of this stage will not be an ideal set of interfaces, but it can be a prototype standard with enough functionality to allow further system implementation within the Ada scheme. Moreover the effort can be carefully screened to ensure that any future standard specification is upward compatible from this base.

Stage 2. Requirements for a KAPSE Interface Standard

Before a standard can be developed, a much better set of requirements must be developed. This could be considered an extension to STONEMAN, but the effort needs a substantially higher level of effort and some interdisciplinary thinking that suggests a major effort with several kinds of "experts" as well as some original thinking and decision making. As an example, the role of database management in the KAPSE must be argued and decided.

It is therefore necessary to both look back, to the stage 1 interface standards, and at the same time forward, to the requirements for the final stage 3 product, since the specification of stage 3 requirements must allow the product of stage 1 to be a feasible "level 0" implementation. At the same time some further consideration must be given to the provision of a validation suite when the requirements are implemented (in the next stage) by specifying a standard interface set.

This stage is not at all easy and will probably be expensive and time consuming, partly because there are many questions that have only been slightly discussed and are still undecided. The effort requires talent and consulting

similar to the Ada effort, but different in kind.

The KITIA would act as a reviewing team during this stage.

Stage 3. Development of a KAPSE Interface Standard Specification

Much of the work to be achieved in this stage is discussed in the appendix.

Appendix 1

THE NEED FOR A STANDARD KAPSE WITH A MOTION FOR CONSIDERATION BY KITIA

Edgar H. Sibley

Background

The KITIA was formed with several objectives that have been discussed in full meetings, in single groups, and in rump sessions over meals and in the evenings. These objectives were all centered around the concept that there was a need for a uniform environment in which an Ada processor could operate and that would provide a stable platform from which tools written in Ada could be "ported." In our discussions, I believe, none has ever suggested that this basic idea is wrong.

There is some disagreement about what goes into a KAPSE; there is also some discussion as to how "portability," of programs, data, and commands can best be achieved. But I think we all agree that these basic "-ilities" are extremely important for the future of DoD systems, and consequently for contractors and all support personnel.

My basic premise, upon which we must first agree, is:

1. A KAPSE interface standard is essential for the success of Ada.

My argument is as follows ...

If we wish to move programs or data from one hardware system to another without any standardization, there are many problems that

must be solved, such as "generalized translation" from one system to another. Such a general translator is probably impossible (I believe that it could be shown that this is non-computable; i.e., it cannot be achieved on a Turing machine). We were all worried, I think, when we saw how incompatible the design of the Intermetrics interfaces were to the implementation of the SofTech design. Of course, given some co-operation, these "differences" can be ironed out (and I understand that the two companies are making substantial progress in producing a "more uniform" set of interfaces). However we are already faced with a proliferation of commercial and world-wide environments, and the future is unlikely to see a halt to this proliferation unless something positive is done. If you need an example, I suggest you think back to the two vendor presentations and ask whether you could imagine the mess in trying to move a debugged Ada program that made extensive use of files (with different naming techniques and different accessing paths). This says nothing of the problems that we would experience when moving the data files.

"But," I hear some of you saying, "Ada is for the HOST/TARGET (H/T) environment, and we should not have to port either data or substantial programs that use data files of the data from one H/T system to another." I think that this argument is incorrect for several reasons:

- (i) The move towards distributed processing (even with H/T systems) will mean that someone is bound to add a different machine environment at some time (in fact, it is already happening).
- (ii) When several, possibly machine environment incompatible, hosts need to communicate with the same target, there will be problems.
- (iii) In spite of the fact that Ada was developed for the process-control like H/T environment, I have it on good authority (and it only confirms my own feelings) that the operational (logistics, etc.) part of DoD will be looking to the advantages of Ada for their future systems. Then we can

expect to have to communicate between many disparate systems and there is no doubt that standard interfaces will be important.

I rest my case.

Next, I must ask: "What chance has the KITIA of defining a standard KAPSE?" I have watched our meetings and realized that we are no worse than most committees and maybe a little better, but does that mean that we could produce a standard definition? I doubt that we can collectively do so, even if we would like to. Again, I ask: "Who do we think could produce a standard in a 'reasonably short' period of time?" I could not convince myself I knew the answer, even though I feel it is terribly important. Thus I come to the second premise:

2. The production of a standard KAPSE is going to need lots of talent, lots of support (bring money), and lots of time.

This immediately spawns some important questions, such as:

(i) Why is it necessary to do this when we already have Stoneman?

I do not think that the KITIA is likely to ask this question, but I am sure that there are many people, not only in DoD, that will be surprised that Stoneman was useful in its time, but little better than a "good basis from which to diverge." In fact, we have seen this when we look at the two current contractors and realize that they cannot be faulted (at least not much) with a charge of "violation of Ada". None the less, we must provide cogent arguments to show the uninitiated that Stoneman is simply not enough.

The second question is likely to be:

(ii) Why cannot this be done by a single (small) group of relatively dedicated people?

My response to this is that the problem of defining a standard KAPSE is probably twice as hard as defining a new language (like the Ada effort) and may be an order of magnitude harder. Thus it seems unlikely that a small group will have all the necessary talent to make it possible: Control Systems, Configuration Management Systems, etc. (You will notice that I put in some of my own preferences, but I think the argument still holds if you substitute your list). But much more to the point, even if I am overstating the difficulty, there is little doubt that we would need more than a small team of part timers or a large team of hacks. I really believe that the effort needs the bringing together of an essentially unique group -- such as we have seen in the Ada specification.

Basis for a Proposal

I would like therefore to give my proposed solution, and ask you to discuss this with a view to making recommendations to AJPO about the way to proceed towards a standard KAPSE:

1. DoD must realize that the work will be time consuming and need full time expensive talent. This, therefore, needs a well planned effort that has the committed resources.
2. A scenario like that which was successful for Ada should be adopted.
3. The success in adoption of the future standard requires proper planning for cut over of operational systems once the standard has been implemented. This means that current developments (the two contractors and any more that matter to DoD) must be watched and guided with the future in mind. Thus some very stringent guidelines on the "strategy of using current Ada environments" should be developed before it is too late.

The scenario that I like would be as follows:

- A. AJPO uses the KITIA recommendations to obtain funding commitment of about \$12M to be used for definition of a standard KAPSE. (to be honest, I am not at all sure that this is enough)...
- B. An RFP is issued as soon as possible to start the process. This would provide funds for one year's effort in the definition of an SKAPSE (standard KAPSE). Four groups would be funded, at a level of about \$1M each. The RFP could require that this be a multi-company effort, with no one company taking more than 45% of the funds.
- C. A run-off after one year would reduce the field to two groups.
- D. A further year of definition would be let at a funding level of \$2M per group. Once again the same multi-company restriction would apply.
- E. The final "winner" will be selected.
- F. A final year of definition will then be funded at \$4M.

Finally, we must set some plausible schedules. And while I do not believe in miracles, I feel that we collectively can act as the right pressure group to help sell the need in a hurry; so here is my best guess:

- A. We give our report to AJPO (they know it is coming, of course)
.....Spring 83
- B. AJPO, knowing our ideas, has started early and succeeds
.....Late Summer 83
- C. RFP issued

D. Four contracts let

.....March 84

E. First run off

.....March/May 85

F. Two contracts continued

.....June 85

G. Second run off

.....June/December 86

F. Presentation of Standard

.....January 87

Proposal

KITIA shall form a small group which will work to prepare a formal statement to AJPO.

This statement shall be based on the memorandum titled "The need for a standard Kapse" distributed by Edgar H. Sibley on Oct. 1, 1982. It will provide AJPO with both the proposed work and the rationale that they will need to start the approval chain for an effort that might be very expensive.

Action will be taken on this proposal at the next meeting of the KITIA, and therefore the group must keep the membership informed by using ARPANET at least every two weeks to give the current status of the proposal.

ARPANET MESSAGES

re: POLICY DISCUSSIONS

-- Messages from file: [USC-ECLB]<POBERNDORF>MAIL.TXT.1
-- Sunday, April 11, 1982 12:26:36 --

Date: 8 Mar 1982 1039-PSY
From: STENNING at USC-ECLB
Subject: KIT/ OPTIONS OR END PRODUCTS
To: CORNHILL at HI-MULTICS, GARGARO at UTAH-20,
To: APSE-BALCH at ECLB, KRAMER at ECLB, POBERNDORF at ECLB,
To: TRVKIT at ECLB, STANDISH at ECLB, YELOWITZ at SRI-KL
cc: STENNING

Colleagues

As promised, here is my explanation of the suggested options we have

0 - INTRODUCTION

There seem to be a number of possible goals or end products towards which the KIT/KITI teams might be working

Some of the discussion is based on the Stoneman aim of life cycle support to contain maintenance costs. That is, the customer wants to be sure that he will be able to use the Apse on which his application system was developed, for his own staff to maintain the system.

The contracting scenarios are only included to assist in explanation. They are not intended to imply that DoD could, would or should act in the way described. (They intentionally blur distinctions (if any) between DoD and Army/Navy/Air force)

1 - N STANDARD KAPSES

The teams accept that there are (and will be) a number of different (incompatible) Kapses, and for some specify strict and complete Kapse interface standards for those Kapse, and for others specify some other organisation which is responsible for similar strict standards for that Kapse. Tools would be fully transportable between different implementations of any one standard, but would not be portable between different standard Kapses.

Contact scenario - The DoD and/or KIT/KITI would satisfy themselves that the various approved standards met the requirements of providing the right maintenance information. The various standards would be published widely so that any organization could build their own environment to that standard, and be sure that they could use other peoples tools for that standard and that the environment would be acceptable for supply to DoD.

2 - NEW KAPSE DESIGN

The teams develop an entirely new Kapse design essentially ignoring the current Kapse designs. That is, the design is not based on any of the current designs, although of course it will learn from the current designs. The design would be forward looking to be considered as a good design at the end of the four year team life. The team would produce a detailed design including detailed Kapse interface specifications

When we consider how this design might be implemented, the design might resemble one of the current Kapse designs, and then that design

might be tweaked. However in general, there is no particular reason why the new design should resemble any of the existing ones. In this case, the new design could be implemented on top of an existing Kapse, but it is most likely that only the very lowest level of Kapse calls of the existing design could be used (eg for the database just read a block and write a block). Hence there would be a very thick layer of software implementing all the Kapse facilities of the new Kapse design in terms of a few simple primitives of the existing one. This thick layer led to this being called the "marshmallow fluff" approach. This approach to implementation would be horribly inefficient, as well as making absolutely no use of the clever, carefully engineered features of the existing Kapse designs

Contacting scenario - obviously DoD is delighted to get any Kapse which meets their own perfect design (shame we didn't get a chance to try out the existing designs before we embarked on a new one)

3 - HIGHEST COMMON FACTOR

The teams develop a detailed Kapse interface standard which consists only of those facilities which are exactly the same (or which can fairly easily be made the same) on the existing Kapse designs

On the face of it, this would restrict the Kapse primitive operations to reading and writing text files to standard input and standard output. File open could not be used because the file name parameter is implementation dependent (and wildly different between ALS and Intermetrics). Program invocation could probably not be used because of the file name problem in specifying the file to be used for standard input and output; there might also be other incompatibilities.

Nevertheless, even in this worst case with a very restricted set of operations, some very useful tools can be written and by means of the standard be guaranteed to be portable. The complexity of the tool can be very great, only its interactions with the outside world need be simple (eg theorem provers etc)

However, it is most important to realise that such a standard would not include the sophisticated database access mechanisms that are built into the current Kapse designs (since they are so different). To some extent the whole Stoneman rationale for a database to hold all the information about a project would be lost - even the tools which were standard and portable would be unable to use the database facilities to record associations etc. In particular, it would not be possible to build portable tools of the configuration control etc type, since these are precisely the tools which use the sophisticated database facilities. If such a tool were to be build using just simple text files to hold all its information, the whole point of the (K)Apse database design would be lost, and you might just as well go back to a simple filing system

Contracting scenario - The DoD can procure and invest in tools which use the standard interface, safe in the knowledge that they are portable. However, there is no standard for the Kapse facilities above the MCF level - and hence DoD would need some other separate method of assurance that they would be able to maintain systems using the Apse (perhaps one of the other methods listed here). Also DoD would not be able to produce eg configuration control tools to this standard.

4 - DATA TO BE CAPTURED

The teams develop a detailed specification of the data which is to be captured by the (K)Apse, and other required facilities, but do not specify a detailed means of meeting that requirement (and do not produce Ada package specs for the Kapse interface). In effect this would be to update, revise and tighten up Stoneman, but still to keep it at the requirements level rather than implementation level.

Contracting scenario - Any organisation could design and build their own (K)Apse which could approach the requirements in quite different ways. The DoD would accept delivery of any (K)Apse which met the requirement.

Determination of whether a (K)Apse met the requirements could probably not be a mechanical process of checking the operation of certain features, but would probably be more like contract negotiation where the supplier points out the features of his system which meet each paragraph of the requirement. The KII/KIII teams could act as standards bodies to rule on whether particular (K)Apses met the requirements

This option would not allow (much) transportability, but would give the greatest freedom for innovation, and for experimentation while allowing DoD to be sure that they could get the necessary maintainability.

5 - DO NOTHING

The teams determine that in the face of existing divergent Kapses, the likely strong favoring of their own systems by sponsoring authorities and further divergence by other organisations, it is not appropriate to produce a standard Kapse interface. This allows freedom for innovation and for experimentation with the existing designs, to determine the requirements and options for the next generation of (K)Apses.

Contracting scenarios - the relevant sponsoring authorities have their own Kapse designs which they will specify for supply

6 - ABANDON ONE

The teams produce a detailed kapse interface standard, including Ada package specs, which is closely based on one existing Kapse design (perhaps making some small changes or improvements to that design)

Contracting scenario - the DoD specify the standard Kapse design for all procurements. Any organisation can develop and supply their own (K)Apse provided it meets the interface standard, All tools, including configuration control etc would be transportable.

A possible variation on this option is to choose one current Kapse design as the current standard, and to aim for or develop a different standard in a longer timescale (say 5 years) - perhaps based on another current Kapse.

7 - OTHER POSSIBILITIES

At the meeting, a further possibility of merging the ALS and AIE designs - or forcing them together - was proposed. I have not included this, as I do not see how it could work. The designs are so fundamentally different in the database areas that I don't see how common ground could be reached except by moving almost totally towards

one or the other - this seems impractical.

Tim Lyons.

-- Messages from file: [USC-ERLE]KPPPPR"DOFF"MAIL.107.1
-- Wednesday, July 7, 1982 17:14:12 --

Mail-from: AFFAIR site HI-MULTICS rovd at 7-Jul-82 1515-EDT
Date: 7 July 1982 17:09 edt
From: Cornhill.ADA at HI-Multics
Subject: KITIA Options
To: (KITIA) at HI-Multics
cc: cornhill at HI-Multics

During our February meeting, we listed a number of alternative paths that we could use to achieve (or at least approach) our goal of a KAPSE standard. Then Tim summarized it all in a useful message which keeps on being referenced. But still we don't seem to be making any progress towards deciding which way to continue.

I suggest that we set as a goal for October's meeting the resolution of this issue. I would like to see this goal clearly identified on the agenda.

In rereading Tim's message, I find that I have more definite opinions now than I did the last time I looked at it.

1 - A STANDARD KAPSE

I don't see how we can adopt this approach as stated. From the DoD's perspective, I suppose $N=2$. But where does this leave the rest of the implementations? (The sum of the rest is greater than 2.) And even if $N=2$, where does that leave future KAPSE designs which are likely to be superior to the ALS/AIE because they will have the opportunity of benefiting from the ALS/AIE experience.

In particular, I expect to see some very nice Ada environments developed on UNIX simply because there is more commercial interest in UNIX based tools than in VMS, MVS and OS/32. It would be a shame to hinder that work.

There are two relatives to this approach that displease me less. Not the whole KAPSE interface is controversial. We could set a single standard for those parts which can be specified. For the others, we could either adopt optional standards, ie, each implementation could choose from a number of approaches (ie ALS-like hierarchical database or an AIE-like relational database), or we could just leave those difficult areas undefined, allowing the implementors complete freedom.

2 - NEW KAPSE DESIGN

I agree with those who said that the KITIA should not try to design the KAPSE. We are not the right group of people, we lack the correct organization.

3 - HIGHEST COMMON FACTOR

I think that this approach would be self defeating. The resultant KAPSE would not be what anybody wanted. It would be too limiting to allow the development of the kinds of tools that we expect the KAPSE to contain.

4 - DATA TO BE CAPTURED

This approach only tries to solve the interoperability problem. I can

see adopting it only after we have decided that the transportability problem has no solution.

5 - DO NOTHING

A good solution if we believe that neither transportability nor interoperability can be solved.

6 - ABANDON ONE

The corollary is "adopt the other." Doesn't this really mean "adopt the ALS?" As I recall the only argument for considering this alternative was that one (possibly poor) KAFSE was better than several good ones.

CONCLUSIONS

I think that we need to put ourselves in a long term perspective. The ALS and AIE are short term projects. We cannot influence them, but if we are long term, we can learn from them. We have four years, lets plan on incorporating four years of experience into our KAFSE interface standard. It may turn out, as has been suggested, that the ALS will become the standard because it appears that 2.5 of the three services are using it. If so, then I suppose that us doing something different (or the same for that matter) will be irrelevant. If the ALS, a short term solution, becomes the long term solution, then we are unnecessary. We become useful if the ALS does not become a long term solution. Which seems to give us more freedom to diverge from the ALS/AIE:

Given a long term perspective, I think that we should strive to achieve both I and T (realizing that they are absolutes which can be approached but not reached). For me, this rules out options 4 and 5 ("Data to be Captured" and "Do Nothing"). I see no reason to abandon a short term project, hence I rule out option 6 ("Abandon One" aka "Adopt the ALS"). I don't see how anybody is going to find the result of option 3 ("Highest Common Factor") useful. The standard would be ignored, just like really bad standards are. This only leaves two options ("Standard KAFSEs" and "New KAFSE Design") and while I don't think either will work, I think that this is where we have to start.

We have agreed that Stoneman is nice but does not go far enough. So why don't we do something about it? We can't do a new KAFSE Design, but can't we at least write some detailed, machine independent requirements that would allow the implementation freedom that is needed but also provide for a sufficient degree of I and T? I believe that we can and I think that this is where we need to focus our work. We can at least start with an iteration on Stoneman (called whatever is going to give us the least trouble), using the experience gained from the ALS, the AIE and other implementations, but without using any of them as a point of departure because they are at a too low level of abstraction. We may find that we need a couple of iterations to achieve the correct tradeoffs between I and implementation freedom. And if it helps, I am not at all uncomfortable with requirements that have alternatives from which

- NEXT MESSAGE IS:
 (MSG. # 252, 7670 CHARS)
 MAIL-FROM: ARPANET SITE USC-ECLB RCVD AT 29-APR-82 0115-PDT
 DATE: 29 APR 1982 0115-PDT
 FROM: HALHART AT USC-ECLB
 SUBJECT: OPTIONS FOR KIT/KITI GOALS
 TO: BALDWIN AT ECLB, CASTOR AT HVSAIL, CONVERSE AT ECLB,
 TO: CSCKIT AT ECLB, DUDASH AT ECLB, HALHART AT ECLB,
 TO: HOUSE AT NUSC, IIKIT AT ECLB, JHAPL AT ECLB,
 TO: JOHNSTON AT ECLB, KRAMER AT ECLB, LINDLEY AT ECLB,
 TO: LUBBES AT ISI, LWEISSMAN AT ISIA, MWOLFE AT ISIA,
 TO: NELSON AT ECLB, NSWC-DL AT ECLB, NUSCKIT AT ECLB,
 TO: PEELE AT ECLB, POBERNDORF AT ECLB, PURRIER AT ECLB,
 TO: SOFTKIT AT ECLB, TIKIT AT ECLB, TRWKIT AT ECLB,
 TO: WALD AT ECLB, WHITE AT RADC-MULTICS, WLOPER AT ECLB,
 TO: CLAUSEN.IAB6 AT MIT-MULTICS, CORNHILL.APSE AT HI-MULTICS,
 TO: FCOX AT ECLB, FELLOWS AT ECLB, FREEDMAN AT ECLB,
 TO: GALKOWSKI.ADM1B AT MIT-MULTICS, GARGARO AT ECLB,
 TO: GLASEMAN AT ECLB, GRIESHEIMER AT ECLB, HFISCHER AT ECLB,
 TO: KERNER AT ECLB, KOTLER AT ECLB, KRAMER AT ECLB, LAMB AT ECLB,
 TO: LINDQUIST AT ECLB, LOVEMAN AT RADC-20, LYONS AT ECLB,
 TO: MCGONAGLE AT ECLB, MOONEY AT ECLB, MORSE AT ECLB,
 TO: POBERNDORF AT ECLB, REEDY AT ECLB, RJOHNSON AT ECLB,
 TO: RUBY AT ECLB, SHIB AT ECLB, SIBLEY AT ECLB,
 TO: STANDISH AT ECLB, WESTERMANN AT ECLB, WILLMAN AT ECLB,
 TO: WREGE AT ECLB, YELOWITZ AT SRI-KL
 CC: AUPD AT USC-ECLB, STENNING AT USC-ECLB

TRICIA, EDGAR, KITI MEMBERS, KIT MEMBERS, ETC.:

I KNOW THAT MANY OF YOU IN DC NOW FOR THE KITI (NAME SUBJECT TO CHANGE) MEETING JUST LIVE FOR YOUR CHANCES TO RUN TO A TERMINAL & CHECK YOUR ARPANET MAIL. SO, HOPEFULLY SOMEONE WILL LIST THIS & INTERJECT IT AS A CONTRIBUTION TO WHAT I AM SURE WILL BE A FAVORITE TOPIC THIS WEEK -- WHERE ARE WE ALL GOING. MAYBE EVEN SOMEONE CAN HAND A PILE OF COPIES OF THIS FOR HANDING OUT TO TRICIA OR EDGAR. MAYBE I WILL EVEN GET TO LIST & REPRD IT BEFORE I FLY IN FOR FRIDAY'S SEGMENT OF THE KITI MEETING.

PEACE,

-- HAL HART

INTRODUCTION

AT THE KIT MEETING IN INDY APRIL 20, TRICIA OBERNDORF DISTRIBUTED COPIES OF AN ARPANET MESSAGE FROM KITI (NAME SUBJECT TO CHANGE) MEMBER TIM LYONS, PRESENTING MANY ALTERNATIVES OF "POSSIBLE GOALS OR END PRODUCTS TOWARDS WHICH THE KIT/KITI TEAMS MIGHT BE WORKING." TRICIA SOLICITED ADVOCATES AND RATIONALE FROM ANY KIT ATTENDEE FOR ANY OF THESE OPTIONS. ALTHO I WAS THE MOST OUTSPOKEN, IN FAVOR OF TIM'S OPTION 2 ("NEW KAPSE DESIGN"), OF THE FEW THAT SPOKE UP, I SUFFERED ONLY LIMITED SUCCESS IN GAINING MANY ENDORSEMENT. THE MAIN OBJECTION SEEMED TO BE SOME UNFORTUNATE PHRASEOLOGY IN THAT OPTION (E.G., "IGNORING THE CURRENT KAPSE DESIGNS") WHICH IF INTERPRETED LITERALLY OR UNGENEROUSLY AREN'T REALLY CONSISTENT WITH MY INTENTIONS. FOLLOWING DISCUSSION CLARIFIED POSSIBLE CONSISTENCIES OF OPTIONS 6 & 5 WITH OPTION 2. TRICIA THEN INVITED THAT FURTHER THOUGHTS ON THIS BE SHARED VIA THE ARPANET AFTER THE MEETING.

THEREFORE, THIS LITTLE TREATISE INCLUDES MY OWN MIND REWRITE OF TIM'S OPTION 2 (TO MAKE IT MORE GENERALLY PALATABLE, AND HOPEFULLY SELF-DEFENDING); MY VERY BRIEF REACTIONS TO EACH OF TIM'S 6 CONCRETE OPTIONS; AND MY BRIEF STATEMENT THAT I REGARD OPTION 2 AS THE GOAL, OPTION 6 (ESPECIALLY THE MENTIONED VARIATION) AS AN EVOLUTIONARY STEP TOWARD 2 (OR A FALL-BACK), AND OPTION 5 AS A FALL-BACK IF PREFERRED OPTIONS ARE LATER DETERMINED INFEASIBLE.

OPTION 2 - NEW KAPSE INTERFACE DESIGN

[FOLLOWING IS MY REWORDING OF THE FIRST PARAGRAPH (WHICH IS THE DEFINITION OF THE OPTION) OF TIM'S OPTION-2 WORDS. (CHANGED WORDS ARE CAPITALIZED. NOTE ALSO ADDITION OF WORD "INTERFACE" IN OPTION 2'S NAME.)

"THE KIT/KITI TEAMS WILL PLAN FOR A NEW KAPSE INTERFACE DESIGN. (WHAT IS) THE DESIGN IS NOT NECESSARILY BASED ON ANY OF THE CURRENT DESIGNS. THE DESIGN WOULD BE FORWARD LOOKING TO BE CONSIDERED AS A GOOD DESIGN AT THE END OF THE FOUR-YEAR TEAM LIFE. THE TEAM WOULD PRODUCE A DETAILED DESIGN INCLUDING DETAILED KAPSE INTERFACE SPECIFICATIONS."

CRITIQUING THE OPTIONS FOR KIT/KITI GOALS

1 - N STANDARD KAPSES: TOLERATING THE IMPLIED SPREAD OF DIVERSITY WILL LIKELY EXTRACT HIGH PENALTIES IN FAILURE TO ACHIEVE IT. IN FACT, THIS SITUATION MAY PREVENT MOST OF THE GROSSEST LIFE-CYCLE COST SAVINGS POSTULATED FOR ADA FROM BEING ACCOMPLISHED.

2 - NEW KAPSE INTERFACE DESIGN: THIS IS CLEARLY MY VISION, AND I HAD THOUGHT THIS WAS THE BASIC VIEW HELD BY EVERYONE INVOLVED WITH FORMULATION OF THE IT EFFORT OVER THE PAST MANY MONTHS. I REGARD THE AIE AND ALS AS PROTOTYPES FROM WHICH KIT/KITI WILL LEARN MUCH ABOUT NEEDED "IDEALIZED" KAPSE INTERFACES AND ADJUNCT APPROACHES (E.G., TOOL-WRITING STANDARDS) TO ACHIEVE PRACTICAL IT COST SAVINGS.

3 - HIGHEST COMMON FACTOR: I DOUBT THAT THIS COMMON SUBSET OF FACILITIES (ESP. IF KPSE-LIKE EFFORTS INDEPENDENT OF THE AIE & ALS ARE CONSIDERED) WILL BE EITHER RICH ENOUGH OR USABLY STANDARDIZED (FROM TOOL-KAPSE INTERFACE POINTS OF VIEW) TO ACHIEVE A REAL KPSE; IN FACT, ANY IT SAVINGS RESULTING (IF ANY) ARE LIKELY EXCEEDED BY PRODUCTIVITY LOSSES IN THE FAILURE TO SUPPORT A REAL INTEGRATED ADA ENVIRONMENT.

4 - DATA TO BE CAPTURED: I & T SEEM IMPOSSIBLE IN THIS APPROACH. I DO NOT EVEN UNDERSTAND THE LAST SENTENCE WHICH SAYS THIS ALLOWS DOD TO GET NECESSARY MAINTAINABILITY (HOW, ON DIFFERING KPSE'S?).

5 - DO NOTHING: WE SHOULD STRIVE FOR MORE STANDARDIZATION THAN THIS (BECAUSE OF OBVIOUS IMPEDIMENTS TO IT BETWEEN CURRENT ALS & AIE), BUT ACKNOWLEDGE THE POSSIBILITY THAT OUR FONDEST DREAMS AND APPROACHES MIGHT ONE DAY BE PROVEN INFEASIBLE (ON A COST-BENEFITS BASIS). SOME PARTS OF TO-BE-ADOPTED STRATEGIES FOR IT ADVANCEMENT MAY BE INFLUENCED BY THIS CONTINGENCY, E.G., PERHAPS A LOT OF EMPHASIS SHOULD BE DIRECTED TOWARD INTER-TOOL DATA-FORMAT-TYPE INTERFACES (OF WHICH DIANA MAY BE ONE EXAMPLE FOR SOME CLASSES OF TOOLS) AND OTHER TOOL-WRITING CONVENTIONS (NOT DEPENDENT ON KPSE INTERFACES OR DESIGNS) SINCE THOSE COULD YET RESULT IN PRACTICAL STANDARDIZATION FOR TOOLS EXTENDING THE AIE & ALS KPSE'S; WHO KNOWS; COST-BENEFITS ANALYSES MAY DEMONSTRATE THAT THESE TYPES OF PURSUITS ARE MORE FRUITFUL THAN STANDARDIZING CONCRETE KPSE INTERFACES.

6 - ABANDON ONE: I DON'T WANT TO NECESSARILY PLAN ON THIS OR ANNOUNCE THIS NOW (FOR SEVERAL NON-TECHNICAL REASONS. HOWEVER, IF OPTION 2 IS CLARIFIED AS OUR GOAL, OPTION 6 MAY BE A REASONABLE INTERMEDIATE STEP TO EXPECT TO REACH IN ABOUT 1984-1985 AS PART OF AN EVOLUTIONARY PROGRESS TOWARD THE END GOAL. IT MAY EVEN THEREAFTER BE DEEMED A LEAST UNACCEPTABLE FALL-BACK END GOAL IF THE EVOLUTION CAN NOT FEASIBLY COMPLETE.

7 - OTHER POSSIBILITIES: [STAY TUNED FOR FALLOUT FROM POSSIBLE TECHNICAL EXCHANGES (COLLABORATIONS?!) BETWEEN AIE & ALS CONTRACTORS.]

CONCLUSIONS / RECOMMENDATIONS

WELL, I CONCLUDED MY INTRODUCTION WITH MY RECOMMENDATION: DRIVE FOR OPTION 2; CONSIDER OPTION 6 AS A POSSIBLE INTERMEDIATE MILE-STONE IN AN EVOLUTIONARY PROCESS; AND LET THE "THREATS" OF OPTIONS 6 & 5 BE REALIZED IN DEVELOPING & REFINING KIT/KITI STRATEGIES FOR ADVANCEMENT. I SEE LITTLE PROMISE OR COST JUSTIFICATIONS IN 1, 3, AND 4.

HOPEFULLY THIS LITTLE (POORLY FORMATTED - SORRY, MY PREFERRED WORD PROCESSOR IS DOWN TONITE) EXPOSITION WILL FOSTER FURTHER THOUGHTS AND DISCUSSION FROM OTHER KIT/KITI PARTICIPANTS. I THINK A RATHER CLEAR VIEW OF THIS GLOBAL OBJECTIVE NEEDS TO BE AGREED UPON RATHER QUICKLY SO WE CAN THEN PLAN AND IMPLEMENT A DETAILED, TECHNICALLY AND MANAGERIALLY SOUND, COST-JUSTIFIED APPROACH FOR ACHIEVING THE DESIRED GOALS.

THANK YOU!

-- HAL HART, TRW

4/29/82

PROGRAM INVOCATION AND CONTROL

Anthony Gargare
COMPUTER SCIENCES CORPORATION

Abstract

This interim technical note develops concepts previously presented in an original position paper [1], and reflects ongoing working group (WG.1) participation in support of the following KITIA charter activities [2]:

- further definition of Stoneman
- study of KAPSE interfaces and services
- approaches to KAPSE design and implementation.

In particular, this note presents some preliminary observations on the program invocation and control interface with respect to distributed processing and software security. The note also reflects the author's initial response to several Arpanet communications that have referenced APSE distribution and security, and the debugging interface.

Comments pertaining to this note are solicited in order to assist in the preparation of a formal working paper suitable for wider visibility.

The program invocation and control interface defined in the KAPSE Interface Worksheets (KIW 1.a) presents an abstraction of user functionality derived from the informal requirements of Stoneman [3], and their materialization of support documented in the ALS [4] and AIE [5] B5 specifications. A more comprehensive view of this interface was presented in the referenced position paper in the framework of code execution services. In this canonical framework, program invocation and control is advocated as properly subsumed by

the granularity of the thread of control that is managed by the interface, and several suggestions for further research were identified.

This perception of program invocation and control has become justified with the recognition in the KIWs (4.b,c,f) [6] of distributed processing, security, and extensibility considerations on the KAPSE interfaces. Requirements pertaining to these considerations were not specifically included in the Stoneman definition, and have therefore received little attention in the KAPSE interfaces of the current designs. Also, the notion of Capacity Transparency introduced in the referenced position paper seem applicable to accomodate program distribution and security, and is compatible with the flexibility essential for potential interface extensions offered through portable run-time support systems [7].

Recently, there has become an increasing awareness for both survivability and security in target resident programs for tactical battlefield applications, where survivability has become synonymous with program distribution. Since a host may be isomorphic with the target, these requirements are legitimately propagated to the host environment. A consequence of this propagation is the realization that when the target is not a host, the target must be supported with analogous functionality to that provided by the KAPSE; the Target Ada Support Kernel (TASK). While this concern is strictly outside the scope of the KITIA activities, it should not be ignored in the KAPSE, since it is believed that reaching a standard KAPSE interface is optimally achieved through a family interface approach that does not exclude TASK oriented members. This approach deemphasizes a traditional layered KAPSE, in favor of exploiting the generic units, library facilities, and package specifications available in the Ada language. This is commensurate with the expected trend that encourages the applications of programming language principles in the design of software systems [8].

The facility for a KAPSE to be adaptable to totally distributed and secure environments extends beyond proved software technology. In addition, understanding the implications of such an adaptation are not readily foreseen. However, the founding rationale for the KAPSE is to ensure that these environments have a minimal impact on the transportability of the APSE tool set. While the posed problems are being formally addressed, useful principles obtained from existing and experimental designs can be factored into early interface requirements and KAPSE design by understanding distribution and

security issues as they relate to code execution services. To expedite the following discussion, familiarity with some of the fundamental principles for distributed and secure software is assumed.

Before the security policies and mechanisms relevant to an APSE are investigated, the implications of an APSE that can execute on a distributed configuration must be delineated. It is interesting to note that recent study [9] has presented substantial arguments for secure systems to be conceived as distributed systems, where security is achieved partly by the physical separation of the individual components (tools), and partly by the trusted functions performed by some of the components. As a result, the various security requirements of APSE components are recognized and enforced by the individual component. The APSE becomes a network of cooperating, distributed, and independent programs, where physical separation of each program provides an alternative to the conventional security kernel. The specification and verification of the properties of the trusted functions, and the functionality to support the distribution of the components are seen as critical KAPSE design considerations. This approach is to be elaborated upon in subsequent technical notes, but in the interim it will be used to influence and justify decisions regarding distributed processing capabilities for the APSE that are motivated from both a logical and physical perspective.

The Stoneman report does not directly address the requirements for a distributed APSE. A general guideline is proposed that the APSE should be designed to exploit the underlying hardware of the host system. Presumably, if the hardware comprises a network of interconnected machines, the KAPSE is expected to provide sufficient functionality to utilize the network efficiently. For example, a user may logon to a local node of the network, and yet have session processing performed by any node in the network that satisfies the processing resource requirements. Such a capability would be characteristic of a KAPSE that provided capacity transparent interfaces that guarantee consistent tool execution on any node selected. The attainment of this level of distribution is not anticipated in either of the current APSE implementations. Each design has recognized that distributed processing requirements must not be obviated by an overly restrictive interface, but there is little evidence of any serious commitment to scope or service the requirements.

As a consequence, it is necessary to stipulate formal guidelines for studying

the performance of an APSE on a physically distributed host system. For convenience, distributed processing is used to denote any hardware environment that supports either logical or physical separation of processing. Justification for this definition has been suggested earlier, and will be argued subsequently in terms of functional isolation for system security.

The need to consider one aspect for the physical separation of processing was identified in the Stoneman report as a result of down-line testing requirements [10]. This style of testing envisaged a target resident program under the supervision of a host resident debug tool, thereby distributing the APSE functionality between the host and target machines. Thus, the host and target form a limited, but legitimate, distributed configuration, especially when the target is another host that is used to provide the required resources necessary for testing the application program.

The outcome of this observation has an impact on the program monitoring interface that must now be perceived as a variant of the program invocation and control interface. The debug tool communicates and controls the executing application program through a debug implant that has been linked with application object code. As a result, this variant motivates the need for a consistent extension to the program invocation and control interface category to include semantics for distributed processing. In addition, it established the rationale for the APSE to be distributed on different machines since multiple users may wish to test programs on distinct or heterogeneous targets. The case of rehosting the APSE is an excellent paradigm for down-line testing. The user employs the debug tool executing on the host to test a tool rehosted on the target (the new APSE host). If the rehosted tool is the debug tool, it may in turn be testing a program executing on the host which is then considered a target. In this instance, the functionality of the APSE is not only distributed, but a machine is configured in the role of both a host and a target.

In the context of the code execution services elaborated in the referenced position paper, distribution of processing may be considered at two levels of user visible functionality, interprogram and intraprogram. Interprogram functionality is representative of the program invocation and control (including the program monitoring variant) services that are essential to support the Command Language Processor, and the requirements for tool synergism. This is clearly demonstrated by the minimal facility for the

Command Language Processor to enable a nonstandard command language to be made available in the APSE by program invocation. Interprogram functionality is representative of the services required to support Ada tasking that are profitably implemented outside of the run-time support library. This suggests that seven classes for logical and physical distribution may be postulated for the code execution services within an APSE.

Class-0 formulates a basis for logical distribution where the code execution services are provided on a single machine (uniprocessor) with no facility for physical down-line testing. The machine is multiplexed among separate threads of control for interleaved execution of programs and tasks. This achieves the minimum properties required by the APSE. From this basis additional levels (1..6) are derived to introduce increased physical processing distribution for interprogram and intraprogram functionality.

Class-1 provides for physical down-line testing as described previously. The code executions services of the KAPSE are upgraded to facilitate the initiation and control of a program on the target machine. The debug implant in the program uses a KAPSE compatible interface that must be supported on the target machine. Through this interface the program is loaded, initiated, and controlled via the implant. The functionality of this interface promotes the concept of target software (i.e., The TASK) that is separate from the run-time support appendage to the application program. If this software was not present a potential lack of capability would occur when the target machine must be shared among several users of the APSE debug tool.

Class-2 is derived to support separate threads of control on a multiprocessor configuration. Both interprogram and intraprogram functionality are revised to exploit the opportunity for improved performance through parallel execution of programs and tasks. It is expected that this upgrade would not change the KAPSE interface specification, but would influence its design and implementation. A suggested reorganization of the ALS KAPSE design for multiprocessor support was outlined in the referenced position paper.

Class-3 is derived to support the physical distribution of APSE tools. Interprogram code execution services are enhanced to facilitate communication among APSE tools executing in a network of homogeneous noded (machines). Each

node (in its role as a host), is required to execute a family member of the same implementation of the KAPSE, and a program execution environment is constrained to a single node: i.e., a program executes on the node on which it was initiated. This does not preclude different executions of the same tool on different nodes. For example, the Command Language Processor might have multiple executions on every node, whereas the compiler may be restricted to nodes with sufficient resources, e.g., the Ada library may be dedicated to a specific node. In this instance, invocations of the compiler would attempt initiation on the appropriate node. Extensions to the code execution services would be accompanied by significant enhancements to the KAPSE to support distributed processing supervision, network topology management, packet handling, and protocol services.

Class-4 is derived from Class-3 to support the physical distribution of Ada tasks on machines hosting the same KAPSE implementation. Intraprogram functionality is upgraded to activate and control the execution of Ada tasks on separate machines. Significant revisions to the KAPSE and run-time support library must be implemented to compensate for the fragmentation of a shared program environment. Popular implementation strategies based upon procedure calls [11] would become obsolete, and many severe problems associated with its feasibility are anticipated. These difficulties of distributed tasks warrant elaboration in a future technical note.

Class-5 and Class-6 dominate Class-3 and Class-4 respectively, and are derived to facilitate APSE tools executing in a network of heterogeneous nodes. Although different KAPSE implementations would exist in a network, each KAPSE would be required to supply an identical (or proper subset) APSE-KAPSE interface. For example, one KAPSE, hosted on a multiprocessor, might provide in addition to Class-5, a Class-2 interface, whereas another KAPSE would only provide the stipulated Class-5. Class-1 can be viewed as a degenerate case of Class-5, where the TASK supports a subset of the Class interface.

Bibliography

- [1] KITIA Position Paper No. 6; 1982-02-17.
- [2] Preliminary KITIA Charter; 1982-04-30.
- [3] Requirements for Ada Programming Support Environment; Stoneman; 1980-02.
- [4] Ada Language System; KAPSE B5 Specification; SofTech, Inc.; Report CR-CP-0059-883; 1981-08.
- [5] Computer Program Development Specification for Ada Integrated Environment; KAPSE Database Type B5; Intermetrics, Inc. and Massachusetts Computer Associates, Inc.; Report IR-678; 1881-03-13.
- [6] Another New Category; Arpanet Communication; 1982-05-27.
- [7] Ada Integrated Environment: Computer Program Development Specification, Part 1; Computer Sciences Corporation and Software Engineering Associates Inc.; 1981-03-15.
- [8] SIGPLAN'83 Call for Papers - Programming Languages Issues in Software Systems.
- [9] Rushby, J. M. The Design and Verification of Secure Systems; Proceedings 8th. ACM Symposium on Operating System Principles; 1981-12.
- [10] Fairley, R. E. Ada Debugging and Support Environments; SIGPLAM Ada Symposium; 1980-12.
- [11] Habermann, A. N., Nassi, I. R. Efficient Implementation of Ada Tasks; CMU-CS-80-103; 1980-01.

Specifying KAPSE Interface Semantics

Roy S. Freedman
HAZELTIME CORPORATION

1. Introduction

Part of the requirements for the design of standard KAPSE interfaces is for an understandable formal semantics. The importance of this is seen in sections 5.0, 6.3.1 and 6.3.2 in 5. We wish to discuss the issues involved in developing a formal semantics for the standard KAPSE interface. These issues include the nature of the formal semantic model (operational vs. denotational), the problems with the "layered" KAPSE as opposed to the "distributed" KAPSE, and the nature of side-effects in executing an interface call. We also recommend an approach to specifying the interface semantics and discuss a possible conversion strategy between the two KAPSE Interface sets.

2.0 Semantic Models

2.1 Operational vs. Denotational

The operational approach to formal semantics involves specifying an abstract machine and the behavior of this machine (i.e., how the machine is supposed to work). The semantics of a program is then defined in terms of the actions of the machine. Operational semantics corresponds to a "black-box" approach. The NYU Ada/ED is an example of an operational semantics. In order to develop an operational semantics, it is necessary to understand exactly how an action is to be performed on an abstract machine (with the ultimate goal of defining the meaning of a program in terms of the input-output relations of the machine).

The denotational approach involves specifying mathematical meanings to objects and parts of programs (abstract syntax categories) in such a way that the semantics of a program is defined in terms of the semantics of its parts. This approach is modular and in one sense distributed, since the semantics of a program can be described in terms of its (distributed) modules. The Ada formal definition ² is an example of a denotational semantics. In order to develop a denotational semantics, it is necessary to understand exactly what a program is doing so that the semantics of its modules can be described.

We assess the differences between the two approaches. An operational semantics is a bottom-up approach (since one starts with the machine and eventually defines the program). A denotational semantics is top-down (since one specifies a module in terms of its constituent submodules). The operational approach is strongly dependent on the implementation (the abstract machine). The denotational approach is independent of a particular implementation, since these semantics emphasize what is being computed and not how these computations are being done.

Implementations, however, are ultimately what designers want. For pragmatic reasons, most languages are traditionally defined operationally, and later a denotational semantics is supplied to describe the language in a "machine independent" way. Unfortunately, the operational semantics also strongly influences the implementation. This is an important problem if one desires guidelines for machine independent semantics. This is one

aspect of the problem associated with developing standard KAPSE interfaces, if one considers the ALS (or AIE) as an operational specification of an APSE.

It has been suggested that the traditional roles of operational and denotational semantics be reversed ³. In this respect, a denotational semantics can be first provided to guarantee implementation independence, and an operational semantics be later provided to implement the denotational specifications. This approach has been followed (with varying degrees of formalism) in the definition of the Ada language (with the exception of tasks). It is too late to apply this prescription to the design of KAPSE interfaces, since operational semantics exist already. (Stoneman is too weak to be regarded as anything but an informal specification.) The problem of designing a standard set of KAPSE interfaces corresponds to the problem of constructing a machine-independent (denotational) semantics given that different sets of operational semantics already exists (ALS and AIE). Consequently, it is recognized that a good part of the semantic specification of the interfaces will involve "working backward" in order to "abstract" the details of the existing operational models. (An alternative strategy is to scrap both operational models and to start with a newer, stronger operational version of Stoneman as the "single" KAPSE. This has been rejected because of the lack of experience and legacy in implementing APSE's.)

2.2 Recommended Approach

We formalize the semantics of "working backward" from an operational model. Wand has shown ¹⁰ that given a denotational semantics of a language, an abstract machine (operational semantics) can be derived by considering representations of the continuation domains. Wand's method (initially applied to compiled languages) produces different machine language implementations from a denotational specification. The trick is to "encode" the results of the semantic equations (continuations) into expressions that look like machine language statements. An inverse function (an "abstraction" function) is also demonstrated that "decodes" the machine representation into its actual denotational semantics. The method is important because a particular implementation of a language can be derived directly from the denotational semantics without explicit use of an operational approach. A pictorial representation of Wand's method is in figure 1.

We apply these concepts to specifying standard KAPSE interfaces. We regard the KAPSE of ALS and AIE as representations of a standard KAPSE denotation. Their interfaces are consequently representations of the standard interfaces. The problem of "working backward" is then formally equivalent to deriving a denotational semantics of a KAPSE, with continuation semantics represented by either ALS or AIE. Figure 2 is a diagram that corresponds to this approach.

At this point we discuss the actual nature of the representations of the continuation semantics of the "standard" KAPSE interfaces. Preliminary guidelines emphasize that these interfaces follow Ada syntax. Consequently, REP-ALS and REP-AIE (in figure 2) are syntactically Ada programs. IO_MAP corresponds to the description of these Ada programs following the standard Ada semantics. Since the Ada semantics has been formally defined, it may be useful (some time in the future) to specify a standard KAPSE representation (figure 3). This would correspond to the previously mentioned strong Stoneman model of developing "a single KAPSE".

2.3 Formal Semantics and the "Layered" KAPSE

We observed above that the design of implementation independent KAPSE interfaces is best based on a denotational approach. This approach is modular and corresponds closely with the structures in the Ada language. In the existing KAPSE implementations, interfaces are Ada procedures embedded in Ada packages. These operational models have the (denotational) characteristics of being modular and distributed. Consequently, a "distributed" KAPSE model offers the best chance of designing implementation independent interfaces.

A "layered" model of a KAPSE can be thought of as a restricted special case of a distributed KAPSE, where modularity is replaced by complicated nesting. The semantics of the interfaces of a layered KAPSE must also be symmetric: inter-layered KAPSE calls cannot violate the order of the layers. This implies that the number of KAPSE

calls increases linearly with the number of layers. The semantics of layering also implies that the number of possible interfaces to be specified increases exponentially with the number of layers. This adds unnecessary complication to the KAPSE in terms of KAPSE validation ⁶, as well as substantial implementation overhead. It is possible to construct a distributive KAPSE model in which the number of KAPSE calls increase exponentially with the number of nested modules, but this is not observed in the package specifications for the ALS KAPSE. The more general distributed KAPSE model corresponds both to the denotational and existing operational models, and is rich enough to permit a taxonomy of KAPSE architectures ⁴. It is recommended that the distributed KAPSE model be followed in designing standard KAPSE interfaces.

3.0 KAPSE Interface Semantics

3.1 Side Effects

KAPSE interfaces are represented in the ALS and AIE as Ada procedures encapsulated in Ada packages. The semantics of these interfaces follows the semantics of Ada. What is not obvious is that many interfaces are characterized by their side effects. These Ada procedure calls (as interfaces) can change the state and environment of computation in ways not associated with the program unit that contains them. In fact, most input-output semantics

are associated with side effects.⁸ For example, using package DIRECT_IO ¹

```
READ (F, X);
```

An element is read from the "current index" of the file associated with F. This element is assigned to X and the side effect is that this "index" is incremented by one. These side effects occur whenever there is an implicit use of the Ada input-output package. Side effects will be seen in the Ada representations of the KAPSE interfaces. In the ALS, these side effects are seen in many interfaces prefixed by MAKE_, DELETE_, SET_, and GET_. They are necessary in order to change the external environment of an Ada program. These side effects should be specified in the (denotational) KAPSE interface semantics when one works backwards from their representations.

3.2 Procedure Semantics

KAPSE interfaces are specified with the semantics of Ada procedures. In order to derive a standard interface specification, certain properties of these procedures should be identified so as to "abstract" these features for a denotational (machine independent) semantics. We follow the guidelines proposed in 6.3.2 of 5 and also 9. We suggest:

- (i) Interfaces must follow Ada syntax.
- (ii) The number of parameters in the interface calls should be explicit. Default parameters should be eliminated. Defaults may be KAPSE implementation dependent.

- (iii) Interface parameters should be machine independent. Since these KAPSE interfaces are specified in Ada packages, it is important that these package specifications do not contain machine dependent programs, attributes, representation specifications, or any other feature listed in Appendix F of ¹.

4.0 Strategy for KAPSE Interface Conversion

We conclude by suggesting a conversion strategy for Ada program portability that is modelled after ⁷ and ⁹. We show how standard KAPSE interfaces on one APSE may be converted to standard KAPSE interfaces on another APSE. The tool that does this conversion will be called TRANSFORM. One requirement will be that TRANSFORM run on both APSEs.

We assume a pathname in APSE_1 exists called TOOL.APSE_1. Corresponding to this pathname is an Ada program that makes use of a standard APSE interface:

```
with KAPSE_1;

procedure MY_TOOL (      ...      ) is
    ...
    SOME_INTERFACE (      ...      );
    ...
end MY_TOOL;
```

The procedure SOME_INTERFACE is implemented in KAPSE_1. This program can be compiled on APSE_1. We call TRANSFORM (on APSE_1):

```
TRANSFORM ("TOOL.APSE_1", "TOOL.APSE_2");
```

where the arguments are strings that denote appropriate pathnames. The pathname TOOL.APSE_2 corresponds to the following Ada program that is constructed by TRANSFORM:

```
with KAPSE_2;
with KAPSE_1_X_KAPSE_2;
procedure MY_TOOL ( ... ) is
use KAPSE_1_X_KAPSE_2;
...

    FILTER_INTERFACE ( ... );
...
end MY_TOOL;
```

TRANSFORM replaced the procedure call SOME_INTERFACE (which is a representation of the standard interface semantics on KAPSE_1) with FILTER_INTERFACE. FILTER_INTERFACE is implemented in KAPSE_1_X_KAPSE_2 and reflects the semantics of the standard KAPSE interface as represented on KAPSE_2.

The procedure FILTER_INTERFACE is not an interface but a set of calls to the standard interfaces represented in KAPSE_2. This program can now be compiled and executed on APSE_2.

5.0 References

1. Reference Manual For the Ada Programming Language
(Draft Revised MIL-STD 1815), U. S. DoD., July 1982.
2. Formal Definition of the Ada Programming Language, U. S. DoD., November 1980.

5.0 References (Cont.)

3. Ashcroft, E. Wadge, W., "RX for Semantics", ACM TOPLAS, Vol. 4(2), April 1982, pp. 283-294.
4. Gargaro, A., KITIA Interim Technical Note, WG.1-A002.
5. Hart, H., "Interoperability/Transportability Criteria/ Requirements for the Design of KAPSE Interface Specifications" (Draft), September 26, 1982.
6. Lindquist, T., Lee, J., Kafura, D., "Validation Methods for the KAPSE Interface", May 1982.
7. Samet, H., "Experience with Software Conversion", Software-Practice and Experience, Vol. 11 (10), October 1981, pp. 1053-1069.
8. Stoy, J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, Mit Press, Cambridge 1977, p. 377.
9. Waite, W., "System Interface", in Software Portability, Cambridge University Press, Cambridge, 1977, pp. 127-135.
10. Wand, M., "Deriving Target Code as a Representation of Continuation Semantics", ACM TOPLAS, Vol. 4 (3), July 1982, pp. 496-517.

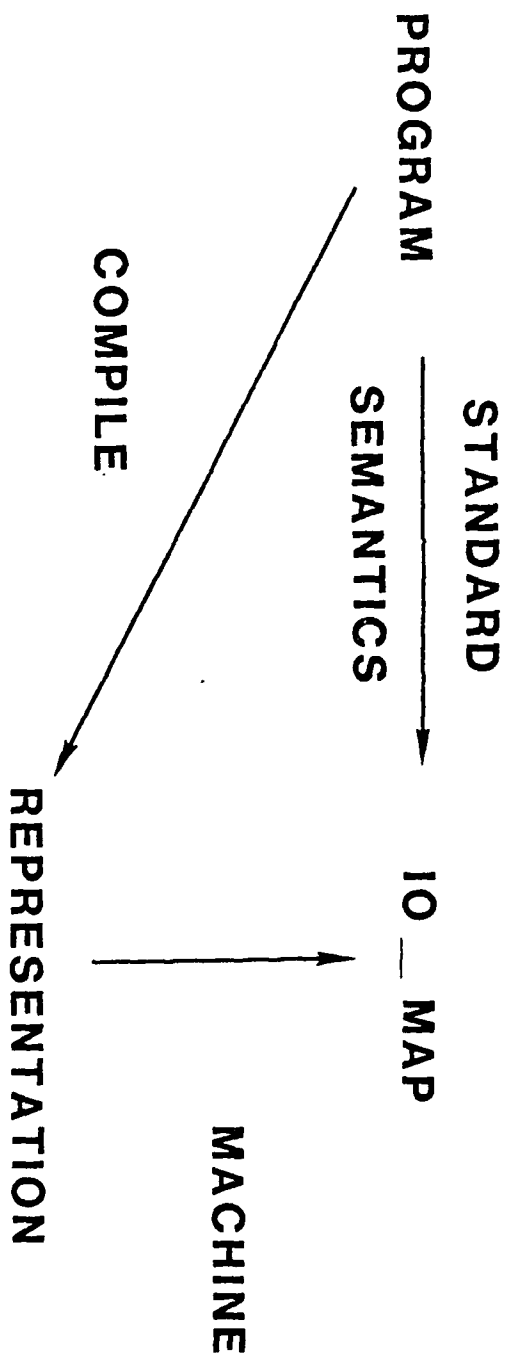


FIGURE 1

WAND'S METHOD. THE TRADITIONAL APPROACH TO SEMANTICS IS INDICATED AS THE FUNCTION (ARROW) FROM A PROGRAM DOMAIN PROGRAM TO A DOMAIN OF INPUT-OUTPUT MAPPINGS (IO_MAP).

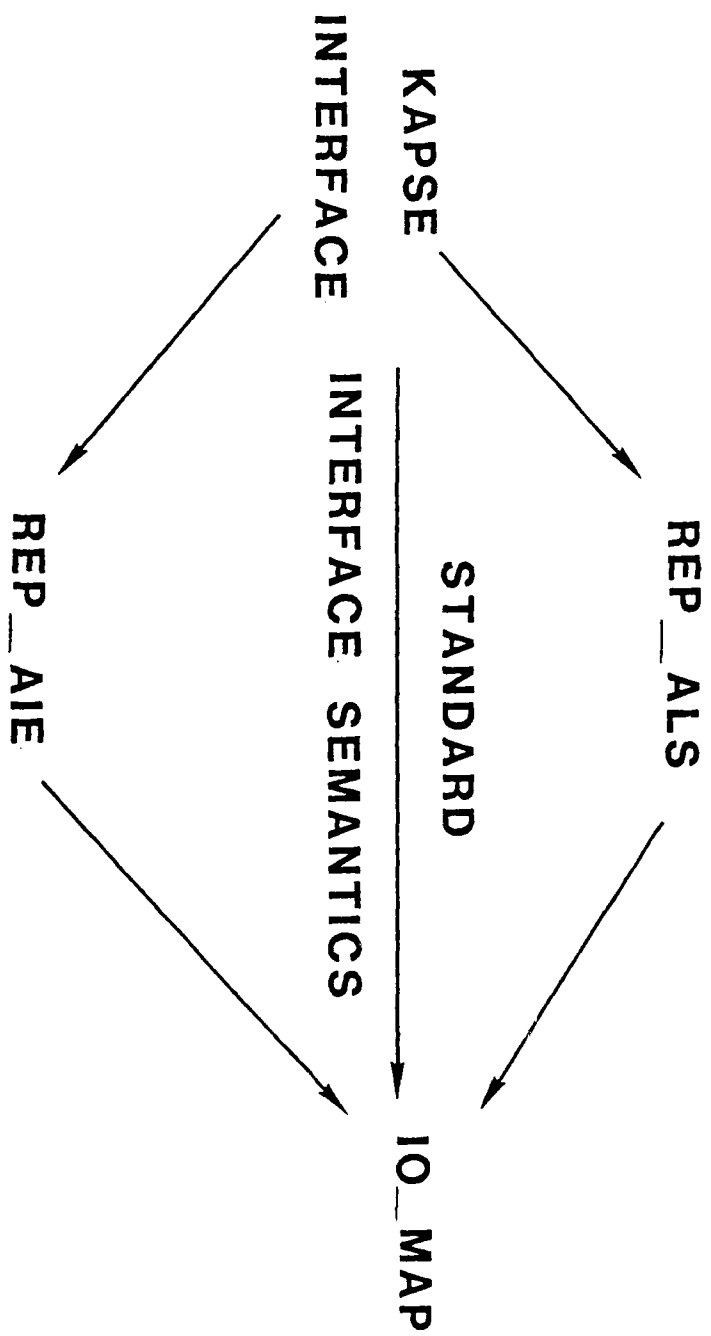


FIGURE 2

ALS AND AIE AS REPRESENTATIONS OF AN INTERFACE SEMANTICS.

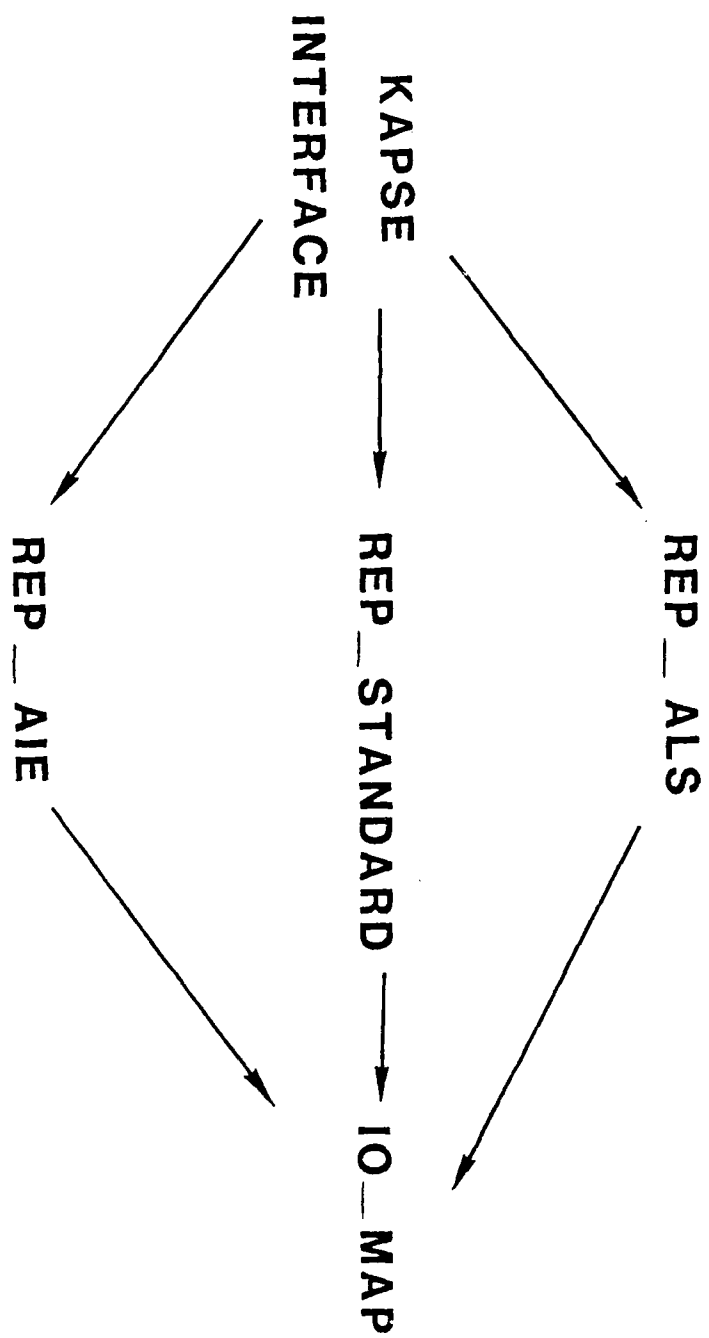


FIGURE 3

THE "SINGLE KAPSE" (REP_STANDARD) AS A REPRESENTATION OF AN INTER-FACE SEMANTICS.

A Machine Architecture for Ada*

Pekka Lahtinen
OY SOFTPLAN AB

Introduction

The NOKIA MPS 10 is a new computer system developed by NOKIA Electronics since 1978. The MPS 10 machine architecture is strongly based upon the requirements of its programming language - Ada. This paper presents an overview of the key features of the MPS 10 architecture.

The most essential general features of the architecture are:

- true stack machine, no 'visible' registers
- word length 32 bits + a tag bit
- virtual memory both segmented and paged
- hardware supported working set mechanism
- microprogrammed instruction set, over 300 instructions
- 32-bit integer and 64-bit floating point data types.

Memory Organization

The MPS 10 architecture implements a single level addressing scheme. All the data in the virtual memory, whether located in the main memory or on the disc, are accessed consistently using virtual addresses (of 40 bits).

The memory is allocated in segments. A virtual address consists of a unique segment identifier and a word displacement within the segment. Hence a virtual address is unique within the whole system. In the memory a virtual address is represented as a two word capability pointer. A capability pointer is protected by means of a tag bit. The hardware permits two words to be used as a pointer only if the tag bits on both words are set on. A capability may also contain access control information (e.g. read only).

Segments are further divided into fixed length pages. The hardware together with the run time support routines perform the necessary page transfers between the main memory and the disc.

The working set of a task is the minimum set of pages required in the main memory to run the task efficiently. In the MPS 10 the working set model is approximated by a set of pages most recently used by the task concerned. The size of the working set is kept within specified bounds by a firmware routine activated at regular intervals by a hardware timer. On the other hand, the working set may be dynamically extended by the address calculation firmware when the task refers to new pages.

* Ada is a trademark of the United States Department of Defense
(Ada Joint Program Office)

Program Organization

The execution time representation of an Ada program consists of a set of segments. Static program segments remain existent thru the life time of a program. Dynamic program segments are allocated as needed for dynamic data areas of a program. Dynamic segments may vary in size during their life time.

All the code of a program is contained in a single code segment. The data segments contain the global data areas for the static packages of a program. (A package is said to be static if it is a library unit or if it is declared immediately within a static package.) The Capability List records all the static segments accessible to a program.

Two dynamic segments are allocated for each task of a program: a data stack for the local data and parameters and a task status stack for the control flow information. Space for large (dynamic) arrays and access type collections may also be allocated as separate segments. The task status stacks of a program are linked to each other to form a 'cactus stack' (see also figure 2).

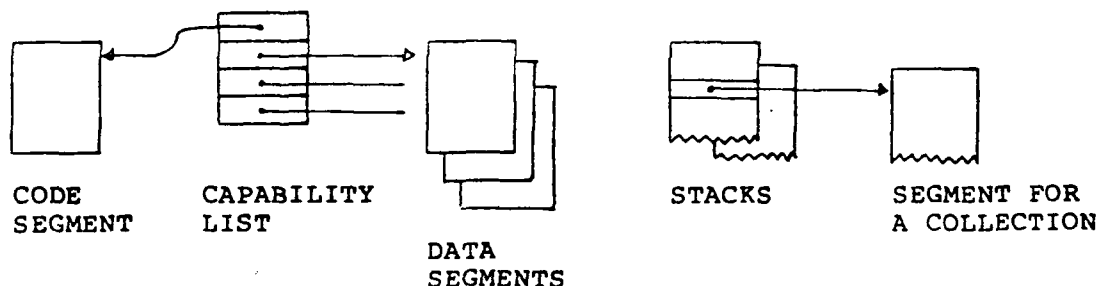


Figure 1: Program Segments.

Addressing Data

A program can access the data stacks and the segments recorded in its capability list. Indexing and indirection are available for addressing, if both are used indirection has effect first. Capabilities are used as indirect addresses.

Global variables on the data segments are generally accessed thru the capability list. An instruction gives a local segment number - i.e. an index to the capability list - and a displacement. The actual virtual address is computed by the micro program.

One of the data segments at a time can be referenced relatively to the Primary Base register. To achieve the best possible efficiency the register should point to the data segment most frequently referenced. Currently this segment is chosen by the compiler to be the data segment of the enclosing package, if any. The Primary Base can also be set explicitly by means of a pragma.

Local variables and parameters on the current stack frame of the data stack are accessed relatively to the Local Base register. The Local Base register is maintained by the subprogram and task control instructions, no explicit control is required.

Dynamic variables, e.g. objects designated by access values, are accessed by means of capabilities.

Variables of lexically enclosing tasks and subprograms are stored in outer stack frames possibly in 'outer' stacks. They are accessed by means of a lexical level number. As a part of the address calculation the firmware scans the task control stacks up to the level referenced. Although this implementation may be relatively inefficient for uplevel references from deep recursion, the extra cost of chain or display maintenance on every entry and exit point is avoided.

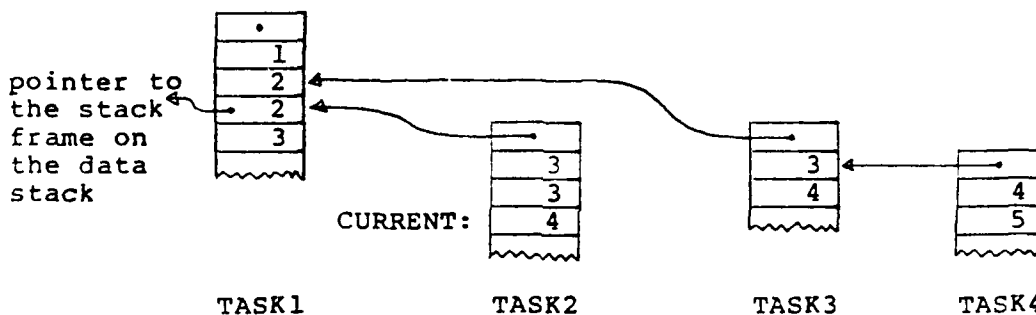


Figure 2: The cactus of task status stacks. Referencing level 2 from level 4 of Task 2.

The memory reference instructions of the MPS 10 generally have two functionally equivalent formats: 32-bit long and 16-bit short format. The short format can be used for the Local Base and the Primary Base relative references.

Addressing Code

Subprograms are referenced by their word displacement within the code segment. Interprogram subroutine calls are thus made possible; they are implemented by means of special capabilities in the Capability List.

Jumps are encoded as program counter relative. Both short jumps with an 8-bit offset and long jumps with a 16-bit offset are provided. Special jump instructions are provided for case statements, for loops and blocks.

Subprograms

The code for a subprogram body on the MPS 10 architecture is illustrated below:

```

-- subprogram header
-- elaboration code
  for the declarations
PELAB
-- code for the sequence
  of statements
-- exception handler
  part (optional)
PXIT n

```

The subprogram header records the number of words occupied by the parameters, the lexical level of the subprogram, the Primary Data Segment and the address of the exception handler part, if any.

The PELAB instruction informs the run time support system that the coming exceptions must be handled by the local handlers, no more by those of the caller.

The immediate operand of the subprogram exit instruction (PXIT) tells the number of words of the stack frame to be left on the stack for the caller (e.g. the function return value). Deallocation of the local dynamic segments (e.g. access type collections) is implemented in the PXIT instruction.

A subprogram call consists of three steps. First the actual parameters are evaluated (copy) or capabilities for them are loaded (reference) onto the stack. The PCAL instruction causes the actual execution of the subprogram. Finally the out parameters are stored back to the memory.

Tasking

Ada tasking on the MPS 10 is implemented by the micro program together with run time support routines of the operating system.

The MPS 10 Ada compiler generates a Task Type Descriptor for each task specification. A Task Type Descriptor records the following properties of a task:

- the address of a parameterless subprogram representing the task body
- the entries
- priority and (initial) storage size.

Each entry has a unique entry number within a task. For each entry the index range and the interrupt address, if any, are given. A unique task_id is assigned for a task object as a part of the execution time elaboration of the task object.

An accept statement is implemented as follows:

AD-A123 136

KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM: PUBLIC REPORT VOLUME II(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P A OBERNDORF 28 OCT 82

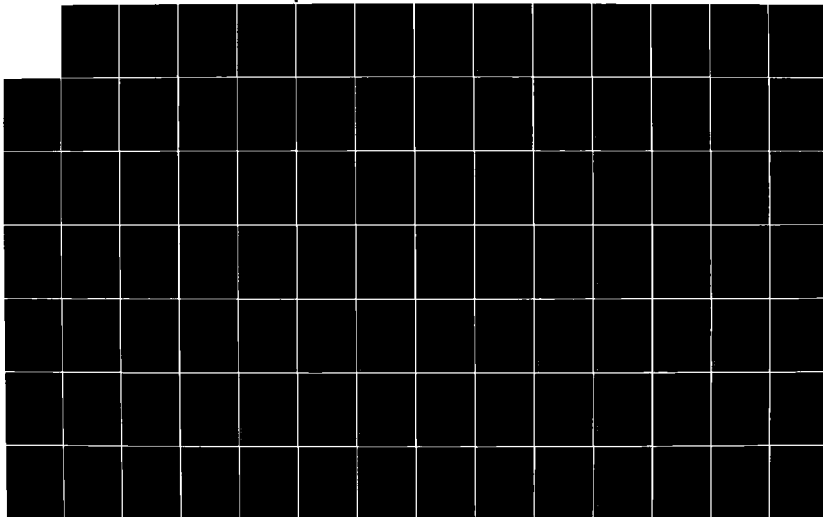
4/6

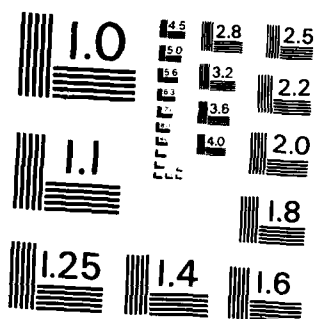
UNCLASSIFIED

NOSC/TD-882

F/G 9/2

NL





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

```

PUSZI      n      -- reserve space for the entry parameters
LDIQ       E      -- entry number
-- evaluate the entry index
-- (or load zero)
ACCEPT     body
BR         next
body:      -- subprogram representing
-- the accept body
next:

```

The corresponding entry call is implemented as follows:

```

-- evaluate entry parameters
-- onto the stack
LDIQ       E      -- entry number
-- evaluate the entry index
-- (or load zero)
LDCAP      T      -- task_id
ECAL
-- store out parameters

```

The ACCEPT instruction copies the entry parameters from the stack of the caller to the stack of the callee before entering the accept body. The reverse copying is done upon return.

The control flow information for subprograms and blocks - stacked onto the task status stack - record among other things the dependent tasks. The subprogram and block exit instructions implement the waiting for the termination of the dependent tasks.

In addition to those mentioned above the MPS 10 instruction set includes instructions for task declaration, activation and allocation. Select, delay and abort statements and the task attributes are implemented in the instruction set, too.

Exceptions

Unique identifiers are assigned at compilation time to all the predefined and user defined exceptions.

The exception `NUMERIC_ERROR` is generated by the hardware. Access values are implemented as capability pointers, the null value is detected by the hardware and the exception `CONSTRAINT_ERROR` is raised as required.

The following classes of instructions are provided for constraint checking; the exception `CONSTRAINT_ERROR` is raised by these instructions if the constraint is violated.

- check integer range
- check floating point range
- check and calculate index
- check discriminant.

The RAISE instruction is provided for explicit raising and propagation of exceptions. When an exception is raised the

firmware invokes a run time support routine to direct the control to the proper exception handler.

Concluding Remarks

Ada is the first language for the MPS 10, a subset - implemented in SOFTPLAN - has been available since October 1981. We believe on the efficiency and reliability of Ada on the MPS 10 : all the systems software - e.g. the operating system, the data base system and the Ada compiler - are written in Ada.

Validation in Ada* Programming Support Environments**

Donnis Kafara

J.A.M. Lee

Timothy Lindquist

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Thomas Probert

MITRE CORPORATION

Abstract

To this date validation has been applied in only two areas, in the validation of programs and the validation of compilers, and then not to any degree which can truly be classified as more than "empirical". This study was established to investigate the steps which would be needed to extend those previous experiences into the realm of programming environments and in particular the environments being proposed for use in the Ada program. A model of such environments already exists [] but is found to be lacking in essential detail necessary for an implementation to prescribe a model by which validation can be specified. This report does not itself provide any details of specific validation procedures or mechanisms, but rather investigates the processes for Ada Programming Support Environment (APSE) implementation in terms of the Ada Programming Language, and uses those specifications to suggest a mechanism for validation suite development.

Further in order to accomplish these goals it is suggested that the conceptual model of the "STONEHAM" document be extended to express the wider computing environments in which the APSE would reside. This extended model would also provide a fundamental basis for the design of Ada systems which respond to the need to provide networking, distributed processing and security enclaves.

* Ada is a registered Trademark of the Ada Joint Program Office of the U.S. Dept. of Defense.

** This research was supported by the Ada Joint Program Office, U.S. Department of Defense, through the Office of Naval Research under contract number N00014-81-K-0143 and work unit number SRO-101. The effort was supported by the Engineering Psychology Programs, Office of Naval Research under the Technical Direction of Dr. John J. Whare. Reproduction in whole or in part is permitted for any purpose of the United States Government.

Introduction

A fundamental objective of the Department of Defense (DoD) initiative to develop Ada is to increase the portability and maintainability of embedded software [XXX]. The Ada language will be the common high order language for use in future DoD embedded systems, and a Ada Validation Organization (AVO) has been established to ensure that Ada compilers implement the same common language. A major objective of the Ada Joint Program Office (AJPO) is to ensure that Ada remains as independent of computing systems and applications as possible, and has undertaken a standardization process to accomplish this objective. The Ada language is a Military Standards (MIL STD 1815) and has been proposed as both an American National (ANSI) and International (ISO) Standard. Requirements for a common (standardized) Ada Programming Support Environment (APSE) have been defined but the details have not yet been settled; however there is a growing realization that some form of the APSE or its kernel computing system interface (KAPSE) may eventually be standardized and conforming products be subjected to validation.

Although validation has usually been an afterthought in language design and implementation, this is not the case for Ada. At least in the case of the programming language the development of the validation suite of test programs has been accomplished hand-in-hand with the language standardization process and the ongoing implementation

[XXX] Carlson, W.E., Druffel, L.E., Fisher, D.A., and Whitaker, W.A., "Introducing Ada", Proc. 1980 ACM Ann. Conf., ACM, New York NY, 1980, 539 pp.

activities. This report surveys the problem of developing methods and techniques for the validation of APSEs based on the preliminary design requirements available at this time. To date, validation activities within the U.S. Federal Government have been restricted to implementations of programming languages according to Federal Procurement requirements and independent of the needs of the general industry. In some ways this has fulfilled this need and at the same time has had the side-effect of providing a measure of "quality assurance" to the non-government consumers. The same effect may hold true for Ada.

Initially the rationale for validation is the support of the goals of the overall Ada program (which is much larger than just a programming language) and is given additional impetus by the need to provide a mechanism for the protection of the Trademark which has been registered for the name. Thus unless it is intended to make the conformance requirements so lax that they can be enforced by "inspection", then a validation mechanism will be required for each element of the Ada program including not only language implementations but also support tools which are inherent to the program.

The Programming Environment Model

When considering the validation of a programming environment one must consider the underlying model which is used to construct such entities initially. While it is true that the model which is proposed here is

selected on the basis of its adequacy in a validation environment, this same model can readily be the basis of the development of the programming environments.

The model displayed by Buxton [] indicates a core-plus-ring structure which logically elucidates, from a functional point of view, the relationships between KAPSE, MAPSE and APSE systems but does not clearly delineate between functional and communications requirements. For example, the diagram of Figure 1.

Figure 1. The STONEMAN MODEL

infers the existence of interfaces between elements of the environment(s) but lacks the depth to indicate data flow requirements which may be superimposed. That is, the model is two dimensional when the problems to be solved are (at least) three dimensional.

An alternative model, though not one which was designed specifically for the design of programming environments is the Open Systems Interconnection Reference Model (OSI) [] developed by the International Organization for Standardization (ISO) Technical Committee 97, Subcommittee 16. This model is primarily concerned with

the modelling of networks and may contain facilities which are too detailed for a direct one-to-one mapping onto a programming environment. However, it is a feature of the model, to be able to combine "layers" in order to implement an actual system.

The working environments of Ada developed (and developing) systems may well include OSI-like environments when one considers the complexity that can be created by tasking, multi-processor and multi-targetable systems. Thus the use of this network model will provide a basis for further considerations of extended environments other than those of "simple" program development in a single family of architectures. A salient feature of Ada program/system development must be the portability of software (c.f. Steelman []) and a natural development from that requirement will be the development of such software in a network environment where both the development and target systems are accessible to the on-line developer.

It has already been hinted [] that there may exist some variations in KAPSE configurations due to the differences in hardware architectures which underlie them; the unfortunate side effect of such developments should not be that those same KAPSEs could not exist in a common (Open Systems) network. A start towards this goal would be the use of a common (standardized) development model.

A third advantage of the OSI Reference Model is the world-wide acceptance of this model for Open Systems Interconnection by network developers. The American National Standards Committee X3 -- Information Processing -- has already accepted this model as the basis for all future standards development work and requires that all proposals for both development projects and draft standards clearly

identify how the work fits into this model. Several mainframe manufacturers have announced their intention to build systems which conform to this model and some significantly sized users have expressed the desire to purchase systems which utilize this architecture.

The OSI Reference Model

*In the concept of OSI, a system is a set of one or more computers, associated software, peripherals, terminals, human operators, physical processes, information transfer means, etc., that forms an autonomous whole capable of performing information processing and/or information transfer. An application-process is an element within a system which performs the information processing for a particular application.

The Reference Model contains seven layers:

- a) the Application Layer (layer 7): This is the layer in which "real work" of the system is accomplished; the remainder of the layers provide the services by which this layer communicates with other application layers in a network. The application layer interfaces with the outside world.
- b) the Presentation Layer (layer 6): this layer provides for the representation of information that application-entities either communicate or refer to in their dialog. The Presentation Layer is concerned only with the syntactic view of the presentation image and of transferred data and not with its semantics, i.e. its meaning to the Application Layer.
- c) the Session Layer (layer 5): the management of (for example) a terminal session is the responsibility of the session layer. Such responsibilities include the necessary resource management for the support of the application.

* Extracted from DP 7498, ISO.

- d) the Transport Layer (layer 4): the transfer of data between session layers is the task of this layer. For example, if the application needed access to a data base, then it would be the responsibility of the transport layer to negotiate the data exchange between the session layers which support the specific application and the data base system (which is itself treated as an application-entity in the model).
- e) the Network Layer (layer 3): the connection between two nodes of the overall network must be managed by the third layer by providing the services of relaying data between end systems with network connections.
- f) the Data Link Layer (layer 2): the essential element of any network is the data link between them in order to exchange data-link-service-units which implement the network activities.
- g) the Physical Layer (layer 1). the physical layer provides mechanical, electrical, functional and procedural means to activate, maintain and deactivate physical connections between systems through the use of the physical media on which it is built.

These layers are illustrated in Figure 2. The highest layer is the Application Layer and it consists of the application-entities that cooperate in the OSI environment. The lower layers provide the services

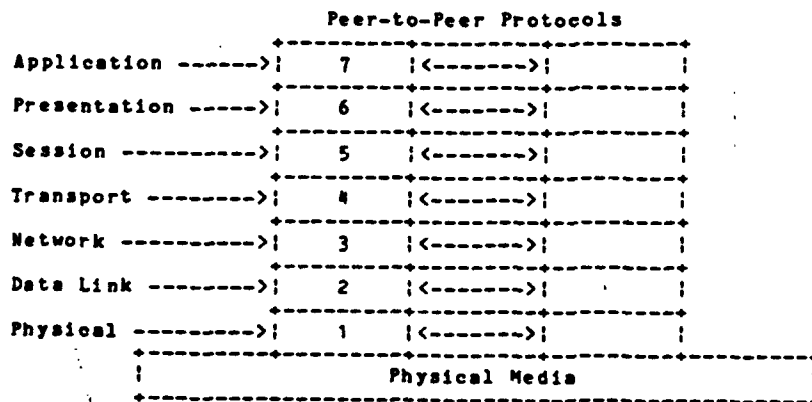


Figure 2. Seven Layer Reference Model

through which the application-entities cooperate. An essential element

of the OSI model is the clear definition of protocols and interfaces between layers of the model. The primary communication processes in the model are those that communicate with processes which exist in the same level layer as they themselves reside. This is peer-to-peer communication. That is, application layer processes communicate with other application layer processes in layer 7, networks communicate (in layer 4) with other networks and so forth. This manner of communication is a logical transmission which requires the use of peer to peer protocols to implement the data exchange. The logical communication process is physically accomplished through the (vertically stacked) interfaces between consecutive layers, and each layer provides services to the layer above it while receiving service from its lower layer.

The Application of the OSI Reference Model to an APSE

The original intent of the OSI Reference Model was not to actually represent an implementation strategy but instead to model those elements of a communications environment which need attention in the domain of networks and distributed systems. However such a model is clearly applicable to communications within both so-called "federated" systems and stand-alone environments. Moreover the facilities provided in the layers of the model have affinities with the tasks to be undertaken by other administrative aspects of networking and distributed processors. Thus rather than proposing to add other layers of complexity to the OSI

Reference Model it is proposed here that additional features be added to the layers in order to represent the non-communication facilities of these environments.

The OSI Reference Model can be implemented in a variety of manners, one of which would be to combine several layers into one. Thus the model can be applied to the design and implementation of programming support systems. Further it is not required that a layer be composed of only a single process (or processor); in fact it is clear that in many situations a layer will have to be composed of a collection of facilities which provide the support to the layer above, or receive data from lower layers. In an Ada Programming Support Environment, the majority of tools will exist as application layer processes and the facilities of the KAPSE will be contained in the session layer. The presentation layer will contain the mechanisms which provide the communication between the application (an Ada program or a tool) and the KAPSE facilities. This can be viewed as the argument to parameter mapping function and is specified through the specification parts of the packages which constitute the KAPSE facilities. This mapping of the APSE models onto the OSI model is shown in Figure 3.

In a "stand-alone" environment, layers 1-4 may be composed into a single module which represents the hardware system on which the Ada system is implemented. One other visualization would provide a subdivision of each layer either by the vertical stacking of sublayers or the division of each layer into a set of elements each of which interfaces with the lower layer. In this latter case, the tools which constitute the application layer are built individually on the presentation layer and have no other means of communication with each

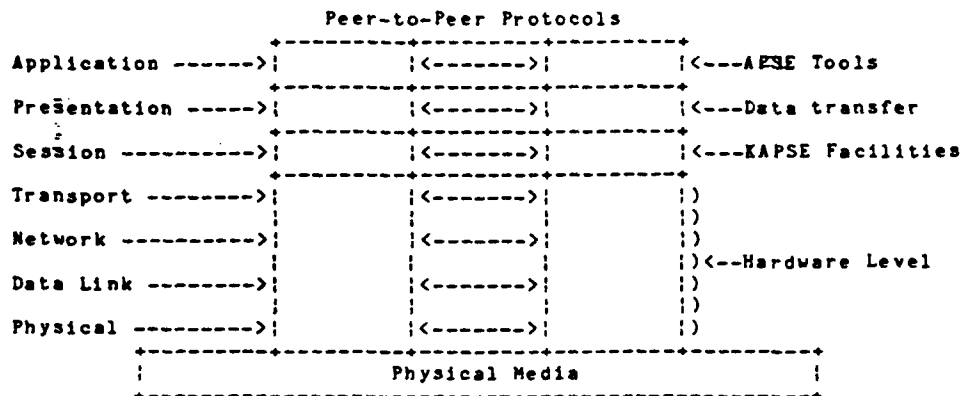


Figure 3. APSE Reference Model

other than through the KAPSE (session) layer. On the other hand, it is more than likely that such systems will be required to communicate in order to provide such cross-compilation, on-line debugging and similar inter-system activities. Thus it is imperative that a networking model such as the OSI Reference Model be an integral part of the APSE implementation system. This model also clearly indicates where there needs to be validation systems installed.

Validation Methods and Problems

The problems of validating a Programming Support Environment are akin to those of validating an operating system, a task which itself has not been attempted previously. Not the least of the problems to be

resolved is the validation of the collection of tools and facilities which make up the environment but also the communications between those elements when they are composed into the support system. Let us consider these elements separately.

Validation Elements

The OSI Reference Model presented in the previous section contains two basic elements. The first element is the set of objects each of which performs a specific, well-defined function. The internal mechanism which implements this function is concealed from view and the object may be readily interchanged with any other object, differently implemented, which performs the same function.

In the ADA context, we may conveniently identify two types of objects: facilities, either KAPSE facilities or other predefined library facilities (e.g., INPUT_OUTPUT), and tools. This distinction between facilities and tools is useful because they are designed with different goals in mind. Facilities are intended to provide generic capabilities out of which a large number of tools could be constructed. Conversely, the facilities are not concerned with any specific tool nor with the needs or characteristics of any eventual end-user, human or otherwise. By contrast, a tool is intended to provide a specific type of service to an end-user. The nature of this service is derived from the requirements of that user and from the operation of other tools

with which the service may interact. The distinction between facilities and tools is also useful from a validation point of view because, as will be seen later, the techniques used to validate facilities are different from those used to validate tools.

The second basic element in the OSI model is the mechanism by which objects interact. An interface is the means of immediate interaction between a tool and a facility or between two different facilities. Recall, also, that the model requires that the two object related via an interface be located on strictly adjacent levels. As an example, an interface may exist between an editor tool and a KAPSE facility. Also an interface may exist between a library facility and a KAPSE facility.

An interaction between two objects on the same level is achieved via a protocol. A protocol is the means by which an object assigns meaning to tokens which it sends to, or receives from, another object. An interaction between two tools, then, involves two protocols - one for each tool. Proper interaction between these two tools is only achieved, of course, when both tools use the same protocol. For example, an editor tool and a compiler tool typically interact through files which are passed between them. The editor creates a source text file for subsequent compilation. The compiler produces an annotated source listing, with possible diagnostic information, which may be viewed by a user through the editor. The two protocols involved in this example are manifested in the assumptions which each tool makes about the structure of these files. The following example illustrates how these two tools may fail to communicate properly because they are using different protocols. Suppose the editor tool assumes that the

first line of each file which it creates or edits contains the settings for various editor options (line length, search modes, user defined keys, terminal characteristics, etc.). On the other hand, suppose that the compiler assumes that all lines in a file contain compileable source text. In this situation, an otherwise valid source program file produced by the editor would, when presented to the compiler, result in error messages because the first line in the file would not be valid source text. Furthermore, these error messages could not be related to the user view of the source text because the first line as seen by the compiler is not visible to the user. In addition, the source listing file created by the compiler might not even be readable by the editor because the editor expects that the first line of text contains the option settings which will not be the case for files produced by the compiler. More subtle, but equally important, forms of protocols will be described later.

The distinction made between objects and the communication paths between objects provides natural points at which the validation effort may be directed. A complete validation may be structured into two phases. The first phase is the validation of the objects by themselves. Attention in this phase is limited to a validation of the functionality of each object, where the object is treated as an isolated entity. The second phase validates the interfaces and protocols through which the objects interact with each other. In this second phase, attention is directed away from the functionality of the object being validated and becomes focussed on the assumptions which the object makes in its interactions with other objects.

This division of the validation process into two disjoint phases appears to be both natural and technically feasible. There are, however, certain questions which must be further studied. For example: Is this division equally applicable to both APSE tools and KAPSE facilities? Is it always useful to separate the validation into these two phases? Is there a difference in the order in which the two phases are performed? These questions are meant to suggest the further inquiry that must be made into the question of validating tools and facilities.

Validation of Facilities and Tools

As indicated above, the first phase of the validation effort establishes to what degree an object, considered as an isolated entity, provides the function(s) expected of that object. This first phase of the validation process requires both a proper specification of the object's function(s) and a conformance mechanism. The specification, of course, must be a clear, complete, precise and unambiguous statement of the object's desired function against which the actual behavior of the object may be evaluated. The conformance mechanism is a, hopefully automated, process through which it is ascertained whether the object, in fact, displays all of the functions described in the object's specification. The conformance mechanism is typically a functional testing tool which is driven by a suite of test cases. These test

cases, in turn, are derived from an analysis of the specification. There is a large body of literature dealing with the techniques of functional testing [] and test case selection []. It is critical, however, to realize that these techniques cannot be applied rigorously and effectively in the absence of a properly stated specification.

It is more difficult to write a proper specification for Ada tools than it is for facilities. The difference lies in the fact that facilities are always expressed as packages and typically contain only a few functions with narrowly defined semantics. For example, common facilities might be "read a single character from a stream" or "create a new process". The use of packages themselves gives a concrete form to the specification of all syntax information for the facility. In a later section, the methods for specifying the semantics of KAPSE facilities will be considered. A tool, however, will typically exist as an Ada program (not a package) which is invoked through the command language mechanism. The tool will typically provide a broad range of individual, but interrelated, functions which may involve extensive user interaction (e.g. an editor or debugger tool) or may have input with complex syntax (e.g. a compiler tool or text formatting tool). Unlike facilities whose syntax of use is captured by the package specification, the syntax of tool use is not embodied in any Ada language construct.

The conformance mechanism is also more difficult to construct for tools than for facilities. Given a validation suite for a facility, a specialized conformance tool can be built which exercises the facility

as dictated by the validation suite and which compares the behavior required by the validation suite with the result produced by the facility. Discrepancies between the desired and actual function are detected and reported. The conformance mechanism for a tool is more complicated when the tool involves significant user interaction. As an example, consider a full-screen editor tool which uses the cursor position to determine a reference location for editing commands. Presumably the editor tool uses standard KAPSE facilities for terminal I/O. It may also be the case that the terminal facilities are distinct from the facilities for file I/O or interprocess communication. A part of the validation suite for the editor tool would involve, perhaps many, test cases which exercise the various editing functions at different cursor positions (e.g., a deletion/insertion at the beginning/end of the screen/line). One conformance mechanism is, of course, to use a human operator to perform these test cases. This approach is both expensive and error-prone. To create an automated process which could perform these tests would require not only a conformance tool to drive the delivery of the test cases and check the results of each test but, more significantly, would require some means of intercepting or simulating the terminal I/O. This may either be done by a separate computer system which is attached to the system under test through a terminal port or by the creation of a new terminal I/O facility which would allow the output produced by the editor and destined for the terminal to be diverted to the conformance tool itself. Similarly, the input expected by the editor tools would be taken from the output stream of the conformance tool and presented to the editor tool as if it were a terminal stream. Whether this new

facility could be easily constructed from the existing KAPSE facilities depends entirely on the way in which those facilities have been implemented and, possibly, on the underlying operating system.

Validation of Interfaces and Protocols

The second basic element in the Ada environment model is the means by which objects interact through interfaces, providing communication between objects at adjacent levels, or protocols, providing communication between objects at the same level. The validation of the interfaces or protocols relating a given pair of objects assumes that a validation suite is available which exercises the full range of possible communications. In the following paragraphs the validation requirements for interfaces and protocols are discussed in more detail and their relationship to the validation suite is described.

The critical interface to be validated in the Ada environment is the interface between a tool and one or more facilities. While there may be additional interfaces present within the tool itself, these interfaces are concealed from the view of the tool validator, which is the validation viewpoint taken here. The relevant question then is: "For a given tool how can it be determined whether that tool uses only standard KAPSE facilities and uses them only in a manner consistent with the definition of these facilities?". Dealing with this question is important for two reasons. First, focussing the validation on the

tool-facility interface assures that the validation suite is sufficiently robust to exercise the full range of the interaction. Without this focus it would not be difficult to construct an otherwise impressive validation suite which did not exercise some exceptional circumstances of the tool-facility interface (e.g. a memory fault). Second, even a thoroughly tested tool may have been developed and tested on a non-standard KAPSE or on a system which incorrectly simulated the KAPSE facilities. In this case the interface validation would reveal the level of conformance with standard KAPSE facilities.

The obvious process for validating the tool-facility interface is by exercising the tool portability. That is, the tool and its validation suite are removed from the development environment and transported to a validated, standard KAPSE. The tool validation suite is then applied to the tool. If the tool operates successfully in this standard environment then the interface is considered to be validated.

Two implications follow from this validation approach. First, it must be possible to know how extensively and in what ways the validation suite exercised the interface. This knowledge can only be obtained by instrumenting the KAPSE facilities themselves. This seems to imply that a specialized "validation KAPSE" should be developed which contains the required instrumentation. This answer, however, only reveals a further question: "What interpretation can be given to the measurements obtained by the instrumentation?" or conversely "What events should the instrumentation be recording?" The answer to this question involves the way in which the tool interface specifications are themselves stated. That is, the specification of the tool must

contain an explicit and complete description of exactly how the tool will use the KAPSE facilities. This specification should include at least the following items:

- (1) names of all KAPSE facilities used
- (2) name of each procedure, type and data item used in each facility
- (3) ranges of parameter values for all, IN, IN-OUT, and OUT parameters for each procedure
- (4) important sequences of facility usage
- (5) error exceptions anticipated

This list is only suggestive of the further thought that must be given to this issue. Only an agreed upon specification form can lead to the development of an instrumented validation KAPSE which, in turn, is necessary to perform the validation act itself.

The second implication which follows from the approach of interface validation by portability concerns the way in which the validation suite is applied to the tool being validated. Presumably, an automated driver was created in the development environment which performed this task. The role of this automated driver (conformance mechanism) was addressed above. However, to efficiently perform the validation, this driver itself must be transported to the validation KAPSE. Thus, not only must tools be portable, but their test drivers must be portable as well. It may well be made a concrete validation requirement that such a portable, automated driver must be supplied along with the tool to be validated and the validation suite. One difficult problem which arises, and perhaps reflects on how the KAPSE itself should be defined,

is that, as noted above, the test driver may need to use modified KAPSE facilities to simulate, or intercept, the tool's terminal I/O without modifying the tool itself. It is also unrealistic to achieve this goal by modifying the validation KAPSE. Perhaps, the original KAPSE design should provide a mechanism which allows a driver to capture the terminal I/O stream generated by a tool. In any event, some means must be found for resolving this problem.

The validation of protocols between Ada tools is also an area where additional work will be required. Many basic tool complexes possess protocols in one form or another. Typical tool pairs which interact through protocols are: compiler-linker, linker-debugger, librarian-configuration manager, text formatter-device manager, etc. The ways in which tools can be related through protocols are varied. Some of these ways are:

- (1) intermediate files
- (2) message stream communication
- (3) timing synchronization
- (4) resource contention

While there may be other ways in which the tool protocol is manifested, the important point is that the tool validation must take into consideration the interaction of the tool being validated with other tools. This wider viewpoint, which encompasses the validation of the protocol, is necessary because the tool is not an isolated entity which functions in a vacuum. Rather tools cooperating through protocols form an integrated entity whose combined function must also fall within the purview of the validation process.

As with interfaces, the validation of protocols between Ada tools is based on an accurate specification of the protocol itself. Two classes of protocols may be distinguished in terms of their difficulty of specification. Protocols in the first class are characterized by the use of a file to pass a complete body of information from one tool to the next (e.g., the file passed from the editor to the compiler). The communication established by this protocol is uni-directional and the tool which produces the file (the editor) completes before the tool which consumes the file (the compiler) is initiated. The specification of these protocols is no more difficult than detailing a file format and indicating the meaning of each part of the format. This can easily be done using some variation of a data structure diagram []. The validation of this protocol is also straightforward.

The second class of protocols are those in which there is a bi-directional communication between two simultaneously active tools (e.g., a debugger tool and an application tool). This protocol may achieve either the exchange of information bearing messages or the exchange of synchronization signals. This second class of protocols is more difficult to specify and to validate. There are at least four possible methods for the specification of these types of protocols. First, there are informal descriptions of the protocol with the usual ambiguity and imprecision which that entails. Second, formal representations, such as finite state machines or petri nets, can be used. In these representations each tool is considered to be in one of several states. The transitions between states of a tool are triggered by the transmission or the receipt of a message or signal. Using these methods it is possible to analytically determine important properties

of the protocol (e.g. deadlock). This method of specification seems to provide a more suitable basis from which the validation suite can be derived. A third possible method for specifying the protocol is by appeal to a predefined inter-tool protocol. For example, a standard debugger-application tool protocol may be defined in advance of the construction of either tool. This predefined protocol may be established as a package whose procedures implement the protocol. The validation amounts to certifying the correct use of these procedures in each tool. Fourth, a protocol "specification" language may be used which contains high level constructs for defining synchronization relationships. This language may be procedural (using, for example, monitors or a CSP-style type of primitive operation) or non-procedural (using techniques like path expressions to describe legitimate sequences of concurrent procedure invocations).

Regardless of the specification technique employed, the conformance mechanism must ultimately apply the validation suite derived from the specification to the tool(s) being validated. Given a tool, A, whose protocol with another tool, B, is to be validated, the conformance mechanism would replace tool B by a previously validated equivalent of tool B or by a stub tool designed to respond appropriately to the circumstances created by the validation suite. Tool A is then exercised as determined by the validation suite and its behavior is compared to that defined as correct by the validation suite.

Since there is a clear distinction between protocols and interfaces, the validation of protocols can be done in the development environment itself. That is, the protocols do not depend on their implementation

being achieved by the use of standard KAPSE functions. If, in fact, the protocols are implemented using non-standard facilities that fact will be revealed when the interface validation is attempted. Recall that the interface validation requires that the tool being validated must be moved outside of the development environment. This separation of validation between interface validation and protocol validation is one of the important benefits which is derived from the use of the reference model presented earlier.

Specification of KAPSE Facilities

It was noted above that the validation process is based on a precise and complete specification of the object being validated. In this section possible methods for specifying KAPSE facilities will be considered. One promising specification method, using an abstract machine, will be illustrated by example.

The specification for each KAPSE facility must include the following details:

- (1) syntax
- (2) semantics
- (3) limits
- (4) hidden protocols

Accurate specifications of this information is necessary in order to

design a robust validation suite for the facility. The meaning and role of each of these items in the validation process is explained below.

The syntax information is easy to specify since each KAPSE facility is defined as an ADA package. The package specification itself, containing operation names, type names, etc., serves to define the syntax of the KAPSE facility.

The semantic information is both the most difficult and the most important part of the specification because it defines the intended "meaning" or "function" of the KAPSE facility. The semantic information can be provided in four different ways. First, a natural language specification of the semantics merely describes in English prose the intended operation of the facility. While natural language specifications are common (see for example the ALS KAPSE package shown in the example in the next section), it is not possible to verify if the natural language specification is complete, consistent or unambiguous. Second, there are several formal methods for specifying semantic properties including axiomatic specifications, denotation semantics, or validation assertions. These methods employ a mathematical formalism to define the semantic information, which are precise and subject to mathematical analysis but they are also expensive to construct and difficult for typical programmers and system designers to understand. However, since the KAPSE facilities will be defined only once and implemented numerous times the precision of a formal specification may be worth its cost. The value of a formal specification can also be enhanced by including an informal, natural

language commentary which increases the understandability of the formal specifications. It should be remembered, however, that the specification is ultimately the formal description and not the additional commentary since the validation suite would be constructed in agreement with the formal specifications. A third method for specifying the KAPSE semantics is through an abstract machine. This abstract machine contains as primitive objects the elements appearing in the KAPSE environment (e.g., streams, flags, histories, processes, etc.). The specification of a KAPSE facility is given by writing a "program" for this abstract machine which, if executed, would perform the function intended for the KAPSE facility. That is, the semantics of the program are the semantics of the KAPSE facility. An example of this approach is given in the next section. By comparison with the other methods, the abstract machine approach is more formal and concise than the natural language specification, easier to comprehend than the formal methods, but is less precise than the formal methods because the semantics of the language used to write the "program" must now be defined. This later problem can be minimized by using ADA itself as the language in which to write the program for the abstract machine. The fourth, and final, method of semantic specification is by example. In this method the validation suite is constructed first and, by definition, anything which acts in accordance with that validation suite implements the intended semantics. In this method the semantic information is described implicitly.

The importance of the KAPSE semantics to the validation effort implies that several of the four specification methods described above may be used in conjunction to specify the KAPSE semantics. By using

several specification methods concurrently it is possible to satisfy the conflicting demands of precision and understandability of the specification. For example, a complete specification might contain either an abstract machine program or a formal mathematical description supplemented by a natural language commentary and illustrated by a validation suite. Disagreements among these three descriptions can be avoided by arranging each part of the commentary as an expansion of a single, specific part of the formal description rather than creating the natural language description separate from and differently organized than the formal specification. Similarly, each case in the validation suite should be tied to a specific part of the formal specification and commentary. In this way, the different levels of description are reinforcing and provide both high precision and ease of understanding.

The third part of the KAPSE specification must include details of any limits which are applicable to the facility being specified. Such limit information describes the sizes of objects (e.g. identifier strings, maximum file size, maximum number of entries, maximum number of processes, etc) and the number of times that operations may be repeated. The importance of this limit information to the validation process is illustrated by the following example. Suppose that an APSE tool is designed to create 10 subprocesses and to produce a file that is 10 megabytes in length. Without careful attention being paid to the limit specifications it is easily possible to implement this tool on a KAPSE supported by a large machine which uses only the functions provided by the KAPSE on that machine. However, when that tool is transported to a KAPSE supported by a much smaller machine, the tool

may be unable to operate because of the more limited environment of the second system. It is important to see in this example that the impediment to portability is not the function of the KAPSE facilities, but the limits which circumscribe the extent or repetition of that facility. Since the KAPSE may be implemented on machines with widely varying machine resources, it would not be unusual to consider validating a tool relative to a defined set of limits. While this does not make a tool any more or less portable, it does bring into clear view the degree of portability of that tool. Such conscious statement of the limit specifications may also serve to control the tool design if a highly portable tool is the design goal.

The fourth, and last, part the KAPSE specification is a description of the "hidden protocols". In a previous section the issue of validating the protocols between APSE tools was discussed. On the surface there does not appear to be a similar validation requirement for KAPSE facilities because each KAPSE facility appears as an independent function. However, beneath the level of the KAPSE interface many of the KAPSE facilities are, in fact, related through data objects in the implementation of the facilities. These underlying connections, or protocols, are "hidden" from the view of the validator because they are implementation dependent and cannot be directly observed - they can only be validated indirectly by observing the joint behavior of those KAPSE facilities interrelated by a given "hidden protocol".

For example, the procedure ANNOTATE in the ALS KAPSE contains the following specification:

"This procedure adds annotation to the derivation record. The text parameter contains the text to be added. ... When a file is closed and receives derivation information, the annotation is placed in the file's deriv_text attribute."

In this case the annotation which is supplied as a parameter to the ANNOTATE procedure is retained in some implementation object until the file being annotated is closed. At that time the text is retrieved from the implementation object and copied to the file's "deriv_text attribute". Notice that the existence of this implementation object is only implied by the specification. This implementation object embodies a "hidden protocol" since it allows a lateral flow of information between objects at the same level -- in this case between KAPSE facilities.

Since the validation process must assess the behavior of facilities related by such "hidden protocols", it is important that the KAPSE specifications make visible the relationships between KAPSE facilities. This visibility can be achieved by a simple mechanism. If there are N KAPSE facilities, an $[N \times N]$ table can be constructed where each entry in the table describes the protocol connecting a pair of KAPSE facilities. For example, using the ALS procedure described above, the corresponding entry in the hidden protocol table, named HPT, might be:

HPT[ANNOTATE,CLOSE] =

"annotation information supplied to the ANNOTATE procedure is added by CLOSE to the deriv_text attribute."

The validation suite would then contain a set of test cases which evaluates the correctness of the interactions identified by the KAPSE

hidden protocol table.

Example of KAPSE Specification

The following example is taken from the preliminary design specification of the ALS KAPSE. The comments which follow are intended to illustrate a method of specification, using a program for an abstract machine, which can be more precise and more comprehensible than a natural language specification. This example, is not meant to be critical of the design or specification developed by SOFTECH but reveals, we believe, common and fundamental problems in KAPSE specifications which all implementors face. The brief example concerns two procedures defined in the FILE_DERIV package: the procedure ANNOTATE and the procedure CITE_INPUT. The natural language specifications for these two procedures are as follows:

ANNOTATE:

procedure annotate

```
( annot_text : in string_util.var_string_rec;  
  result : out io_defs.io_result_enum;  
  result_string : in out string_util.var_string_rec  
);
```

This procedure adds annotation to the derivation record. The text parameter contains the text to be added. A null value for the text parameter is a special case, causing all current annotation to be cleared. When a file is closed and receives derivation information, the annotation is placed in the file's deriv_text attribute. The result parameter indicates the success or failure of the request.

CITE_INPUT:

procedure CITE_INPUT

```
( stream      : in io_defs.stream_id_prv;
  cite_flag   : in boolean;
  result      : out io_defs.io_results_enu;
  result_string : in out string_util.var_string_rec
);
```

This procedure sets or clears a flag indicating if an input file is to be cited in derivation histories. The stream parameter indicates the stream associated with the input file. The stream must be associated with an environment database file opened for in or inout access. The cite_flag parameter indicates if the flag is to be set or cleared. If it is true, the input file will be cited in subsequent derivations, if it is false the input file will not be cited. The result parameter indicates the success or failure of the request. The flag for any given input file may be set and reset as often as necessary. The value of the flag at the time a derivation is being written determines if the file is cited in that particular derivation or not. (Derivations are written only when output files are closed.) To be cited in a derivation means that the file will be referenced in the output file's derived from association. Input files that are not cited are referenced by the output file's other inputs association. In other words, if an input file is "cited" in a derivation the derivation information is "hard", i.e., the derivation count attribute of the input file is incremented when the derivation is written to the output file. Input files that are not "cited" in the derivation are still mentioned in a "soft" manner, i.e., the derivation count of the input file is not incremented.

Notice that in the above specifications, certain basic objects are referred to: derivation_records which may be added to or cleared, input/output streams which are of different types and have derivation and record flags which may each be set or cleared, and string used for returning result information. Since these objects are fundamental entities in the universe being defined by the KAPSE, these objects are explicitly defined in the program of the abstract machine. Thus, the following abstract program contains the declaration of three objects: derivation_record, string, and io_stream. The bodies of the two procedures ANNOTATE and CITE_INPUT are also given. Additional comments

on this abstract program specification are given after the abstract program.

Abstract Machine Specifications for the ALS package FILE_DERIV:

```
OBJECT derivation_record HAS
  clear, add : OPERATIONS;
  capacity : ATTRIBUTE;
end derivation_record;
```

```
OBJECT string HAS
  "=", "!=" : OPERATIONS;
  length : ATTRIBUTE;
end string;
```

OBJECT code is ENUMERATED TYPE;

```
procedure annotate(
  annot_text: in string;
  result : out code;
  result_string: in out string );
begin
  if annot_text = null
    then clear (derivation_record);
    elseif derivation_record.capacity < annot_text.length
    then begin
      result := "failure code";
      result_string := "insufficient space"
    end
    else begin
      add(annot_text, derivation_record);
      result := "normal code";
      result_string := "success::"
    end;
  end;
end;
end;
```

```
OBJECT io_stream HAS
  access_type, file_type : ATTRIBUTES;
  OBJECT derivation_flag HAS
    set, clear : OPERATIONS;
  end derivation_flag;
  OBJECT record_flag HAS
    set, clear : OPERATIONS;
  end record_flag;
end io_stream;
```

```

procedure CITE_INPUT(
  stream      : in io_stream;
  cite_flag   : in boolean;
  result      : out code;
  result_string: in out string);
begin
  if stream'file_type /= "environment database"
  then begin
    result := "failure code";
    result_string := "not environment database file";
  end;
  elsif stream'access_type /= in or
        stream'access_type /= inout
  then begin
    result := "failure code";
    result_string := "incorrect access type";
  end;
  else begin
    result := normal;
    result_string := "success";
    if cite_flag
    then set(stream.derivation_flag);
    else clear(stream.derivation_flag);
  end;
end;
endif;
end CITE_INPUT;

```

Several points are of interest to note in the above example. First, in the specification of the ANNOTATE procedure, the fact that insufficient space is one of the possible outcomes of a call on this procedure can only be determined in the original specification by examining the possible values of the enumeration type which defines the result parameter. However, in the abstract machine approach, this fact is clearly visible. Second, by comparing the natural language specification of the CITE_INPUT procedure with the abstract machine specification it appears that much of the specification given in the former case is misplaced because it has little to do with the operation of the CITE_INPUT procedure but it has more to do with the close operation and its effects. Third, the natural language specification

only implies the nature of the basic objects in the environment whereas the declaration in the abstract machine approach identify the attributes and operations for each such object. For example, to determine the total characteristics of an io_stream, it is necessary to read several parts of the specification of CITE_INPUT and other procedures while there is a single place in the abstract program which provides this information.

The abstract machine approach is certainly far from complete and there are many unanswered questions regarding its use. However, it is put forward in this report to indicate a possibly fruitful avenue of research which would permit the KAPSE facilities to be specified in a precise and comprehensible manner.

Validations of Run-Time Environments

The purpose of APSE's (and their subsets) is to provide a standardized (programmer portable) environment in which Ada programs can be developed, compiled, debugged and tested. In the latter situations it will be necessary to construct model run-time environments in which to perform these tasks since it is unlikely that the actual environments will be secure enough to permit on-line testing.

"Stoneman" suggests (section 2.B.11) that "the KAPSE is a virtual support environment for Ada programs". But it our contention that this should be more correctly stated as "the KAPSE is a virtual support environment for the development of Ada programs". In the model, and as

shown in Figure AA, the KAPSE is replaced by a run-time environment which contains the support facilities defined by the Ada LRM and the necessary pragmas. It is likely that there will be some overlap in the facilities provided in the run-time environments and the KAPSE, but it should not be overlooked that these environments have disjoint members. Thus there are two further validation activities which must be considered:

- (1) The validation of run-time environments as a separate activity from general compiler validation, and
- (2) the validation that the model or test environments not only conform to the general validation requirements (1) above, but that they also correctly "mirror" the run-time environments which they are intended to model.

This latter requirement is necessary since it can easily be anticipated that actual environments will contain facilities which, while they themselves are conforming programs, are peculiar to and modify the general run-time environments required by the Ada LRM. On the other hand, this requirement for validation is not one which should be imposed on a general validation organization, but instead should be part of the requirements contained in contractual agreements with software vendors.

At run-time, the reference model contains not the support environment elements but instead the actual Ada program at the application level, the necessary argument-parameter transfer facilities in the presentation layer, and the run-time support facilities in the session layer as shown in Figure 4.

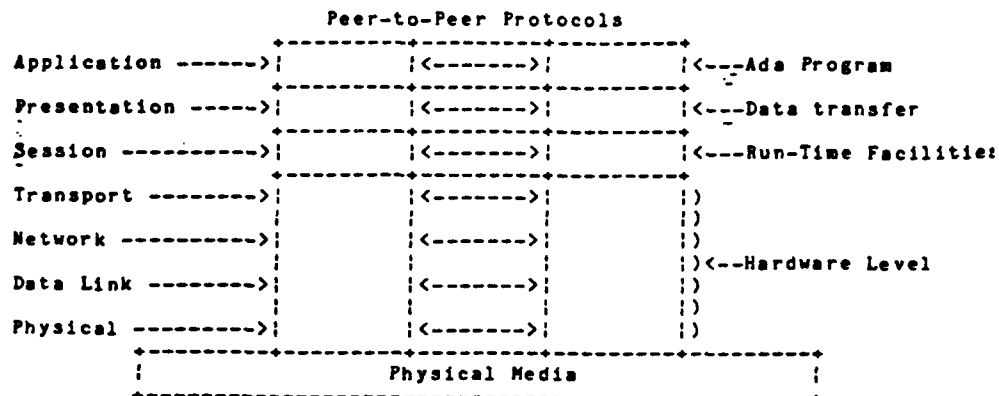


Figure 4. Run-Time Support Model

Validation of KAPSEs and Their Facilities

The validation of a programming support environment can primarily be considered in two stages: the validation of the KAPSE and its facilities and subsequently the validation of the tools which are contained in either a MAPSE or a "full" APSE. On the basis of the Reference model, the validation can be considered as the validation of each layer. Within a layer it is necessary to show that the processes contained therein fulfill the service requirements of the higher layer or in the case of the application layer (7), that the application requirements are met. Between layers it will be necessary to show that the interface requirements are met and from any layer communication with a peer layer is in conformance with the protocol requirements of the system. This latter situation will occur when (for example) a

debugging tool exists at the application level and is monitoring or debugging an application elsewhere. This situation is shown in Figure

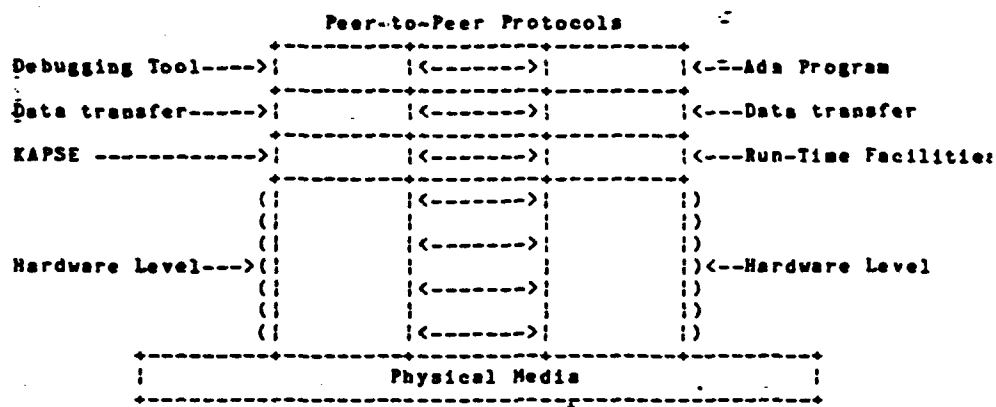


Figure 5. The Debugging Model

5. In this model it is assumed that the two systems are separate and that communication takes place through the physical media; in a cross-compilation and debugging environment this model would require extensive facilities to "make sense" of the debugging information which is exchanged between the two environments. Similar models, but all exhibiting the same layered characteristics can be constructed for cross-compilation, data-base access and multiprocessor systems. Thus it is clearly apparent that a generalized validation schema can be developed by the use of this reference model.

KAPSES AS ADA PACKAGES

The STONEMAN requirements [] specify that:

"5.E.1 The KAPSE shall implement interface definitions which shall be available to APSE tools. Such interface definitions shall be given in the form of package specifications in the Ada Language.

5.E.2 Interface definitions provided by the KAPSE shall encompass

(i) the primitive operations that the KAPSE makes available to APSE tools; these include any operations that may be necessary to supplement the facilities of package INPUT_OUTPUT ... in order to allow an APSE tool access to all the functional capabilities of the database,

(ii) the abstract data types (type declarations plus operations) that are required to interface the various stages of compilation; these include the data types that are produced by a compilation stage for later use by analysis, testing, or debugging tools."

This implies that the process of generating a support environment (whether at the level of a MAPSE or an APSE, but definitely above the level of a KAPSE) is to possess a "standard" library of KAPSE facilities which exist as a collection of package specifications and bodies which are used during the compilation of the tools. These tools are then added to the library and new tools added subsequently building on this foundation until the "whole" is tied together through some command language or access system.

This will require that the KAPSE facilities exist in a form which is accessible to the compilation system (which of course resides at the level of tools) as a library. However, since the facilities have to interact with the hardware system and may be distinctly implementation dependent, it is possible that the facility bodies will not be implementable in Ada itself! Thus it will not be possible to simply validate the facilities as Ada programs primarily; they will only be subject to functional validation. To accomplish both the accessibility

of the facilities to the compilers, the tools and the validating processors, it will be necessary to establish "internal standards" for the representation of the bodies and their interfaces with their own specifications. Interfaces between the specifications and the tools which use the KAPSE facilities must conform with the other standards for the Ada language level of interaction. That is, whatever the compiler generates (or is required to generate, depending on which is implemented first) to facilitate the interaction between USE statements and library packages will also be applied to the non-Ada generated facilities bodies. Such standards could well be defined for each implementation but it would be advantageous to specify such interfaces at some common language level such as through the intermediate language Diana, or at the very least through a PDL, and will be implemented as the specification part of the KAPSE packages.

One of the drawbacks to the current definition of the Ada language is the inability to place assertive statements which depend on values associated with parameters in package specifications. There is a modicum of this ability in the inclusion of constraints in type declarations and an inkling of this ability appears in discriminants in record specifications. An extension of this ability to include assertive statements connecting the input and output parameters of procedures and functions would provide an on-line validation facility which would greatly enhance the verification of conformance in KAPSE specifications.

THE USE OF LIBRARIES AND THE USE FEATURE

The concept of implementing a KAPSE as a library of packages gives rise to a facilities validation procedure which is the initial test suite in a process which ultimately encompasses the Minimal APSE set (MAPSE). That is, the specifications of the facilities (which actually prescribe the interfaces with the tools) should be defined in a standard KAPSE document in the programming language Ada, and the functionality of the facilities can be specified in a PDL. However it will not be possible to require that the specifications be validated in a character by character comparison system since:

- (1) there should be no requirement that the authors of a KAPSE maintain the exact same naming conventions for declarations except to maintain a homomorphic relationship between entities, and
- (2) the authors may need to add private parts to the specification in order to accommodate other unique procedures (functions and packages) which are not a part of the required facilities set.

On the other hand validation of the interfaces between a tool and the facility should be capable of being conducted as an integral part of the validation of the facility itself, particularly if the facility is designed (specified) to be responsive to changes in individual parameters and these responses can be readily ascertained. Thus a facility should be specified in conjunction with a test tool which would exercise its capabilities. Such a validation tool should be a part of the KAPSE description, and be a valid Ada program which can be simply compiled and executed as a stand-alone test of the facility. Overall testing and validation which would validate the interactions between facilities and the interdependence of facilities would be

separate*.

Communicatability Versus Portability

The validation of tools requires that tools be regarded as separate, individually defined entities, each with its own specifications document. For example, the compiler is a tool which has its own Language Reference Manual and its own validation suite. By comparison therefore it will be necessary to write specifications for the editor, debugger, etc. and each specification will have to include the necessary conformance statements by which a validation suite can be constructed and be judged relative to those conformance statements.

It would appear, from the Stoneman requirements, that it is necessary that tools be written in the Ada language (see sections 2.B.10 re portability, 2.B.11 which can be interpreted as saying that it refers to tools including those written in Ada, and 4.E.3 which asks that tools be "written in Ada") but it is not a requirement of current contracts that the fundamental tool, the compiler, is written in Ada. The only requirement should be that the tools have interfaces (with the KAPSE) which are "standardized". Thus tools act and perform in the same manner as Ada programs but are not required to be Ada conforming

* If facilities are defined as independent complete entities, the intricate interdependence of facilities can be minimized. If on the other hand the implementer desires to "extricate" common features of facilities, this interaction would not be apparent to a tool and thus not subject to validation.

programs in their source form.

By the model which has been proposed here, tools may communicate with other tools only through the KAPSE; this can be accomplished best through the provision of a "pass-through" facility in the KAPSE. However, indirect communication is possible through the data-base facility (see "Stoneman" section 2.B.4). This ability to communicate also provides an alternative to portability in an open systems environment. That is, while the "Stoneman" requirements specify the portability of tools (2.B.11), this can only be achieved at the source code level between differing architectures; however, the ability to communicate with tools which exist at other nodes of a network will effectively resolve this problem for non-Ada tools. This concept of tools which are implementation restricted through their non-use of Ada can be characterized by distinguishing between Stoneman-conforming tools which are Ada written and transportable, and APSE-conforming tools which are non-portable but accessible through a communications system. That is, portability can be replaced by communicability.

KAPSE Implementation Strategies -- Effect on Validation

Although programs in the Ada language may be portable, tools that are written using operating system facilities not directly included in the language would be portable only if those facilities are identically implemented on each machine. In an APSE, KAPSE facility specifications provide the interface between the underlying operating system and the

support environment tools. So as to gain the highest degree of portability of APSE tools, of Ada programs, and of APSE databases, the Department of Defense has promulgated as a distinct goal the evolution of a single KAPSE interface [] Despite the goal, two distinct KAPSE interfaces exist as part of APSE development efforts. The KAPSE of the Army supported Ada Language System (ALS) and of the Air Force supported Ada Integrated Environment (AIE) both have elements in common, but are not compatible for the purposes of transportability.

In [] Lyons details seven likely evolutionary paths given that different KAPSE interfaces currently exist and that some degree of transportability will continue to be needed as evidenced by the memorandum of agreement. Two of the most likely alternatives, the standardization of more than one existing KAPSE and the design and standardization of an entirely new KAPSE interface, are interesting when examined from the point of view of validation for transportability. Both alternatives have drawbacks and advantages over the other, but from the pragmatic level, satisfying the memorandum of agreement will require the design and standardization of a machine independent set of KAPSE facilities. These two alternatives are now discussed from the perspective of validating for transportability. Although the desired goal is one KAPSE interface, a third alternative, a mixed approach, is also discussed as a practical implementation strategy that allows for transportability.

N-Standard KAPSE's

(ref MOA and Stoneman).
(ref tim's options)

Momentum is already well established along the path toward N-standard KAPSE interfaces. The evolution of a small number, presumably one for each Armed Service, of accepted KAPSE's characterizes this path. Each of the KAPSE's is likely to undergo a defacto standardization process as multiple implementations are developed. To assure portability of tools within one of the standard KAPSE's, that is across implementations of the same KAPSE, the development of a validation suite for KAPSE services would be required. In light of the two currently existing and distinct KAPSE's, the next step in this path will be the proliferation of these systems by multiple KAPSE implementations. This is certainly anticipated of both the ALS and AIE through rehosting of their KAPSE facilities.

Supposing that a standard for each of the KAPSE's did evolve, validating the transportability of tools across implementations could proceed in one of two ways. The first approach parallels the big-bang approach to testing by submitting the entire set of rehosted KAPSE facilities to an extensive validation suite. Doing so would ascertain that any tool developed on the rehosted KAPSE could be transported to another APSE, and that any tool developed on another validated implementation of the KAPSE could be transported to the rehosted APSE. This gives the appearance that one validation effort for all the facilities provided by the KAPSE would assure the transportability of tools, but in reality such would not be the case. In almost every instance of a tool to be ported, there is another form of interface that must be validated in the new environment. Very few tools do not themselves communicate to other tools either directly through KAPSE facilities or through commonly accessible data bases. For example,

consider a text editor that is to be transported from one APSE to another. As mentioned above, intertool dependencies, or protocols, must be validated whenever tools that depend upon or use other tools are ported in isolation. That is, aside from a direct interface with KAPSE services through procedure calls, the editor has indirect interfaces, or protocols, with other tools accessing a text file created by the editor. It may be the case that the APSE's compiler expects that Ada source files to be in a special format somewhat different than other text files. Although the big-bang approach allows the functionality of all KAPSE services to be tested at once, there exists a need to validate the interaction that ported tools have with other tools.

The second approach is essentially divide-and-conquer. Each time that a tool is to be ported to a system, the specific facilities used by the tool and the intertool dependencies are validated. In most cases it would not be necessary to validate the entire KAPSE to port a single tool, but instead, validation of only those facilities used by the tool would be required. Although a particular implementation's KAPSE facilities may be validated several times, the advantage of divide-and-conquer is that test cases originally used to validate the functionality of the tools augment the validation of the specific KAPSE facilities.

In general, portability of tools across different implementations of one of the N-standard KAPSES poses no additional problems to validation than portability of tools given that there exists just one standard KAPSE. Although validating the functionality of the KAPSE facilities is relatively straightforward through validation suites, there is

certainly the additional overhead of standardizing N different KAPSE's and developing and maintaining N validation suites. The most damaging drawback, however, is that nothing can be done to provide for the portability of tools from one of the standard KAPSE's to another. Returning to the concept of one standard KAPSE for each Armed Service and realizing that several contractors may work for more than one Armed Service, they would be required to maintain and use different, possibly radically different, Ada environments. Additionally, many programs are identified with more than one Armed Service and the software produced on these programs must integrate into each involved Armed Service. To elucidate the options available for validating transportability across standard KAPSE's, suppose that a tool or group of tools are to be transported from one KAPSE, with KAPSE-A, to another with a different KAPSE, say KAPSE-B. Since it is very likely that the underlying KAPSE facilities being used are radically different, one must either simulate the original KAPSE or alter the tool to achieve mobility.

Simulation of KAPSE-A facilities using those provided by KAPSE-B would provide a relatively simple porting mechanism that would not require tool modification. The subset of KAPSE-A facilities that are used by the tool could be implemented using either the facilities of KAPSE-B or the host level operating system. Once the KAPSE-A facilities had been simulated they could be validated using the relevant portion of the suite for KAPSE-A. This is, however, a rather naive view of the requirements for moving tools from one standard KAPSE to another. In the most common situation, major sections of the target KAPSE (KAPSE-A in the above example) would have to be simulated for even the simplest of tools. For example, suppose that the tool to be moved

performed file manipulation. Because of the strong dependence of one file control primitive upon others, all file control services would have to be simulated. This strong dependence filters up to the tool level as well, and it is likely that just a single isolated tool could not be ported from one APSE to another unless that tool did not interact with any other tool on the system. A more obvious drawback of this approach is that it is generally quite inefficient to simulate one operating system's primitives in terms of those of another. It appears that the technique of simulation of one set of KAPSE facilities in terms of another set is reasonable only for transporting very simple tools without strict performance constraints, and the technique can take little advantage of validation efforts already expended on the involved kernels.

The other approach that can be used to move tools from one standard KAPSE to another is to alter the tool itself to accommodate the new set of KAPSE facilities. Unfortunately, the only way that validation can aid in the process is when a set of test cases has been established to validate the functionality of the tool. Such a set of test cases can be used, with minor modifications, to test the new version of the tool. A distinct disadvantage to changing the tool to transport it is that the effort required to effect the move is highly variable and dependent upon the original design and implementation of the tool. Tools whose KAPSE service calls are few in number and isolated to a specific section of code have an obvious advantage. Even though well written tools, independent of all else, are easier to transport, leaving such a large portion of the burden for transportability in the hands of the tool implementor is not advisable.

If the current momentum is not altered, more than one defacto standard KAPSE's may evolve. In this event, the relationship between the Ada language and its supporting set of operating system facilities would be little changed from the current relationship between high level languages and already existing operating systems. A major software engineering advancement of the Ada Program will be significantly diluted, and only pure Ada language programs will be transportable. Glaseman and Wrege have addressed this problem in a KITIA position paper [] in which they identify the problems arising from the current Department of Defense efforts and recommend a logistically sound path leading toward a resolution. Although they do not analyze how multiple standard KAPSE's affect the cost of transportability, their paper certainly supports our thesis that validating the KAPSE services to assure transportability can be practically applied only to moving tools between different implementations of the same KAPSE.

A Single Standard KAPSE

KAPSE validation for transportability between different implementations of the same KAPSE is no different for the scenario of N-standard KAPSES than it is for a single KAPSE. Approaches to this validation are now discussed in the context of a single standardized set of KAPSE facilities. Several alternative paths may eventually lead to one single KAPSE interface. One of the two existing KAPSE's could be selected to be the standard, or parts from each could be selected. From the point of view of pure transportability, the desired route would be to design an entirely new set of KAPSE facilities based on the Stoneman requirements, experience from the ALS and AIE implementations.

and foreseeable tool requirements. Such a design could be performed with ease of validation, machine independence, and rehostability on various architectures as primary design criteria.

As mentioned above, two different strategies could be used to validate that tools written using KAPSE facilities could be transported. Using the big-bang approach, a suite to test the functionality of all routines making up the KAPSE interface are developed and used to test new implementations for compliance to the standard. This form of test is presumably large and quite time consuming to use. The advantage, however, is that early in the re-implementation of a KAPSE several of the subtle errors are encountered and corrected alleviating possible data or program errors that might otherwise be promulgated through the environment's tools. The test suite itself could take the form of an Ada tool using KAPSE services. Thus, KAPSE facilities defined through Ada package specifications along with their machine dependent implementation in the form of package bodies could be tested by executing the validation tools. The other approach, divide-and-conquer, validates the all that is necessary when transporting. Each time that a tool or group of related tools are to be transported four different levels of validation must take place. The functionality of the rehosted KAPSE facilities must be validated, each tool must be checked to assure that it uses only standard KAPSE facilities, and shared protocols with different tools must be validated.

(1) Validating the necessary KAPSE facilities. All KAPSE facilities used by the tool to be transported must be considered. Assuming that a validation suite exists to test the functionality and interactions among KAPSE facilities, that suite must be employed to validate the

facilities used by a tool in the new environment. Certainly it is not necessary to revalidate facilities that have already been shown to conform. But, it is necessary to establish that the facilities used by the tool in its home environment have the same meaning as those in the new environment. To do so it may also be necessary to establish the conformance of the facilities in the home environment of the tool. Additionally, if a tool uses only a small number of a set of highly related facilities then it may be necessary to validate the entire set as they exist in the new environment. An example of this is the set of KAPSE facilities that manipulate the intermediate form of an Ada compilation unit. Included in the set are routines to build the intermediate form as well as those needed to access an already existing intermediate structure. If a tool to be transported uses only those facilities for building the form it may be necessary to validate the conformance of the entire set of facilities.

(2) Tool use of KAPSE facilities. It may also be necessary to assure that a tool uses only standard KAPSE facilities. Non-standard facilities may be hidden within the tool in various forms. One such form might be that a certain KAPSE routine have an additional parameter that is implementation specific. Although non-standard usage such as additional parameters or new routine names may be easily detected with static checks, a more difficult deviation would be the use of a non-standard parameter value to a standard parameter. The use of enumerated types for values of parameters to standard KAPSE facilities would ease this problem, but since some procedures have parameter values for which type enumeration would not be possible the problem would still exist.

(3) Tool to tool interactions (protocols). APSE tools that do not communicate with other tools either directly or through common data bases are rare. We call such communication intertool protocol, and consider it an important part of tool transportability. If a tool that communicates with other tools is moved in isolation, i.e., without moving the related tools, then the tool validation must include tests examining the tools communication in its new environment. Tools such as compilers, linkers, and debuggers are so highly interrelated that transportation of only a single tool of the group would not be practical. This is one instance where the protocol between the tools is so complex that the effort required to transport a single tool would exceed the advantage gained. Tools such as a general text editor, however, have a protocol with other tools simple enough to allow for isolated transportation of the tool.

In terms of the Open System's model of an APSE presented above, the validation of the KAPSE facilities and the tools use of those facilities is nothing more than validation that all levels below the application are the same, at least as seen by the tool, in the old and new environments. Validation of protocol as described above checks at the same level as the tool to assure that interactions at that level are the same in the new environment as in the old. From the standpoint of the cost required to validate for transportability the alternative KAPSE evolution leading to one standard KAPSE is most desirable independent of the means used to arrive at a single KAPSE. If the criteria includes, however, hostability on various architectures the most appropriate path to a standard KAPSE is a new design based on already existing systems and the needs of validation for

transportability.

A Mixed Implementation Strategy

The final implementation strategy discussed deviates slightly from those above by allowing the KAPSE supporting an individual APSE to contain both standard and nonstandard features. Although the mixed approach is not optimal from the standpoint of transportability, it does provide an alternative in which transportability can be achieved, and it permits flexibility within each installation. The approach centers around a minimal set of functionally complete KAPSE services which are machine independent. As a standardized set of facilities they would be implemented and validated on each installation of an APSE, and tools would be written based upon these facilities. The term minimal is used to describe the KAPSE indicating that a truly machine independent set of facilities may not be efficiently implementable on all systems upon which the KAPSE is to be hosted. Consequently, the mixed approach would attempt to define the smallest group of services necessary to implement APSE tools. Although the standard set might not allow tools to be developed requiring specialized facilities, through controlled mutation, installations could enhance or extend the services for the purposes of efficiency or added capability. Such enhancements or extensions would have to be controlled not by the criteria of functionality, but rather by the criteria of how they are implemented. The concern in using nonstandard facilities in the implementation of tools is that transportability is affected. But, by controlling the enhancements so that their use may be detected automatically by an analysis of the tool, the cost of transportability of a tool could at

least be determined, if not minimized. An installation's enhancements to the set of KAPSE facilities, in the form of new routines, would be relatively easy to discover. By automatically generating a list of all services used by a tool, either directly or indirectly, and by comparing the list with the standard KAPSE specifications, the set of nonstandard facilities could be isolated to specific sections of program text. On the other hand, however, the use of nonstandard facilities can be more difficult to detect. An example might be a tool that provides a runtime generated nonstandard parameter to an acceptable KAPSE service. Simple analysis of the procedure names invocable by a tool would not reveal this type of extension, and detection would instead require a runtime monitor of KAPSE services. A further question is how the implementation of the extensions relate to the implementation of the standard KAPSE facilities. Certainly the meaning of a standard facility could not be changed by an extension, although an extension could require that a standard service perform actions in addition to those standardized. For example, if a service called CREATE_PROCESS were part of the standardized set then it may be that an extension would require CREATE_PROCESS to store information about the process that augments what is required by the standard. The additional information may later be used by a nonstandard feature.

The mixed approach would only be applicable in the case that a standardized set of efficiently implementable services cannot be designed as the KAPSE. Such an approach admits that transportability can be achieved only by sacrificing either or both flexibility and efficiency. The advantage to this approach is that each tool may be designed and implemented according to its intended use. That is, if

the tool is to be usable on several different APSE's then it would be implemented in terms of the standard facilities only. If the tool were to resident only on one APSE then it could be implemented taking full advantage of underlying host peculiarities.

Of the alternative implementation strategies presented in this section, the evolution of a single set of standardized KAPSE facilities, to which strict adherence required, is the most desirable from the criteria of cost of validation and transportability. Currently however, not enough is known about the specific architecture of a machine independent set of facilities to determine whether a single standard KAPSE can evolve. The important question is whether a single set of services can be defined that are economically implementable on various existing systems and that are computationally efficient in terms of time and space requirements. If such a set can indeed be defined then the most desirable alternative is a single standard KAPSE whose facilities and whose overlying tools are validated using either the big-bang or divide-and-conquer approach. In the event that a reasonable KAPSE cannot be defined then careful attention and further work should be devoted to the mixed approach. Questions that need to be addressed should this alternative be adopted include: What levity can be taken in developing extensions and enhancements to the standard KAPSE? To what extent if any can the meanings of standard facilities be augmented? Is there a specific syntax that should be used in enhancements? What facilities should be included in the minimal standard KAPSE? Should they only be sufficient to implement the MAPSE level or should they also support projected APSE tools? Without careful guidance the mixed approach could easily lead to the

current situation in which vastly different APSE structures are supported.

Conclusions and Recommendations

AN APSE REFERENCE MODEL

This report has raised the issue of the need for a more substantial and extensible model for the definition of APSEs that was presented in the Stoneman requirements []. The model presented by Burton was intended as being merely illustrative and limited in scope to that report and not intended to be followed closely in constructing or implementing actual support environments. It is the recommended that the Open Systems Interconnection Model be accepted as the underlying model of APSEs and that implementations be required to clearly delineate the layers and restrict modules layers in which they are specified to reside.

Extensions to Include Networking Environments

It is clear to us that the Ada systems are likely to exist in environments which are more extensive than considered in previous reports. That is, the Ada systems, due to their reliance on multi-processing environments and cross-system development will be installed in networks which will require a more detailed consideration of inter-system communications than has been previously presented. Such environments will need to be developed specifically for the Ada systems and should be implemented so as to permit their clarity of interconnection with Ada Programming Support Environments and, by implication, with Ada run-time environments. It is recommended that there be developed a "Strawman" to extend Ada systems into a networking environment, based on the OSI Reference Model.

The Need for Security Considerations

It is clear that Ada environments will be required to contain security elements which will provide both access security and "physical" security. Considering the OSI model, a security sublayer could be introduced into the presentation layer which would insulate the Ada programs or the APSE tools from the KAPSE layer. Other elements may need to be added at lower layers to provide security between the software and hardware systems. By insisting that the only means of access to a system is through an application layer, full security can be ensured. It is recommended that the security aspects of the design of APSEs be investigated and that the results of this study be

incorporated into the Stoneman requirements.

The Need for A Single KAPSE Definition

In view of the potential for violating the general Ada program requirements for consistency and portability as, at least implied by Steelman and Stoneman requirements, by the continued development of separate KAPSEs by the Dept. of the Army and the Air Force, it is recommended that the policy that there shall be only one KAPSE definition and that by (say) 1986 all interim KAPSEs be required to conform to this single model.

Defining a Standard KAPSE Completely

Following the decision required above regarding the use of a single KAPSE model, it is recommended that work be initiated to define such a KAPSE in a form which would permit conforming implementations and ensure the adequacy of a validation procedure.

THE NEED TO DEFINE CONFORMANCE AND VALIDATION WITHIN APSE SPECIFICATIONS

The need to develop procedures and test suites to validate APSEs itself requires that some guidance be given to those responsible for the administration of those procedures. It is recommended that guidelines be established to ensure that the specifications for APSEs be accompanied by statements which specify the requirements for conformance and the conditions to be met to satisfy the validation requirements.

Continued Development of Formal Definition Techniques for Ada

Work is already under way, supported by AJPO, to develop a formal definition of the programming language Ada; it is recommended that this work be extended to consider the use of a semantic description method in connection with APSEs and specifically for the definition of conformance and validation requirements.

References

[] Buxton, J.N., Requirements for Ada Programming Support
Environments, "STONEHAM", U.S. Dept. of Defense, February 1980, pp.50.

INTRODUCTION

The Data Interfaces Working Group of KITIA (Working Group 2) has been spending most of its time on several key issues relating to KAPSE data interfaces. Working papers are included here which present some current thinking on a number of these topics. Ann Reedy and Judy Kerner have put together a brief overview of several of these topics, both in the context of short-term problems to be defined, and longer term issues to be addressed. Herman Fischer has written a time-line analysis of the KAPSE calls to the database that would be made during compilation of a simple program in the ALS and the AIE environments, pointing out discrepancies with the Stoneman KAPSE definitions. Erhard Ploedereder has contributed a suggested refinement of the Stoneman KAPSE concept based on definitions discussed and favorably received as possible alternatives at the Workshop on Intermediate Languages and Interfaces in APSEs held in Bernried, Germany, on July 5-9, 1982.

Further work is being done in each of the areas discussed, both in the United States and abroad, and we welcome comments on any of the relevant issues.

OVERVIEW OF CURRENT WORK OF THE DATABASE INTERFACE WORKING GROUP

Judith S. Kerner

UNITED TECHNOLOGIES NORDEN SYSTEMS

Ann E. Reedy

PLANNING RESEARCH CORPORATION

Working Group 2 of the KITIA has been concerned primarily with interfaces to the KAPSE database and how they affect transportability of programs and interoperability of data. There are a number of issues particularly affecting tool transportability which have been raised and are being considered by the group.

These issues are being looked at in several contexts. First, it is desirable (or perhaps necessary) in the near term to achieve transportability of at least some tools between the ALS and AIE APSEs now being developed. Second, for the long term, requirements, and eventually specifications, should be developed for a set of standard KAPSE interfaces. These should evolve if possible from the common set between the current environment designs, as experience is gained in the use of programming support environments.

A basic problem in transporting tools between the ALS and AIE is that the ways in which database objects may be related can vary widely. Thus, tools that depend heavily on the database structure may not be portable. A sophisticated configuration manager might fall into this category. Conversely, if a tool is restricted to using only simple database objects, portability may reasonably be achievable.

A number of questions have been identified already that will need addressing in future systems. One which seems very complex involves the "level" at which KAPSE calls are made. Specifically, in both the

ALS and AIE, a simple object is a file, and the internal structure of its data cannot be described to the KAPSE. Therefore, to the KAPSE, the contents of a file are treated identically whether they represent source code, abstract syntax tree nodes, program data, or anything else. Thus, in the current designs, to achieve any tool portability one must also port the package -- outside the KAPSE -- which translates the tool's structured view of the data to KAPSE calls and vice versa. Moreover, dependencies between compilation units are not all controlled within the KAPSE. This must be looked at for the future; perhaps more powerful functionality is needed in a database manager.

This situation, that the tool's view of the data is determined outside the KAPSE, raises another issue, that of data integrity. If a tool can read data via a package outside the KAPSE, and then write it out directly via the primitive KAPSE calls, the internal structure of the data objects cannot be guaranteed. (In the AIE, access controls may avoid this problem.) Similarly, many relationships between file objects are controlled by packages outside the KAPSE. In a future KAPSE, serious attention must be given to whether these data structuring packages must reside within the KAPSE, thus providing abstract data type KAPSE interfaces at the file level more completely than in the current designs, and also at the level where the data within a file is structured.

Thus, another concept to be considered for future KAPSE development is the flexibility of the database structure. Both the ALS and the AIE have a certain structure implicit in their databases. These structures reflect a particular view of the management of program development that may not be suitable for all applications or all organizations. This area requires a great deal of further study since automated configuration management support is still relatively new.

Inter-tool data interfaces is another area for consideration. Defining a set of requirements in this area is an issue that has barely been addressed in Working Group 2. Work on intermediate languages is being done in many places, and may lend some insight into the problems involved. In the short term, a reasonable course of action might be to simply port tools as a set (e.g. compiler, linker, loader, etc.). This is an area which will require a great deal of work in interface definition and standardization for a future development effort, if tools are to be ported independently.

Many other issues have been addressed in Working Group sessions which are not described here. As a clearer definition of each problem and its place in KAPSE interface standardization concerns is reached, further papers will be presented.

KITIA Data Base Group (#2)

Time Line Analysis of KAPSE Interfaces During a Compilation

Herman Fischer
LITTON DATA SYSTEMS

Summary

A Time Line Analysis was performed to determine the nature of, and order of, KAPSE interfaces to database services, by both the Softech (ALS) and Intermetrics (AIE) Compilers and their respective environments. The analysis was conducted by analyzing, at the respective companies, the compiling process for a example program.

The results were somewhat startling. Neither company defined the KAPSE database services interfaces to include either namespace operations, on the "program library", or abstract syntax operations, on the code. In both cases, the compilers were linked with program library namespace access tools, and with sophisticated abstract syntax access routines, which make simple "read/write" calls over their definition of the KAPSE interface. In neither company's case was either the namespace operations (which would be used by configuration management tools) or the abstract syntax access (which would be used by code manipulative tools) available as a "Stoneman"-defined KAPSE interface.

Example Program

The example program was picked to be simple, so that it would allow the

KITIA team to walk through the KAPSE interfaces in the order they are called during a compilation. The program is:

```
with TEXT_IO;

use TEXT_IO, INTEGER_IO;

procedure EXAMPLE is
  A,B: INTEGER;

begin

  GET (A);

  B := A;

  PUT (B);

end EXAMPLE;
```

The example program is very simple; it gets an integer, reassigns it, and prints it out.

This program has been compiled and successfully executed on TeleSoft Ada (tm). (There is some debate as to whether "INTEGER_IO" need appear in the with statement. It runs as shown. It is, however, now illegal to name things in the use clause which are not in the with clause. A solution with the TeleSoft compiler would be to "use TEXT_IO.INTEGER_IO" since in their case INTEGER_IO is part of TEXT_IO.)

Figure 1 provides an abstract syntax tree for the program. Figure 2 provides the Diana fragments for the program. Figures 1 and 2 are provided because the Stoneman definition of the KAPSE interfaces for the database are as quoted:

"5.E.2 Interface definitions provided by the KAPSE shall encompass

(i) the primitive operations that the KAPSE makes available to APSE tools; these include any operations that may be necessary to supplement the facilities of package INPUT_OUTPUT in order to allow an APSE tool access to all the functional capabilities of the database,

(ii) the abstract data types (type declarations plus operations) that are required to interface the various stages of compilation;

these include the data types that are produced by a compilation stage for later use by analysis, testing, or debugging tools."

Given these definitions, the team vainly expected to traverse the Diana code of figure 2 in examining SofTech's and Intermetrics' implementations.

Time Line Analyses

The time line analyses were conducted at Intermetrics and Softech by starting with the assumption that something called source code (as prepared by a human user via a text editor) resides on disk, and that the compiler has been "invoked" (whatever that means), by methods being studied by KITIA group #1.

Using the definition of the "compiler" and the KAPSE, as provided by each company, the time line analysis seeks to examine each activity, in sequence, which occurs on the interface connecting the KAPSE to the compiler.

Intermetrics Time Line Analysis

The Intermetrics time line appears in figure 3.

When the compiler is "invoked" it is passed, in parameters, the name of the library, the name of the source file to be compiled, and options. The program library, named via this parameter is a composite object. Within the library are separate objects for each of the compilation units that have already been compiled in the library, and additional information that represents the structure needed.

Rather than being a large simple object of which the KAPSE knows nothing about, the program library is treated as a composite object so as to take advantage of some of the KAPSE structuring services. The KAPSE provides an ability to build composite objects, and the composite objects allow a KAPSE interface to select their components by giving their names. The composite object may consist of other composite objects and/or of a collection of not necessarily similar simple objects.

Items in the library will include something for each compilation unit. A package spec for STANDARD is an example, of something which is "already there", but looks like a compilation unit. There will also be compilation units for all the library units, bodies, and subunits which have been previously successfully compiled. There is one object per "subdomain", a

simple object. That is to say, a program unit is stored as a simple (single) object rather than as (for example) a number of objects, such as one per Diana fragment. (A "subdomain" is defined by Intermetrics to be a simple object representing a single compilation unit.) The subdomain's simple object is managed, within the compiler, by a Virtual Memory Manager, VMM. The KAPSE has no access to the abstract syntax or Diana representation within a subdomain, other than as passing it in the form of a bunch of bytes to the compiler's VMM function.

There is another simple object for a compilation unit's abstract syntax tree, but as with the simple object collecting Diana fragments, the tree is not "pruned" into separately KAPSE-accessable leaves, but treated as a compiler accessible (via VMM function) collection of bytes.

Figure 4 shows the representation of a program library containing simple objects for EXAMPLE, STANDARD, and TEXT_IO. The simple objects containing Diana for STANDARD and TEXT_IO are preexisting and accessed in a read-only mode. There is a simple object created by the Compiler's internal VMM function during the first phase. This object is created by processing the source code object, to create an abstract syntax tree. The abstract syntax tree becomes a read-only to the second pass of the compiler, which uses that abstract syntax to create the resulting Diana simple object.

Once the compiler is started, it begins by opening up the program library. This is performed by giving the library's name to the library manager package (which is part of the compiler). (The name of the program library was a parameter passed to the compiler.) (At this time the library contains only the preexisting simple objects containing, respectively, Diana trees for STANDARD and TEXT_IO.)

A second parameter passed to the compiler, the name of the source text file.

The compiler, through its preamble, uses the interface to "get parameter", to retrieve the parameters to the compiler. The two parameters of concern to this discussion are the name of the source file (a database name) and the name of the library composite object (also a database name). The KAPSE services perform an open on the object. KAPSE composite object indexed I/O services are later used to access the object.

The next action is to open the source text file. The package, TEXT_IO, is used to read the characters of the source file. The entire source file text is parsed without worrying about any of the semantics of that source file. The program library manager function of the compiler creates, within the program library, a temporary object for each compilation unit encountered, within which it creates the abstract syntax tree for those compilation units encountered as the source file text is parsed. The VMM function of the

compiler creates and manages the nodes of the abstract syntax tree. The KAPSE is only aware of indexed direct I/O used to transfer the VMM-managed segments of the abstract syntax tree between processor memory and disk storage.

(The decision of whether to parse entirely all source text units into abstract syntax before compiling the first, or to compile each as encountered has not been made as of the time of this analysis.)

The compiler, upon finishing parsing the source text for one unit (or the entire source file, to be determined), then is ready to process the semantics of the program. The semantics phase creates a new simple object to receive the Diana fragments for the unit being semantically analyzed. The temporary object containing abstract syntax is input to this phase and the VMM function performs reads and writes on the newly created Diana simple object. Actually, the subdomain to receive the Diana starts as a complete copy of the abstract syntax, to which the Diana nodes are added by the VMM.

The temporary abstract syntax tree object is not deleted at the end of creating the Diana code, but is retained for ease of subsequent recompilation.

The objects in the library are not identified by the "name" of the unit being compiled. The distinguishing attribute is a compiler managed number, assigned to be unique among versions, etc. The unit name is retained as a nondistinguishing attribute.

The syntax of the Ada program (EXAMPLE, in this case) was felt by the Intermetrics designers to have little impact on the type of KAPSE calls occurring as a result of VMM's adding nodes to the abstract syntax tree (in the process of creating the Diana fragments). Basically, VMM adds nodes to existing tree segments by using pointers and access-like variables which appear only as bytes of data to the KAPSE interface.

A with statement, however, would cause the KAPSE interface to access the library object containing the Diana for the "withed" object. The VMM allows cross-object pointers, so that references from the generated Diana can point into nodes which are located in Diana trees of the previously existing objects.

The program library manager takes a program unit name, version number, and its own tables, and produces the distinguishing attribute number used to access Diana objects stored within the library composite object. For this reason, knowledge of how to locate objects within the program library is not available at the KAPSE interface, just as the ability to access nodes within Diana code is not available at the KAPSE interface. It is not possible to

create tools which operate on compilation unit name and version identification, or which operate on Diana nodes, using the KAPSE interface.

(The program library manager is expected to use the VMM to access the tables of unit name, version number, and distinguishing object attribute number. For this reason, the VMM will actually perform opens and read/writes on the table object prior to performing reads/writes on the Diana objects.)

(Package specs and package bodies, being separate compilation units, have their own Diana simple objects.)

The history attribute of each (Diana) object created during the compilation will refer back to the source text object that was read. The abstract syntax tree contains sufficient information to regenerate the source object, as well as referring back to that source object. Because the KAPSE knows which simple objects were read during the writing/update of a new simple object, it can record this information in the history attribute of the new/updated object. In addition, the program library's tables maintain sufficient information to know the dependencies between the units involved (in the program library's VMM-managed tables). Recompilation uses the program library's tables, and not the KAPSE-maintained history attribute.

In order to access these data in the program library's tables, a configuration management tool would (hypothetically) need to incorporate the same VMM and program library management function routines as used by the compiler. A configuration manager could not operate strictly by using KAPSE calls.

A listing is produced by a separate phase of the compiler which kicks in at the end, reads the abstract syntax tree or Diana (not known which yet), and reads the (rewound) source text object again. Errors were recorded in the Diana nodes.

Softech Time Line Analysis

The SofTech compiler time line appears in figure 5.

The compiler gets "invoked" using "call wait" or "call no wait". The first thing that it does is to get the parameter list, using the parameter list utility package (which is bound [linked] with the compiler, not a KAPSE function). The parameter list utility breaks the list down into a set of local strings which can then be obtained using the invoke strings. There are two positional and a number of named parameters. The first is the source text file name, and the second is the program library name.

Using the AUX_IO facilities, the compiler reads the values of attributes for the two files (source and library). The compiler internally checks the contents of the variable length strings returned by AUX_IO using its internal string utilities. (These checks use the KAPSE to assure that the files exist, that they have the proper access, and the compiler uses the attribute strings to verify that the file is either source or a library.)

Next the individually named arguments following the source and library file names in the compilers parameter list are processed by the compiler. (Examples are "options-list", output, etc.) Each option's name is verified and the value of the option is verified also, using the program string. If there is a reformat specified, this is deferred until later.

At this time, the BASIC_IO facilities are used to open the source file. The "stream id" of the source file is passed from the KAPSE back to the parser which passes it on to the lexer which then starts using BASIC_IO to read the source file. (The Stream id is an internal identifier which identifies the file control block to the KAPSE. It is the same as the "file handle" in the Intermetrics design.)

As part of the initialization, the control of the compiler calls the Program Library Manager to create a container. The Program Library Manager uses the KAPSE to create the physical file to receive the container. The Container Data Manager Initialize function is called by the compiler's control function, with the just created file, to set up the data structures for the Container Data Manager.

To process the source file, the lexer consists of a number of subroutines in the parser. When the parser needs a token, it invokes the lexer. The parser has a one-token look-ahead and the lexer has a two-character look-ahead.

Once the parser gets a reduction, it potentially creates a node which defines that reduction. In the EXAMPLE program, when it sees "TEXT_IO" it realizes that this is an identifier and it asks the Container Data Manager (CDM), a set of routines in the compiler, to create a node for that identifier.

The parser creates as its output the abstract syntax tree, for one compilation unit at a time. The entire compilation process is completed for the first compilation unit before the source file is read to start the compilation process for the second unit.

In the example program, when the parser sees the semicolon of the with clause, it realizes that it has processed an entire with clause. The lexer uses the Container Data Manager to create two kinds of nodes for the clause,

a token node and id nodes. The id node is strung off the with node, in the abstract syntax representation in the container.

The container contains Diana IDL; IDL has about 25 different modes which define the possible data structures which can be created by the Container Data Manager. (A mode is an encapsulation of a set of IDL, like a record definition.)

The particular implementation of containers in SofTech's present compiler is processor memory based. The entire container is held in virtual memory, managed by VMS, during the compilation process. As the parser goes from semicolon to semicolon, no KAPSE IO occurs for the container whose nodes are being generated. (One can, however, expect a small to considerable amount of VMS I/O for swapping virtual pages among available real memory. Such I/O is invisible to the KAPSE.)

The parser continues through the source file, creating the abstract syntax tree in a container, via the services of the Container Data Manager. The only KAPSE services called during this process (since the container is held in virtual processor memory in entirety) is the BASIC_IO to access the source text.

(In the near future, SofTech intends to go to a "paging" type system to manage the container's use of virtual memory.)

At the end of parsing the source file, there is a single unit's complete abstract syntax tree in a container, with the tokens representing it. (In SofTech's implementation of Diana, there is a lex source position node, which is a reference to the token which starts the construct, which provides source reproducibility, as needed for the symbolic debugger.)

The next phase of the compilation is context processing. (Between the parsing and context processing phases, maintenance options may cause the status of the container to be written out, via the KAPSE.)

Context processing next walks the abstract syntax tree. It is only concerned with the with and separate clauses. When it sees the example program's "with TEXT_IO" it calls the Program Library Manager (again, linked with the compiler) to translate that name into a program library name, which is an ALS file name. It then calls the Container Data Manager to open the TEXT_IO program library file, which in turn calls BASIC_IO via the KAPSE to bring the entire TEXT_IO file into virtual memory. (The entire abstract syntax plus tokens for each of the previously compiled containers thus also occupy virtual memory during compilation. Once read in, there is no further I/O activity during the compilation on that given container.) (In theory, main memory size under VAX VMS is limited to two gigabytes or disk swap memory,

whichever is smaller. SofTech is thus unconcerned about main memory requirements (virtual) during compilation.)

As part of the processing of the with-referenced containers, the compiler creates a symbol table access tree (hash table) for all of the context containers, because Diana only has the symbol table definitions, and not symbol table access. The symbol table access tree is created by calls to the Container Data Manager to create nodes, initialize them, and hook them up with the previously compiled container and also with the current compilation container.

Since the referenced previously compiled containers are accessed by the Program Library Manager, that program is responsible to assure that only the latest compiled container version is accessible. In order to utilize an earlier version of a referenced container, the user must manually set up a separate program library and use the "acquire" function to obtain his own personal copy of the non-latest version of the container.

The implementation of "history attributes" is handled by file derivations embodied by the KAPSE package "file_deriv". If one wishes to have a derivation, the user must set logging on from the command interpreter. Assuming that logging is on, for every file which is modified and subsequently closed by the compiler, there is a derivation associated with it. The derivation consists of two attributes and three associations. The attributes are derivation count (a count of the number of other files which have used this file in their derivation), and a derivation text (the name of the program which created the file, all of the parameters invoked of the initiation, and anything else which the program [compiler] wishes to place there, such as the name of the source file, withed objects, etc.) Extra information is posted by the ANNOTATE entry point in the file derivation package. Any file opened by the compiler and read from is recorded in the "used to derive" association. (Those files opened but which have nothing to do with the derivation can be quashed, by using the "cite input" entry point with the "cite flag" false. Quashed inputs are then recorded in the "other inputs" association, and do not increment the derivation count.) The third association is a list of backward pointers.

Context processing does not process the use clauses, its only purpose is to establish a visibility tree for past access.

Name processing is the next phase. This performs most of the traditional "declaration" processing. Name processing builds its own visibility tree for each scope, and it hooks it on up with the scopes of things which have been imported with with clauses. The tree is built through calls to the Container Data Manager to create the nodes, to modify the attributes within the nodes, and so forth.

Name processing starts setting the values of the semantic attributes by putting the values of the appropriate attributes in nodes, again by making calls to the Container Data Manager, with no KAPSE calls during this process.

Errors encountered are recorded in the container with an error number, the statement and token it occurred on, for the post-compiling display tool to print.

Unlike the Intermetrics approach, with one object for the abstract syntax tree and another for Diana storage, the SofTech approach expands the nodes of the abstract syntax tree to become, in the original container, the Diana. Effectively, the abstract syntax tree is not preserved as a separate entity, but the Diana tree is in the container when the compilation completes. The Diana is preserved subsequently also, even though the code generator later continues to modify and add information to the Diana, for separate compilations and for the symbolic debugger.

(The compiler, in the VAX VMS environment, is one huge virtual storage program. Rather than utilizing overlays to partition the compiler, the swapping facilities of VMS accomplish this de facto.)

The compiler's container is named by the name resolution function of the Program Library Manager.

Eventually, code generation produces object code. The object code is part of the container. When the code generation is completed, the control function comes back and Container Data Manager is called to close the container. This function writes the container, as a string of bytes, out to the file associated with the container at the start of compilation. BASIC_IO is used to write out the container. The container image file is then closed. (Only permanent parts of the container are written out. Temporary parts are discarded.)

The Program Library Manager is called to install the container. It takes the temporary file (containing permanent parts of the container) and moves it into the proper position in the Program Library, and at that point creates the super structure in which the container sits. There are a large number of BASIC_IOs and AUX_IOs performed in this process. The container is moved into the Program Library hierarchy relative to its name, rather than its dependencies.

References between dependent containers are expressed two ways. They are contained within the Diana, because its part of the context, and there are associations, which are the with and referenced by, held as text strings in

the nodes for the container files. (These associations are KAPSE controlled, but are independent of the Program Library Manager controlled associations. Program Library Manager associations are access controlled to prevent anything except the itself from accessing them.)

Subsequent to compiling, the display tool is invoked using program call. It produces the listings. The source remains open in case there are more compilation units in the source file.

The display tool opens STANDARD_OUT and MESSAGE_OUT, does its thing, terminates, and comes back to the compiler. The compiler, if it has end of file, quits. If not, compilation of the next unit proceeds.

The compiler control checks for voluntary termination before continuing with subsequent compilation units, after the display function.

The display tool, since it can be invoked separately of the compiler, does not use the container as was resident in virtual memory. Instead, it calls the Program Library Manager to obtain its own virtual memory copy of the program's container. The whole container is, as before, read into virtual memory when opened, in a single operation.

Conclusions

The purpose of performing the time line analyses was to learn about the KAPSE interfaces which might be used in the creation of interoperable and transportable tools. The two classes of tools considered as candidates for experimenting with interoperability and transportability between the Internetrics and SofTech environments were: (1) configuration management tools and (2) code manipulation (code analysis and debugging) tools. Configuration management tools require access to the "name space" describing object and unit names, versions, and linkages. Code manipulation tools require access to the code, basically in forms of Diana and/or abstract syntax.

Neither of the two efforts reviewed above provide the necessary interfaces for the above classes of tools, at the KAPSE level. In both cases, the respective companies have buried the true interfaces needed in routines bound or linked as part of the compiler, and used the KAPSE services to transfer byte strings with structures recognizable only to those internal compiler routines.

Thus, unless the KIT and KITIA can cause a reorganization of the two companies' internal compiler/library manager/abstract syntax tree and Diana

structure, the NOSC interoperability and transportability effort has little chance of success with the two environments examined.

The time line analysis, since it found that the necessary interfaces were not present in either effort's KAPSE, has thus failed to analyze the interoperability and transportability interfaces required by the above tools. It has, on the other hand, succeeded in a limited way, because it uncovered the discrepancy between compiler implementation and Stoneman requirements.

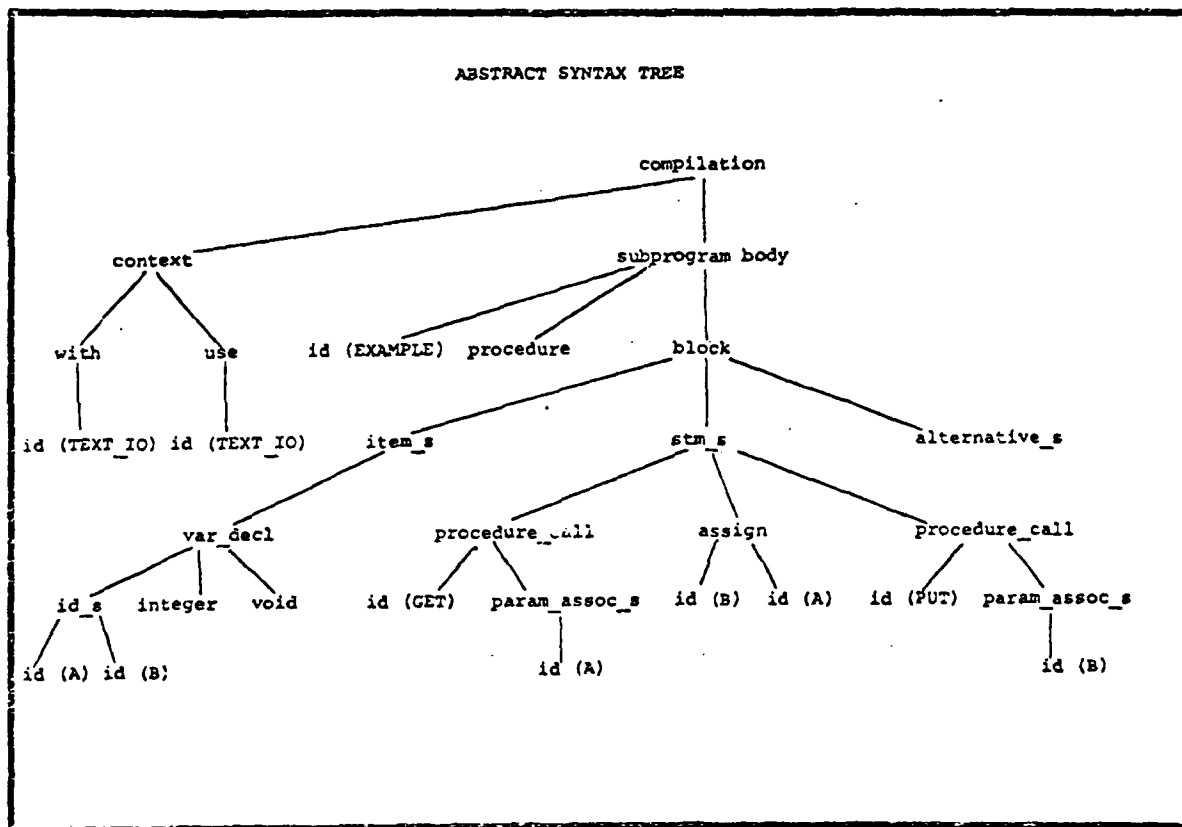


Figure 1. Abstract syntax tree for example program

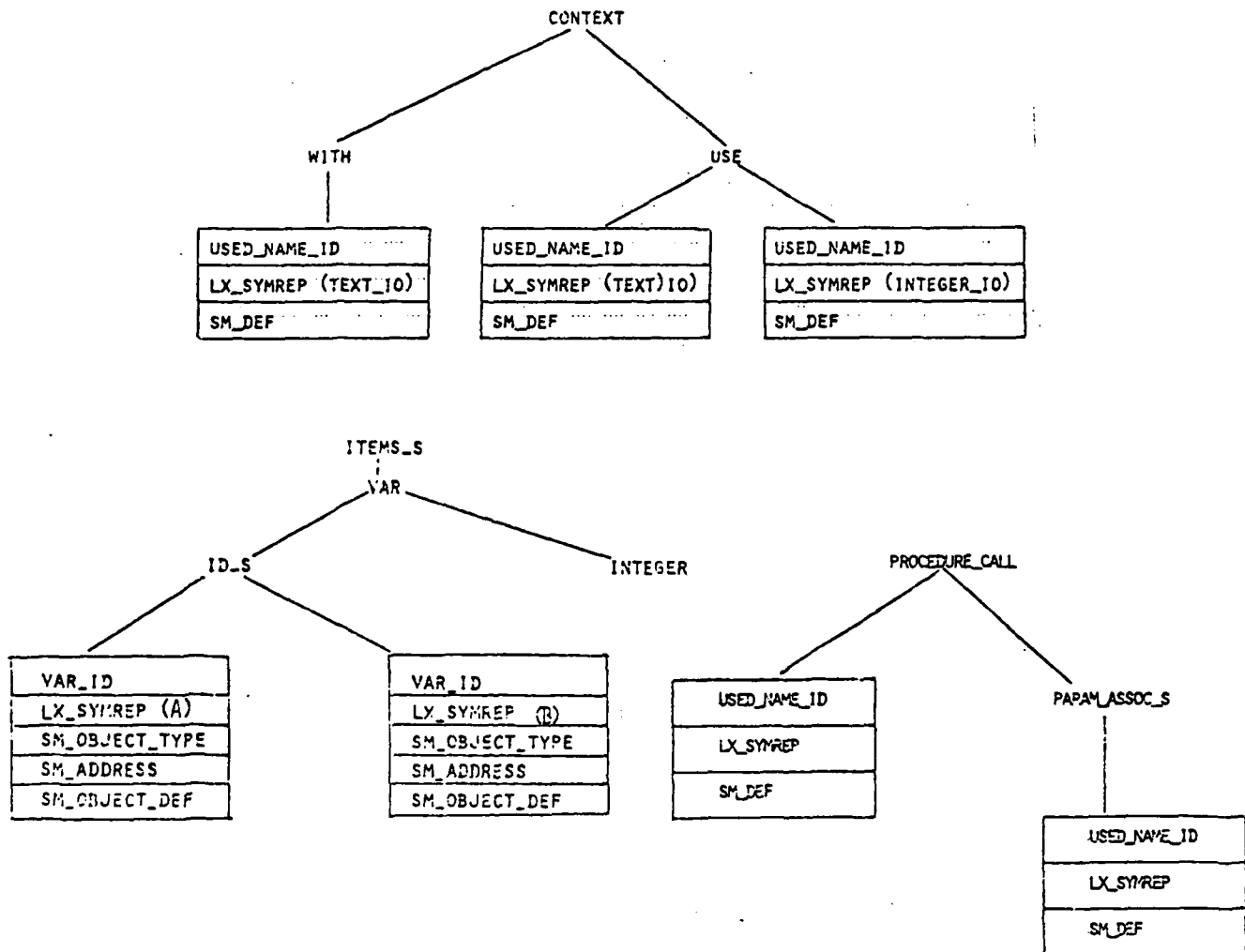


Figure 2. Diana fragments for example program.

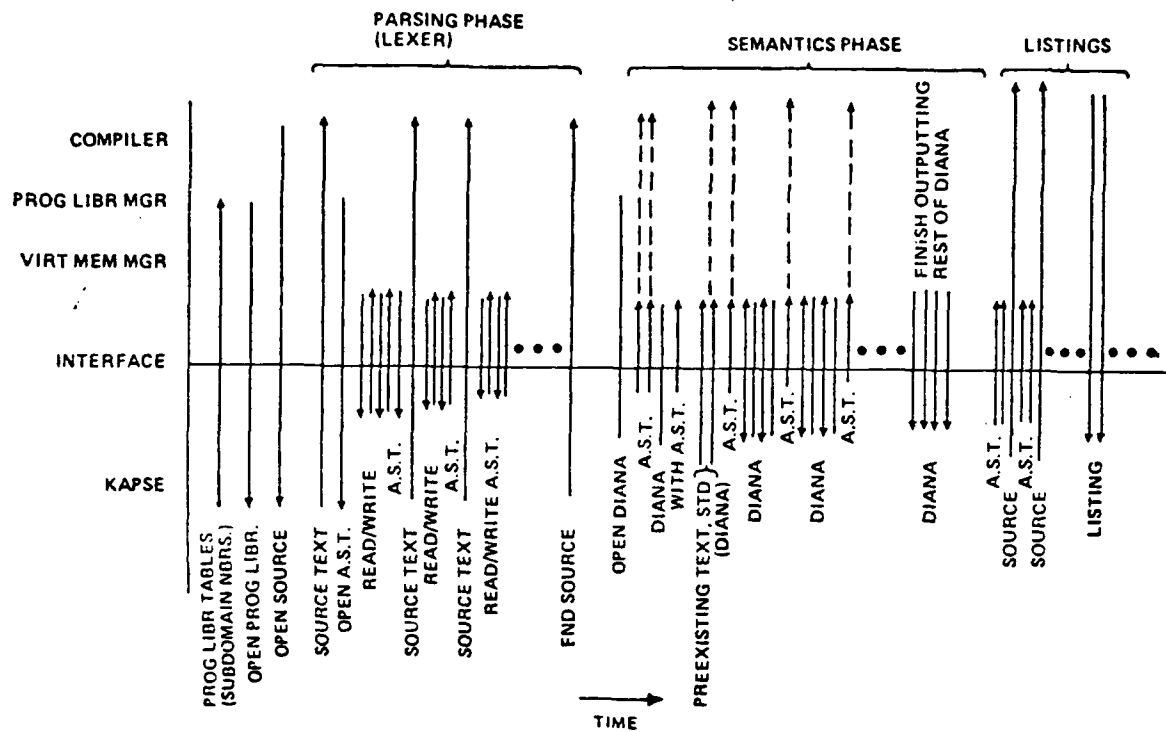


Figure 3. Intermetrics compiler. time line.

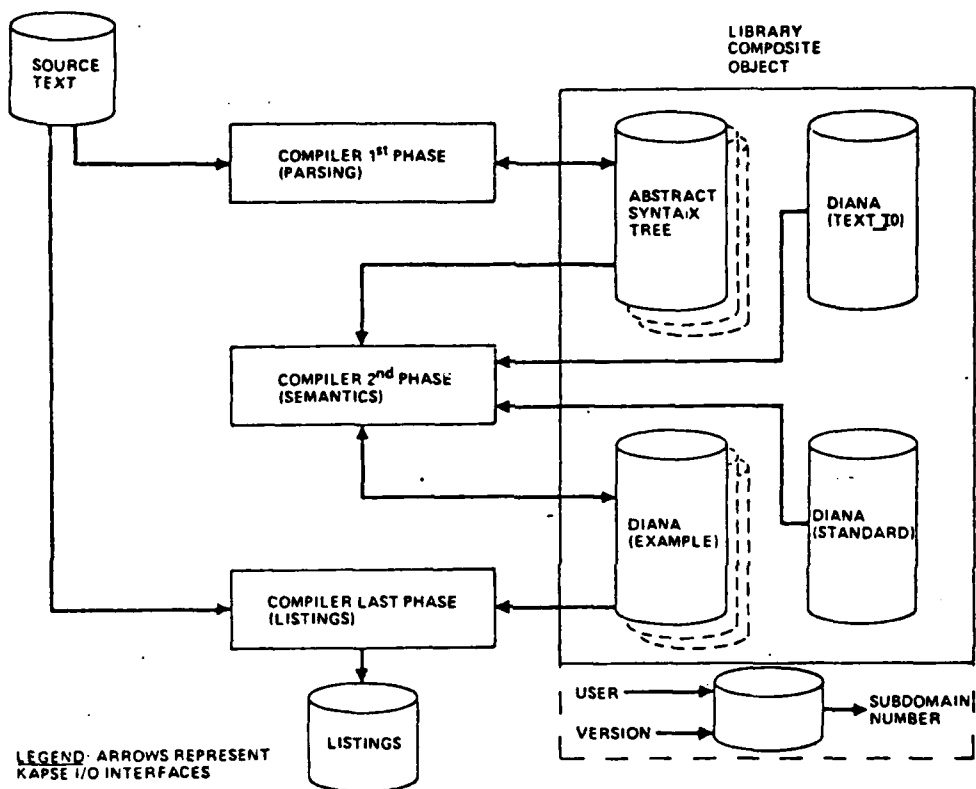


Figure 4. Intermetrics program library for example program.

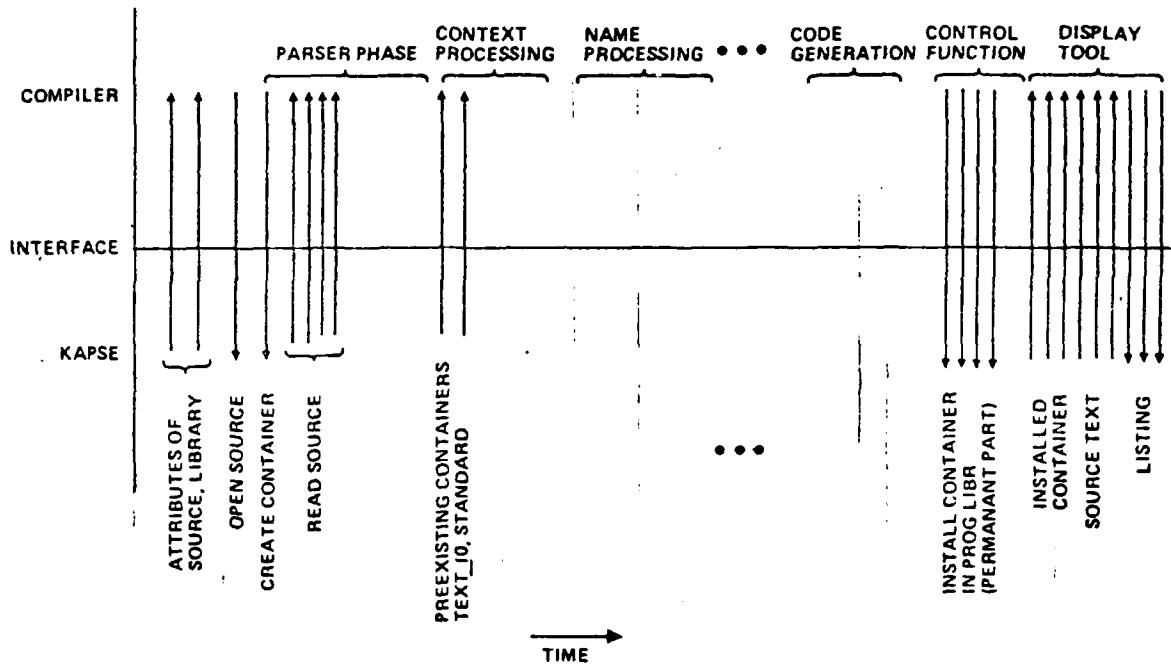


Figure 5. SofTech compiler time line.

THE NEED FOR A REVISION OF THE STONEMAN KAPSE CONCEPT

Erhad Ploedereder
IABG, DEPT. SZT

Introduction

Stoneman provided a framework for the construction of Ada Programming Support Environments (APSEs). One of the main objectives of this framework was to minimize the effort both of rehosting entire APSEs and of porting individual tools or toolsets from one APSE to another. For this purpose, STONEMAN introduced the concept of an inner layer in APSEs, the Kernel APSE (KAPSE). The KAPSE encapsulates all host dependencies thus providing a host-independent interface to all APSE tools. In addition, it incorporates the tool-to-tool interfaces. Under the premise that the KAPSEs on two different host systems provide identical, or at least semantically isomorphic interfaces, all tools programmed host-independently in Ada could be ported with minimal effort. Consequently, rehosting an entire APSE requires only the implementation of such a conforming KAPSE on the new host system.

In order to fulfill this premise, it would be desirable to standardize the KAPSE interfaces. This, however, is not possible at the present time, due to our insufficient understanding of the full scope of required KAPSE interfaces, in particular in the area of tool-to-tool interfaces. At best, we can attempt to standardize on subsets of the KAPSE interfaces and extend this standardized subset gradually as our understanding of the required interfaces increases and conventions can be agreed upon.

This paper attempts to delineate an approach compromising between our need for a standard KAPSE and our limited ability to foresee the various interfaces required by the many APSE tools yet to be specified.

In the following, we consciously avoid using the word "KAPSE" to avoid conflicts with preconceived notions of what a KAPSE should

encompass. Much of the diversity in these notions has been caused by applying the term KAPSE to that subset of the Kernel APSE whose standardization appears to be, even remotely, feasible today.

Host Encapsulation Interface

Host dependencies need to be encapsulated and hidden from all APSE tools by means of a host-independent interface. This interface provides services that are absolutely required for enabling the execution of APSE tools, i.e., tool invocation, execution and control, basic I/O, file and device handling, Ada run-time system, etc. This interface must be the prime target for standardization since, without it being standardized, the portability of tools would be severely restricted and endangered.

This standardized interface should, in principle, be sufficient to support model implementations of all APSE tools or toolsets that do not require interfaces with other APSE tools. (Model implementations provide the same functionality as production quality implementations but are not necessarily targeted to satisfy quantitative performance requirements.)

The standardized interface should be modest in scope and general enough to be implemented with reasonable effort on a variety of host machines.

Preliminary experiences with ALS and AIE show substantial *commonality* of the two systems in this area. One can therefore expect that preliminary standards at this elementary level could be established with reasonable effort.

Porting of tools and toolsets independent of tool-to-tool interfaces will be substantially facilitated among APSEs adhering to these conventions.

Tool-to-Tool Interfaces

Another important concept of STONEMAN was the granular structure of the APSE in the sense that, if a particular functionality was required in realizing a variety of tools, it should be provided by a common "subtool". As a consequence, an implementation of an APSE conforming with STONEMAN principles incorporates many tool-to-tool interfaces. In order to make tools dependent on such interfaces portable among APSEs, conventions are needed for the nature of these interfaces.

Unfortunately our understanding of APSE tools and their interactions is still very limited. With some notable exceptions, e.g. the intermediate representation of programs in DIANA after semantic analysis, the tool-to-tool interfaces are not sufficiently explored, and in many cases not even recognized, to allow us to establish conventions. At best, we can monitor the evolution of APSEs and attempt to reach agreements, as interfaces are recognized and practical experience has been gained with the approaches

taken. This will be a slow and painful process as individual efforts may have to backtrack since a divergence of interfaces among the typical, most needed tools in an APSE (which constitute the MAPSE) must be avoided, if the benefits of the STONEMAN model are to be realized.

Tools will communicate with other tools mostly through storing data in, and retrieving it from, an APSE database. While there is little chance to reach early conventions on the syntax and semantics of these data for the reasons given above, it will be of paramount importance to establish conventions about the framework into which these data are embedded in the database, since fundamentally diverging methods of storing, relating and retrieving information will almost certainly destroy all hopes for compatible interfaces of tools developed on different APSEs. In other words, a "generic" framework of the database must be established, flexible enough to satisfy the semantic requirements of existing and future APSE tools, but nevertheless suitable for efficient implementations of the database. To satisfy the latter requirement, it can be envisaged that, as these semantic properties are better understood and conventions are established, they migrate into instantiated elements of the framework of the database. As an example, the database mechanism will have to support arbitrary relations among, and attributes of, database objects, whose semantics are dealt with by individual tools. As these relations and attributes are better understood and standardized, utilization of their semantics can migrate into the data base administration as well, potentially yielding more efficient implementations.

ALS and AIE appear to diverge substantially in their approach to providing the database capabilities (and neither one appears to be sufficiently flexible for accommodating the need of future tools).

Apart from the host-encapsulation, the area of database capabilities must be the most important short-range target for standardization, if our chances for exchanging granular tools among APSEs shall be preserved.

Conclusions for System Structure and Design Considerations

STONEMAN provided an idealized view of APSEs based on the premise of a fully standardized KAPSE including all tool-to-tool interfaces. This premise is not achievable today; it may only be achieved by approximation through evolution. Correspondingly, the STONEMAN model needs to be refined to account for such evolution.

Maintaining the principal purpose of the KAPSE as that part of an APSE whose rehosting rehosts the entire APSE, we define the KAPSE interface to be the interface which encapsulates all host-dependencies and provides the "operating system" capabilities for enabling the execution of APSE tools. The standardized KAPSE interface must support model-implementations of all APSE-tools and toolsets that do not require interfaces provided by other tools.

We introduce the concept of a Common MAPSE (CMAPSE) which comprises:

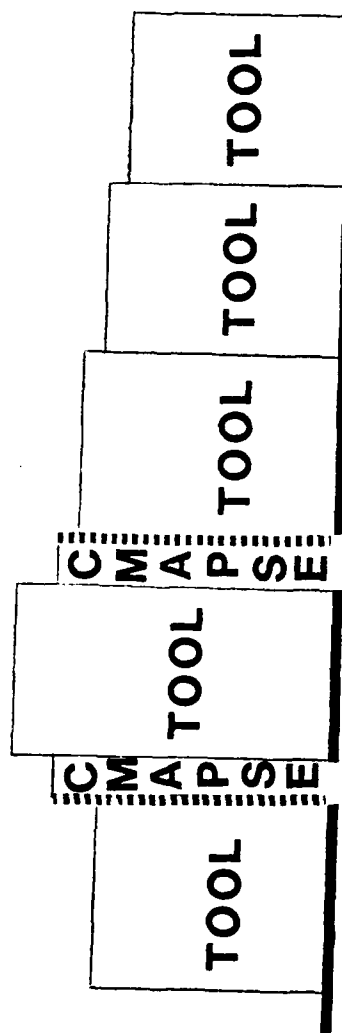
- a) the standardized KAPSE interfaces.
- b) tool-to-tool interfaces for which conventions have been established.
- c) tools with functionality and tool-to-tool interfaces for which such conventions have been established.

Hence the CMAPSE-interface can be regarded as the set of standardized interfaces in the APSE. (Care must be taken in the design and implementation of these interfaces, so that the combined usage of standard and non-standard KAPSE interfaces as well as CMAPSE interfaces cannot create inconsistencies in the system.)

Porting APSE tools whose interfaces belong to the CMAPSE will be substantially facilitated among APSEs with conforming CMAPSEs. Since the standardized KAPSE must support model implementations of the tools that create the CMAPSE interface, the CMAPSE can be rehosted by rehosting the standard KAPSE. It is, however, conceivable that the tools creating the CMAPSE interface are reimplemented host-dependently in a particular APSE installation, in order to gain efficiency. For the purpose of rehosting this APSE, such CMAPSE interfaces must be considered as having migrated into the KAPSE and need to be reimplemented on the new host system (unless, of course, their model implementation is ported.)

The CMAPSE will be the major vehicle for expanding toward a fully standardized set of tool-to-tool interfaces (i.e., the idealized STONEMAN KAPSE). The standardized KAPSE, on the other hand, provides a stable basis for a relatively inexpensive rehosting of APSE model implementations, which can be tuned, exploiting the properties of the new host system.

It is unclear at present which database capabilities, if any, should be incorporated in the standardized KAPSE. In the interest of a quick stabilization of a KAPSE standard it appears desirable to include at most very primitive capabilities and leave more elaborate interfaces to be defined in the CMAPSE.

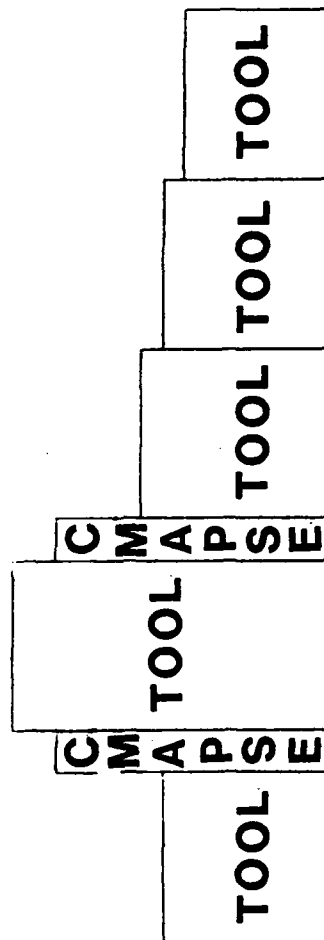


HOST ENCAPSULATION INTERFACE (STANDARD KAPSE)

HOST DEPENDENT SERVICES

FIGURE 3

INDIVIDUAL APSE INSTALLATIONS MAY CHOSE TO RE-IMPLEMENT (PARTS OF) THE CMAPSE HOST-DEPENDENTLY IN ORDER TO GAIN EFFICIENCY

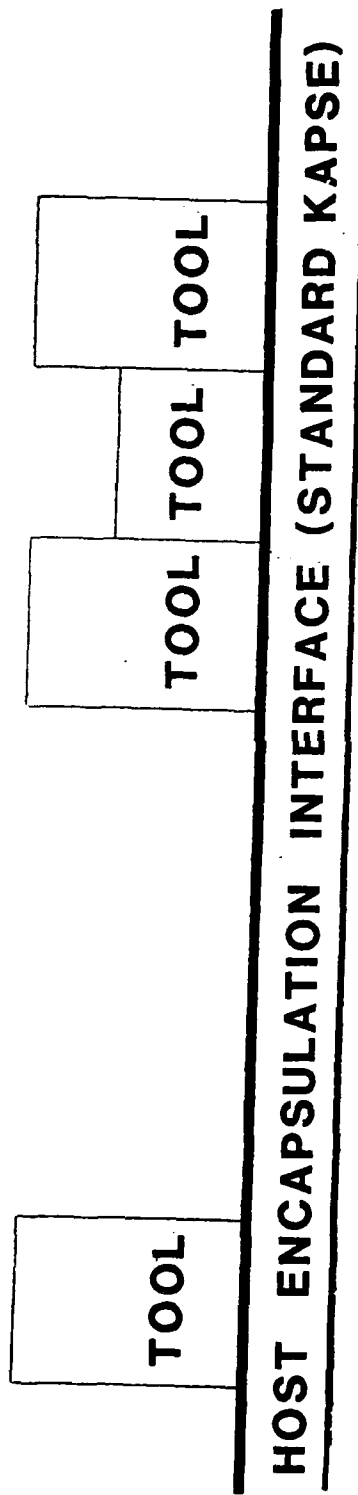


HOST ENCAPSULATION INTERFACE (STANDARD KAPSE)

HOST DEPENDENT SERVICES

FIGURE 2

TOOLS AND TOOLSETS WITH TOOL-TO-TOOL INTERFACES ARE PORTABLE
(AS MODEL-IMPLEMENTATIONS) ON THE BASIS OF THE CMAPSE AND STANDARD
KAPSE; THE CMAPSE CAN BE MODEL-IMPLEMENTED ON THE BASIS OF THE
STANDARD KAPSE



HOST-DEPENDENT SERVICES

FIGURE 1

TOOLS AND TOOLSETS WITHOUT TOOL-TO-TOOL INTERFACES ARE PORTABLE
(AS MODEL-IMPLEMENTATIONS) ON THE BASIS OF THE STANDARD KAPSE

KAPSE Semantics and the Layered KAPSE

D.E. Wrege

CONTROL DATA CORPORATION

Introduction

The STONEMAN was motivated by the recognition that an integrated approach was necessary for the entire software lifecycle. This was not a revolutionary concept, yet military embedded systems had historically lacked such support. Development contractors had neither the resources, time or money, nor the motivation to consider more than merely the development phase of a system. Further, from a competitive standpoint, they could seldom afford to implement even those tools necessary for the development phase of their project. When they did, the tools tended to address a rather narrow problem and tended to be of mediocre quality. Portability became a requirement of STONEMAN in an attempt to provide availability of Ada Environments to solve these problems.

The services are currently developing and adopting particular environments for many reasons. One is the desire to provide support for the portions of the software life-cycle that they are intimately involved in, most notably post-deployment support. Another is to clearly define the development cycle data that must be captured to support the rest of the life-cycle. Clearly, the DoD desires that software be developed using an Ada Programming Support Environment (APSE) that is to be used for the remainder of the software life-cycle. Availability of these APSEs is the key to obtaining this goal. The mere existence of an APSE will not satisfy the DoD's goals unless it is both used and widely available. The desired cost savings cannot be realized if the government must supply every contractor with a host upon which to run the APSE, for example.

But tools are required to be portable! With portable tools all of the above problems are solved! STONEMAN suggested a software architecture that should guarantee portable tools: the familiar KAPSE/MAPSE/APSE layered structure.

We must not delude ourselves into thinking that the KAPSE/MAPSE architecture itself will solve the problem of availability of Ada Environments. For example, if most of the MAPSE tools were to be placed within the KAPSE, the implementation of the KAPSE would be so difficult that re-hosting would be impossible even though tools would be portable. In the following sections, the need for developing the semantic definition of the KAPSE and an approach for obtaining this definition through a layered KAPSE model will be discussed.

KAPSE Semantics

Portability of APSE tools is achieved through a fixed KAPSE interface definition and by utilizing Ada as the tool implementation language. Programs written in Ada, utilizing only machine independent Ada features and relying only on the KAPSE for external support, should be portable. The issue being discussed here, however, is not portability but availability of APSEs: i.e. rehostability. There must be someplace to port these portable tools. To rehost an environment one must:

- o Implement the Ada language for the new host.
- o Build a linker for the new host.
- o Implement the KAPSE for the new host.
- o Port the tools.

With many languages prior to Ada, the first requirement has been difficult to achieve. The reason was that the semantics of the language have been ill defined. Consequently, the compiler defined the language rather than vice versa. Thus when a compiler is implemented, it may have subtle differences between it and the semantics of the language within which the "portable" tools were written. Fortunately, Ada does not suffer from these problems as the Language Reference Manual and the Formal Definition serve as defining documents. One does not have to empirically determine the characteristics of the language by "running some programs." Indeed, since there are requirements for retargeting, the current design of the compilers is allowing for a portable front end and semi-automated techniques for code-generator implementation. In short, considerable attention is being paid to this aspect of the problem.

Although some of the features of Ada complicate the Linking process, notably generics and the separate compilation model, linkers are generally not as difficult to build as compilers or the routines to implement the KAPSE interface. Therefore, the major problem is the KAPSE interface.

Difficulties in implementing the KAPSE for the new host is a relative unknown and the primary subject of this discussion. Given that MAPSE and APSE tools are developed properly (are target independent), and that the other requirements for rehosting have been satisfied, tools should port easily. The remainder of this development will focus on KAPSE rehostability.

Consider the present situation. The Army is currently contracting for the development of the Ada Language System and the Air Force contracting for the Ada Integrated Environment. The fact that the two KAPSEs are different is being addressed by the KAPSE Interface Team (KIT) and others. Putting aside the issue of portability between the APSEs built on two different KAPSEs, there is still a potential problem regarding rehostability of either of the environments. Foremost among these difficulties is that the KAPSE interface is going to be defined by an implementation. The KAPSEs are to be hosted on particular machine operating systems with the resultant danger that implementation details will show through the KAPSE semantics. Indeed many characteristics exist for efficiency reasons but will likely be regarded as required for a valid KAPSE interface. By KAPSE semantics is meant "those things in the KAPSE interface which must be true to insure the portability of APSE tools." It is crucial that the semantics of the KAPSE interface be clearly defined, much like the semantic definition of Ada itself.

The clean separation of the KAPSE semantics from particular implementation characteristics will allow the understanding of (1) how well defined KAPSE semantics can be implemented in different ways, (2) where implementation details or dependencies can show through the KAPSE (and isolate them), (3) how a standard semantics can be implemented in less or more efficient ways, and (4) how the semantics may determine what kinds of machines may host an APSE. Consider, for example, the "program call" interface. Ada programs are required to be able to call other Ada programs. Both the ALS and the AIE provide such a KAPSE interface routine in two flavors: one suspends the calling program until completion of the called program, and the other allows the two programs to proceed in parallel. Yet information is passed between the two programs only on call and return. Thus, it seems that a valid implementation could treat both calls identically (no parallelism) with parallel execution a detail of the implementation. There is no question that both types of call should exist, since greater efficiency is desirable, but requiring simultaneous execution could make implementation of the KAPSE on some hosts prohibitive. If such parallel execution is to be made part of the KAPSE semantics, the consequences must be made obvious.

The Layered KAPSE Model

Rehostability of a KAPSE means the degree to which it is portable. The idea of a portable KAPSE seems contrary to its definition. Nevertheless, let us consider what this might mean.

First, separate out of the KAPSE the portion necessary to implement the Ada Language. This portion is normally termed the Run Time System, and we will dismiss this as part of the Ada Language implementation problem. Some portions of the RTS will, of course, call on other portions of the KAPSE, e.g. Input_output will need to access KAPSE I/O.

Increased portability is usually gained at the expense of efficiency. This derives from the fact that highly portable software makes the least demands on an underlying interface. There are methods for obtaining both highly portable software yet maintaining the capability of obtaining desired efficiencies. Such tools usually define two (or more) interfaces and implement the higher level interface with portable routines depending only on the more primitive inner level. Once the software is ported, requiring implementation of only the primitive level, additional effort can be expended, recoding the intermediate routines, to gain desired efficiency of operation. Let us apply this methodology to the KAPSE.

A good way of approaching the problem of defining KAPSE semantics is to define a low level interface upon which the remainder of the KAPSE could be implemented in a portable manner (ignoring efficiency issues). There are two additional benefits to be derived from this approach. First, it will tend to give an operational definition of the KAPSE semantics, but from a different point of view from that of the Softech implementation, in that its orientation is toward making the KAPSE available to many hosts rather than efficiently on a particular host. Second, most of the KAPSE will be written in Ada which has well defined semantics itself. Let us call the lowest level interface that the KAPSE can be built upon the "minimum host interface." We would like to define a sequence of interface layers at successively higher levels until the KAPSE interface level is reached. The requirement being that outer layers always be implementable in "portable Ada" depending only on inner (or adjacent) layers. In this way, the opportunity to take advantage of host operating system features for efficiency reasons will exist.

As an example, let us sneak up on the minimum host interface by looking at what would be needed to put the KAPSE on a base machine. A first layer would contain physical device drivers. But, every host operating system contains such drivers, so this layer need never be build. Next, build the layer to implement a device independent interface. The nature of this layer is not common to all operating systems since within this layer may be found the "filing system". Filing systems are notoriously different on different operating systems. Nevertheless, non file-structured devices, and the allocation of such devices to users, are strikingly similar on all hosts. Thus, as long as the file-structured devices are not included, the rest of the I/O system can be provided by the host operating system. Every host should be able to provide a single file object which appears externally as a logical disk. Let us use this as a point of departure for building the KAPSE database/filing system in a portable manner (using Ada).

It is suggested that attempting to define a minimum host interface in conjunction with multiple layers of portable KAPSE routines implementing higher and higher interface levels could be a valuable approach toward developing a KAPSE Semantic Definition. Implementation of the KAPSE with portable routines outside of this minimum host interface would provide both an operational definition of these semantics and facilitate the rehosting of Ada Environments. In addition, if the semantics of the KAPSE have been carefully defined, some parts of the portable KAPSE that are (initially) in Ada could be rewritten to take advantage of particular OS services to result in a more efficient KAPSE.

Introduction

The KITIA Working Group on KAPSE services has established a preliminary set of recommendations and some supporting papers for these recommendations. In addition, the group has a number of papers on important issues regarding the KAPSE and an expanded outline for the final report. These papers have been collected together in a working paper which includes:

1. Recommendations
2. The Adaptation of VAX VMS Services to KAPSE to Accommodate Transportability of Tools
by R. D. Johnson (Boeing)
3. KAPSE Model of VMS System Services
by R. D. Johnson (Boeing)
4. Outline for Ada Interoperability and Transportability Guidelines by R. D. Johnson (Boeing)
5. Ada/APSE Portability, A Recommendation for Pragmatic Limitations by Herb Willman (Raytheon)
6. An Executive Summary of the TOPS-20 Operating System Calls by Thomas A. Standish (UC Irvine)
7. New Category--Extensibility
by Thomas A. Standish (US Irvine)
8. KAPSE Interface Category: G.1 Debugger Support
by Douglas E. Wrege (Control Data)
9. The Consequences of Multiple DoD KAPSE Efforts
by Douglas E. Wrege (Control Data)
10. KAPSE Services--Expanded Outline
by Hert Willman (Raytheon)

AD-A123 136

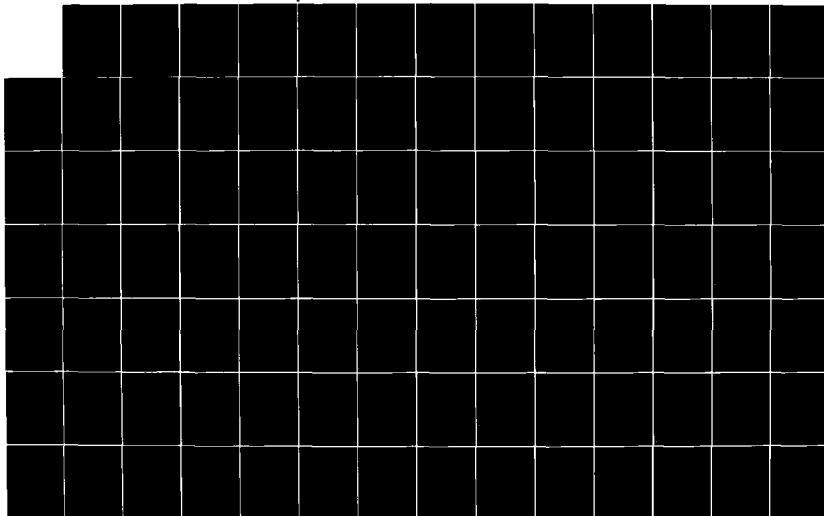
KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT (KAPSE)
INTERFACE TEAM: PUBLIC REPORT VOLUME II(U) NAVAL OCEAN
SYSTEMS CENTER SAN DIEGO CA P A OBERNDORF 28 OCT 82
F/G 9/2

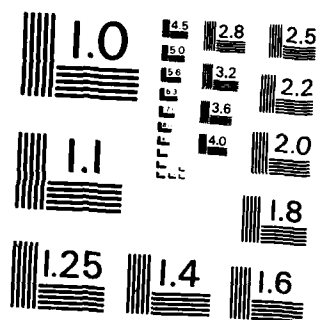
516

UNCLASSIFIED

NOSC/TD-882

NL





Recommendations

1. That AJPO shall produce a Guideline for the development of Transportable and Interoperable software.

The outline of such a guideline is presented in the "Outline for Ada Interoperability and Transportability Guidelines" by Ron Johnson.

2. That AJPO shall define a single, standard set of "system services" (i.e., tool-callable KAPSE functions and procedures) that are necessary and sufficient for a KAPSE/tool (or application program) interface.

To assist AJPO in selection of the standard services, descriptions of similar capabilities of existing operating systems are included in the following sections. These are:

VAX VMS - Ron Johnson

TOPS 20 - Thomas A. Standish

Recommendation

The KIT/KITIA should draft, or cause to be drafted, a set of Ada Package Specifications (together with a clear, unambiguous description of the associated semantics) for a set of operating system services that could serve as a basis for promoting the interoperability and transportability of Ada programs. Such a draft should be circulated to the community for tuning, upgrade, and general reactions.

End Recommendation

Discussion

In STONEMAN, a promise is given for a set of appendices providing provisional examples of an Ada package spec. giving operating system services. If such a set of provisional services could be drafted and circulated, we might have a basis for moving toward standardized operating system service calls, which, if used, could greatly improve the chances that the Ada programs we write would be transportable. We should carry out the STONEMAN recommendation and fulfill its promise to supply these appendices.

End Discussion

The paper, "An Executive Summary of the TOPS-20 Operating System Calls," by Thomas A. Standish, supports this recommendation.

Recommendation

"That an INTERFACEMAN document be prepared presenting policies, procedures, and means of technical support, for adding user-interfaces to new APSE tools to an APSE so as to preserve the simplicity and uniformity of the user interface."

End Recommendation

This extends and completes the work of STONEMAN and addresses the means of implementing STONEMAN requirement 3.C as it pertains to APSE tool user-interfaces.

The paper, "New Category -- Extensibility" by Thomas A. Standish, supports this recommendation.

Recommendations

- There will be a well defined boundary between a minimal set of system services called the KAPSE and user oriented services called the MAPSE.
- There will be a well defined set of system services which support all target computers.

THE ADAPTATION OF VAX VMS SYSTEM SERVICES TO KAPSE TO ACCOMODATE TRANSPORTABILITY OF TOOLS

R. D. Johnson

BOEING AEROSPACE CO.

Macro Outline

Paragraph

Number

Title

1	Introduction
2	VAX VMS System Services
2.1	Event Flag Services
2.2	Asynchronous System Trap Services
2.3	Logical Name Services
2.4	Input/Output Services
2.5	Process Control Services
2.6	Timer and Time Conversion Services
2.7	Condition Handling Services
2.8	Memory Management Services
2.9	Change Mode Services
3	KAPSE System Services for Transportability
3.1	Event Flag Services
3.2	Asynchronous System Trap Services
3.3	Logical Name Services
3.4	Input/Output Services
3.5	Process Control Services
3.6	Timer and Time Conversion Services
3.7	Condition Handling Services
3.8	Memory Management Services
3.9	Change Mode Services

```

--
--
--
--      FILL_STR(MSG,"Is Mission Exercise Tape available?"
--              & "Answer 'yes' or 'no'");
--      SEND_MSG_TO_OP(MSG, REPLY);
--      if IS_EQUAL_STR(REPLY, "yes") then
--
--
--
--      end if;
--      -- NOTE: The following are part of
--      -- package VARIABLE_STRING:
--      --      (1) abstract data type VARIABLE_STRING
--      --      (2) procedure FILL_STR
--      --      (3) function IS_EQUAL_STR
--
--      (e) Example of Application: TBD

```

- 3.1 Event Flag Services
- 3.2 Asynchronous System Trap Services
- 3.3 Logical Name Services
- 3.4 Input/Output Services
- 3.5 Process Control Services
- 3.6 Timer and Time Conversion Services
 - This section will also note use of a standard system
 - base date. VMS uses 17 November 1858, the Smith-
 - sonian base date and time for the Astronomical
 - Calendar.
- 3.7 Condition Handling Services
- 3.8 Memory Management Services
- 3.9 Change Mode Services

Detailed Outline

1. Introduction

- 1.1 Purpose of the Report
- 1.2 Scope of the Report
- 1.3 Structure of the Report
- 1.4 References

2. VAX VMS System Services

- There will be an introductory paragraph in which the following are
- discussed.
- . General description of VMS system services
- . Language interfaces
- . Privileged usage
- . Form of Section 2: Each of the sub-sections
- 2.1 thru 2.9 will contain a general description
- of the category and identification of the VMS
- services within the category. For each service
- there will be a terse explanation of its function
- and an appraisal of its portability aspects.
- Appropriate privileges will also be noted.

2.1 Event Flag Services

- 2.1.1 \$ASCEFC -- Associate Common Event Flag Cluster
- 2.1.2 \$DACEFC -- Disassociate Common Event Flag Cluster
- 2.1.3 \$DLCEFC -- Delete Common Event Flag Cluster
- 2.1.4 \$SETEF -- Set Event Flag
- 2.1.5 \$CLREF -- Clear Event Flag
- 2.1.6 \$READEF -- Read Event Flag

- 2.1.7 \$WAITFR -- Wait for Single Event Flag
- 2.1.8 \$WFLOP -- Wait for Logical OR of Event Flags
- 2.1.9 \$WFLAND -- Wait for Logical AND of Event Flags

2.2 Asynchronous System Trap (AST) Services

- 2.2.1 \$SETAST -- Set AST Enable
- 2.2.2 \$DCLAST -- Declare AST
- 2.2.3 \$SETPRA -- Set Power Recovery AST

2.3 Logical Name Services

- 2.3.1 \$CRELOG -- Create Logical Name
- 2.3.2 \$DELLOG -- Delete Logical Name
- 2.3.3 \$STRNLOG -- Translate Logical Name

2.4 Input/Output Services

- 2.4.1 \$ASSIGN -- Assign I/O Channel
- 2.4.2 \$DASSGN -- Deassign I/O Channel
- 2.4.3 \$QIO -- Queue I/O Request
- 2.4.4 \$QIOW -- Queue I/O Request and Wait for Event Flag
- 2.4.5 \$INPUT -- Queue Input Request and Wait for Event Flag
- 2.4.6 \$OUTPUT -- Queue Output Request and Wait for Event Flag
- 2.4.7 \$FAO -- Formatted ASCII Output
- 2.4.8 \$FAOL -- Formatted ASCII Output with List Parameter
- 2.4.9 \$ALLOC -- Allocate Device
- 2.4.10 \$DALLOC -- Deallocate Device
- 2.4.11 \$GETCHN -- Get I/O Channel Information
- 2.4.12 \$GETDEV -- Get I/O Device Information
- 2.4.13 \$CANCEL -- Cancel I/O on Channel
- 2.4.14 \$CREMBX -- Create Mailbox and Assign Channel
- 2.4.15 \$DELMBX -- Delete Mailbox
- 2.4.16 \$BRDCST -- Broadcast
- 2.4.17 \$SENDACC -- Send Message to Accounting Manager
- 2.4.18 \$SEND SMB -- Send Message to Symbiont Manager

- 2.4.19 \$SENDOPR -- Send Message to Operator
- 2.4.20 \$SENDERR -- Send Message to Error Logger
- 2.4.21 \$GETMSG -- Get Message
- 2.4.22 \$PUTMSG -- Put Message

2.5 Process Control Services

- 2.5.1 \$CREPRC -- Create Process
- 2.5.2 \$DELPRC -- Delete Process
- 2.5.3 \$HIBER -- Hibernate
- 2.5.4 \$WAKE -- Wake
- 2.5.5 \$SCHDWK -- Schedule Wakeup
- 2.5.6 \$SUSPND -- Suspend
- 2.5.7 \$RESUME -- Resume Process
- 2.5.8 \$CANWAK -- Cancel Wakeup
- 2.5.9 \$EXIT -- Exit
- 2.5.10 \$FORCEX -- Force Exit
- 2.5.11 \$DCLEXH -- Declare Exit Handler
- 2.5.12 \$CANEXH -- Cancel Exit Handler
- 2.5.13 \$SETPRN -- Set Process Name
- 2.5.14 \$SETPRI -- Set Priority
- 2.5.15 \$SETRWM -- Set Resource Wait Mode
- 2.5.16 \$GETJPI -- Get Job/Process Information
- 2.5.17 \$SETPRV -- Set Privileges

2.6 Timer and Time Conversion Services

- 2.6.1 \$GETTIM -- Get Time
- 2.6.2 \$NUMTIM -- Convert Binary Number to Numeric Time
- 2.6.3 \$ASCTIM -- Convert Binary Time to ASCII String
- 2.6.4 \$BINTIM -- Convert ASCII String to Binary Time
- 2.6.5 \$SETIMR -- Set Timer
- 2.6.6 \$CANTIM -- Cancel Timer Request
- 2.6.7 \$SCHDWK -- Schedule Wakeup
- 2.6.8 \$CANWAK -- Cancel Wakeup
- 2.6.9 \$SETIME -- Set System Time

2.7 Condition Handling Services

- 2.7.1 \$SETEXV -- Set Exception Vector
- 2.7.2 \$SETSFM -- Set System Service Failure Exception Mode
- 2.7.3 \$UNWIND -- Unwind Call Stack
- 2.7.4 \$DCLCMH -- Declare Change Mode or Compatibility Mode Handler

2.8 Memory Management Services

- 2.8.1 \$EXPREG -- Expand Program/Control Region
- 2.8.2 \$CNTREG -- Contract Program/Control Region
- 2.8.3 \$CRETVA -- Create Virtual Address Space
- 2.8.4 \$DELTVA -- Delete Virtual Address Space
- 2.8.5 \$CRMPSC -- Create and Map Section
- 2.8.6 \$MGBLSC -- Map Global Section
- 2.8.7 \$UPDSEC -- Update Section File on Disk
- 2.8.8 \$DGBLSC -- Delete Global Section
- 2.8.9 \$LKWSET -- Lock Pages in Working Set
- 2.8.10 \$ULKWSET -- Unlock Pages from Working Set
- 2.8.11 \$PURGWS -- Purge Working Set
- 2.8.12 \$LCKPAG -- Lock Pages in Memory
- 2.8.13 \$UNLPAG -- Unlock Pages from Memory
- 2.8.14 \$ADJWSL -- Adjust Working Set Limit
- 2.8.15 \$SETPRT -- Set Protection on Pages
- 2.8.16 \$SETSWM -- Set Process Swap Mode

2.9 Change Mode Services

- 2.9.1 \$CMEXEC -- Change to Executive
- 2.9.2 \$CMKRNL -- Change to Kernel Mode
- 2.9.3 \$ADJSTK -- Adjust Outer Mode Stack Pointer

3. KAPSE System Services for Transportability

- In this section we'll identify services listed in section
- 2 which should also appear in KAPSE.
- There will be an introductory paragraph in which the
- following are discussed:
 - . General description of KAPSE services
 - . Language interfaces
 - . Motivation
 - . Role of KAPSE vs. that of MAPSE
 - . Privileged usage
 - . Form of Section 3: Each of the sub-sections
 - 3.1 thru 3.9 will correspond in title and func-
 - tion to corresponding sub-sections 2.1 thru
 - 2.9. For each service included in these sub-
 - sections the following information will be
 - provided
 - (a) Reference to an associated VMS system service, by
 - paragraph number
 - (b) Rationale for including this service in KAPSE
 - (c) Tool/KAPSE interface description (i.e., Ada
 - <subprogram_declaration>)
 - (d) Example of usage
 - (e) Example of an application

-- The following example illustrates a KAPSE system service
-- description:

-- 3.4.5 Send Message to Operator

-- This system service allows a program to send a message
-- to an operator's terminal and optionally to receive a
-- reply. The operator's terminal may be specified, or the
-- default master terminal may be implied.

```

--
-- (a) Associated VMS System Service: $SENDOPR (2.4.19)
--
-- (b) Rationale: Tools and application programs need
--               an easily-used operator interface.
--
-- (c) Interface: The <subprogram_declaration>'s for the
--               overloaded procedure are:
--
--               procedure SEND_MSG_TO_OP -- no reply, master terminal
--                   (A: in VARIABLE_STRING);
--               procedure SEND_MSG_TO_OP -- no reply, specified terminal
--                   (A: in VARIABLE_STRING;
--                    B: in OP_ID);
--               procedure SEND_MSG_TO_OP -- reply, specified terminal
--                   (A: in VARIABLE_STRING;
--                    B: in OP_ID ;
--                    C: out VARIABLE_STRING);
--               procedure SEND_MSG_TO_OP -- reply, master terminal
--                   (A: in VARIABLE_STRING;
--                    C: out VARIABLE_STRING);
--
-- (d) Example of Usage:
--       MSG      : VARIABLE_STRING(50);
--       REPLY    : VARIABLE_STRING(10);
--
--               --
--               --
--               --
--
--       begin

```

**KAPSE MODEL
OF
VMS SYSTEM SERVICES**

Ren Johnson
BOEING AEROSPACE

4 October 1982

To: TRICIA OBERNDORF

Subject: KAPSE Model of VMS System Services

From: RDN JOHNSON

In Appendix C of the 9 June 1982 Corrected KITIA Minutes

I was tasked with producing a VAX/VMS model couched in Ada.
The first (albeit incomplete) cut of this model is in the
following pages, presented as an Ada package.

Because of our (i.e., Group 3) interest in run-time services,
the model is defined as a set of system services which would
be useful to a tool builder. I've discarded a number of VMS
services that did not seem too useful in this context.

Another large group of VMS services were omitted because
of their obvious dependence on VAX.

As I've already stated, this is a first cut; you'll probably
note that some services have been intentionally omitted, for
later examination. But October has come, so I'd better
submit this to you as is. I'm also wondering if this is a
worthwhile effort anyway. If it is, or if you'd like to
see a change in this direction, then let me know. I'll
suspend this activity until I hear from you.

RDN

package KAPSE_SERVICES_MODEL_FROM_VMS is

-- VMS System Services are described in Chapter 11 of the
-- "VAX Software Handbook", 1982, by Digital Equipment Corporation.
-- The Services "... are procedures incorporated into and used by
-- the operating system to control resources available to processes
-- (NOTE: process = executing_program + context), to provide for
-- communication among processes, and to perform basic operating
-- system functions, such as the coordination of input/output
-- functions." Services are grouped into ten functional
-- categories:

--
-- . Event Flag Services
-- . Asynchronous System Trap (AST) Services
-- . Logical Name Services
-- . Input/Output Services
-- . Process Control Services
-- . Timer and Time Conversion Services
-- . Condition Handling Services
-- . Memory Management Services
-- . Change Mode Services
-- . Lock Management Services
--

-- These categories are associated with the (sub) package
-- specifications which follow. Services that I do not consider
-- appropriate to the general KAPSE case have not been included
-- (Even so, the subset may be too large.) In the main, KAPSE
-- services presented here form a true subset of VMS, though
-- there are a few exceptions.

-- The following, quoted functional descriptions of service
-- categories are from the VAX Software Handbook.

type PROGRAM_ID is private;

type CLUSTER_NAME is new STRING(1..15);

subtype FLAG_NUMBER is INTEGER range 0..32; -- FLAG_NUMBER = 0..32

NONE_EXIST : constant PROGRAM_ID;

SELF : constant PROGRAM_ID;

ALL_PROGRAMS : constant PROGRAM_ID;

-- \$\$\$\$

package EVENT_FLAG_SERVICES is

-- Event Flag Services allow a process or group of
-- cooperating processes to read, wait for, and manipulate
-- event flags. A process can use event flags to
-- synchronize sequences of operations in a program."
--
-- All VMS services are represented in the model.

type CLUSTER_TYPE is (LOCAL_PERM, LOCAL_TEMP,
GLOBAL_PERM, GLOBAL_TEMP);

type FLAG_MASK is array(0..FLAG_NUMBER'LAST-1) of BOOLEAN;

subtype FLAG_CLUSTER is FLAG_MASK;

subtype SETABLE_FLAG is FLAG_NUMBER range 0..FLAG_NUMBER'LAST - 1;

procedure ASSOCIATE_COMMON_EVENT_FLAG_CLUSTER

(A : CLUSTER_NAME;
B : CLUSTER_TYPE);
-- Create a cluster (if it doesn't already
-- exist) and associate the caller with it.
-- Creation of a permanent cluster requires
-- privilege.

procedure DISASSOCIATE_COMMON_EVENT_FLAG_CLUSTER

(A : CLUSTER_NAME);
-- Implicitly happens at termination anyway.
-- If no programs are still associated, the
-- cluster is removed.

procedure DELETE_COMMON_EVENT_FLAG_CLUSTER

(A : CLUSTER_NAME);
-- Causes cluster to become temporary.
-- Privilege is required.

procedure SET_EVENT_FLAG

(A : CLUSTER_NAME;
B : SETABLE_FLAG);

procedure CLEAR_EVENT_FLAG

(A : CLUSTER_NAME;
B : FLAG_NUMBER);

function READ_EVENT_FLAGS

(A : CLUSTER_NAME) return FLAG_CLUSTER;

procedure WAIT_FOR_SINGLE_EVENT_FLAG

(A : CLUSTER_NAME;
B : SETABLE_FLAG);

procedure WAIT_FOR_LOGICAL_OR_OF_EVENT_FLAGS

(A : CLUSTER_NAME;
B : FLAG_MASK);

```

procedure WAIT_FOR_LOGICAL_AND_OF_EVENT_FLAGS
  (A : CLUSTER_NAME;
   B : FLAG_MASK);

```

```

end EVENT_FLAG_SERVICES;

```

```

package ASYNCHRONOUS_SYSTEM_TRAP_SERVICES is

```

```

-- "Various system services allow a process to request that it be
-- interrupted when a particular event (such as I/O completion)
-- occurs. Since the interrupt occurs asynchronously with
-- respect to the process execution, the interrupt mechanism is
-- called an asynchronous system trap (AST). The trap provides
-- a transfer of control to a user-specified routine that handles
-- the event." In the model are procedures which enable traps.
-- AST procedures are specified elsewhere. (See, for example,
-- procedure SET_TIMER in package TIMER_AND_TIME_CONVERSION_
-- SERVICES.)
--
-- The service Set_Power_Recovery_AST is not represented
-- in the model.

```

```

type ENABLE_MODE is (ENABLE, DISABLE);

```

```

type ACCESS_MODE is (SUPERVISOR, USER);

```

```

type AST_BLOCK is      -- described like VMS'
  record
    AST_PARAMETER : INTEGER;
    R_0           : INTEGER;
    R_1           : INTEGER;
    P_C           : INTEGER;
    P_S_L         : INTEGER;
  end record;

```

```

-- procedure SET_TIMER is described in package
-- TIMER_AND_TIME_CONVERSION_SERVICES, even though
-- AST's may be involved, so it is not included here.

```

```

procedure SET_AST_ENABLE
  (A : ENABLE_MODE);

```

```

procedure DECLARE_AST
  (A : ACCESS_MODE;
   B : AST_BLOCK);

```

```

end ASYNCHRONOUS_SYSTEM_TRAP_SERVICES;

```

```

-- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ --

```

package LOGICAL_NAME_SERVICES is

```
-- These services are very VMS-dependent and are
-- not general. They have been omitted from the
-- model, but are identified (in VMS) as:
--
--      Create Logical Name
--      Delete Logical Name
--      Translate Logical Name
--
-- These may be reviewed in Chapter 11 of the
-- VAX Software Handbook.
```

end LOGICAL_NAME_SERVICES;

-- \$

package INPUT_OUTPUT_SERVICES is

```
-- "The I/O system services permit the user to utilize the
-- I/O resources of the operating system directly in a
-- device-dependent manner. The I/O system services can
-- perform the following functions:
```

```
--      Assign and deassign channels
--      Queue I/O requests
--      Synchronize I/O completion
--      Allocate and deallocate devices
--      Create mailboxes
--      Perform network operations"
```

```
-- The following VMS services are presently not represented
-- in the model. Their status is TBD.
```

```
--      Assign_IO_Channel
--      Deassign_IO_Channel
--      Queue_IO_Request
--      Queue_IO_Request_and_Wait_for_Event_Flag
--      Queue_Input_Request_and_Wait_for_Event_Flag
--      Queue_Output_Request_and_Wait_for_Event_Flag
--      Formatted_ASCII_Output_with_List_Parameters
--      Allocate_Device
--      Deallocate_Device
--      Mount_Volume
--      Dismount_Volume
--      Get_IO_Channel_Information
--      Get_IO_Device_Information
--      Get_Device_Volume_Information
--      Cancel_IO_on_Channel
--      Send_Message_to_Accounting_Manager
--      Send_Message_to_Symbiont_Manager
```



```

--      Send_Message_to_Error_Logger
--      Get_Message
--      Put_Message

NBR_OF_OP_CONSOLES : constant := 2; -- 2 operator consoles

NBR_OF_TERMINALS   : constant := 50; -- 50 terminals

subtype BROAD_STRING is STRING (1..56);

type CONTINUATION_INDICATOR is (MORE_TO_COME, TERMINATE_MESSAGE);

type FILE_TYPE is private;

subtype MAILBOX_NAME is STRING (1..15);

subtype OPERATOR_ID is INTEGER range 0..NBR_OF_OP_CONSOLES;
-- 0 => "all"

type PERMANENCE_INDICATOR is (PERMANENT, TEMPORARY);

subtype TERMINAL_ID is INTEGER range 0..NBR_OF_TERMINALS;
-- 0 => "all"

procedure FORMAT_ASCII_IO; -- TBD

procedure CREATE_MAILBOX
  (A : MAILBOX_NAME;
   B : PERMANENCE_INDICATOR;
   C : out FILE_TYPE);
-- PERMANENT requires privilege.

procedure DELETE_MAILBOX
  (A : FILE_TYPE);
-- Deletion of a permanent mailbox requires privilege.

--
-- Receiving and sending mail are done by using GET and PUT.
-- See package TEXT_IO in Ada LRM.
--

procedure BROADCAST -- to terminal
  (A : TERMINAL_ID;
   B : BROAD_STRING;
   C : CONTINUATION_INDICATOR);

procedure OPERATOR_MESSAGE -- note overload
  (A : OPERATOR_ID;
   B : BROAD_STRING;
   C : out BROAD_STRING); -- response required

procedure OPERATOR_MESSAGE -- note overload
  (A : OPERATOR_ID;
   B : BROAD_STRING); -- no response

private -- is really implementation-dependent

type FILE_TYPE is new INTEGER range 1..1000;

end INPUT_OUTPUT_SERVICES;

```

```
-----  
-- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ --  
-----
```

```
package PROCESS_CONTROL_SERVICES is
```

```
-- "Process control services allow the user to create,  
-- delete, and control the execution of processes."
```

```
--  
-- The following VMS services are presently not in  
-- the model. Their status is TBD.
```

```
--  
--      Exit  
--      Force_Exit  
--      Declare_Exit_Handler  
--      Cancel_Exit_Handler  
--      Set_Process_Name
```

```
subtype PRIORITY is INTEGER range 1..100;
```

```
subtype PROGRAM_NAME is STRING(1..15); -- Size really impl. dependent
```

```
type PROGRAM_INFORMATION is  
  record  
    P_NAME : PROGRAM_NAME;  
    -- TBD Accounting information  
    -- TBD Status  
  end record;
```

```
type RESOURCE_WAIT_MODE is (ON, OFF);
```

```
procedure CREATE_PROCESS  
  (A : PROGRAM_NAME;  
   B : out PROGRAM_ID);  
  -- Spawn an independent program.
```

```
procedure DELETE_PROCESS  
  (A : PROGRAM_ID);  
  -- A program can delete itself or another  
  -- program that it has spawned.  
  -- It can also delete some other program  
  -- in the system, provided it has proper  
  -- privileges.
```

```
procedure HIBERNATE;  
  -- A program goes into hibernation, awaiting  
  -- awakening from an outside event.
```

```
procedure WAKE  
  (A : PROGRAM_ID);  
  -- A hibernating program is immediately  
  -- awakened.
```

```
-- procedure SCHEDULE_WAKEUP  
-- See package TIMER_AND_TIME_CONVERSION_SERVICES
```

procedure SUSPEND_PROCESS

-- (A : PROGRAM_ID := SELF);

(A : PROGRAM_ID); -- NYU won't allow default by deferred cons

-- Suspend SELF or another program. Special

-- privileges are required to suspend any

-- but self or one created by this program.

procedure RESUME_PROCESS

(A : PROGRAM_ID);

-- Resume execution of a program previously

-- suspended. If suspension was not by the

-- caller, then special privilege is required.

procedure SET_PRIORITY

-- (A : PROGRAM_ID := SELF;

(A : PROGRAM_ID; -- NYU won't allow default

P : PRIORITY);

-- A program can change SELF or another's

-- priority. Special privilege is needed to

-- change another's or one's own to exceed

-- one's initial base priority.

procedure SET_RESOURCE_WAIT_MODE

(A : RESOURCE_WAIT_MODE := ON);

procedure GET_PROGRAM_INFORMATION

-- (A : PROGRAM_ID := SELF;

(A : PROGRAM_ID; -- NYU won't allow default

B : out PROGRAM_INFORMATION);

-- Special privilege is needed to inquire

-- after other programs. One can inquire

-- after SELF.

procedure SET_PRIVILEGES; -- T B D

function GET_PROGRAM_ID -- not in VMS

(A : PROGRAM_NAME) return PROGRAM_ID;

-- NONE_EXIST could be returned.

end PROCESS_CONTROL_SERVICES;

-- \$ --

```
package TIMER_AND_TIME_CONVERSION_SERVICES is
```

```
-- "Timer services can:
```

```
--  
--      . Schedule setting an event flag or queing an AST  
--      for the current process, or cancel a pending  
--      request that has not yet been honored  
--      . Schedule a wakeup request for a hibernating  
--      process, and cancel a pending request that has  
--      not yet been honored  
--      . Set the system time"
```

```
-- The Smithsonian base date and time for the Astronomical  
-- Calendar is used by VMS. This is 00:00 on 11/17/1958.  
--
```

```
-- All VMS services (plus others) are represented in the model.
```

```
type ABSOLUTE_IND is (ABSOLUTE, DELTA_OFFSET);
```

```
subtype ASCII_TIME is STRING(1..23);  
-- dd-mmm-yyyy hh:mm:ss.cc
```

```
subtype DELTA_TIME is STRING(1..15);  
-- ddd hh:mm:ss.cc
```

```
type NUMERIC_TIME is
```

```
record
```

```
    YEAR   : INTEGER;  
    MONTH  : INTEGER range 1..12;  
    DAY     : INTEGER range 1..31;  
    HOUR    : INTEGER range 0..23;  
    MINUTE  : INTEGER range 0..59;  
    SECOND  : INTEGER range 0..59;  
    HUNDRTS: INTEGER range 0..99;
```

```
end record;
```

```
type SMITHSONIAN_TIME is private;  
-- 100-nanosecond units offset from  
-- 00:00:00.000 hours on 11/17/1958
```

```
type WAKE_TYPE is (ABSOLUTE, DELTA_OFFSET, INTERVAL);
```

```
function GET_TIME return SMITHSONIAN_TIME;
```

```
function SMITHSONIAN_TO_NUMERIC  
    (A : ABSOLUTE_IND;  
     B : SMITHSONIAN_TIME) return NUMERIC_TIME;
```

```
function NUMERIC_TO_SMITHSONIAN -- not in VMS  
    (A : ABSOLUTE_IND;  
     B : NUMERIC_TIME) return SMITHSONIAN_TIME;
```

```
function SMITHSONIAN_TO_ASCII  
    (A : SMITHSONIAN_TIME) return ASCII_TIME;
```

function ASCII_TO_SMITHSONIAN

(A : ASCII_TIME) return SMITHSONIAN_TIME;

-- Current values will be substituted for blanks in 'a.

function ASCII_DELTA_TO_SMITHSONIAN_DELTA

(A : DELTA_TIME) return SMITHSONIAN_TIME.

procedure SET_TIMER -- note overload

(A : PROGRAM_ID; -- AST identification

B : SMITHSONIAN_TIME;

C : ABSOLUTE_IND := ABSOLUTE);

procedure SET_TIMER -- note overload

(A : CLUSTER_NAME;

B : FLAG_NUMBER;

C : SMITHSONIAN_TIME;

D : ABSOLUTE_IND := ABSOLUTE);

procedure CANCEL_TIMER_REQUEST -- note overload

(A : PROGRAM_ID); -- AST identifier

-- A can be ALL_PROGRAMS

procedure CANCEL_TIMER_REQUEST -- note overload

(A : CLUSTER_NAME := "ALL";

B : FLAG_NUMBER := FLAG_NUMBER 'LAST);

-- Defaults both mean "all".

procedure SCHEDULE_WAKEUP

(A : PROGRAM_ID;

B : WAKE_TYPE;

C : SMITHSONIAN_TIME);

procedure CANCEL_WAKEUP

(A : PROGRAM_ID);

-- All scheduled wakeup requests (made by

-- call to SCHEDULE_WAKEUP by this and other

-- programs) are cancelled.

procedure SET_SYSTEM_TIME

(A : ABSOLUTE_IND;

B : SMITHSONIAN_TIME);

-- For operator use.

private

type SMITHSONIAN_TIME is array(1..4) of INTEGER;

end TIMER_AND_TIME_CONVERSION_SERVICES;

-- \$ --

```
-----  
- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ -  
-----
```

```
package CHANGE_MODE_SERVICES is
```

```
-- "The Change Mode Services allow a process to change  
-- to either executive mode or kernel mode to execute  
-- a specified routine. Use of these services requires  
-- privilege."
```

```
-- The model supports only executive mode.  
--
```

```
-- The service Adjust_Outer_Mode_Stack_Pointer is not  
-- represented in the model.
```

```
generic  
  with procedure CALLED_PROGRAM;  
  procedure CALL_IN_EXECUTIVE_MODE;
```

```
-- This generic procedure allows one to:  
-- (1) change to executive mode  
-- (2) enter a procedure  
-- (3) return to normal mode  
-- Use of the generic procedure insures  
-- that normal mode is re-instated.
```

```
-- To instantiate the above for two procedures:
```

```
--  
-- procedure CALL_XYZ is new CALL_IN_EXECUTIVE_MODE(XYZ);  
-- procedure EX_GRS is new CALL_IN_EXECUTIVE_MODE(GRS);
```

```
-- And to use the instantiated programs, we say:
```

```
-- CALL_XYZ;  
-- EX_GRS;
```

```
-- The program body may appear thus:
```

```
--  
-- procedure CALL_IN_EXECUTIVE_MODE is  
-- begin  
--   SET_EXEC_MODE; -- defined elsewhere  
--   CALLED_PROGRAM; -- generic parameter  
--   SET_NORMAL_MODE; -- defined elsewhere  
-- end;
```

```
end CHANGE_MODE_SERVICES;
```

```
-----  
-- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ -  
-----
```

```
package LOCK_MANAGEMENT_SERVICES is
```

```
-- These services "are a tool to help users develop complex
-- resource-sharing applications; for example, database
-- systems. It (sic) provides a resource nametable for
-- defining a resource with a variety of lock modes for
-- controlling access to it, and the means for processes
-- to queue lock requests."
```

```
-- All VMS services are represented in the model.
```

```
subtype EVENT_FLAG is INTEGER range -1000..0;
```

```
type LOCK_MODE is (NO_LOCK, CONCURRENT_READ, CONCURRENT_WRITE,
PROTECTED_READ, PROTECTED_WRITE, EXCLUSIVE);
```

```
type LOCK_STATUS is (UNLOCKED, ALREADY_LOCKED);
```

```
subtype RESOURCE_NAME is STRING(1..15);
```

```
procedure ENGUE_LOCK_REQUEST      -- note overload
```

```
(A : RESOURCE_NAME;
B : LOCK_MODE;
C : out LOCK_STATUS);
```

```
procedure ENGUE_LOCK_REQUEST      -- note overload
```

```
(A : RESOURCE_NAME;
B : LOCK_MODE;
C : PROGRAM_ID);      -- AST Identification
```

```
procedure ENGUE_LOCK_REQUEST      -- note overload
```

```
(A : RESOURCE_NAME;
B : LOCK_MODE;
C : CLUSTER_NAME;
D : SETABLE_FLAG);
```

```
procedure ENGUE_LOCK_REQUEST_AND_WAIT_FOR_EVENT
```

```
(A : RESOURCE_NAME;
B : LOCK_MODE);
```

```
procedure DEQUE_LOCK_REQUESTS      -- note overload
```

```
(A : RESOURCE_NAME);
```

```
procedure DEQUE_LOCK_REQUESTS      -- note overload
```

```
(A : RESOURCE_NAME;
B : LOCK_MODE);
```

```
end LOCK_MANAGEMENT_SERVICES;
```

```
-----
-- $$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$ --
-----
```

private

-- The following declarations do not necessarily bear
-- any resemblance to VMS, but are supplied so that
-- I can compile the package. I guess I could also
-- make similar statements in other places.

type PROGRAM_ID is new INTEGER range 0..1000;

NONE_EXIST : constant PROGRAM_ID := 1;

SELF : constant PROGRAM_ID := 0;

ALL_PROGRAMS : constant PROGRAM_ID := 2;

old KAPSE_SERVICES_MODEL_FROM_VMS;

**OUTLINE FOR
ADA INTEROPERABILITY
AND
TRANSPORTABILITY GUIDELINES**

**R. D. Johnson
BOEING AEROSPACE**

*** VERY PRELIMINARY ***

Outline for

A D A

I N T E R O P E R A B I L I T Y

A N D

T R A N S P O R T A B I L I T Y

G U I D E L I N E S

*** VERY PRELIMINARY ***

ADA IT GUIDELINES

<u>CHAPTER</u>	<u>TITLE</u>
1	INTRODUCTION
2	ADA LANGUAGE ISSUES
3	APSE ISSUES
4	HARDWARE ISSUES
5	THE TRANSPORT Process
6	AUTOMATED TRANSPORT AIDS
7	CASE STUDIES
8	APPENDICES : SYSTEM-UNIQUES GUIDELINES

CHAPTER 1 INTRODUCTION

Purpose of document

Scope of document.

Definition of terms

Statement of problem

Solution of problem

Bibliography

Structure and contents of document

Chapter 2 ADA LANGUAGE ISSUES

2.1 Character Sets

- Standard and extended sets
- Writing with portability in mind
- Set transformation (mechanical and otherwise)

2.2 Representation Specifications

- How they assist transportability
- How they hinder transportability
- Locating them in pragmas
- Effects on efficiency
- Alternatives

2.3 Pragmas

- Language-defined pragmas
- Implementation-defined pragmas
- Locating pragmas in programs
- Alternatives

2.4 Use of Standard Types

- Implementation defined
- Possible problems
- Locating usage in programs
- Alternatives

2.5 Efficiency Considerations

- Differences in computers and APSEs
- Differences in optimization
- Measuring efficiency
- Program modification

2.6 Attributes

Usefulness in developing portable programs
Relationship to Package System

2.7 Unchecked Programming

Dangers
Locating unchecked programming in programs

2.8 Package Standard

Care with type descriptors
Locating use of particular types
Package SYSTEM

2.9 Appendix F of ADA LRM - Discussion

Chapter 3 APSE ISSUES

3.1 Use of Other Languages

- What one might expect
- Porting a multi-language program

3.2 Libraries

- The "Standard Library"
- Other needs
- Installing other Libraries

3.3 KAPSE System Services

- "Standard KAPSE Services"
- Extensions to the standard set

3.4 KAPSE EXTENSIBILITY

3.5 Data Base Considerations

3.6 Debugging Considerations

3.7 I/O Considerations

- Normal
- Handlers for unique hardware

3.8 Downloading to Target Systems

3.9 Efficiency Measurements

3.10 Fine-Tuning a Ported System

3.11 Command Language

Chapter 4 HARDWARE ISSUES

4.1 Word Size Implications

- Accuracy
- Magnitude
- Efficiency

4.2 Internal Data Formats

4.3 I/O Data Formats

4.4 Virtual Systems

- KAPSE Services for control
- Porting between virtual and non-virtual systems

4.5 Memory Size

- Efficiency considerations
- Different program structures
- Restructuring during porting
- Tools which assist

4.6 Unique External Devices

4.7 Architectural Implications

- Micros
- Pipe line
- Array processor
- Ochers

Chapter 5 THE TRANSPORT PROCESS

5.1 Media

- Tape
- Disc
- Data link

5.2 Data Transfer

- Files
- Data Base
- Catalogues

5.3 Program Transfer

- Source representation
- DIANA

5.4 Manual Procedures

5.5 Automated Procedures

Chapter 6 AUTOMATED TRANSPORT AIDS

-- This chapter will contain summaries of
-- tools to assist IT. The description
-- of each will address the following (as
-- appropriate).

--

-- Function

-- Implementation Language

-- Host system

-- Target system

-- Method of acquisition

-- Reference material

Chapter 7 CASE STUDIES

- Several case studies will be documented.
- Each will be realistic and representative
- and will be described in terms of:
 -
 - Scope of effort to port
 - What was ported
 - Problems encountered
 - Solutions
 - Automated tools used
 - Manual processes

Chapter 8 APPENDIXES SYSTEM-UNIQUE GUIDELINES

- Appropriate sections reflecting chapters 2-7
- will be included in each appendix. How the
- issues are (or are not) resolved will be
- explained.
-
- Each APSE developer will provide the full
- book with an appendix addressing his
- particular implementation.

Ada/DPSE Portability
A Recommendation for Pragmatic Limitations
Herb Willman
Raytheon Company, Missile Systems Division

1. The General Problem of Software Portability

The general problem of transporting a program from one system to another can be stated as follows:

A program, P, which has compiled correctly on a host, A, and executes correctly on a target, X, is transported to a target, Y, by recompilation on a host, B.

Several possible outcomes of this rehosting/retargeting exist:

- (a) P compiles correctly on B and
executes correctly on Y, producing results
exactly equal to those from execution on X
- (b) P compiles correctly on B and
executes correctly on Y, producing different results,
but less correct, than those from execution on X
- (c) P compiles correctly on B and
executes correctly on Y, producing different results,
but more correct, than those from execution on X
- (d) P compiles correctly on B and
executes incorrectly on Y
- (e) P compiles incorrectly on B

Outcome (a) is the usually desired result: no portability problem exists.

Outcomes (b) and (c) may result when a program is retargeted to a machine which has a different representation for real variables. This might occur when a program is retargeted from a 16 bit word length machine to a 32 bit word length machine, or vice versa. These outcomes may also result from different execution times on Y yielding

~~~~~  
Ada is a registered trademark of the US Department of  
Defense (AJPO)

September 29, 1982

differing control signal or message transmission timing.

Outcome (d) may occur when the run time environments differ between X and Y. This may result in unacceptably different numerical results, message transmission timing, or control signal timing. The differing numerical results may occur because of different mathematical libraries being used or because of different representations, or both. The different control signal timing may result from different run time control programs having different priority structures or from execution time differences. Limitations on the features of the language supported by the run-time environment, such as the number of simultaneously active tasks allowed, may also cause differences. More on this later.

Outcome (e) may result from different compilers having different built in limitations on the source code which is acceptable to the compilers. Both compilers may be verified compilers. One compiler may compile a program correctly and another may not be able to do so. This may happen as a result of different pragmatic limitations for each compiler and is the focus for the remainder of this paper.

## 2. Pragmatic Limitations - an Ada Portability Issue

With the formalization of Ada syntax and the eventual formalization of Ada semantics, one potentially troublesome area that remains is the Ada pragmatics.

The pragmatics of a mechanical language usually refers to its applicability to a user problem domain. It addresses the languages utility for a particular application. In part, this utility may be limited by the compiler system designers. Because of the finiteness of the implementation environment or because of a concern for compiler system operating efficiency, features of the language may be further restricted by design or implementation. These limitations arise due to the real need by implementors to place a bound on certain language features not otherwise limited by specification or standardization. These restrictions are derived from the finite sizes of records, arrays, lists, stacks, strings, etc. within the compiler system itself and within the run time environment. Other limitations are explicitly identified by the Ada Language Reference Manual as "implementation dependent".

Therefore portability problems may arise because the pragmatics of different compilers may be different. When a program is transported from one computer type to another by recompilation, there may be areas of that program that have to be rewritten due to these differences.

September 29, 1982

Examples of these differences are:

- o Maximum identifier length
- o Maximum number of identifiers in a compilation unit
- o Maximum number of operators in an expression
- o Maximum number of enumeration values in a declaration
- o Maximum depth of recursion

These differences may not be limited to the compilers. The pragmatics of the APSEs themselves, especially in the KAPSE/MAPSE interface, may impose similar constraints. For example, the maximum number of compilation units allowed or the maximum number of external references that the link editors can process may be different.

### 3. Discussion of the Issue

Appendix A contains a list of items from the Ada Reference Manual (July 1980) which are subject to pragmatic limitation by compiler system implementors, and which therefore may present obstacles to portability. However, it may not be simple to identify the values for each of these items for a particular implementation. Many of these items may share storage area during compilation and the pragmatic limitations may occur with combinations of items. For example, a string containing all of the identifiers in a compilation unit may also contain all of the labels. The maximum number of identifiers then depends on the number of labels, and vice versa.

Depth of recursion is also a pragmatic consideration which is difficult to measure. Here the run time environment plays the critical role. We believe that the best that can be done here is to "plumb the depths" by using a standard benchmark for dynamically measuring the depth. The benchmark should be written to provide a measure of the depth at the point that the exception for stack overflow occurs. This data could then be published for the users information.

These difficulties aside, it is still important to know the pragmatic limitations of the relevant compilers when faced with transporting programs from one system to another.

Consider the following hypothetical situation of rehosting a program from target X compiled on host A to target Y compiled on host B. The pragmatic limitations of both compilers for a small set of items is listed in the table below:

September 29, 1982



| Items                         | Host A/Target X | Host B/Target Y |
|-------------------------------|-----------------|-----------------|
| max identifier length         | 20              | 31              |
| max length of strings         | 256             | 4096            |
| max dimension of ARRAYS       | 9               | 3               |
| max # enumeration values      | 32              | 65536           |
| max # PACKAGES in WITH clause | 48              | 32              |
| max # explicit exceptions     | 16              | 128             |
| max # compilation units       | 512             | 400             |
| max # statements in comp unit | 4095            | 512             |
| max # active tasks            | 256             | 8               |

It can be readily seen from the comparison above that unless these limitations are known to the user, it is highly likely that transporting a program from X to Y or from Y to X will be a major problem requiring significant redesign, rewriting, and recoding of the rehosted program.

This type of problem is a commonly occurring one during the rehosting of programs which are components of large DoD systems. At Raytheon, we have encountered it often when rehosting JOVIAL/J3 tools from one system to another. The overflowing of internal tables is the common symptom for such a problem and, if you are lucky, the compiler usually detects the overflow and degrades softly with explanation. Occasionally they just crash.

Since portability is a major reason for the development of Ada, we recommend that compiler pragmatics and also APSE/MAPSE/KAPSE pragmatics be recognized as a potential obstacle to portability and we make several recommendations in this regard.

#### **4. Recommendations for Requirements for Ada Pragmatics**

First, since portability is one of the principle reasons for the existence of Ada and we have illustrated that one of the possible obstacles to portability is the lack of standardization of Ada pragmatics, we recommend that the KIT and/or the AJPO establish a standard for a "least common capability" for Ada compilers which provide for a minimum capability for use in the development of portable software.

Appendix A of this paper contains a recommended list of "least common capabilities" for Ada Compiler Systems. This is a initial effort establishing such a specification and should be viewed as a strawman. (Or since "strawman" has an Ada history, perhaps we should view such a specification of "pragmatically applicable capability" as a pacman?)

September 29, 1982

Criteria for the establishing of such constraints are threefold:

- a) Compatibility with existing DoD standard languages
- b) Use of modern programming practices
- c) Use of automatic generation tools

We believe that compatibility with existing DoD standard languages is important from the standpoint of conversion of existing systems to Ada. Any existing DoD code is a candidate for translation to Ada via manual or automatic means. An Ada compiler system should have the capability of handling software so translated. This certainly has implications for pragmatic limitations.

The trend of modern programming is toward small, functionally simple modules and this implies that a compiler's capability also may be small. However, working against this is the trend toward higher level tools which do some amount of automatic generation of software. And since these are not typically limited to small modules, large compiler capability may be required. Also the technique of modularity via insertion of code segments may require large capacities in a compiler. So there are no simple guidelines in this area.

Therefore, we recommend that the KIT and AJPO review the recommendations contained in Appendix A and set up a mechanism for arriving at a standard or at least a set of guidelines.

Second, we recommend that the Ada Compiler Validation Suite (ACVS) be extended to measure the compliance with these "least common capabilities". The AJPO has a choice of just measuring the capabilities of compilers submitted for validation and publishing the results, thus making the portability considerations a matter of the procuring agency and the users concern, or the AJPO may require compliance with a standard as a necessary requirement for compiler acceptance.

Third, we recommend that a similar standard be established for the APSE/MAPSE/KAPSE's being developed. The portability of software tools may depend on similar pragmatic limitations of the Ada programming environment. Since this is also an area where portability may reap large benefits for cost reduction, standards for a "least common capability" should also be established here.

September 29, 1982

## APPENDIX A

### RECOMMENDATIONS FOR PRAGMATIC LIMITATIONS - ADA COMPILEBS

| <u>LANGUAGE ELEMENT</u>                             | <u>LEAST COMMON CAPABILITY</u> |
|-----------------------------------------------------|--------------------------------|
| <b>General:</b>                                     |                                |
| Length of Identifiers (Unqualified)                 | 31                             |
| Length of Qualified Identifiers                     | 72                             |
| Length of Attributes                                | 72                             |
| Length of Expanded Names                            | 72                             |
| Length of Labels                                    | 31                             |
| Length of Strings                                   | 4096                           |
| Dimensionality (Rank) of Arrays                     | 9                              |
| Number of Compilation Units                         | 1000                           |
| Number of Library Units                             | 5000                           |
| Number of PACKAGE names in a WITH clause            | 50                             |
| Number of PACKAGE names in a USE clause             | 50                             |
| Number of Simultaneously Active TASKs               | 100                            |
| Number of PRIORITyS                                 | 100                            |
| <b>By Compilation Unit:</b>                         |                                |
| Number of Identifiers                               | 1000                           |
| Number of TYPE Declarations                         | 200                            |
| Number of SUBTYPEs of a TYPE                        | 200                            |
| Number of Enumeration Values (per Declaration)      | 255                            |
| Number of elements in an ARRAY                      | 4096                           |
| Number of Record Components                         | 100                            |
| Number of Variant Parts                             | 50                             |
| Number of Declarations                              | 200                            |
| Number of Attributes                                | 50                             |
| Number of Literals                                  | 200                            |
| Depth of Nesting of IF statements                   | 50                             |
| Number of ELSIF Alternatives                        | 50                             |
| Depth of Nesting of CASE Statements                 | 100                            |
| Number of CASE Alternatives                         | 255                            |
| Depth of Nesting of LOOP Statements                 | 100                            |
| Number of Declarations in a Block                   | 100                            |
| Number of Statements in sequence_of_statements      | 100                            |
| Depth of Nesting of Blocks                          | 50                             |
| Number of Statements                                | 1000                           |
| Number of Explicit EXCEPTIONS                       | 100                            |
| Depth of Recursion (measured by standard benchmark) | ---                            |

September 29, 1982

| LANGUAGE ELEMENT                              | LEAST COMMON CAPABILITY |
|-----------------------------------------------|-------------------------|
| 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 ENTRYs                              | 100                     |
| Number of SELECT Alternatives                 | 100                     |
| Number of DELAYs                              | 100                     |

September 29, 1982

## NEW CATEGORY --- EXTENSIBILITY

Thomas A. Standish

The purpose of this memorandum is to raise the issue of whether or not the KITIA should consider Extensibility as a new category.

Part of the KITIA's responsibility is to determine the nature of the KAPSE services required to support future APSE toolsets. In the STONEMAN's "strategy for advancement," it is envisaged that environment tools, written in Ada, can be compiled in the MAPSE and can thus be added to an APSE toolset. The KAPSE must naturally provide the underlying operating system support services and database services to support the orderly growth of such APSE toolsets of the future. So the question arises of how the KAPSE can be structured to support extension of APSE toolsets. The issue of extensibility is therefore concerned with the nature of support services required to support tool extensions in APSEs and with any policies or standards that would govern the growth of such toolsets to ensure that they possess desirable properties.

One such issue that arises is whether or not we can devise policies of orderly, controlled growth of toolsets in APSEs that promote simplicity and uniformity of the user interface. Let us develop this issue a bit further in this memorandum to enable the KITIA better to decide whether it wishes to pursue the matter further.

One of the most important requirements in STONEMAN is  
requirement (3.C) for simplicity, quoted here:

3.C SIMPLICITY The structure of an APSE shall be based on simple overall concepts which are straightforward to understand and use and few in number. Wherever possible, the concepts of the Ada language will be used in an APSE.

Suppose it is our desire to produce APSE toolsets with user interfaces obeying this requirement, and suppose further, that when we extend the APSE toolset by adding new tools, we should like to have the new tools "blend" with the old in such a fashion as to maintain the simplicity of the original interface.

A model that seems not to provide for this property is the model of adding tools to current operating system toolsets. When tools in a normal operating system accumulate by ordinary evolution, they tend to exhibit a bewildering variety of different, often inconsistent, user-interface conventions. The same control characters mean different things in different tools, and there are a jumble of different methods for exiting subsystems, jumping to a system's top level, obtaining help, specifying commands, and so forth. This tends to impose a high cognitive load on the user as there is a considerable and probably needless learning burden imposed on him as he attempts to deal with the thicket of inconsistent interface methods which the tools require him to use.

If it were possible to devise some method or policy for controlling the growth of APSE toolsets, so that new tools added to an APSE would be required to obey one of a small, bounded number of "standard" user interface methods, the STONEMAN requirement for simplicity could be better met. But this must be done in such a fashion as not to inhibit the addition of new interface methods of substantial effectiveness and novelty that are discovered in the future. What is needed is a proper balance between reaping the benefits of future exploration (by incorporating new interface methods) and keeping the interface simple and easily useable (by enforcement of standards). How might this be accomplished? Here is a suggestion.

We know that there are certain popular interface methods for accessing tools in use at the moment. These include:

- (a) the normal conversational interface in which the system prompts the user to give a command (usually issuing a "prompt" character explicitly), and in which the user types a command in response. This type of interface may have various control characters for exiting to the top system level, suspending execution, obtaining help, and so forth.
- (b) the form-filling interface in which the user is presented with a form (usually on a CRT) with holes in it to fill in, and he advances from hole to hole entering suitable data. Here, there may be conventions for advancing to the next hole, returning to the top of the form, aborting, and so forth.
- (c) the tree of menus interface in which a top-level menu gives a set of choices and when one choice is made, a new level of choices is exhibited as a sub-menu. (Example: the UCSD Pascal interface).
- (d) a graphic interface with windows and iconics, as in the Xerox Star.

Some interfaces may make use of mixtures of the capabilities given in (a) through (d), as in a graphics interface which uses iconics to pop up a menu, which in turn selects a tool, which tool, in turn, asks the user to fill out a form or to conduct a conversation consisting of commands and responses.

One might be able to prescribe a handful (say seven to ten) of standard interface methods, designed to cooperate consistently on the use of universal, pervasive signals, such as a "suspend execution signal" or a signal to obtain help. One could furnish a library of reusable software parts (written in Ada), which parts could be used to create a tool interface according to one of the standard methods. For example, in creating a form-filling interface, one might have parts to "get a date from the user" or "get an integer from the user." One might have "application generators" that, for example, conduct a conversation with the user to extract, say, the contents of menus in a tree-of-menus, or the visual appearance of a form with holes to be filled in. The "names" of the holes in the form or the items in the tree of menus might be tied to certain tables of actions: actions giving error messages, actions giving help, and data validating actions. The application generator would "interview" the programmer to extract all the information needed to generate a "complete" interface, and would generate an Ada program to act as a complete instance of one of the standard tool interfaces (forms, menus, conversations, iconics, or whatever).



If one could make the addition of a new tool interface very easy technically, through the use of libraries of reusable interface components and application generators, then it might be reasonable to put the growth of APSE toolset interfaces under configuration management using a policy somewhat like the following.

"We, the APSE tool interface configuration management authority, will accept a new tool to be put into the standard APSE tool library provided its interface obeys one of the given standard interface methods to the letter. If this is not suitable and you believe you have a new interface method to contribute along with your proposed tool, we will make you go through the procedure for adding a new standard interface method. Only when you have succeeded in adding your new method to the list of approved standard methods, would we permit you to use your method, and in that case, everybody else can use your method, too. But there are some restrictions. A new method must be judged sufficiently new and beneficial, in comparison to the existing methods, that the authority believes it will bring in more benefits from its use than it costs in the diversification of methods and the imposition of new learning burdens on the APSE user community. It must be brought into consistency and harmony with the existing methods, insofar as practicable. And reusable parts and application generators must be supplied to make it easy for others to add tools using the new interface method."

Were it feasible to provide such a policy and its required technical support, it might be possible substantially to meet the letter and spirit of the STONEMAN simplicity requirement, regarding controlled extension of APSE toolsets. In the absence of some means for controlling the growth of APSE toolsets and enforcing some consistency in tool interfaces, we are likely to witness a recapitulation of the current state of affairs with the growth of operating system toolsets --- namely, random, haphazard, inconsistent growth, with its detrimental effects.

A step that we could take to investigate extensibility further, should the KITIA decide it is worthwhile to pursue it further, would be to draft an INTERFACEMAN document, giving requirements for APSE toolset interfaces. This is, in a sense, a continuation of the STONEMAN exercise, and amounts to our attempting responsibly to carry out the intent of STONEMAN in this area.

If, after circulation and tuning, INTERFACEMAN were to be judged a worthwhile pursuit, it might be possible to put together draft configuration management plans and a prototype reusable software component library to support APSE toolset extensibility. After further evaluation, trial use, and tuning, this could eventually lead to a widely adopted shared agreement about a beneficial way of doing business with respect to the use of APSE tools.

Date: 10 Mar 82

KAPSE INTERFACE CATEGORY: G.1 Debugger Support (RTS)

(KAPSE INTERFACE GROUP: 3. KAPSE Services)

#### 1.1. EXPLANATION:

An Ada symbolic debugger will place certain requirements on the Ada run-time system, and consequently the KAPSE, not otherwise required within the APSE. Certain information, normally hidden within the RTS and "implementation dependent", must be accessible to a debugger. Such information may include, but is not limited to, information in scheduler tables, implementation details of the stack model, access methods to data, and mechanisms for "breakpointing" control to the debugger. Debuggers will also need to "influence scheduling" itself. For example when a breakpoint is reached, execution of the Ada program may need to be totally or partially suspended.

#### 2.1. KEY ISSUES:

- o What partitioning between the Debugger tool and the RTS will allow its use for varying targets.
- o What partitioning between the Debugger tool and the RTS will allow its use for varying host-target architectures (e.g. simulated target, connected target, simulated environment, etc.).
- o What requirements must be placed on the RTS to allow debugger functionality.
- o Do proposed approaches allow for advanced concepts in debuggers which address tasking? What additional RTS requirements would these impose.
- o What requirements must be placed on the linker to provide binding information necessary for debugging? What RTS requirements must be satisfied for cases of execution-time binding (for debugger access to objects/code)?
- o Debugger/Compiler Interaction
  - Is it reasonable to require that debuggers work in the presence of optimization?
  - Should certain PRAGMAs be required of compilers?
  - What effects do the above considerations have on run-time efficiency in the absence of the debugger (i.e. operational programs).
  - What impact would such pragmas have on recompilation checking stra-

tegies in support of Ada's separate compilation model.

G3.3. RELEVANCE:

G4.1. PRIORITY:

G5.1. RELATED CATEGORIES:

G6.1. PROPOSED APPROACHES:

6a. ALS:

6b. AIE:

6c. Existing Standards:

6d. Other:

G7.1. RISK:

G8.1. ACTIONS TO TAKE:

(Preliminary formulation of long-term KIT approach:)

# THE CONSEQUENCES OF MULTIPLE DOD KAPSE EFFORTS

Douglas E. Wrege  
CONTROL DATA CORPORATION

## Introduction

The KAPSE Interface Team (KIT) was organized by the AJPO to identify and document requirements for transportability and interoperability of tools and data bases between Ada Programming Support Environments (APSEs). In conjunction with the KIT, a team of industry and academic representatives (KITIA) was established whose purpose is to present questions and issues, alternatives and recommendations concerning this interface effort, as well as to serve as a technical resource to the KIT. As we of the Industry Academia Team undertake our charge, we would like to make a point about the current direction taken by the Ada Program. The existence of multiple DoD- sponsored APSEs may be deleterious to the Ada effort.

## Background Information

The major goals of the DoD common high-order language effort are to address the problem of software life-cycle costs, and to improve the quality and reliability of software for embedded computer applications. The DoD recognized that in order to address the goal of developing cost-effective, high quality software, one must also promote the development of portable software -- especially portable software tools. In short, the concept of portability is seen as crucial for improved software development [STONEMAN 2.B.10].

Portability requires the definition of some framework within which tools will execute. Accordingly, STONEMAN defined three distinct layers in the environment: The Kernal Ada Programming Support Environment (KAPSE), the Minimal APSE (MAPSE), and the APSE. The KAPSE is viewed as central to portability [STONEMAN B.11]. The intent is that any program, using only standard Ada constructs and making external reference only to the KAPSE, be portable to any host. This can be achieved provided that KAPSE interfaces are identical across environments. Thus a standardized KAPSE interface is critical to the development of portable tools. STONEMAN indeed anticipated a standard KAPSE interface, and, moreover, expected a single standard to emerge [STONEMAN 2.B.13; 2.C.9].

However, the DoD is currently funding the development of two APSEs: The Army Ada Language System (ALS) and the Air Force Ada Integrated Environment (AIE). As presently envisioned, these environments will have different KAPSE interface definitions.

## Multiple APSEs

The existence of several APSEs is an expected consequence of progress. It is envisioned that users will debug, update, extend and otherwise manipulate the tools supported by an interim standard APSE. Indeed, its success depends on such activities to uncover latent problems, to test tool portability and interoperability, and to discover new and more effective applications for software tools -- in this way driving the concept of a programming support environment forward towards the next APSE implementation. Thus, more than one APSE may be in use for a short time, as users shift their attention from one standard APSE to its newly released successor -- much as the large operating systems of the 1960's and 1970's evolved.

What is unexpected, and almost surely counterproductive, is the concurrent promulgation of two or more formally sanctioned APSEs, each of which is viewed as a standard by its developing community, and each of which is fundamentally incompatible with the other. The fact is that there will be at least two such APSEs -- the Army's ALS, and the Air Force's AIE -- undergoing simultaneous testing. There will be, as a result, some positive, but probably more negative consequences for the Ada experiment as a whole. The following discussion articulates some of the issues and potential problems injected into the Ada initiative by the existence of both the ALS and the AIE.

## Is a Single Environment Crucial?

By the phrase "single environment" we mean APSE. Given that there is a single environment, there would necessarily be a single MAPSE and a single KAPSE.

PRO: With a single environment there would be optimal portability of both tools and programmers. As non-DoD sponsored APSEs are developed there will be maximum pressure to conform to the government APSE, thus promoting a common way of developing and supporting software.

CON: Forcing the use of a single APSE will tend to cause the evolution of environments to stagnate. Certainly the use of environments is immature enough that there is no clear agreement as to the components of a full APSE. Even STONEMAN expected a wide variety of different APSEs to exist, exhibiting, for example, different programming design methodologies. What is crucial to the Ada Initiative is portability of tools (thus affording availability of sophisticated environments), which can be achieved through the porting of the Command Language Interpreter (CLI) or standardizing on that tool as a part of the MAPSE. Advocates of a single APSE are really asking for a preliminary MAPSE and a full MAPSE or don't understand the term APSE.

## Recommendations

The arguments for multiple APSEs are decisive. Portability of tools and programmers can be achieved without difficulty.

### Is A Single MAPSE Crucial?

STONEMAN defines the MAPSE as the APSE which provides a minimal but useful Ada programming environment and supports its own extension with new tools written in Ada. Unfortunately, the question of whether a single MAPSE is crucial is somewhat clouded by different assumptions about what should be included in the MAPSE and which tools should be considered as part of the APSE. In this discussion the set of tools which must exist in every APSE will be used to define the MAPSE, viz. Ada Compiler, Linker, CLI, Library Manager (configuration manager), and Debugger. If a single MAPSE is crucial, we will assume that the KAPSE upon which it is built is also crucial.

PRO: STONEMAN expected that one MAPSE, and its kernal (KAPSE), would become a de facto convention. Although STONEMAN expected that eventually the KAPSE specifications would be considered for standardization within DoD, it did not indicate that the MAPSE would be a candidate for standardization also. Proponents of a standard MAPSE view the STONEMAN as not going far enough since it did not foresee the proliferation of environments via multiple DoD developments. A single MAPSE will facilitate tool portability and interoperability since the basic tools will be standard and there will be a single KAPSE. There will tend to be a basic set of data structures produced by these fundamental tools providing de facto standards for inter-tool communications. Programmer portability will be provided for through a common CLI, i.e. the user interface being part of the MAPSE will be standard.

CON: STONEMAN expected that a KAPSE specification would be considered for standardization within the DoD through the emergence of a de facto standard MAPSE. This was the only reason for mentioning the MAPSE in a paragraph discussing standardization. The MAPSE was intentionally not considered for standardization. As long as sufficient KAPSE commonality exists to satisfy the goal of tool portability, there is no need to standardize on a MAPSE. Indeed, standardizing on a MAPSE is counter-productive as the MAPSE needs to be fluid, allowing incorporation of continually improved MAPSE tools. It is true that some standards regarding inter-tool communication data structures need to be established so that individual tools can be effectively ported to an APSE, but this can be achieved without standardizing on a particular MAPSE. Programmer portability can still be provided by porting the CLI tool between APSEs, and maintaining multiple CLI tools within a single environment.

### Recommendations

Do not standardize at the MAPSE level, however, encourage the emergence of a de facto standard MAPSE through DoD wide adoption of a preferred set of basic tools. The intent is to maintain flexibility in the makeup of the MAPSE tools yet maintain uniformity between individual services.

### Is a Single KAPSE Crucial?

PRO: All evidence supports some form of standardization at the KAPSE level. STONEMAN clearly states that "it is intended that conventions and, eventually, standards be developed at the KAPSE interface level." The Memorandum of Agreement (MOA) establishing the KAPSE Interface Team (KIT) sets the goal of "establish[ing] the necessary interface conventions so that multiple [APSE] efforts may converge to a single set of interface standards in the 1985 time frame." Initial KIT and KITIA requirements for interoperability and transportability of tools revolve around establishing a Standard Interface Set (SIS) which will become a kernel for all KAPSEs, i.e. a conforming KAPSE will implement all of the SIS but may implement a superset of the SIS. A requirement will be that the "SIS should be the same everywhere and for everyone."

CON: Nil.

Controversy centers on the questions of: when is "eventually", how strong should the standards be, and does the DoD have a viable plan which will allow such commonality to be achieved. The remainder of this discussion centers around the last question.

### The Stoneman Plan

STONEMAN set forth requirements for an Ada Programming Support Environment and suggested the MAPSE/KAPSE approach as a mechanism for achieving those requirements. It was never intended to set forth a specific plan for the acquisition of an APSE, however, the STONEMAN was influenced by an "expected" strategy. STONEMAN's 'Strategy for Advancement' [2.C.7] states "Progress towards the long-term goal of a wide measure of portability is expected to be achieved by a process of competitive design and evaluation of APSEs and their associated KAPSEs."

The plan was for the Air Force to act as the lead service in the development of an integrated software environment for Ada. The Army was to develop a few critical tools to enable the use of Ada in the context of existing software environments, i.e. to provide an interim solution [STONEMAN, Preface].

### The Situation Today: The Stoneman Plan O.B.E.

The current situation is that the Army is about to introduce the ALS, a complete APSE based on the MAPSE/KAPSE approach. The Air Force, meanwhile, has contracted for the AIE following a competitive design of three candidate APSEs with public evaluation and selection of a winner. Thus, it appears that there will be the concurrent promulgation of two formally sanctioned APSEs, each of which is viewed as standard by its developers.



There are a number of important problems with the "two-standards" approach. There is a high risk factor associated with STONEMAN's expectation that one MAPSE and its basic kernel or KAPSE would achieve a sufficiently wide measure of acceptance for it to become a de facto standard [STONEMAN 2.C.9]. Certainly, there is no precedence for the emergence of de facto standards from the private sector without incentives imposed from outside (i.e. the Government). Thus the existence of two DoD APSEs will diffuse any tendencies toward commonality of KAPSEs or MAPSEs outside of DoD. The lack of a uniform position within DoD regarding APSE use is probably a sure way to guarantee that no de facto standards emerge. The government's posture tends to be magnified in the private sector; uniformity within DoD may lead to commonality in the private sector, but a two-APSE approach will surely lead to a total lack of commonality in industry. Let us examine issues bearing on the two-standards situation.

1. Are the ALS and AIE research prototypes or production-quality final systems?

If they are prototypes, it is advantageous to have both developed since we will gain valuable experience in practicalities regarding the makeup of an APSE as well as in the features necessary to support them (the KAPSE). We will certainly be better able to build an APSE "right" later, given the opportunity. If they are production systems, it is probably counterproductive to have both as gained experience will come too late.

It appears that both the Army and Softech view the ALS as a production system. Army embedded system developments are to begin using Ada in 1983. All indications are that the Air Force views the AIE as a production system.

2. Will the Army and the Air Force adhere to and promote their systems to the exclusion of all others?

If they do, there is always a natural tendency to adopt one's own, if only to make good on initial investment. Initially, factors such as the particular host machine upon which the environment is implemented and which code generators exist will dictate the environment to us. At first, the ALS will be adopted in the Army if only as a result of VAXs being available. The Air Force with inventories of IBM and PE 8/32s will use the AIE. As an environment is used there will be an increasing functional dependency on that environment, as problems are solved, and as users develop work habits that include the new capabilities. Economic dependencies on the given environment will increase as start-up and recurrent costs (debugging, reimplementation, retargeting, etc.) accrue, and as return on investment becomes a critical factor. As user groups, unique support facilities, special products, etc., grow, based on the particular features of the ALS or AIE, incentives will decrease to ever move to a future standard environment. Past experience indicates that once entrenched in a way of doing business, there is always a tendency to resist change.

On the other hand, Wright-Patterson AFB has a contract to build a 1750A code generator for the ALS, thus the Air Force will experience at least some use of the ALS. The Navy is planning to use the ALS as a baseline for their APSE, thus the ALS may become a de facto standard if only because two out of three of the major Services are using it.

Thus, the indications are that if the Air Force initially adopts the AIE to any uniform extent then the two-standard situation will result. The Army will almost certainly adopt the ALS to the exclusion of the AIE, given the current situation.

#### Current Efforts To Remedy The Situation

A Memorandum of Agreement was signed by the Assistant Secretaries of the Army, Navy, and Air Force recognizing the multiple KAPSE problem and defining the long term goal of establishing a single set of interface standards for the KAPSE. The MOA resulted in the formation of the KIT and KITIA to evolve these standards.

The developers of the AIE, Intermetrics, have been tasked to examine the ALS and endeavor to incorporate commonality, wherever possible, as part of their development effort. The current feeling among KITIA members is that the data base structures of the ALS and AIE are sufficiently different that the KAPSEs will necessarily be incompatible. For the AIE to adopt the ALS database would be tantamount to ignoring the very features that were the determining factors in the competitive selection of Intermetrics as the AIE contractor. If the AIE were to adopt the ALS database, AIE development would result in "re-inventing the wheel" and largely be a waste of scarce resources.

It is the author's feeling that these efforts are not sufficient to achieve the benefits expected from the Ada Initiative. Assume that the KIT and KITIA do establish a generally agreed upon set of requirements for transportability and interoperability of APSE tools. Both the ALS and AIE will probably satisfy these criteria, with the exception of requirements like:

The Standard Interface Set shall be the same everywhere and for everyone.

This requirement could not be satisfied just because there are two KAPSE definitions, one for the ALS and one for the AIE. Either an arbitrary choice must be made, or somehow the "better" environment must be determined. The latter case will most assuredly result in a largely political battle, introducing delays sufficient to cause the two environments to become entrenched, i.e. no resolution will be possible. The services are sufficiently autonomous that it is difficult to imagine who could or would be willing to make the arbitrary choice.

The statement has been made in the KITIA that "We would be better off with the worse of the two environments than with both." The point here is not to pick the poorer of the two, but that the undesirable consequences are so great that an arbitrary decision is preferable to the two-standards approach even if the chosen environment is inferior. We must not allow the situation to proceed to the point where two accepted environments exist within DoD.

There is a slim chance that "everything will turn out all right." The ALS or AIE may prove to be so superior to the other that one becomes a de facto standard. One may exhibit sufficient superiority in adaptability that it evolves into a clearly preferable environment. The ALS may become so widespread due to its earlier release (and the fact that the Navy is adopting it as a baseline) that the AIE never has a chance. Intermetrics may figure out how to make the AIE a strict superset of the ALS in which case it will be both superior and could easily supplant the ALS. But we cannot afford the risk, considering the consequences.

#### Recommendations

The DoD must develop a clear plan resulting in a single KAPSE definition for common use within the government. This policy must be approved at an extremely high level and must have the necessary power of enforcement. The following are some of the possible plans:

1. Designate the ALS as the baseline APSE to be used by all Services. View the AIE as a research prototype to investigate evolutionary directions for the ALS.
2. Designate the ALS as the baseline APSE to be used by all Services. Change the AIE contract so that a new APSE is developed based on the ALS, for mandated replacement of the ALS.
3. Designate the ALS as an interim APSE with mandated replacement by the AIE when available, but do not constrain the AIE design to be compatible with the ALS.
4. Evolve the Navy APSE from the ALS, to be the designated APSE. View the AIE as a research prototype. This is similar to 1 above.
5. Have a competition between the ALS and AIE to pick one. Mandate use of the selected environment. This may not be viable due to the difference in development schedules.
6. View both as prototypes. Establish a requirement that a new APSE will be developed based on knowledge gained from the ALS and AIE, and mandate the adoption of the new APSE. This approach will tend to slow the widespread availability of Ada outside of the DoD and hence, delay its introduction and widespread use.

There are surely many other alternatives. All of the above should benefit from the standards/conventions/requirements developed by the KIT effort. The point to be gleaned from the above alternatives is to adopt a plan to establish a standard KAPSE definition with sufficient strength to do so. We cannot allow the opportunities possible within the Ada Initiative to be Overcome By Events.

# KAPSE SERVICES - EXPANDED OUTLINE

Herb Willman  
RAYTHEON COMPANY

This paper is an expanded outline for the KAPSE Services Sections of the KAPSE Interface Report for the KAPSE Interface Team of the Ada Joint Program Office. It contains the outline for Sections G (Ada Program Run-time System), H (Bindings and Their Effect on Tools), and I (Performance Measurement).

## 1. ADA Program Run Time System (RTS) - Section G

### 1.1. G1. Explanation

The Ada RTS provides the basic run-time support facilities that are required by Ada programs that execute within the APSE and the non-host target environment. Ada programs operate (execute) in two potentially different environments. The first is the self-hosted target environment in which the RTS is part of the APSE environment itself (i.e. the host and object machines are the same). The second is the "remote" target environment in which the target system is different than the host and requires cross-compilation. In the second case the RTS is not a part of an APSE and must be "stand-alone".

These facilities may include, but are not limited to, closed routine provisions for:

- Multitasking Support
- Exception Handling
- Arithmetic Operations
- Type Conversions
- String Operations
- Structure Operations
- Program Initiation
- Program Termination
- Standard Input/Output
- Resource Allocation
- Debugging Support
- Performance Measurement Support

---

Ada is a trademark of the US Department of Defense

### 1.1.1. Multitasking Support

Task ACCEPT  
Task execution initiation  
Task execution suspension  
~~Task execution termination~~  
Task DELAY  
Task scheduling (PRIORITY)  
Task STORAGE\_SIZE  
Task entry 'COUNT  
Task ABORT  
Task Exceptions  
    TASKING\_ERROR  
        Attempt to call a terminated task  
    CONSTRAINT\_ERROR  
        Out of range index for family identification  
Task termination by exception  
    Incomplete rendezvous termination  
    Non-locally handled exceptions in ACCEPT  
SELECT\_ERROR  
    All alternatives closed and no ELSE part  
FAILURE  
    Raised by another Task

Pragmatics

### 1.1.2. EXCEPTION HANDLING

CONSTRAINT\_ERROR  
NUMERIC\_ERROR  
STORAGE\_ERROR  
TASKING\_ERROR  
SELECT\_ERROR  
FAILURE

Pragmatics

### 1.1.3. ARITHMETIC OPERATIONS

Necessary Operations (may include NUMERIC\_ERROR checks)

All included in Package STANDARD

Desirable Operations (may include CONSTRAINT\_ERROR checks)

sqr (integer)  
sqr (real)  
sort (real)

|             |        |
|-------------|--------|
| sin (real)  | Units? |
| cos (real)  | "      |
| tan (real)  | "      |
| asin (real) | "      |
| acos (real) | "      |
| atan (real) | "      |

Pragmatics

### 1.1.4. TYPE CONVERSIONS

Numeric Conversions

integer to/from fixed  
integer to/from real  
fixed to/from real

### 1.1.5. STRING OPERATIONS

Move Strings

Byte Aligned

Not Byte Aligned

Catenate Strings

Byte Aligned

Not Byte Aligned

Compare Strings

Byte Aligned

Not Byte Aligned

Pragmatics

1.1.6. STRUCTURE OPERATIONS

Move Record/Array  
Compare Record/Array

1.1.7. PROGRAM INITIATION

Initiate Program  
    Initialize, as necessary  
    Support Debugging  
    Support Performance Measurement

1.1.8. PROGRAM TERMINATION

Terminate Program  
    Report Status, as necessary  
    Report Exceptions, as necessary  
    Support Debugging  
    Support Performance Measurement

1.1.9. STANDARD INPUT/OUTPUT

Generic Package INPUT\_OUTPUT  
Package TEXT\_IO  
Package LOW\_LEVEL\_IO

1.1.10. RESOURCE ALLOCATION

Resource Allocation at the PTS level - What is required?



#### 1.1.11. DEBUGGING SUPPORT

##### Non-symbolic Basic Debugging

- Memory Dump
- Registers Dump
- Trace
  - All
  - Control Path Only
- Snapshot
- Breakpoint Insert
- Breakpoint
- Breakpoint Remove

##### Symbolic Debugging

- Trace
  - All
  - Control Path Only
- Snapshot
- Breakpoint Insert
- Breakpoint
- Breakpoint Remove

##### Pragmatics

#### 1.1.12. PERFORMANCE MEASUREMENT SUPPORT

There are three types of interfaces which support performance measurement in computer systems:

- Hardware only
- Hardware/software
- Software only

For the purposes of the KAPSE Specification, it seems practical to concern ourselves with only the last type, viz. a software only interface. Performance Measurement through the use of software may be accomplished by the use of:

- Predefined Hooks
- Implanted Hooks
- Dummy Loads, periodically sampled
- Other techniques?

## 1.2. G2. Key Issues

### 1.2.1. Specification

Level of the Specification

A Level (Stoneman)?  
Interface Specification?  
B5 Level?

### 1.2.2. Access Control and Usage Restrictions

Command Language Level  
Tool Level

### 1.2.3. RTS/Other KAPSE Interfaces

How does RTS interface with other KAPSE elements? What are the interfaces? Any standardization? Any similarity between tool and command level interfaces?

### 1.2.4. Distributed Systems Considerations

How do we specify in such a way as to allow the interfaces to RTS support uniprocessor, multiprocessor, and distributed approaches?

### 1.2.5. Security Considerations

How do we specify in such a way as to not impact the future addition of security requirements on the RTS?

## 1.3. G3. Relevance

High. This area alone could significantly impact the portability of Ada programs in the DoD application environment.

## 1.4. G4. Priority

High.

## 1.5. G5. Related Categories

- A. Program Invocation and Control.
- C. Device Interactions.
- E. Inter-tool Data Interfaces.
- H. Bindings and Their Effect on Tools.
- I. Performance Measurement.
- J. Recovery Mechanisms.
- K. Distributed APSE's.
- L. Security.
- M. Support for Targets.
- N. Pragma & Other Tool Controls.

1.6. G6. Proposed Approaches

1.6.1. Ada Language System

1.6.2. Ada Integrated Environment

1.6.3. Existing Standards

None.

1.6.4. Other

1.7. G7. Risk

Probably High. To early to assess.

1.8. G8. Actions to Take

Establish a suitable requirements document and interface specification.

## 2. H. Bindings and Their Effect on Tools

### 2.1. H1. Explanation

Binding is the assigning of a value or referent to an identifier. We are concerned with link- or execution-time assignment of concrete subprograms, devices, and database objects to identifiers in Ada programs. In military applications, Ada must be capable of supporting the full range of binding possibilities. Ada programs must be capable of being bound completely, partially, or loosely. Complete binding means that the link-edit-load process must be capable of supporting a predefined resource allocation (static binding) for an application, which typically uses the complete target machine. Partial binding means that the link-edit-load process makes decisions about resource allocation during the link-edit process. Loose binding means that only the loader and target operating system or executive are concerned with the resource allocation (dynamic binding).

### 2.2. H2. Key Issues

#### 2.2.1. Specification of Binding

Level of the Specification  
A Level (Stoneman)?  
Interface Specification?  
B5 Level Specification?

What operations must be supported by the KAPSE regarding diagnostic (debugging), interpretive, and full speed execution of programs incorporating static and dynamic binding to subprograms and data?

Static binding?

Dynamic binding?

Tradeoffs in binding?

#### 2.2.2. System Recovery

How do we specify in such a way as to not impact system recovery?

#### 2.2.3. Distributed Systems

How do we specify in such a way as to support uniprocessor, multiprocessor, and distributed approaches?

#### 2.2.4. Security

How do we specify in such a way as to not impact the future addition of security requirements?

#### 2.2.5. Growth

How do we specify in such a way as to not limit the capabilities for target dependent functionality?

#### 2.3. H3. Relevance

Moderate.

#### 2.4. H4. Priority

Moderate.

#### 2.5. H5. Related Categories

- A. Program Invocation and Control.
- C. Device Interaction.
- E. Inter-tool Data Interfaces.
- G. Ada Program Run-Time System (RTS).
- I. Performance Measurement.
- J. Recovery Mechanisms.
- K. Distributed APSE's.
- L. Security.
- M. Support for Targets.
- N. Pragmas & Other Tool Controls.

#### 2.6. H6. Proposed Approaches

##### 2.6.1. Ada Language System

##### 2.6.2. Ada Integrated Environment

##### 2.6.3. Existing Standards

##### 2.6.4. Other

#### 2.7. H7. Risk

Moderate.

2.8. H8. Actions to Take

Establish suitable requirements and interface specifications.

### 3. 1. Performance Measurement

#### 3.1. 11. Explanation

Performance Measurement is the selection, collection, and recording of dynamic information on APSE activity and resource use as a function of time. From this follows performance evaluation, which is the technical assessment of a system (APSE) or system component (APSE tool or KAPSE component) to determine how effectively operating objectives have been achieved.

#### 3.2. 12. Key Issues

##### 3.2.1. Specification

Level of the Specification

A Level (Stoneman Update)?  
Interface Specification?  
B5 Level?

What mechanisms should exist for the selection, collection, and recording of dynamic information on resource use as a function of time?

What mechanisms should exist for tuning KAPSE characteristics based on such information?

What access should exist to KAPSE software, host operating systems, and host hardware to support performance measurement and tuning?

What mechanisms should exist to support event recording? Examples of such events are:

Logon/Logoff  
Program initiation/termination  
Tool invocation  
Scheduling information  
Input/output activity  
Resource allocation/activity  
Multitasking activity  
Exception processing  
Command level activity

### 3.3. I3. Relevance

Moderate to Low. This area is important, but not critical. Performance measurement and fine tuning may be possible via the underlying operating system for those APSE's that have one.

### 3.4. I4. Priority

Low.

### 3.5. I5. Related Categories

- A. Program Invocation and Control.
- B. Logon/Logoff Services.
- C. Device Interactions.
- D. Database Services.
- E. Inter-tool Data Interfaces.
- G. Ada Program Run-Time System (RTS).
- H. Bindings and Their Effect on Tools.
- J. Recovery Mechanisms.
- K. Distributed APSE's.
- L. Security.
- M. Support for Targets.
- N. Pragma and Other Tool Controls.

### 3.6. I6. Proposed Approaches

#### 3.6.1. Ada Language System

#### 3.6.2. Ada Integrated Environment

#### 3.6.3. Existing Standards

#### 3.6.4. Other

### 3.7. I7. Risk

Low.

### 3.8. Actions to Take

Establish suitable requirements and interface specifications.



# AN EXECUTIVE SUMMARY OF THE TOPS - 20 OPERATING SYSTEM CALLS

Thomas A. Standish  
COMPUTER SCIENCE DEPARTMENT  
UNIVERSITY OF CALIFORNIA

## Abstract

This brief document gives a high-level, executive summary of the principal features, and organization of the TOPS-20 system of operating system services.

## Introduction

Tops-20 provides its users with a system of operating system service calls. These calls are expressed in machine language using JSYS (or Jump to SYStem) commands. JSYS commands consist of a DEC-20 special instruction code whose address field is set to contain the numerical code for the particular service being requested. Arguments are passed to the service routines in accumulators or using pointers in accumulators. Results are returned in the same fashion. There are error return conventions, which specify, for example, that erroneous returns occur at the instruction location immediately after the call, and successful returns occur at the second location after the location of the call.

Underlying the set of operating system services is a set of models for: (a) Processes, (b) Files, (c) Interrupt Handling, (d) Interprocess Communication, and (e) Contention for Resources. These models are machine-independent, in the sense that they could be implemented, in principle, on non-DEC equipment. But if one did so

it is not at all clear that one could escape the bias toward a particular manufacturer's operating system that we seek. The value, then, in reading briefly about TOPS-20 services in relation to our effort may lie in the fact that it suggests that a system of KAPSE Virtual Operating System Services (VOSSes) might have to have an underlying model specifying the details of processes, communication, files (together with directories, protection, versions, etc.), a software interrupt handling system, and a means for inter-process communication.

Let's proceed, then, to give thumbnail sketches of each of the five components of the overall TOPS-20 model:

#### The Process Model

A process in TOPS-20 is an executable entity that is associated with its own (virtual) address space for programs and data. A process may spawn children (called inferior processes, or sometimes, "forks"). A given process may have more than one inferior process, and these children may execute in a concurrent (or interleaved) fashion. Children may, in turn, spawn their own children processes, giving, as it were, a family tree of processes with one superior process at the top. In general, processes can control their descendants and themselves, but cannot control their ancestors.

Processes can communicate by a variety of methods: (a) they can share pages of memory so each can see what the others are reading and writing, (b) they can pass messages (or information packets) using inter-process communication services furnished by the monitor, (c) they can raise interrupts which are serviced by interrupt service routines in the others, and (d) they can read and write files in the file system.

For example, when a user logs on to the system and is assigned a job, he is put in contact with a process that executes EXEC-level commands (i. e., operating system commands). If the user runs a tool or runs one of his own programs, an inferior process, running under the EXEC, is established with its own fresh address space. If the inferior process reaches an error condition, an interrupt is generated which is serviced by the superior EXEC-level process, allowing the user to kill, re-enter, or continue the job. Also, user-generated interrupts (such as hitting Control-C) are serviced by the suspended EXEC, and can generate the action of suspending the execution of the inferior process.

#### The File Model

Files for a given user are stored in that user's directory with associated protection codes, account numbers, version codes, creation dates, etc. A TOPS-20 file name is of the form:

<USER> NAME. EXTENSION. VERSION\_NUMBER; Protection; Account

A process that needs to read or write a file has to call an operating system service that accepts a string giving the file name and returns a Job File Number (JFN). This JFN acts as a "handle" on the file. The process then opens the file, reads or writes it, and closes the file. Files on disk have byte pointers that give random access to bytes in the file. Modes of transacting with files are: (a) sequential byte-by-byte, (b) in a multiple byte or string manner, (c) in a random byte-by-byte manner, and (d) in a page-mapping manner.

There are many details of the protection system and simultaneous file usage convention system not covered in this brief summary. (Nonetheless, these issues must be dealt with in a mature set of conventions for a set of Standard KAPSE Operating System Services).

## Interrupt Handling

TOPS-20 provides a system of software interrupts. There are thirty-six software interrupt channels. Each process can set up its own interrupt handling table. To each of the thirty-six channels can be associated: (a) a type of interrupt, such as the machine-generated floating point overflow interrupt, or the user-generated Control-C signal, (b) an interrupt service routine (called when an interrupt is handled), (c) a priority code (either 1, 2, or 3 with 1 the highest priority), and (d) an enabling bit to say whether interrupts on the given channel will be handled or ignored. The TOPS-20 operating system queues up interrupts waiting service on the channels for a given job that are enabled to service interrupts.

## Interprocess Communication

Processes can obtain PINs (Process Identification Numbers) of other processes, and using those PINs (which are furnished by calling the operating system), they can call services which send each other buffered messages (or information packets). This is, so to speak, a mail system implemented by the operating system service model.

## Contention For Resources

Finally, TOPS-20 provides an ENQueueing/DEQueueing model for processes to use that must share and contend for resources (such as line printers). A process can queue up a request to use a device, and when its request is granted, it seizes the device, uses it, then releases it. A process waiting to use a device can query the status of the usage queue to see whether it is available.

## Conclusion

If we are going to furnish a machine-independent, non-manufacturer-biased set of KAPSE VOSSes (Virtual Operating System Services), we probably have to have a crisp, clean, precise, well-thought-out model that covers processes, interrupt handling, interprocess communication, resource contention, and the file system.

# **KAPSE Interface Team Industry and Academia (KITIA)**

S. Glaseman  
J. Beane  
D. Cornhill  
E. Griesheimer  
D. Loveman  
J. Ruby  
R. Westerman  
L. Yelowitz

## **1.0 INTRODUCTION AND SUMMARY**

### **1.1 Objectives of This Document**

This document represents the first formal output from Group IV of the KAPSE Interface Team - Industry and Academia (KITIA). Group IV is concerned with a variety of issues, each of which embodies questions that cut across the boundaries of the other KITIA subgroups. Group IV's issue areas currently include:

- o APSE Support for Targets
- o APSE Recovery Mechanisms
- o APSE Extensibility
- o Ada/APSE Program Policy Issues
- o Distributed APSEs
- o APSE Security
- o Pragmas and other APSE Tool Controls

One objective of this document is to present, for each of these areas, what has been learned thus far that is of relevance to APSE interoperability and transportability. This objective is met by a position paper for each area. Each paper takes a firm stand on the area in question, and sets forth its arguments as though the issues were being laid to rest instead of barely opened. The purpose is to clearly display the status of our analyses, and to provide hard targets for discussion and criticism. A second objective is to develop, in each area, a set of recommendations in the form of guidelines, requirements, and activities needed for moving closer to interoperable and transportable APSE tools and data bases. Each position paper is followed by a set of applicable recommendations. As before, these are stated with unwarranted assurance in order to inspire focused debate.

## **1.2 Sources of Information**

In the interest of providing a complete record of the information available to our analyses, each section of the document will contain a list of references. Common to all sections are the following documents:

- o Requirements for Ada Programming Support Environments: "STONEMAN", Department of Defense, February 1980.
- o Reference Manual for the Ada Programming Language, U.S. Department of Defense, July 1980.
- o KITI Working Session: KAPSE Interface Worksheets, KAPSE Interface Team, February 1982.
- o Charter: KAPSE Interface Team - Industry and Academia, March 1982.

### 1.3 Document Overview

Sections II through VIII contain position papers on each of the Subgroup IV issue areas listed earlier. Each paper will adhere to the same format: a topic statement defining the issue in question; a discussion of the issue from the perspective of interoperability and transportability; a summary of recommendations that were explicitly or implicitly referenced in the discussion; and a list of pertinent references.

Section IX then presents the currently envisioned future activities of Subgroup IV as an aid to better inter-group coordination on the KITIA during 1983.

### 1.4 Summary of Results.

The papers contained herein present, in explicit or implicit form, a number of recommendations aimed at both furthering our grasp of currently known issues pertinent to APSE interoperability and transportability, and exploring new issues as these emerge from our efforts. These recommendations, in the form of suggested activities, guidelines, and requirements, are summarized below.

1.4.1 APSE Support for Targets. Two general guidelines emerge, thus far, from Group IVs work in this area.

1. Consider, as part of a target support system, a three part, target resident monitor. The three parts would comprise a kernel, a resource manager, and a MAPSE-like tool set.
2. The session layer of the OSI model (see reference in text) should be considered in designing the host-target communications interface that would be part of the target monitor's kernel.



1.4.2 APSE Recovery Mechanisms. We have identified five guidelines in this area.

1. Failure of any part of the database should not compromise the utility of the remaining structure.
2. The system's resolution of all recovery actions, complete or partial, should be communicated to all relevant users. Recovery processing should include feedback to users as to the nature of the failure and of the resources known to be presumed to be, and potentially impacted.
3. Recovery from any failure involving user-resource connections should include recertification. At a minimum, recovery from a dropped remote line should require password recertification.
4. Wherever possible, recovery processing should be localized to maintain transparency for the unaffected user population.
5. Equivalent failures on different APSEs should result in equivalent recovery processing and post-processing status.

1.4.3 APSE Extensibility. Thus far Group IV has identified one recommended activity and one guideline in the area of Extensibility.

1. Recommended action: Re-interpret STONEMAN intentions as regards a fixed KAPSE interface - especially in light of a rapidly expanding tool environment. Can a fixed interface be expected to provide services for the range of tools that can reasonably be postulated?
2. Guideline: An APSE must support the front-end activities of the software life-cycle as well as the later software development and test activities that are now the focus of APSE tool efforts.

1.4.4 Ada/APSE Program Policy Issues. This is an area having to do primarily with questions of implementation and management of the Ada experiment. The issues involved are thought to bear importantly on the long range benefits, both technical and economic, that Ada could bring to the software engineering community. Four recommended activities and one guideline of overriding importance are summarized below.

Recommended activities:

1. The position paper identifies a set of 16 questions for which answers should be sought.
2. Explore the notion of an Ada Consortium whose purpose would be to organize and coordinate the efforts of Ada communities around the world, and to gather, analyze, and disseminate information among them.
3. Develop a long range plan to using the emergent results of Ada/APSE implementations as a catalyst for progress in software engineering practice.
4. Devise an approach for improving the data security characteristics of near and mid-range Ada/APSE implementations.

Guideline: Any APSE should provide, independently from its host, near state-of-the-art data security for all data under APSE control.

## **2.0 APSE SUPPORT FOR TARGETS**

JOHN R. BEANE, HONEYWELL SYSTEMS & RESEARCH CENTER

### **2.1 Topic Statement**

Target support is that set of services (tools) needed to load, execute, debug, test, evaluate, and otherwise maintain Ada objects on the target computer. Of necessity, part of these services (tools) are resident on the host and part are on the target. The target-resident portion will be referred to as the target monitor, or monitor for short. It is the position of this paper that the target monitor may be divided into three parts--a set of basic services called the monitor kernel, a resource manager, and a tool set analogous to the MAPSE tool set. The interface between the monitor kernel and the rest of the monitor is an extension to the KAPSE interface and deals primarily with host-target communication and realtime organization. The main topic of the paper concentrates on the former, while deferring discussion of the latter to the KITIA group looking at the Ada program runtime system (KAPSE Interface Category 3A). We recommend that the monitor interface for host-target communication conform to the requirements for the Session Layer of the Open System Interconnection (OSI) Basic Reference Model (ISO/DP Draft Proposal TC 97/SC 16 N 719).

### **2.2 Discussion**

Target support consists of the target monitor, the monitor kernel, KAPSE (host) services and MAPSE tools. Each will be described in turn. The exact functions of the target monitor will vary with the nature and services of the testing environment it supports. Each testing service, such as frequency analysis or source debugging, will have a corresponding monitor module tool which knows about and performs that service.

Below the tools is a monitor level (referred to as resource management) which deals with the resources of the target environment whether real or simulated. This level schedules actual resources among the virtual machines running a single user program. Also, in a multiple-user environment this monitor level simulates missing target peripherals, and actual devices which cannot be multiplexed among more than one user. In a single-user environment where accuracy is important and a complete set of target hardware is available, this monitor level may not be present (i.e., the user program interfaces with the real resources directly).

The monitor kernel is a set of basic services which are needed in all testing environments using some form of monitor. The monitor kernel provides input/output facilities to the host-target communication subsystem. The kernel also contains access and control mechanisms for Ada objects executing on the target. These mechanisms know the target runtime organization and the host-resident portion of target support closely parallels that found on the target. KAPSE services (relevant to target support) include input/output using the host-target communication subsystem, and data base facilities to access and manipulate Ada objects of various forms. Each testing capability has a corresponding MAPSE tool to interact with the user, access the host-resident database, and translate user requests into commands for the monitor.

Which parts of the target monitor -- tools, resources management, and kernel -- belong in the KAPSE and which belong outside? This choice can be made by examining the definition of the KAPSE. The role of the

KAPSE is to isolate the MAPSE tool set from dependencies on the host machine or its operating system. The kernel hides the nature of the host-target communication subsystem and the target runtime organization from the rest of the monitor, and hence belongs inside the KAPSE. The tools on the other hand are target-independent, so they belong outside the KAPSE.

Resource management is the tough question. Services like memory management, processor scheduling, and device drivers are a part of the target environment which the KAPSE is meant to hide. But simulators for missing peripherals are more tool-like. In the interest of keeping the KAPSE interface small, resource management shall be considered like an environment simulator, which belongs outside the KAPSE, rather than an operating system (at least with regard to the target).

Is any part of the target monitor transportable between APSEs -- probably not, at least not in all cases. Most monitor tools will rely on the target runtime organization, or make assumptions about the allowable range of transmission rates for the host-target communication subsystem, or both. The kernel mechanisms to access/control an executing Ada object are dependent on the target runtime organization which is a basic facet (assumption) of each Ada compiler and its support tools. At this point, we have no standards to hope that two different compilers would make the same assumptions about runtime organization.

A similar sort of argument applies to host-target input-output. Each monitor tool makes some assumptions about the interaction with its host-resident counterpart. For practical reasons this level of interaction implies a minimal transmission rate required from the host-target communication subsystem. Hence, if some new host-target configuration cannot meet this minimal communication requirement, the tool cannot be transported. Such a situation can be imagined when moving from a tightly coupled configuration to one which relies on communication using tapes.

The minimal services of the kernel (host-target) input-output facility are:

- o Create a virtual link between a program on the host and another on the target.
- o Destroy the virtual link,
- o Send a message over the link,
- o Receive a message,
- o Inhibit message traffic on the link,
- o Enable message traffic.

These services are defined in the Session Layer of the OSI Basic Reference Model. (Rather than define a new standard an existing protocol, the OSI Model, was chosen because it has attracted a large amount of interest as a potential communications standard). Lower layers of the model deal with splitting and reforming variable-length

mechanisms. Without standardization, a portable tool would recover only if the KAPSE did it on the tool's behalf, as in the paternalistic model. This implies that the tool might behave significantly differently from one APSE to another. It also raises worries that the KAPSE recovery might be inappropriate to the more specific needs of a tool, and the recovery might be incorrect.

Recovery mechanisms can be difficult to add onto a system already in operation. Since one of the goals of the KIT effort is to introduce Ada in an acceptable environment, it is important that users not be frustrated by their inability to recover from the errors which will certainly occur in early versions of any system.

#### 3.2.5 Where does recovery happen?

The command language processor must be able to do something about unhandled exceptions in the programs it activates. This is a form of exception propagation which has been available in many systems for some time.

The logon process must be particularly sensitive to problems. It is reasonable that a state of the KAPSE is sufficiently dangerous to prevent further logins without being so bad as to require termination of current jobs.

If the KAPSE input-output facility had a package to read or write an executing Ada object, the debugger tool (on the host) could issue a read request to the KAPSE with the stack frame offset and receive back the variable value directly. If not, the script continues.

3. The KAPSE (host) sends the message to the monitor kernel.
4. The monitor kernel passes the message to the monitor debug tool.
5. The monitor debug tool:
  - a. Interprets the request in the message;
  - b. Accesses the executing Ada object via the kernel runtime organization interface to retrieve the variable value at the absolute offset of the appropriate stack frame;
  - c. Builds a message to satisfy the request;
  - d. Passes the message to the kernel input-output facility for host-target communication.

And a response is delivered to the user via the KAPSE and host debugger tool.

A few observations are appropriate. First, the session layer input-output facility is adequate to get the job done. Second, this simpler facility requires a server on the other machine which contains the smarts of the more complicated facility or knows where to find this information. Third, the powerful input-output facility (to read and write executing Ada objects) could probably be built on top of the simpler service.



Therefore, being conservative at heart, our recommendation is to define the kernel input-output facility for host-target communication in terms of the Session Layer of the OSI Model.

As mentioned earlier, standards proposed by the Ada Program Run-Time System KITIA group will have an impact on the monitor kernel interface for target runtime organization. Similarly, the needs of target support should be addressed when defining such standards.

### **2.3 Summary of Recommendations**

This paper has defined a way of looking at target support that lays out a symmetric structure on the host and target. The monitor portion may be divided into three parts -- kernel, resource management, and tool set -- of which the kernel belongs inside the KAPSE. The monitor kernel supplies input-output facilities to the host-target communications subsystem and an access/control mechanism for the target runtime organization.

Of these two areas, kernel input-output facilities and standards for the same were the focus of this discussion. Our conclusion was to define a kernel input-output interface using the OSI Session Layer. More complicated input-output packages could be built on top of this, and if successful, considered for standardization. As explained, this standard will not result in portable monitor software above the KAPSE level due to a lack of standards for the runtime organization and the nature of host-target communication devices.

Continuing study by both the KIT and KITIA is recommended to verify the adequacy of using the Session Layer input-output in a variety of test monitor situations.

#### **2.4 References**

### 3.0 APSE RECOVERY MECHANISMS

ERIC GRIESHEIMER, MCDONNELL DOUGLAS ASTRONAUTICS

#### 3.1 Topic Statement

It is a Stoneman guideline that an APSE be a highly robust system that can protect itself from user and system errors, that can recover from unforeseen situations, and that can provide meaningful diagnostic information to its users. In order for this to happen, it is necessary for the tools in the APSE to recover from these situations. This can be done on their behalf by the KAPSE (inflexible), or the KAPSE can provide a mechanism for recovery. In order to support portable tools, all KAPSEs should provide the same mechanisms.

#### 3.2 Discussion

##### 3.2.1 What does recovery look like?

When an event occurs which does not permit normal continuation, it generally occurs in one of two places; the tool or the KAPSE. If it occurs in the KAPSE and recovery is possible, it is likely that the tool will see nothing but its own demise, and the summoning process will receive some sort of exception (in the simplest form, the summoning process is the command language processor). When a recoverable event occurs in the tool's execution (or outside of the tool, e.g. a crash), some actions may occur before an exception is raised in the tool. Ordinary Ada exceptions have some serious disadvantages when

they deal with external events, as will be discussed in 3.2.3.

There is a simple model of recovery, which we can call paternalistic recovery. Any event is either critical or inconsequential. Either the KAPSE takes care of everything so that the tool (or user program) doesn't need to worry, or the KAPSE terminates the tool. In either case, the tool is powerless. This solution is inadequate to a robust APSE, since tools cannot recover by their own understanding. A comprehensive database tool should, for example, be able to recover from a parity error in the database without losing the entire base, or even the current transaction if there is some reasonable way to cover the problem.

### 3.2.2 From what does one recover?

An APSE must be able to recover from a large number of events (exceptions). Recovery from many of these will not require any KAPSE action (e.g. recovery from syntax errors during compilation). Many of these exceptions, however, will occur at the KAPSE level, either in that they occur during KAPSE calls or that they occur at a level invisible to an Ada program. Some of the exceptions requiring such KAPSE awareness are:

- System failure (crash)
- Errors in KAPSE components (e.g. "unhandled" exceptions in I/O packages)
- Unhandled exceptions in executing programs

- Release of faulty versions of operational programs. Although this is not an exception at the KAPSE level, it is recovered based on the structure of the database and file structure, which is at least partly in the KAPSE.
- Partial loss of data (e.g. disk failure)
- Parity errors
- Communication line disconnect
- Attempted penetration (as perceived by the KAPSE)

### 3.2.3 How should recovery be undertaken?

For each of the exceptions in 3.2.2, some action must be taken. That actions should cause the least disruption compatible with maintaining the security and integrity of the APSE and its programs. There are certain characteristics one would like a recovery mechanism to have.

A user program or tool should be able to identify the failure which has interfered with its execution. It should be able to get enough information about the failure to do something about it, rather than just estimating the severity of the problem and terminating gracefully.

Where an exception has no consequence (e.g. a parity error which goes away on a retry), an ordinary tool or program

should see nothing of the exception.

Where an exception cannot be hidden (e.g. the parity error persists), some reasonable action should be taken, and the program should be able to predict that action, independent of the KAPSE installation.

Exceptions occurring in user programs or single-user tools should not terminate the entire system. There are probably situations in which this is unavoidable.

When an exception requires termination of a task, user, or data object, some effort must be made to assure that all interlocked entities be closed, excepted, or terminated as completely as remaining data permits.

Exceptions propagating to the command language (e.g. unhandled exceptions in user tasks) should not overinform the user unless he asks for details or is operating in a hatch mode.

An octal dump should not be considered a reasonable recovery.

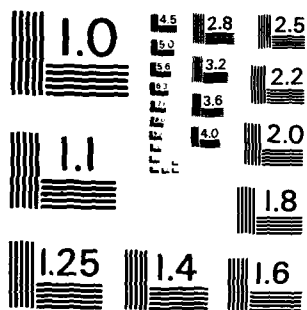
#### 3.2.4 What happens if recovery is not standardized?

Any tool using a KAPSE service is in a position to receive a KAPSE exception due to a failure. In order to rehost a tool, it is necessary to use only standard or common recovery

616

NL

END  
DATE  
FILMED  
2 83  
DTIC



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



mechanisms. Without standardization, a portable tool would recover only if the KAPSE did it on the tool's behalf, as in the paternalistic model. This implies that the tool might behave significantly differently from one APSE to another. It also raises worries that the KAPSE recovery might be inappropriate to the more specific needs of a tool, and the recovery might be incorrect.

Recovery mechanisms can be difficult to add onto a system already in operation. Since one of the goals of the KIT effort is to introduce Ada in an acceptable environment, it is important that users not be frustrated by their inability to recover from the errors which will certainly occur in early versions of any system.

#### 3.2.5 Where does recovery happen?

The command language processor must be able to do something about unhandled exceptions in the programs it activates. This is a form of exception propagation which has been available in many systems for some time.

The logon process must be particularly sensitive to problems. It is reasonable that a state of the KAPSE is sufficiently dangerous to prevent further logins without being so bad as to require termination of current jobs.

Under certain conditions, a part of recovery will include forced logout of some or all jobs.

All device interactions can raise errors. It is important to distinguish the loss of the user's contact, which should be recovered without visibility to the program executing (e.g. re-dial and reattach after a telephone line drops), and the loss of a device selected by the program, which should be corrected by the program. The first error does not require an exception, although in a sense it propagates control above the executing program.

Loss of a part of a database should not result in a total loss of the system's data. It is possible, however, that one bad bit in a directory could compromise the security of the entire APSE. Redundancy and compartmentalization must be built into a KAPSE database design in order to assure that recovery is possible. When recovery is complete, the calling program need not usually be informed. When recovery is incomplete, an exception is appropriate, since errors in KAPSE database access functions are expectable.

The interface between tools must be capable of propagating failures from one tool to another.

The run-time system will encounter exceptions which should

be reported to its monitor if there is one. Standardization of this reporting will simplify the portability of debugging tools.

Distribution of a KAPSE raises many questions about what happens to one part of a network when another part fails. There are situations in which ignoring a failed member is unsafe, since the failed component can no longer be considered free of malicious intent. On the other hand, it is not desirable that the whole network should come down, and it is often necessary to rely on a partially compromised component to forward information or controls.

Security violations must be recovered, and some compromise must be reached between incessant repetitions of "wrong password, guess again" and terminating users with poor typing skills.

It is also necessary to invoke security checks during recovery, since this is perhaps the most dangerous point in a system. If a system is sufficiently compromised, recovery may bring back a significantly different system, about which all of the security postulates may be false. It is certainly necessary to recertify all users after a system crash or a telephone line drop. A failed database must be tested for harmfulness in some sense if it contains windows on data which might be

protected.

### 3.3 Summary of Recommendations

It is apparent from the structure of 3.2.5 that recovery is a highly diffuse problem. It must be invoked at the correct times by other modules, and usually by modules which do not have as one of their principal objectives the solution of exceptional events. Because of this, recovery must be built into the entire system.

There are several guidelines which can be extracted from the above, and should be applied to the entire design of a KAPSE.

- The basic database should be so compartmentalized and/or redundant that failure of a part of it does not compromise the whole.
- Recovery can be partial. The fact that execution can continue does not imply that the user task should not be told that something happened. Great caution should be taken in asserting that a recovery mechanism is complete.
- Any failure which creates doubt as to the identity of any user, data object, or task, should be followed by a recertification of all dubious users, data objects, and tasks. As a very minimum, a dropped dial-in line should require a password for recovery.

- Information should not be withheld from users and tools unless it would compromise the system, or the user had an opportunity to request the information and did not. It is probably desirable that, when a system goes down, all users should be able to find out why it is presumed to have failed when they log back on (or reattach).
- Recovery should be localized to avoid destroying anything which is not related to the failure. This entails a compromise with the need to verify the validity of things which might have been tied to the failure.
- Recovery should extend to all objects which are tied to anything being terminated. When a user of a file is terminated, it may be necessary to check the file for damage.
- The occurrence of an event which is equivalent between two KAPSEs should not cause different recovery on those two KAPSEs. This requires a standardized list of error conditions, actions to be taken, Ada exceptions to be raised, and information to be made available. Such a list is a significant undertaking, and subject to much debate.

### 3.4 References

#### **4.0 APSE EXTENSIBILITY**

JIM RUBY, HUGHES AIRCRAFT CO.

##### **4.1 Topic Statement**

There are at least two possible interpretations of APSE extensibility relevant to the adoption of future guidelines, conventions, and standards for KAPSE interfaces. The first one defines extensibility as a property of the KAPSE itself, as follows:

###### **Definition 1.**

Extensibility is the ability of a KAPSE to incorporate new functions and interfaces without impacting existing APSE tools. Extensibility is measured in the degree to which a KAPSE can be extended without reprogramming existing tools.

This view is somewhat of a departure from the classic STONEMAN concept of a fixed KAPSE, whose service are adequate to enable APSE extension through new tools that rely on those (and only those) services.

The second interpretation deals with extensibility in the more classic STONEMAN sense, and is adapted from a paper by Standish:

###### **Definition 2.**

Extensibility is concerned with the nature of support services required to support tool extensions in APSEs, and with any policies or standards that would govern the growth of such tool sets to ensure that they possess desirable properties.

This view leads to a stronger focus on the APSE tool/user interface as it is affected by tool extensions within the APSE.

## 4.2 Discussion

This section explores the problem of APSE extensibility both in terms of KAPSE interfaces and APSE tool/user interfaces. Since this is a new category whose definition and surrounding issues have neither been widely discussed nor agreed upon, a fair amount of attention will be paid to identifying such issues within the context of specific STONEMAN requirements. Section 4.2.1 highlights a number of critical issues related to the point of view expressed in Definition 1, while the point of view expressed in Definition 2 is expanded in section 4.2.2 (again, taken directly from (2)).

### 4.2.1 Extensibility of the KAPSE Interface.

The first view of extensibility, that the KAPSE itself should be easily extensible, must be taken in the context of STONEMAN requirements. The following quotes highlight the essence of the STONEMAN concept of a KAPSE:

2.B.12 The declarations which are made visible by the KAPSE are given in one or more Ada package specifications. These specifications will include declarations of the primitive operations that are available to any tool in an APSE. They will also include declarations of abstract data types which will be common to all APSEs, including the data types which feature in the interface specifications for the various stages of compilation and execution of a program.

2.B.13 While the external specifications for the KAPSE will be fixed, the associated bodies may vary from one implementation to another. In general all software above the level of the KAPSE will be written in Ada, but the KAPSE itself will be implemented in Ada or by other techniques, making use of local operating systems filing systems or database systems as appropriate



The above statements focus on the basic object-oriented approach that is assumed to be taken in designing a KAPSE, making full use of the power of the Ada package specification construct to define abstract data type interfaces that hide implementation details. Two key points are (1) that such specifications are fixed over some unstated period of time, while the underlying implementation is allowed to vary; and (2) that the implementation may be based on existing software that was not necessarily designed in an object-oriented fashion.

There are some serious problems with this concept that are becoming increasingly evident as more is learned about the evolving ALS and AIE KAPSEs, both of which are being implemented on top of existing operating systems. After review of current documentation and face-to-face discussions with the ALS and AIE contractors, there is no really cogent definition of the KAPSE interfaces in either system. Indeed, one of the key functional ingredients of a KAPSE (according to STONEMAN) is a database manager; yet in the ALS, this function is largely relegated to a MAPSE tool which merely relies on KAPSE services. Moreover, the concept of KAPSE as a well-defined, integrated collection of abstract data types is far from being a demonstrable reality in the foreseeable future.

Let us suppose, however, that such a KAPSE has been designed and implemented on some existing hardware/software base. This hypothetical KAPSE would present APSE tool builders with a number of package interfaces, each having a (hopefully) comprehensive set of procedural operators defined to manipulate objects of a particular type. Such a scenario

supports a limited concept of KAPSE extensibility, in that the object-oriented nature of the overall KAPSE design should permit the addition of new packages. Such additions, possibly made via new instantiations of generic packages, should not corrupt any existing function; and, further, they should have no impact on existing APSE tools which rely totally on existing KAPSE interfaces that are supported by unmodified and unaffected existing Ada packages). The motivation for providing such extensions would presumably be the need to support some new APSE tool for which existing KAPSE services are inadequate or excessively inefficient (more about this later); or to provide a basis for extending an existing tool to perform new functions. In some cases however, creating an entire new package would not necessarily be a desirable or cost-effective solution- it may tend to complicate the overall KAPSE interface by duplicating, in another form, some services already provided. A better approach may be to add a new operator to an existing package; this however, implies a much more direct perturbation of the existing KAPSE interface and may lead to unsuspected impact on existing tools.

The main point to be made here is that KAPSE extensions are feasible in principle, but that the mechanics of providing them in a practical context are not necessarily straightforward. The STONEMAN notion of a fixed KAPSE interface needs to be re-examined, particularly in light of the expressed need to support an ever expanding suite of tools.

Another quote from STONEMAN:

1.J It is possible to take a broader and more general view of programming environments as embodying and supporting the complete integrated process of program design and evolution. This generality is regarded as beyond the present scope of the STONEMAN; however, the aim is that the present document should not exclude a more general view being developed and so it is intended to be "upwards compatible" in all critical areas.

2.B.3 An APSE offers a coordinated and complete set of tools which is applicable at all stages of the system life cycle, from initial requirements specification to long-term maintenance and adaptation to changing requirements.

3.D Life Cycle Support: Support shall be provided to projects throughout the software life cycle from requirements and design through implementation to long term maintenance and modification.

The above statement reflects a clear intent to provide the basis, in the KAPSE, for a far broader range of tools than either DOD-sponsored APSE currently includes. The current efforts support primarily the latter half of the traditional software development life cycle (detailed design, code, test). It must be possible to extend the APSE to cover the full spectrum of software life cycle activities, including requirements specification and analysis, top level design, design verification, and rapid prototyping. It is not at all clear that a KAPSE whose major orientation is towards late life cycle support tools provides adequate services for effective and efficient support of tools that deal with earlier phases. For example, many of the tools gaining favor today for requirements

analysis and structural design provide an interactive graphics interface. To support such tools, a KAPSE should provide services directly related to manipulation of graphics data base objects and graphics device interactions (e.g., draw object x on device y).

There are other existing tools whose peculiar needs have not been examined (e.g., program synthesis tools (3)); and there are surely totally unforeseen classes of tools for which no notion of appropriate services can be established at present. All of this suggests strongly that a KAPSE, if it is to stand the test of time and meet the expressed intent of STONEMAN in terms of life cycle support, must be extensible in a controlled fashion.

There is yet another STONEMAN thrust that suggests a need for KAPSE extensibility:

2.B.8 Extension of an APSE tool set requires knowledge only of the particular APSE and of the Ada language. A new tool -- for example, an environment simulator -- is written within the APSE. This tool can then be installed as part of the APSE and subsequently invoked.

4.E.2 Tools in an APSE shall be designed to meet clear functional needs and shall be composable with other tools in order to carry out more complex functions where appropriate.

4.E.4 The set of tools in an APSE shall remain open-ended; it shall always be possible to add new tools.

In contrast with the orientation towards increased life cycle support characterizing the previous quotes, the above requirements call for a more general kind of "open-endedness". One could plausibly extrapolate to consideration of environments that are far more dynamic than anything explicitly envisioned by STONEMAN. As noted above, a limiting property of the STONEMAN approach is the heavy reliance on Ada itself, a language designed for programming embedded, real-time military computer systems. Many features of Ada (e.g., strong/static typing) are more concerned with reliability than with open-endedness and dynamism. Indeed there exist programming environments today (e.g., interlisp (4), ECL(5)) where extensibility is "built-in", in the sense that the underlying language (LISP or EL-1, respectively) provide truly dynamic data types and/or procedures. This allows for construction of "packages" that, for example, can adapt to new type constructs (defined by evaluation of user expressions) at run time. This degree of extensibility is not achievable in a KAPSE written in Ada without going outside Ada's "nominal system of type" (1) (as is often done in languages like FORTRAN to implement lists through coordinated usage of arrays). While the need for a truly dynamic environment is not clearly established, it is worthy of some attention. The kind of organic evolution of services and tools made possible by such an underlying language system is clearly not achievable within the current KAPSE philosophy.

#### 4.2.2 Extensibility of the Tool/User Interface (taken from (2))

The second view of extensibility, that APSE extensions need to be controlled to ensure that the extended tool set possesses desirable properties, raises several additional issues. One such issue is whether or not we can devise policies of orderly, controlled growth of tool sets in APSEs that promote simplicity and uniformity in the user interface.

One of the most important requirements in STONEMAN is requirement (3.C) for simplicity, quoted here:

3.C SIMPLICITY - The structure of an APSE shall be based on simple overall concepts which are straightforward to understand and use and few in number. Wherever possible, the concepts of the Ada language will be used in an APSE.

Suppose it is our desire to produce APSE tool sets with user interfaces obeying this requirement, and suppose further, that when we extend the APSE tool set by adding new tools, we should like to have the new tools "blend" with the old in such a fashion as to maintain the simplicity of the original interface.

A model that seems not to provide for this property is the model of adding tools to current operating system tool sets. When tools in a normal operating system accumulate by ordinary evolution, they tend to exhibit a bewildering variety of different, often inconsistent, user-interface conventions. The same control characters mean different things in different tools, and there are a jumble of different methods for existing

subsystems, jumping to a system's top level, obtaining help, specifying commands, and so forth. This tends to impose a high cognitive load on the user as there is a considerable and probably needless learning burden imposed on him as he attempts to deal with the thicket of inconsistent interface methods which the tools require him to use.

If it were possible to devise some method or policy for controlling the growth of APSE tool sets, so that new tools added to an APSE would be required to obey one of a small, bounded number of "standard" user interface methods, the STONEMAN requirement for simplicity could be better met. But this must be done in such a fashion as not to inhibit the addition of new interface methods of substantial effectiveness and novelty that are discovered in the future. What is needed is a proper balance between reaping the benefits of future exploration (by incorporating new interface methods) and keeping the interface simple and easily useable (by enforcement of standards). How might this be accomplished? Here is a suggestion.

We know that there are certain popular interface methods for accessing tools in use at the moment. These include:

- (a) the normal conversational interface in which the system prompts the user to give a command (usually issuing a "prompt" character explicitly), and in which the user types a command in response. This type of interface may have various control characters for exiting to the top system level, suspending execution, obtaining help, and so forth.

- (b) the form-filling interface in which the user is presented with a form (usually on a CRT) with holes in it to fill in, and he advances from hole to hole entering suitable data. Here, there may be conventions for advancing to the next hole, returning to the top of the form, aborting, and so forth.
- (c) the tree of menus interface in which a top-level menu gives a set of choices and when one choice is made, a new level of choices is exhibited as a sub-menu. (Example: the UCSD Pascal interface).
- (d) a graphic interface with windows and iconics, as in the Xerox Star.

Some interfaces may make use of mixtures of the capabilities given in (a) through (d), as in a graphics interface which uses iconics to pop up a menu, which in turn selects a tool, which, in turn, asks the user to fill out a form or to conduct a conversation consisting of commands and responses.

One might be able to prescribe a handful (say seven to ten) of standard interface methods, designed to cooperate consistently on the use of universal, pervasive signals, such as a "suspend execution signal" or a signal to obtain help. One could furnish a library of reusable software parts (written in Ada), which parts could be used to create a tool interface according to one of the standard methods. For example, in creating a form-filling interface, one might have parts to "get a date from the user" or "get an integer from the user." One might have "application generators" that, for example, conduct a conversation with



the user to extract, say, the contents of menus in a tree-of-menus, or the visual appearance of a form with holes to be filled in. The "names" of the holes in the form or the items in the tree of menus might be tied to certain tables of actions: actions giving error messages, actions giving help, and data validating actions. The application generator would "interview" the programmer to extract all the information needed to generate a "complete" interface, and would generate an Ada program to act as a complete instance of one of the standard tool interfaces (forms, menus, conversations, iconics, or whatever).

If one could make the addition of a new tool interface very easy technically, through the use of libraries of reusable interface components and application generators, then it might be reasonable to put the growth of APSE tool set interfaces under configuration management using a policy somewhat like the following.

"We, the APSE tool interface configuration management authority, will accept a new tool to be put into the standard APSE tool library provided its interface obeys one of the given standard interface methods to the letter. If this is not suitable and you believe you have a new interface method to contribute along with your proposed tool, we will make you go through the procedure for adding a new standard interface method. Only when you have succeeded in adding your new method to the list of approved standard methods, would we permit you to use your method, and in that case, everybody else can use your method, too. But there are some restrictions. A new method must be judged sufficiently new and beneficial, in comparison to the existing methods that the authority

believes it will bring in more benefits from its use than it costs in the diversification of methods and the imposition of new learning burdens on the APSE user community. It must be brought into consistency and harmony with the existing methods, insofar as practicable. And reusable parts and application generators must be supplied to make it easy for others to add tools using the new interface method".

Were it feasible to provide such a policy and its required technical support, it might be possible substantially to meet the letter and spirit of the STONEMAN simplicity requirement, regarding controlled extension of APSE tool sets. In the absence of some means for controlling the growth of APSE tool sets and enforcing some consistency in tool interfaces, we are likely to witness a recapitulation of the current state of affairs with the growth of operating system tool sets -- namely, random, haphazard, inconsistent growth, with its detrimental effects.

A step that we could take to investigate extensibility further, should the KITIA decide it is worthwhile to pursue it further, would be to draft an INTERFACEMAN document, giving requirements for APSE tool set interfaces. This is, in a sense, a continuation of the STONEMAN exercise, and amounts to our attempting responsibility to carry out the intent of STONEMAN in this area.

If, after circulation and tuning, INTERFACEMAN were to be judged a worthwhile pursuit, it might be possible to put together draft configuration management plans and a prototype reusable software component library to support APSE tool set extensibility. After further evaluation, trial use, and tuning, this could eventually lead to a widely adopted shared agreement about a beneficial way of doing business with respect to the use of APSE tools.

#### 4.3 Summary of Recommendations

1. Recommended action: Re-interpret STONEMAN intentions as regards a fixed KAPSE interface - especially in light of a rapidly expanding tool environment. Can a fixed interface be expected to provide services for the range of tools that can reasonable be postulated?
2. Guideline: An APSE must support the front-end activities of the software life-cycle as well as the later software development and test activities that are now the focus of APSE tool efforts.

#### 4.4 References

- (1) J.W. Goodwin, "Why Programming Environments Need Dynamic Data Types", IEEE Trans. on Software Engr., vol. SE-7, pp. 451-457, Sept. 1981.
- (2) T. A. Scandish, "A Philosophy for a Tool Extension Paradigm", (Preliminary Position Paper Draft), August 1982.
- (3) T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Comm. ACM, vol. 24, pp 563-573, September 1981.
- (4) W. Teitelman and L. Masinter, "The Interlisp Programming Environment", IEEE Computer, vol. 14, pp. 25-34, April 1981.
- (5) B. Wegbreit, "The Treatment of Data Types in EL-1", Comm. ACM, vol. 17, pp. 251-264, May 1974.

## **5.0 ADA/APSE PROGRAM POLICY ISSUES**

**STEVE GLASEMAM, TELEDYNE SYSTEMS COMPANY**

### **5.1 Topic Statement**

In this paper I have taken a position of devil's advocate with respect to certain policies, both stated and implicit, that currently guide the Ada community. Criticisms are stated in exaggerated fashion to highlight the issues involved, and to surface embedded assumptions. Some points discussed here are thought to be settled issues. I open them to re-examination because I question their resolution.

I make no claim to have identified all, or even the most important, of the relevant policy variables. But continual re-examination of policies and their implementations is a vital component of a program as large and complex as the Ada experiment.

Stated and implicit policies have been extracted from a number of documents. For each a discussion and critique is provided, to which I have added material based on the reactions of others to this paper.

### **5.2 Discussion**

#### **5.2.1 PEBBLEMAN (revised) (1)**

5.2.1.1 "It is the expectation that this language will be universally used for DoD embedded computer systems" (1, I.1). Universality is undefined. If all 1985 and beyond ECS projects use Ada for 15% of their software requirements and 85% is assembly language, that could be seen as universality - but little will change in the software world. Do we expect all ECS software: operational flight programs, electronic warfare algorithms, automated test software, trainee software, etc., to be coded in Ada? Do we know if that's a reasonable objective for

for even current approaches to these areas? If it is not, then how can we define the application areas for which Ada is or is not suited?

5.2.1.2 "It is to the DoD's advantage that the language be widely used... both in the U.S. and abroad" (1, I.1). This is a policy that is deceptively simple to implement -- one simply does nothing. However, the implicit assumption is that whatever good comes of other efforts will somehow find its way to the DoD. It would be better to have active cooperation and data sharing. Overall, this implies regular contact, coordinated efforts, and structured cooperation. Specifically, and, for example, are similar metrics, coordinated data collection and analysis efforts, and cooperative analyses. These characteristics are less likely without careful and continual planning by explicitly assigned individuals representing the different countries. For example, the notion of an Ada Consortium may have benefits in terms of consistent and cross fertilized data collection and analyses, providing, overall, a more valuable experiment. With careful planning costs need not be high.

5.2.1.3 "In order to suppress variance, all implementations... will be ... certified as satisfying the language standard ..." (1, I.1). "Defense systems shall require revalidation of translators at the beginning of each project" (1, I.5). "Commercial offerings will be encouraged to regularly maintain the status of their validation" (1, I.5).

This policy, while related to the one in 5.2.1.6, is different enough to warrant separate mention. The idea that any Ada compiler must adhere to an established standard is basic to the notion of standardization. To the extent, however, that the standard can be expected to evolve as more is learned about program validation and as the language itself changes over time, certain policy issues emerge. Will all compilers validated to a given standard be subject to re-validation in the event of a change to the standard? If so, does this imply a sort of second-class citizenship to programs-- APSE tools as well as application suites and re-useable building blocks -- developed under the superceded standard? Does this approach ease or hinder an already strong tendency for proliferation of functionally similar programs?

If older programs are not to be re-validated when the standard changes, does this lock us into a relatively shallow pool of truly standard modules - i.e., only those whose validity has been established under the current standard? If the approach is to re-validate some, but not all tools, how is the choice to be made? This issue gains importance in view of the potential requirement for validating certain MAPSE level tools if experience suggests the value of this.

- 5.2.1.4 "An elementary set of tools for the Ada programming environment will be procured by the DoD..." (1, I.1).

This is a good idea, as far as it goes, and easy to implement using well-known contracting procedures. It also continues the process, started by the Ada initiative, of getting the customer actively involved in

solutions to major problems in which he has a stake. There are, however, no indications that this policy is based on more than short-term expediency. The major issues involved here are how to evolve the software development environment and what, if any, continuing role can be foreseen for the DoD. In the short run, direct involvement with individual system acquisitions will grow tools, procedures and, possibly, the beginnings of a set of reusable software building blocks applicable to somewhat more general environments. But while this is happening other sectors of the community (non-DoD agencies; commercial, industrial, and educational) as well as foreign efforts, will also progress. It is obvious that while some early attention must be focused on specific projects, failure to develop a long-range, high-level planning posture could negatively impact the results expected from the Ada program. Specifically, without explicit plans for examining, evaluating, and incorporating the most valuable contributions from these other sectors -- and this within a context influenced by the general thrusts in the relevant technologies, threat environments, political realities, etc., -- the DoD will find itself in a reactive rather than a controlling posture. The point is that Ada must be managed according to its long range -- not its short term -- potentials.

- 5.2.1.5 "Industry is encouraged to produce and market in the normal fashion ..." (1, I.1).

This policy seems to serve notice that the Ada community understands its lack of control over most of the world. It also indicates an appreciation that much of the progress in software technology comes

from the marketplace itself, and not from the original motivating initiatives. To maintain a measure of control over the Ada effort, the DoD must develop a more formal window on the marketplace. There are precedents for doing such a thing; most notably the VHSIC effort, and the new emphasis given to applied research in communications and other technologies. What needs recognition is that it is now possible to see the early outlines of a real engineering approach to the software life cycle, and to encourage such work with as much energy as has traditionally been applied to hardware-oriented activities.

- 5.2.1.6 "The existence of subsets, supersets, or dialects of a language negate ... benefits which could otherwise accrue" (1, I.3).

This policy, aimed at configuration control and a host of other objectives, is right-hearted but wrong-headed. It appears enforceable, and there is thus the danger that enforcement will be attempted until circumstances show it to be shortsighted. Whatever they are now named, subsets and supersets of Ada already exist and probably will continue to be produced. There seems no reason to require an all or nothing environment. The benefits of Ada as a programming language are substantial, and are available in subset or superset Ada, as well as in full Ada. In addition, there is little reason to believe that an early 198X standard will be relevant for very long. What is important now is to attend to standardization in terms of knowledge emerging from early experience -- leading to a program of incrementally improvable standards. What is perhaps more important as far as cost benefit goes (the choice of language per se impacts less than 20% of software life cycle costs; see ((2) and (3)) is the potential for more effective



software engineering. Holding to a no variance policy with respect to the language, and implementing that policy by requiring all compilers to pass muster at the ACVF, has little to do with this aspect of the Ada experiment. Perhaps a better policy would be to require whatever compiler a contractor employs to pass the ACVF tests that apply. Or, is that feasible? The problem with the policy is not that it is already overcome by events, or that it will probably not be cost effective, or that it discourages innovation; but, rather, that it is inadequate to the task. What is wanted are the economic benefits associated with reusable software and better engineering practices. Thus, besides controlling the language itself, we should control its environment (a potentially more difficult task), and its underlying data structure; in short, the entire software development armament. A more direct approach would be to require demonstrated reusability of all tools used to develop DoD software, and to somehow define and enforce desirable software engineering practices.

5.2.1.7 "The ASF will collect and disseminate information ... will maintain statistics ... (and) will publish periodic reports ..." (1, I.6).

The collection and analysis of data related to Ada is crucial to its political, if not technical, survival. In the near term, as I have elsewhere argued (4), political survival could be of considerable importance. There are several reasons to think so; two are presented here. First, it is likely that in the first year or two of its use the benefits of Ada will be more evident in contractual boilerplate

than in measured fact. It will take some time for experience to build up a picture of Ada as just another programming language, or as a real contributor to a better software engineering environment. While this experience should accrue largely unaided, a completely hands-off approach might be unwise. A visible and active data collection program could have benefits beyond the data it uncovers. Such a program could lend emphasis to the fact that more is expected from Ada than was expected from e.g., Jovial. With such a constant reminder, contractors might be encouraged to stay with a marginal APSE for longer than might ordinarily be the case - especially if the results of data analysis were to be costless to contractors, and shared among them. It is interesting to note in this regard that the notion of an ASF seems not to have survived the transition from PEBBLEMAN to STONEMAN.

Secondly, there is no reason to believe that the data related to Ada projects will be any better defined or easier to collect than that related to pre-Ada projects. This is a venerable problem area of some twenty years standing; we are perhaps slightly better off today in terms of some understanding of what to measure. But, in terms of collection, analysis, and employment of results we are, as before, uninvolved. This fact stems largely from a poor and uneven grasp of which variables should be measured. Moreover, there has long been considerable resistance in the contractor community to collecting data on internal operations and, even if collected, to making it available outside the organization (5). But the Ada experiment may present an effective argument in favor of modifying these ingrained policies. Theoretically, everyone would benefit if Ada attains even a part of

its economic objectives; thus there may be a clearer motivation to bear the costs of data collection. The political issue is to convince decisionmakers of a favorable return on that investment and that means, at the least, coming to grips with the technical problems associated with collecting data of this type. I have elsewhere presented the outlines of how this may be accomplished (5).

#### 5.2.2 STONEMAN (6)

##### 5.2.2.1 "In order to achieve the long term goal of portability of software tools and application systems dependent on them ..." (7, 2.C.5).

I have discussed the implementation of this policy, in an earlier paper (4), from the point of view of user acceptance. Two key paragraphs from that discussion are reproduced below.

With a careful introduction to its tools, the user might ease into an APSE, be receptive to its benefits, and wish to promote them, e.g., by suggesting enhancements and by developing new tools with an eye toward minimal disruption of a recognizable superior software development environment. In such a scenario, the development of installation-specific tools that cannot be fit to an APSE might be minimized - occurring only in special cases where real project peculiarities exist. Under these conditions, the APSE could have the support of its targeted user group, and the experiment as to whether or not a standard HOL and an associated support environment is the key to improved software outcomes will be under way. The verdict should then be based on the merits of the system itself, unclouded by issues of poor implementation, management, and the like.

If, on the other hand, users encounter an initial APSE package with little or no effort to guide implementation, to emphasize its benefits in particular contexts, to anticipate and help to resolve problems, or to provide resources for communicating valuable experiences throughout the user community then, no matter how technically capable, smoothly interfaced, well documented, and configuration controlled, users will resist what can be resisted. In the extreme, as mentioned earlier, they will use the language because they must, but waiver applications will remain common. The environment will languish because other tools are already available - no matter that they do not fit into what is perceived as an unachieved ideal. Portability will go down the drain as tools and environments diverge, and the Ada experiment could fail without ever gathering data. In the end, the language will be just another language, and a major hoped-for benefit -- that of encouraging a software engineering approach to problem solution, with code development a result of that approach instead of an input to the problem -- will not be achieved.

- 5.2.2.2 "Progress towards ... portability is ... achieved by ... competitive design and evaluation of APSEs and their associated KAPSEs." (8, 2.C.7)
- "... one such MAPSE, and ... its KAPSE, will ... become a de facto contention and ... be considered for standardization ..." (8, 2.C.9).

The policy of pursuing standardization through competitive development of APSEs has been addressed in (4). There, however, criticism was based on the additional and unnecessary problems introduced by simultaneous and fundamentally uncooperative initial products. The idea of benefitting from competitive environments is basic to both technical and economic development. However, the realities of a given situation need to be accounted for in such decisions. It can be argued that this has not been done in the case of Ada. For one thing, the usual objective of competitive designs is, after evaluation, to pick one for development and deployment, not to pick two as was done in the case of the ALS and the AIE. Moreover, the military reality -- within which the Ada effort currently has its strongest representation -- is one of strong and multidimensional competition. Because they compete for budget allocations, the service components have vested interests not in cooperation, but in distinguishing themselves from one another as clearly as it is possible to do so. One result of this is to become both operationally and economically dependent upon one's own implementation of any given system. Thus, we can expect the Army to employ the ALS to the exclusion of the probably simultaneously available AIE. And we'd expect the Air Force to use its AIE, and not the ALS, or anything else for that matter. This may not be the best environment for the emergence of one APSE/KAPSE as a widely accepted de-facto standard and as the basis, therefore, of an eventual single DoD standard. What has not been explored as yet that seems basic to progress in this area is the evolutionary relationships between the

ALS, the AIE, and whatever the Navy develops for its own use. Evidence for this -- although it may be a cheap shot, in hindsight -- is the complete incompatibility of ALS and AIE data structures; that fact dooming significant portability of tools and procedures between the two systems.

### 5.2.3 APSE Interoperability and Transportability Plan (8)

5.2.3.1 The current policy that I and T objectives be pursued with respect to the ALS and AIE -- systems that are fundamentally incompatible -- seems questionable. I have no quarrel with the idea that the ALS and AIE provide an opportunity to explore I and T requirements in an empirical rather than in a theoretical way. But, such opportunities abound, and there is nothing particularly attractive about these two systems as compared, for example, to exploring I and T between UNIX and VAX/VMS. If, on the other hand, the policy represents tacit acceptance of dual APSE standards for the visible future, then I am more concerned.

That concern does not focus on the inconsistency of dual standards. If we can achieve two, and limit it to two, then we are ahead of the game and on the right track despite any semantic inconsistency. (There are now, for example, seven "standard" DoD higher order languages.) Instead, it seems to me that, given dual standards, we would serve ourselves better by embracing the differences between the ALS and AIE, and focusing on how to ease the achievement of I and T separately within each environment. This approach deserves consideration, it

seems to me, because it is more in tune with the near-term reality of a two-APSE world, and because, while improving the benefits achievable from each environment, we could learn as much or more about I and T requirements as under the current policy -- without spinning our wheels working towards a questionable objective.

#### 5.2.4 Data security as an APSE requirement.

Any Ada programming support environment, whether ALS, AIE or other, that is employed to produce defense system software must, at least, not degrade existing data security provisions. It is to be hoped that such an environment, being designed, as it is, from the ground up, would rather enhance the security picture for the data under its control. In view of the potential for doing so, the current policy regarding APSE data security seems an especially vacuous one. Having stated that data security must be provided (7), the implementation is left to the APSE contractors with essentially no guidance as to what, explicitly, is required. As a result, the issue has been passed on to the manufacturers of the host computer systems: DEC, in the case of the ALS, and IBM for the AIE. The problem here is that neither of these vendors has anything to offer in the area of data security that goes beyond the current state-of-the-art. But, more than that, neither vendor, as regards hosting an APSE, even approaches what is currently available -- let alone goes beyond it. This would be of somewhat less concern if it were not for the fact that standardized APSEs and a standard development language makes the potential penetrator's job much easier, and increases his return on investment as well -- thus adding motivation to enhanced opportunity (see (7) for

further elaboration of this). The point here is that, for all intents and purposes, the issue has been largely ignored. The potential for significant improvement in data security, especially as resulting from the Ada effort, is not great. Thus there is an understandable tendency to let others worry about a difficult and seemingly marginal concern. On the other hand, the potential damage that is possible -- both to the national interest and to the Ada program itself -- argues strongly against any but the most rigorous application of what is currently possible in this area.

#### 5.2.5 Interim Summary

The preceding paragraphs raise certain general questions with respect to the policy decisions that seem to guide the Ada community.

- (1) Is there a clear idea of the concept of universality with respect to the application of Ada? What are the implications of any answer other than 100%?
- (2) Has there been adequate planning for international cooperation in specific areas of Ada and APSE development?
- (3) How will the community deal with the problem of re-evaluating software developed with tools whose certification has, in effect, expired as a result of evolving compiler standards or techniques for validation?
- (4) Is there an adequate planning structure for managing APSE evolution?
- (5) What specific procedures and organizations are now planned or in place for tracking the technologies most relevant to Ada and APSE implementations.



- (6) Is the no subset/superset policy necessary, and do its short term benefits outweigh possible long term disadvantages?
- (7) Is the current emphasis on achieving Ada's objectives premature? Should there be more emphasis on using first products to better understand the software problem?
- (8) Are existing plans for Ada/APSE implementation incorrectly skewed to favor technique over outcome?
- (9) Are adequate plans and organizations now in place for data collection, analysis, and the dissemination of results?
- (10) Exactly how will contractors be motivated, under existing conditions, to conform to APSE conventions?
- (11) Has there been adequate planning for managing the introduction of the Ada environment?
- (12) Is assuming a natural trend toward a de-facto APSE standard a viable long range posture?
- (13) What consideration has been given to the details of standardizing on one APSE from a starting point of two, and probably three, separate products?
- (14) Why is data security being ignored in view of the comparative ease of improving the situation?
- (15) Should we pursue interoperability and transportability between the ALS and AIE, or should those objectives be sought within each product separately?

- (16) What has been the fate of the Ada Support Facility (ASF) introduced in PEBBLEMAN (1, I.6), but absent from STONEMAN (6)?

### **5.3 Summary of Recommendations**

#### **5.3.1 Recommended Activities: KIT/KITIA Tasks**

- 5.3.1.1 Find answers to questions 1 through 16 in section 5.2.5.
- 5.3.1.2 Explore the idea of an Ada Consortium organized to coordinate the efforts of Ada groups around the world, and to gather, analyze, and disseminate information among those groups.
- 5.3.1.3 Begin to develop a long range plan for using the Ada experiment as a catalyst for improving software engineering practice.
- 5.3.1.4 Devise an approach for improving the data security characteristics of future APSEs.

#### **5.3.2 Recommended Guidelines**

- 5.3.2.1 Any APSE should provide, independent from its host environment, state-of-the-art data security for all data under APSE control.

### **5.4 References**

- (1) Requirements for the Programming Support Environment for the Common High Order Language: "PEBBLEMAN" (revised), U.S. Department of Defense, January 1979.
- (2) Davis, M., S. Glaseman, and W. L. Stanley, "Acquisition and Support of Embedded Computer System Software", the Rand Corporation, R-2831-AF, June 1982.

- (3) "F-111A/E Digital Bomb-Nav System Software Analysis", Battelle-Columbus Laboratories, Technical Report AFAL-TR-79-1043, November 1978.
- (4) Glaseman, S. "Tool Portability as a function of APSE Acceptance", KITIA Concept Paper, Teledyne Systems Company, February 1982.
- (5) Glaseman, S., and M. R. Davis, "Software Requirements for Embedded Computers: A Preliminary Report," The Rand Corporation, R-2567-AF, March 1980.
- (6) Requirements for Ada Programming Support Environments: "STONEMAN", Department of Defense, February 1980.
- (7) Glaseman, S., R. Turn, and R. S. Gaines, "Problem Areas in Computer Security Assessment," Proceedings, 1977 National Computer Conference, and P-5822, The Rand Corporation, February 1977.
- (8) Ada Programming Support Environment (APSE) Interoperability and Transportability (IT) Plan, TRW Defense Systems Group, 1980.

6.0 DISTRIBUTED APSEs

(TBD)

## 7.0 APSE SECURITY

(TBD)

## 8.0 PRAGMAS AND OTHER APSE TOOL CONTROLS

(TBD)

## 9.0 GROUP IV FUTURE PLANS

A major objective is to establish a more productive working relationship with the KIT. Now that some sort of products are available, a better sense of the possibilities for joint action may be forthcoming. KITIA Group IV has, as has our KIT counterpart, been working independently to scope our various interest areas. Now that that is well underway, we can more fruitfully explore the potentials for mutually supporting and cooperative activities.

Similarly, Group IV may now be in a better position to support and be supported by the other KITIA groups. It might be well, for example, for us to schedule periodic meetings of group chairpersons. An improved sense of inter-group coordination, cooperative goal setting, information transfer, and problem solving are some benefits that, somewhat lacking in the recent past, come immediately to mind.

Group IV has identified a number of recommended activities (see Paragraph 1.4), and we expect to contribute to many of them ourselves. This work should, in fact, provide a firm base for increased cooperation among ourselves, other KITIA groups, and the KIT.

One obvious objective is to complete this document, and to evolve its contents based on further examination of existing topics, and those that may be added over time.

Finally, we need to address certain group organizational issues. For example, we have one member we haven't seen or heard from in some time. His status will be reviewed with an eye toward our need for productive individuals.



# Computer Simulation and the Ada Environment

Eric Griesheimer

MCDONNELL DOUGLAS ASTRONAUTICS

A major step in the debugging of software writing in Ada is testing through monitored execution. In the absence of supportive or accessible hardware, it is often necessary or desirable to employ interpretive simulation (instruction level simulation or HOL interpretation). There are a number of problems relating to KAPSE interfaces which can arise in this area.

It is possible to simulate the program under test at either the Ada level or at the target instruction level (even occasionally at the target microinstruction level). In the first case, the simulator would probably interpret a form of DIANA. It would have to be sensitive enough to the target architecture to reproduce exact timing for all Ada statements, exact results for all arithmetic operations on approximate numeric types, precise memory management limitations, representation specification consequences, unsafe conversion effects, execution of machine language insertions, and exactly the same overflow conditions as the target processor. Such a simulation presents two disadvantages. First, it does not fully verify the compiler, since it does not even see the object code. Second, the confidence it provides if no errors are detected is provisional on confidence in the simulator itself, and the simulator is a sophisticated program.

The second, classical type of target simulator simulates at the level of target machine operations. Although this requires more execution time to simulate long algorithms, the simulator is a subset of the high level simulator and thus cheaper to build, easier to verify, and generally a smaller program when executing. In fact, the analysis performed on DIANA by the other simulator might take as long as the interpretation of the more numerous, but simpler target operations. This sort of simulator would be written in Ada, but would probably employ "cheating". Certain variables, representing registers and memory locations, would need to be considered alternately

as bit strings and as integers in Ada. This entails UNSAFE-CONVERSION, and could inhibit portability, particularly since the Ada host compiler would not check for dependence on one's/two's complement differences and word size.

In a growing percentage of applications, the program under test does not operate on a single processor. This implies that the simulator simulates multiple processors. This effect could be achieved through the KAPSE by communication among separate tasks, and when some of the processors in the simulation are not real (not simulated), this form of communication (at least at the message level) would be required.

Regardless of the type of simulator used, there are several useful capabilities which require support from an operating system, and present portability problems.

Plotting is a common form of program analysis, for data values and program coverage information. Although it may not belong at the KAPSE level, a simple core of plotting functions, adapted to a standard set of plotting devices, would permit portable plotting software to be written. Transportation of plot programs often involves more than just character-command transliteration, since timing and permissible sequences can carry.

Since the user of a computer simulator is operating a process, he needs a way to suspend the process without losing the simulator. This entails either program recovery and re-entry to the executive, or intermittent polling of the terminal without wait. The provision of the latter function in a system-independent form would provide great power to the writer of any portable interactive program.

One of the prime causes of simulator slowness is the simulations of large memories and complicated application environments. Efficient random access file primitives (fixed-length records by number) can go a long way toward alleviating system overloads.

# **APPENDIX A**

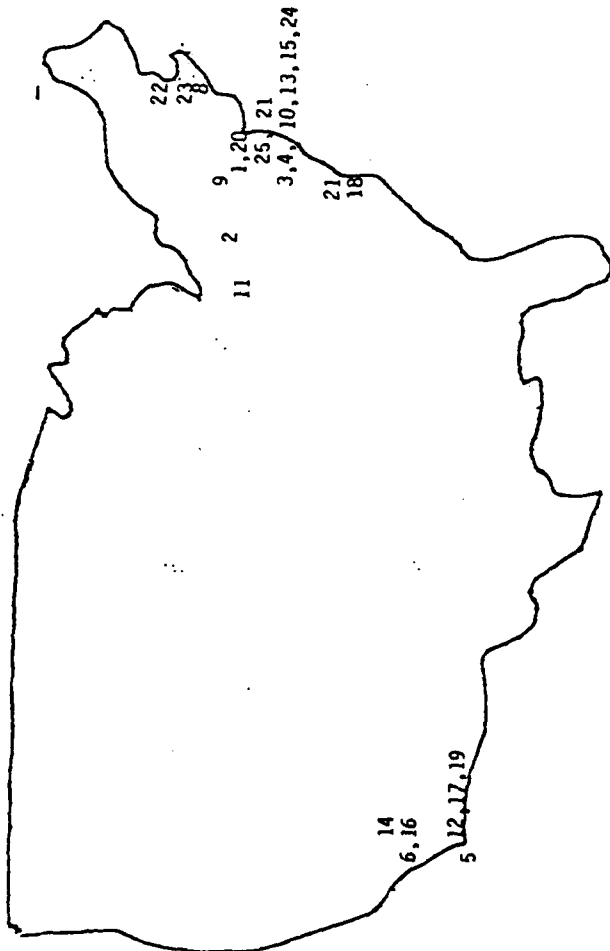
**KIT MEMBERS & MAP**

**KITIA MEMBERS & MAP**

KIT NAME LIST

|                                    |                                 |
|------------------------------------|---------------------------------|
| BALDWIN, Rich                      | U. S. Army (CECOM)              |
| CASTOR, Jinny                      | U.S. Air Force (AFWAL/AAAF-2)   |
| CONVERSE, Bob                      | Naval Sea Systems Command       |
| FERGUSON, Jay                      | National Security Agency        |
| FOIDL, Jack                        | TRW                             |
| FOREMAN, John                      | Texas Instruments, Inc.         |
| HART, Hal                          | TRW                             |
| HOUSE, Ron<br>CONRAD, Tom<br>(Alt) | Naval Underwater Systems Center |
| JOHNSTON, Larry                    | Naval Air Development Center    |
| KRAMER, Jack                       | Ada Joint Program Office        |
| LINDLEY, Larry                     | Naval Avionics Center           |
| LOPER, Warren                      | Naval Ocean Systems Center      |

|                         |                                                            |
|-------------------------|------------------------------------------------------------|
| LUBBES, H. O.           | Naval Electronic Systems Command                           |
| MILLER, Jo              | Naval Weapons Center                                       |
| MILTON, Donn            | Computer Sciences Corp                                     |
| NELSON, Eldred          | TRW                                                        |
| OBERNDORF, Tricia       | Naval Ocean Systems Center                                 |
| PEELE, Shirley          | Fleet Combat Direction Systems Support Activity -Dam Neck  |
| TAYLOR, Guy (Alt)       |                                                            |
| PURRIER, Lee            | Fleet Combat Direction Systems Support Activity -San Diego |
| ROBERTSON, George (Alt) |                                                            |
| SANTANELLI, Barbara     | U. S. Army (CECOM)                                         |
| STEIN, Mo               | Naval Surface Weapons Center                               |
| DUDASH, Ed (Alt)        |                                                            |
| TAFT, Tucker            | Intermetrics                                               |
| MOLONEY, Jim (Alt)      |                                                            |
| THALL, Rich             | SofTech                                                    |
| WALD, Elizabeth         | Naval Research Laboratory                                  |
| EGAN, Jack (Alt)        |                                                            |
| WALTRIP, Chuck          | John Hopkins University                                    |
| WHITE, Doug             | U.S. Air Force (RADC/COES)                                 |
| KEAN, Elizabeth (Alt)   |                                                            |



# KIT MEMBERS

1. BALDWIN, Rich
2. CASTOR, Jimmy
3. CONVERSE, Bob
4. FERGUSON, Jay
5. FOTDL, Jack
6. FOREMAN, John
7. HART, Hal
8. HOUSE, Ron  
CONRAD, Tom (Alt)
9. JOHNSTON, Larry

10. KRAMER, Jack
11. LINDLEY, Larry
12. LOPER, Warren
13. LUBBES, H. O.
14. MILLER, Jo
15. MILTON, Donn
16. NELSON, Eldred
17. OBERNDORF, Tricia
18. PEELE, Shirley  
TAYLOR, Guy (Alt)

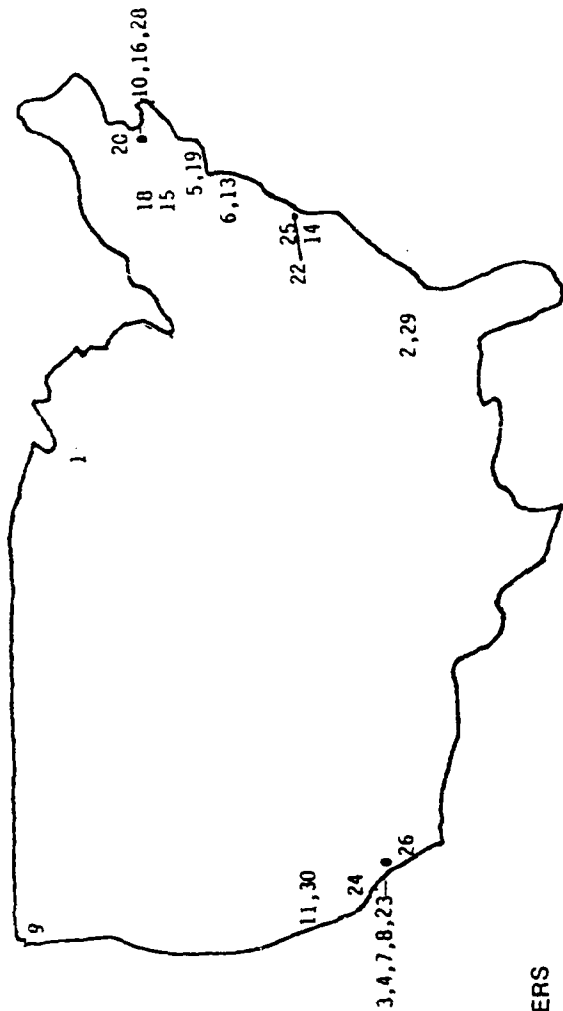
19. PURRIER, Lee  
Robertson, George (Alt)
20. SANTANELLI, Barbara
21. STEIN, Mo  
DUDASH, ED (Alt)
22. TAFT, Tucker  
MOLONEY, Jim (Alt)
23. THALL, Rich
24. WALD, Elizabeth  
EGAN, Jack (Alt)
25. WALTRIP, Chuck
26. WHITE, Doug  
KEAN, Elizabeth (Alt)

# KITIA NAME LIST

| NAME              | AFFILIATION                     |
|-------------------|---------------------------------|
| CORNHILL, Dennis  | Honeywell                       |
| BEANE, John (Alt) |                                 |
| COX, Fred         | Georgia Institute of Technology |
| DRAKE, Dick       | IBM                             |
| FELLOWS, Jon      | System Development Corp         |
| FISCHER, Herman   | Litton Data Sytsems             |
| FREEDMAN, Roy     | Hazeltine Corp.                 |
| GARGARO, Anthony  | Computer Sciences Corp.         |
| GLASEMAN, Steve   | Teledyne Systems Co.            |
| GRIESHEIMER, Eric | McDonnell Douglas Astronautics  |
| JOHNSON, Ron      | Boeing Aerospace Co.            |
| KERNER, Judy      | Norden Systems                  |
| KOTLER, Reed      | Lockheed Missiles & Space       |
| LAHTINEN, Pekka   | Oy Softplan AB (Finland)        |

|                     |                                         |
|---------------------|-----------------------------------------|
| LAMB, J. Eli        | Bell Labs                               |
| LINDQUIST, Tim      | Virginia Institute of Technology        |
| LOVEMAN, Dave       | Massachusetts Computer Association Inc. |
| LYONS, Tim          | Software Sciences Ltd. (UK)             |
| McGONAGLE, Dave     | General Electric                        |
| MOONEY, Charles     | Grumman Aerospace                       |
| MORSE, H. R.        | Frey Federal Systems                    |
| PLOEDEREDER, Erhard | IABG (West Germany)                     |
| REEDY, Ann          | Planning Research Corporation           |
| RUBY, Jim           | Hughes Aircraft Co.                     |
| SAIB, Sabina        | General Research Corp.                  |
| SIBLEY, Edgar       | Alpha Omega Group, Inc.                 |
| STANDISH, Thomas    | University of California at Irvine      |
| WESTERMANN, Rob     | TNO-IBBC (The Netherlands)              |
| WILLMAN, Herb       | Raytheon Company                        |
| WREGG, Doug         | Control Data Corp.                      |
| YELOWITZ, Larry     | Ford Aerospace & Communications Corp.   |





# KITIA MEMBERS

A-7

- |                      |                     |                                        |
|----------------------|---------------------|----------------------------------------|
| 1. CORNWILL, Dennis  | 11. KOTLER, Reed    | 24. SAIB, Sabina                       |
| 2. COX, Fred         | 13. LAMB, Eli       | 25. SIBLEY, Edgar                      |
| 3. FELLOWS, Jon      | 14. LINDQUIST, Tim  | 26. STANDISH, Thomas                   |
| 4. FISCHER, Herman   | 15. LOCKE, Doug     | 28. WILLMAN, Herb                      |
| 5. FREEDMAN, Roy     | 16. LOVEMAN, Dave   | 29. WREGE, Doug                        |
| 6. GARGARO, Anthony  | 18. MCGONAGLE, Dave | 30. YELOWITZ, Larry                    |
| 7. GLASEMAN, Steve   | 19. MOONEY, Charles | EUROPE:                                |
| 8. GRIESHEIMER, Eric | 20. MORSE, H. R.    | 12. LAHTINEN, Pekka (Finland)          |
| 9. JOHNSON, Ron      | 22. REEPPY, Ann     | 17. LYONS, Tim (United Kingdom)        |
| 10. KERNER, Judy     | 23. RUBY, Jim       | 21. PLOEDEREDER, Erhard (West Germany) |
|                      |                     | 27. WESTERMANN, Rob (The Netherlands)  |

