

DTIC FILE COPY

①

AGARD-AR-229

AGARD-AR-229

AD-A219 101

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD ADVISORY REPORT No.229

The Implications of Using Integrated Software Support Environment for Design of Guidance and Control Systems Software

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

DTIC
ELECTE
MAR 16 1990
S E D

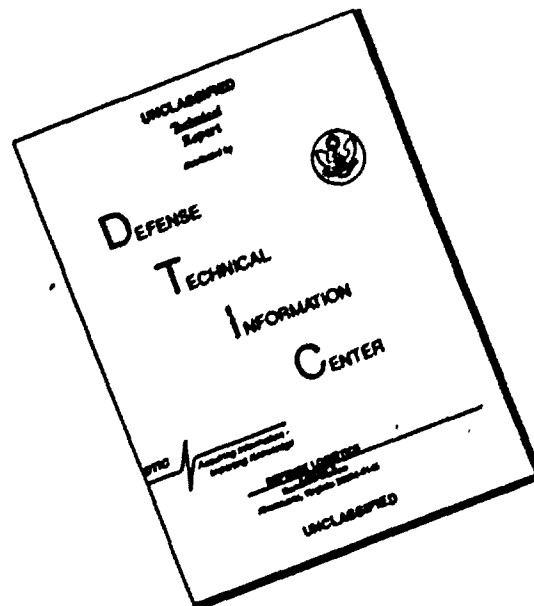
NORTH ATLANTIC TREATY ORGANIZATION



DISTRIBUTION AND AVAILABILITY
ON BACK COVER

90 03 16 106

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

NORTH ATLANTIC TREATY ORGANIZATION
 ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT
 (ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARD Advisory Report No.229

THE IMPLICATIONS OF USING INTEGRATED SOFTWARE SUPPORT
 ENVIRONMENT FOR DESIGN OF GUIDANCE AND CONTROL
 SYSTEMS SOFTWARE

Edited by

Dr Edwin B.Stear
 Executive Director
 The Washington Technology Center
 University of Washington, FH-10
 376 Loew Hall
 Seattle, WA 98195
 United States

John T.Shepherd,
 Research Director
 GEC-Marconi Limited
 Elstree Way
 Borehamwood
 Hertfordshire WD6 1RX
 United Kingdom

Accession Per	
NTIS GEM	X
DTIC TAB	
Unannounced	
Justification	
By	
Distribution/	
Availability Code	
Dist	Avail and/or Special
A-1	

This report has been prepared as a summary of the deliberations of Working Group 08
 of the Guidance and Control Panel of AGARD.

THE MISSION OF AGARD

According to its Charter, the mission of AGARD is to bring together the leading personalities of the NATO nations in the fields of science and technology relating to aerospace for the following purposes:

- Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community;
- Providing scientific and technical advice and assistance to the Military Committee in the field of aerospace research and development (with particular regard to its military application);
- Continuously stimulating advances in the aerospace sciences relevant to strengthening the common defence posture;
- Improving the co-operation among member nations in aerospace research and development;
- Exchange of scientific and technical information;
- Providing assistance to member nations for the purpose of increasing their scientific and technical potential;
- Rendering scientific and technical assistance, as requested, to other NATO bodies and to member nations in connection with research and development problems in the aerospace field.

The highest authority within AGARD is the National Delegates Board consisting of officially appointed senior representatives from each member nation. The mission of AGARD is carried out through the Panels which are composed of experts appointed by the National Delegates, the Consultant and Exchange Programme and the Aerospace Applications Studies Programme. The results of AGARD work are reported to the member nations and the NATO Authorities through the AGARD series of publications of which this is one.

Participation in AGARD activities is by invitation only and is normally limited to citizens of the NATO nations.

The content of this publication has been reproduced directly from material supplied by AGARD or the authors.

Published February 1990

Copyright © AGARD 1990
All Rights Reserved

ISBN 92 835 0538 7



Printed by Specialised Printing Services Limited
40 Chigwell Lane, Loughton, Essex IG10 3TZ

DEDICATION

THIS REPORT OF AGARD GUIDANCE AND CONTROL PANEL
WORKING GROUP 08 IS DEDICATED TO



Dr. Louis Urban

TECHNICAL LEADER, AGARD ENTHUSIAST, THE REAL FATHER OF
WORKING GROUP 08, DEDICATED GUIDANCE AND CONTROL
PANEL MEMBER, VALUED AND TRUSTED COLLEAGUE OF THOSE
WHO HAD THE GOOD FORTUNE TO WORK WITH HIM. WE HOPE HE
APPROVES OF THIS FINAL REPORT OF THE WORKING GROUP.

PREFACE

This report has been prepared as a summary of the deliberations of Working Group 08 of the Guidance and Control Panel of AGARD. The Terms of Reference of the working group as approved by the National Delegates Board of AGARD were:

- (i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint.
- (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i).
- (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i).

The working group attempted to consider all existing software design and development technology which existed or was known to be under development at the time the report was prepared. In particular there are a great number of important software developments taking place in connection with the ADA* and LTR3 high level languages which are under development, and a number of important software engineering environment developments are also taking place in various countries represented on the working group. Where possible, the working group gave special consideration to these developments. However, because of their large number, it has obviously been impossible to do justice to all of them. The particular existing and new technology under development which was considered explicitly by the group is referenced at appropriate points in the report.

The working group started work in May 1983 and met at six month intervals through May 1986. Final editing of the report took place during the last half of 1986 and during 1987.

The working group was composed of members from France, The Federal Republic of Germany, Italy, The Netherlands, the United Kingdom and the United States, all of whom are expert in either guidance and control systems design or software design or both. This report represents the consensus view of the group, but it should not be construed as representing the views or policies of any of the nations, organizations, or individuals represented on the working group.

AVANT-PROPOS

Ce rapport résume les travaux du Groupe de travail 08 de la Commission Guidage et Pilotage de l'AGARD. Les termes de référence de ce groupe, tels que définis par le Conseil des Délégués Nationaux de l'AGARD (NDB) étaient les suivants:

- (i) Développer et discuter un cahier des charges relatif à un environnement de support de développement de logiciel en langage de haut niveau vu sous les aspects guidage et pilotage.
- (ii) évaluer les caractéristiques et les possibilités offertes par les environnements de support de langages de haut niveau soit existants, soit en cours de développement, par rapport aux besoins définis en (i).
- (iii) définir, le cas échéant, les modifications qui seraient à envisager afin de pouvoir répondre à la totalité des besoins exprimés. (i)

Le groupe de travail s'est donné pour tâche d'étudier toutes les technologies de conception et de réalisation de logiciels disponibles ou en cours de développement au moment de l'élaboration du rapport. En particulier, des progrès importants sont réalisés à l'heure actuelle en ce qui concerne la mise en place des langages de haut niveau ADA et LTR3, et de nombreux développements importants concernant les environnements de génie logiciel sont en cours dans les différents pays représentés au niveau du groupe. Dans la mesure du possible, le Groupe de travail a porté une attention particulière à ces développements. Cependant, vu leur nombre, il est évident qu'une étude exhaustive n'a pu être entreprise. Les références des technologies existantes ou en cours de développement qui ont été étudiées directement par le groupe sont données aux endroits appropriés dans le texte du rapport.

Le groupe de travail s'est réuni pour la première fois au mois de mai 1983. Par la suite, les réunions ont été organisées à des intervalles de six mois jusqu'en mai 1986. La révision définitive du document a été réalisée pendant le deuxième semestre de 1986 et courant 1987.

Les membres du groupe, tous experts soit dans le domaine de la conception des systèmes de guidage et de pilotage, soit dans le domaine du génie logiciel, soit dans les deux matières, représentaient la France, la République Fédérale d'Allemagne, l'Italie, les Pays-Bas, le Royaume-Uni et les Etats-Unis. Ce rapport représente le consensus d'opinion du groupe: il ne doit pas être interprété comme étant représentatif d'une opinion ou d'une politique quelconque de l'une des nations, organisations ou personnes représentées au sein du groupe de travail.

AGARD GUIDANCE AND CONTROL PANEL WORKING GROUP-08 MEMBERSHIP

FRANCE

Dr. P. De Bondeli
Aerospatiale
Batiment 19 - 5DTL
59 Route De Verneuil
F-78130 Les Mureaux
France

Mr. J. Ichbiah
ALSYS
Centre Commercial de la
Chataigneraie
F-78170 La Celle St. Cloud
France

Mr. J. Moreau De
Montcheuil
DEN/STEN/GP
26 Boulevard Victor
F-75996 Paris Armees
France

FEDERAL REPUBLIC OF GERMANY

Mr. W. M. Fraedrich
BMVg
RU IV 6
Postfach 13 28
D-5300 Bonn 1
Federal Republic of Ger-
many

Dr. H. M. Hessel
Messerschmitt-Boelkow-
Blohm GMBH
ABT LKE 43
Postfach 80 11 60
D-8000 Muenchen 80
Federal Republic of Ger-
many

Mr. A. H. Kuechle

Dornier System GMBH
ABT ZIT
Postfach 13 60
D-7990 Friedrichshafen 1
Federal Republic of Ger-
many

Mr. O. F. Nielsen
MBB GMBH, ABT FE 324
Military Aircraft Division
Postfach 80 11 60
D-8012 Ottobrunn/
Muenchen
Federal Republic of Ger-
many

Mr. H. Roschmann
AEG Telefunken
ABT A14 E233
Sedanstr 10
D-7900 ULM
Federal Republic of Ger-
many

Mr. J. Stocker
Messerschmitt-Boelkow-
Blohm GMBH
ABT LKE 341
Postfach 80 11 60
D-8000 Muenchen 80
Federal Republic of Ger-
many

ITALY

Mr. F. Ristori
Aeritalia Saipa
Direzione Tecnica Sistemi
Corso Marche 41
I-10146 Torino
Italy

NETHERLANDS

Ir M. Boasson
Hollandse Signaalap-
paraten BV
Software Dept., PO Box 47
7550 GD Hengelo
Netherlands

UNITED KINGDOM

Mr. D. Beck
British Aerospace PLC
Aircraft Group
Warton Aerodrome
Preston, Lancs PR 4 1AX
United Kingdom

Dr. A. A. Callaway
GEC Software Limited
132 Long Acre
London WC2
United Kingdom

Mr. J. R. Carter
British Aerospace PLC
A/C Group, Woodford
Aerodrome
Chester Road
Woodford, Cheshire SK7
1QR
United Kingdom

Mr. P. D. Chinn
GEC Avionics Limited
Elstree Way
Borehamwood, Herts
United Kingdom

Mr. E. Sefton
British Aerospace PLC
Aircraft Group
Warton Aerodrome
Preson, Lancs PR4 1AX
United Kingdom

AGARD GUIDANCE AND CONTROL PANEL WORKING GROUP-08 MEMBERSHIP (con.)

J. T. Shepherd
Research Director
GEC-Marconi Limited
Elstree Way
Borehamwood
Hertfordshire WD6 1RX
United Kingdom

Mr. G. W. Wilcock
Head of Airborne Computing
Section FS(F) Department
Royal Aircraft Establishment
Farnborough, Hants GU14
6TD
United Kingdom

UNITED STATES

R. F. Bousley
Manager, Avionics
Boeing Military Airplane
Co.
P.O. Box 3707 - MS 41-08
Seattle, WA 98124
United States

Dr. T. B. Cunningham
Honeywell Inc., Mgr.
Systems & Control
MN17-2370, Box 312
2600 Ridgway Parkway
Minneapolis, MN 55440
United States

Dr. A. P. Dethomas
AFWAL/FIGX
Wright-Patterson AFB
OH 45433-6553
United States

Lt.Col. L.E. Druffel
Director, ADA Joint Pro-

gram Office
Ballston Tower Two, Room
1210
801 N. Randolph Street
Arlington, VA 22203
United States

Mr. D. T. Green
Code N20E
Naval Surface Weapons
Center
Dahlgren, VA 22448
United States

Mr. G. L. Hadley
Data Processing Supervisor
1401 SW 172nd St.
Seattle, WA 98166
United States

Mr. A. M. Henne
Harris Corporation - GISD
505 John Rhodes Blvd.
Mail Stop 98000
Melbourne, FL 32935
United States

Dr. J. Munson
Systems Development
Corporation
2500 Colorado Avenue
Santa Monica, CA 90406
United States

Dr. E. B. Stear
Director, Washington
Technology
Center
University of Washington
376 Loew Hall - FH-10
Seattle, WA 98195
United States

Mr. H. G. Stuebing

Naval Air Development
Center
Software Computer Direc-
torate
Warminster, PA 18974-5000
United States

Dr. D. Sundstrom
General Dynamics
P.O. Box 748
Fort Worth, Texas 76101
United States

TABLE OF CONTENTS

CHAPTER 1—INTRODUCTION

1.1 General	1-1
1.2 General Requirements for Guidance and Control Software	1-1
1.2.1 Fault Avoidance Issues	1-2
1.2.2 Fault Tolerance Issues	1-2
1.2.3 Computer Language Issues	1-2
1.2.4 Flight Control Systems Requirements Issues	1-2
1.2.5 Integration Issues	1-4
1.2.6 Flight Control System Software Complexity Issues	1-5
1.2.7 General Requirements for Tools and Support Environments	1-6
1.3 Structure of the Report	1-7

CHAPTER 2—PHASES AND METHODS FOR SYSTEM AND SOFTWARE DEVELOPMENT

2.1 Introduction	2-1
2.2 Flight Guidance and Control Systems Specifics	2-1
2.3. The System Development Process	2-2
2.3.1. The Hierarchical Phase Model	2-3
2.3.1.1. General	2-3
2.3.1.2. The Study Phase	2-4
2.3.1.3. The Concept Development Phase	2-6
2.3.1.4. The System Requirements/System Design Phase	2-6
2.3.1.5. Subsystem Requirements/Subsystem Design Phase	2-6
2.3.1.6. The Software Requirements/Hardware Specification Phase	2-8
2.3.1.7. The Basic Software Design Phase	2-9
2.3.1.8. The Detailed Software Design Phase	2-9
2.3.1.9. The Module Coding/Module Test Phases	2-10
2.3.1.10. The Software Integration on Host Computer Phase	2-10
2.3.1.11. The Software Integration on the Target Computer Phase	2-11
2.3.1.12. The Subsystem Integration Phase	2-11
2.3.1.13. The System Integration Phase	2-11

2.3.1.14. The Flight Test Phase	2-11
2.3.1.15 The Production Phase	2-11
2.3.1.16 The In-Service or Postdevelopment Support Phase	2-12
2.3.2. General Methods for Guidance and Control System Development	2-12
2.3.2.1. General Requirements for System Development Methods	2-12
2.3.2.2 Requirements for Methods from the Study Phase through the Software Requirements/Hardware Specification Phase	2-12
2.3.2.3 Requirements for Methods for the Basic Software Design Phase	2-13
2.3.2.4 Requirements for Methods for Detailed Software Design	2-15
2.3.2.5 Requirements for Methods for Coding/Module Test	2-15
2.3.2.6 Requirements for Methods for Integration and Testing	2-15
2.3.2.7 Resume	2-15
 2.4 Project Documentation	 2-16
2.4.1. System Development Documents	2-16
2.4.2. Software Engineering Documents	2-17
2.4.3. System/Subsystem Integration Documents	2-18
2.4.4. Standards Documents	2-18
2.4.5. Quality Assurance (QA) Documents	2-18
2.4.6. Other Project Documents	2-19
 2.5 General Project Support Issues and Measures	 2-19
2.5.1. Organizational Issues	2-19
2.5.2. Project Planning Issues and Measures	2-19
2.5.3. Project Review Issues	2-20
2.5.4. Change and Configuration Control Issues and Measures	2-21
2.5.5. Quality Assurance and Measurement Issues /5/,/6/	2-21
2.5.6. Certification Requirements Issues	2-22
 REFERENCES	 2-24
 Appendix 2.1: MODEL SYSTEM SPECIFICATION DOCUMENT	 2-23
 Appendix 2.2: MODEL SUBSYSTEM SPECIFICATION DOCUMENT	 2-24
 Appendix: DERIVATION OF EQUATIONS & ALGORITHMS	 2-25
 CHAPTER 3—CURRENT STATUS OF TOOLS, TOOL SETS, AND SUPPORT ENVIRONMENTS FOR EMBEDDED COMPUTER SYSTEMS SOFTWARE	

3.1 CURRENT SOFTWARE PRACTICES	3-1
3.1.1. Introduction	
3.1.2. Current Acquisition, Management, and Support Approaches	3-1
3.1.2.1 Federal Republic of Germany	3-1
3.1.2.2 France	3-1
3.1.2.3 United Kingdom	3-3
3.1.2.4. United States of America	3-3
3.2. SOFTWARE REQUIREMENTS AND DESIGN METHODS	3-3
3.2.1 Introduction	3-3
3.2.2 Software Requirements Process Description	3-5
3.2.3 Software Design Process Description	3-6
3.2.3.1 Design Methods	3-6
3.2.3.2 Other Factors in the Design Process	3-7
3.2.4 Tables of Current Methods, Tools, and Tool Sets	3-7
3.3. PROGRAMMING LANGUAGES, TOOLS, AND ENVIRONMENTS	3-7
3.3.1. Introduction	3-7
3.3.2. Generic Tool Descriptions	3-8
Editor	3-8
Compiler	3-9
Assembler	3-9
Linker	3-9
Relocating Loader	3-9
Run-Time Executive	3-9
Simulator/Emulator	3-10
In-Circuit Emulation	3-10
Symbolic Debugger	3-10
Pretty Printer	3-10
Host to Target Exporter	3-10
3.3.3. Current Languages, Tools, and Environments	3-11
3.3.4. ADA	3-11
3.3.4.1. ADA Certification	3-11
3.3.4.2. ADA Run-Time Environment	3-11
3.3.4.3. Ada Programming Support Environment (APSE)	3-11
3.3.4.4. Common APSE Interface Set (CAIS)	3-12
3.3.4.5. Portable Common Tool Environment (PCTE)	3-12
3.3.4.6. ADA Summary	3-12
3.4. SOFTWARE AND SYSTEM INTEGRATION/TESTING	3-12

3.4.1. Introduction	3-12
3.4.2. Test Stages	3-13
3.4.2.1. Execution on Host Computer	3-14
3.4.2.2. Execution on an Emulator	3-14
3.4.2.3. Execution on Target Computer	3-14
3.4.2.4. System Integration	3-15
3.4.2.5. Flight Test	3-16
3.5 TOOLS TO SUPPORT CERTIFICATION, VALIDATION, AND VERIFICATION	3-16
3.5.1. Introduction	3-16
3.5.2. Certification	3-16
3.5.3. Validation	3-17
3.5.4. Verification	3-17
3.5.5. Software Quality Assurance (SQA)	3-18
3.5.5.1. Purpose	3-18
3.5.5.2. Tasks of SQA	3-18
3.5.5.3. Responsibility	3-19
3.5.5.4. Document Review and Checking	3-19
3.5.5.5. Standards, Practices, and Conventions	3-19
3.5.5.6. Reviews, Software Audits, and Software Quality Inspections	3-19
3.5.5.7. SQA of Configuration Management	3-20
3.5.5.8. Tools, Techniques, and Methods for SQA	3-20
3.5.5.9. Quality Factors and Criteria	3-20
3.5.5.10. SQA of Code Control and Media Control	3-23
3.5.5.11. SQA Standards	3-23
3.6 SOFTWARE PROJECT MANAGEMENT	3-23
3.6.1. Introduction	3-23
3.6.2. Management	3-23
3.6.2.1 Project Planning	3-23
3.6.2.2 Project Control	3-25
3.6.2.3 Project Communications	3-25
3.6.3. Documentation	3-26
3.6.3.1. Engineering Documentation	3-26
3.6.3.2. Formal Management Documentation	3-26
3.6.3.3. Informal Documentation	3-26
3.6.4. Configuration Management	3-26
3.6.4.1. Baseline Identification	3-26

3.6.4.2. Control and Tracking of Software Access and Change	3-26
3.6.4.3. Control of Software Releases	3-27
REFERENCES	3-27
Appendix 3-1 - List of Methods vs. Life Cycle	3-28
Appendix 3-2 - Definition of Acronymns	3-32
Appendix 3-3 - Current Languages and Environments	3-36
CMS-2/MTASS	3-36
MTASS HOST COMPUTER SYSTEMS	3-37
FASP	3-38
CORAL-66	3-41
JOVIAL	3-42
LTR3	3-43
THE ENTREPRISE LTR3 ENVIRONMENT	3-46
PEARL	3-48
WERUM PEARL ENVIRONMENT	3-50
GPP PEARL ENVIRONMENT	3-52
Appendix 3-4 - The ADA Language and Current Compilers	3-53
The Ada Language	3-53
ALSYS COMPILER	3-54
AIR FORCE ARMANENT LAB (AFATL)	3-56
DANSK DATAMATIC CENTER (DDC)	3-58
DEPT. OF THE ARMY ALS (Ada Language System)	3-60
DIGITAL EQUIPMENT CORPORATION	3-62
INTERMETRICS, Inc. ADA COMPILER	3-64
RATIONAL ADA ENVIRONMENT	3-66
ROLM AND DATA GENERAL ADA ENVIRONMENT	3-68
SYSTEM KG	3-70
SYSTEM DESIGNERS SOFTWARE	3-71
TELESOFT	3-73
VERDIX ADA DEVELOPMENT SYSTEM (VADS)	3-74
 CHAPTER 4—GENERAL LIMITATIONS OF CURRENT DESIGN SUPPORT ENVIRONMENTS	
4.1 Introduction	4-1
Limitations of Completeness	4-1
Limitations of Integration	4-1
Limitations of Availability	4-1

Limitations of Extensibility and Interchangeability	4-1
Limitations of Performance	4-2
Limitations of Application	4-2
4.2 The Interface between Machine and the Environment - The Operating System	4-3
4.3 The Development Process	4-3
4.3.1 The Phases of Real-Time Software Development	4-3
4.3.2 Requirements, Design, and Implementation	4-4
4.3.3 Verification and Validation	4-5
4.3.4 Development Control	4-6
4.3.5 A Development Process Overview	4-7
 CHAPTER 5—REQUIREMENTS FOR SOFTWARE ENGINEERING ENVIRONMENTS AND TOOLS	
5.1 Introduction	5-1
5.2 The Guidance and Control Environment	5-1
5.3 What Makes a Software Engineering Environment “Integrated”	5-1
5.3.1 The Kernel Environment	5-2
5.3.2 User Interface Requirements	5-2
5.4 General Requirements for Software Engineering Tools	5-3
5.5 System Requirements Tools	5-4
5.6 Software Requirements and Prototyping Tools	5-4
5.6.1 Software Requirements Tools	5-4
5.6.2 Prototyping	5-5
5.7 Basic Software Design Tools	5-5
5.8 Detailed Software Design Tools	5-5
5.9 Software Implementation Tools	5-5
5.9.1 Program-Constructor Tools	5-5
5.9.2 Syntactic Editor	5-6

5.9.3	Compilers	5-6
5.9.4	Program Library Manager (PLM)	5-7
5.9.5	Predefined Package/Module Libraries	5-8
5.9.6	Run-Time-Executives (RTE)	5-8
	Dynamic memory management	5-8
	Exceptions	5-8
	Tasking	5-9
	High level input/output	5-9
	Interrupt Handling	5-9
	Timing	5-9
	The RTE for the Safety Critical Systems	5-9
5.10	Validation and Test Tools	5-10
5.10.1	Symbolic (High Level Language) Debugger	5-10
5.10.2	Concurrency Property Analyser	5-11
	Static Analysis	5-11
	Dynamic Analysis	5-11
	System Simulation: The "Software Rig" Concept	5-12
5.10.3	Failure Modes and Effects Analysis	5-12
5.11	Support Tools	5-12
5.11.1	Text Editor	5-12
5.11.2	Configuration Management	5-13
5.11.3	Project Management	5-13
5.12	References	5-14
APPENDIX 5.1	- THE KERNEL ENVIRONMENT	5-17
A.5.1.1	Introduction	5-17
A.5.1.2	Object Management System (OMS)	5-17
A.5.1.3	Program Execution Primitives	5-18
A.5.1.4	Inter-Process Communication Facilities	5-19
A.5.1.5	Protection Mechanisms to Preserve Data Integrity	5-20
A.5.1.6	I/O Primitives	5-21
A.5.1.7	Distribution Mechanisms	5-21
APPENDIX 5.2	- SOFTWARE DESIGN TOOLS TO SUPPORT THE "OBJECT ORIENTED" METHODOLOGY	5-23
A.5.2.1	- Basic Software Design Tools	5-23

CHAPTER 6—FUTURE TRENDS

6.1 Introduction	6-1
6.2 Control of Complexity	6-1
6.2.1 System specification	6-2
6.2.2 Documentation	6-3
6.2.3 Structured Software Development.	
6.2.4 High-Level Language Implementation	6-4
6.2.5 Standardization	6-5
6.2.5.1 Standardization in the Software Design Process	6-5
6.2.5.2 Standardization in the Implementation	6-5
6.2.6 Software Re-Use	6-6
6.3 Management of Complexity	6-6
6.4 New Techniques	6-7
6.4.1 Very Fast Machines	6-8
6.4.1.1 High Speed Single Processors	6-8
6.4.1.2 Parallel Processors	6-8
Dataflow Machines	6-8
Reduction Machines	6-9
Array Processors	6-9
6.4.2 Multi-Processors	6-9
Systolic Arrays and Similar Arrangements	6-9
Multiple Processor to Memory Mapping	6-9
6.4.3 Distributed Architectures	6-10
6.4.3.1 Tightly Coupled Processors	6-10
6.4.3.2 Loosely Coupled Processors	6-10
6.4.4 Languages	6-11
6.4.5 Artificial Intelligence	6-11
6.4.5.1 Use in the Design Phase	6-12
6.4.5.2 Real-Time Use	6-12

Consistency of Knowledge Bases	6-12
Completeness of Knowledge Bases	6-13
Deletion of Old Data	6-13
Dealing with Data that Cannot Satisfactorily be Explained Using the Available Models of the Environment	6-13
6.5 Conclusions	6-13
The Strong Tendency Towards Distribution of Processing	6-13
The Meaning of Time	6-13
Deadlock	6-13
Distribution of Functions Over the Various Processors	6-14
Testing and Verification	6-14
Technological Innovation	6-14
Distribution	6-14
 7.—EXECUTIVE SUMMARY	
7.1 Introduction	7-1
7.2 General Requirements for Guidance and Control System Design and Implementation	7-1
7.3 Phases of and Methods for Guidance and Control System Design and Implementation	7-3
7.4 Current Status of Tools, Toolsets, and Software Engineering Environments for Embedded Computer Systems Software	7-6
7.5 Limitations of Currently Available Software Engineering Environments	7-10
7.6 Requirements for Software Engineering Environments and Tools	7-11
7.7 Future Trends	7-14

CHAPTER 1

INTRODUCTION

1.1 General

This report has been prepared as a summary of the deliberations of Working Group 08 of the Guidance and Control Panel of AGARD. The Terms of Reference of the working group as approved by the National Delegates Board of AGARD were:

- (i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint.
- (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i).
- (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i).

The working group attempted to consider all existing software design and development technology which existed or was known to be under development at the time the report was prepared. In particular, there are a great number of important software developments taking place in connection with the ADA* and LTR3 high level languages which are under development, and a number of important software engineering environment developments are also taking place in various countries represented on the working group. Where possible, the working group gave special consideration to these developments. However, because of their large number, it has obviously been impossible to do justice to all of them. The particular existing and new technology under development which was considered explicitly by the group is referenced at appropriate points in the report.

The working group started work in May 1983 and met at six month intervals through May 1986. Final editing of the report took place during the last half of 1986 and during 1987.

The working group was composed of members from France, The Federal Republic of Germany, Italy, The Netherlands, the United Kingdom and the United States, all of whom are expert in either guidance and control systems design or software design or both. This report represents the consensus view of the group, but it should not be construed as representing the views or policies of any of the nations, organizations, or individuals represented on the working group.

1.2 General Requirements for Guidance and Control Software

In considering the general requirements for guidance and control software, it is important to distinguish between the two tasks of guidance and control. In general, guidance systems can be considered to be critical to the success of a mission but not to the safety of the aircraft. On the other hand, the flight control and engine control systems incorporated into a modern aircraft are usually essential to the safety of operation of the aircraft. This is reflected in the commonly used specifications for the probability of loss of an aircraft due to a flight control failure being:

$$P_{loss} < 1 \times 10^{-7} \text{ per flight hour (military aircraft)}$$

$$P_{loss} < 1 \times 10^{-9} \text{ per flight hour (civil aircraft)}$$

* ADA is a trademark of the United States Department of Defense.

These very stringent requirements have led to the development of a variety of high integrity design and implementation techniques for flight control systems.

The advent of digital flight control has meant that these failure probabilities have to be applied to the total digital system including the flight software. The failure probabilities assigned to the software are typically an order of magnitude less than those given above, and this has led to great emphasis being placed on fault avoidance and fault tolerance techniques in software design.

1.2.1 Fault Avoidance Issues

Current fault avoidance techniques use structured design methods together with rigorous quality control and systematic testing of the software to insure that the probability of a software 'bug' being introduced or remaining undetected during the software design and development process is extremely low. Such methods should be used in all software development. Their use is especially important for safety critical software which is distinguished from non-safety critical software by the use of very small and very simple modules. Since such modules are relatively easy to verify, it is commonly assumed that high integrity can be achieved through use of such techniques. In practice, however, no matter how carefully the software is designed, it is impossible to establish that it is completely error free because (i) The large number of possible states preclude exhaustive testing (ii) The usual statistical analysis methods which are useful in hardware development are not applicable to software development.

1.2.2 Fault Tolerance Issues

In considering software fault tolerance methods, it is necessary to remember that the adoption of such methods does not mean that structured design techniques should be abandoned. There is a wealth of data available to indicate that, on average, 51% of the faults in a program are not discovered prior to entry of that software into service. Thus, it is necessary to introduce fault tolerance into the design to cope with faults which are not discovered during the design and implementation process. A good fault tolerant design will usually prevent any remaining faults from having a catastrophic effect on the system. However, the belated discovery of such faults and the resultant need to modify, re-verify, and re-validate the program usually increases the life cycle costs dramatically. Thus, fault tolerant methods and fault avoidance methods go hand-in-hand. If both are used, the resultant design has the best chance of being extremely reliable and having the required level of integrity.

1.2.3 Computer Language Issues

The requirement for high integrity, engendered by the knowledge that a system failure could mean the loss of an aircraft and the death of one or more people, has meant that the flight control system designer has taken a very conservative approach to the use of software and to use of the various languages that have been developed. As a result, most digital flight control systems developed to date have had their software written in assembly language.

1.2.4 Flight Control Systems Requirements Issues

Table 1 shows some of the typical requirements placed on flight control systems for modern aircraft and the implications that these requirements have on software design. While these requirements for digital flight control are not unique individually, collectively they represent the most demanding requirements in the field of guidance and control.

From the designer's standpoint, safety critical software requirements provide a variety of constraints, situations, and challenges which are much more stringent than those seen by mission software system designers. Table 2 documents some of these considerations. Because of the direct and strong interaction of the software with the rest of the control systems and with the aircraft itself, the software designers of such systems are usually control engineers who understand software rather than software engineers with a limited understanding of flight control systems technology.

Software development time and cost, including the incorporation of software changes, have been greatly and adversely impacted by computer capacity constraints. One is tempted to view the rapid emergence of high

Characteristic	Major Concern	Implications
Real-Time Operations	Throughput and Timing	Precision Task Execution
Closed-Loop Operation	Performance and Stability	Minimal Time Delays to Avoid Loop Phase Loss
Bandwidth Variations	Improper Balance of Throughput/Loop Bandwidth	Multiple Rate Scheduling
High Integrity	Fault Tolerance Coverage	Similar/Dissimilar Redundancy
Crew Interaction	Individual Pilot (Crew)/ System Match	Robustness
Vehicle Uncertainties and Complex Variations	Non Recoverable Conditions	Robustness
Operational Extremes	Performance and Stability	Automatic Recovery and Boundary Limits
Dispersed/Non-Ideal Installation	Disruptions	Non-Susceptibility
Multi-Loop	Mixed Iteration Rates	Multiple Rate Scheduling
Multi-Mode	Increased Complexity	More Complex Systems
Interface to Guidance Systems	Synchronization and Data Integrity	Sensor Use of Sensor Inputs, Timing of Data Inputs

TABLE 1

DIGITAL FLIGHT CONTROL SYSTEMS CHARACTERISTICS AND IMPLICATIONS

throughput machines as providing a solution to this problem. History, however, documents numerous examples of mistaken throughput requirement assumptions, the U.S. Space Shuttle flight control system being a prime example. An often heard industry axiom is still true until disproven: "The capacity of any flight computer is exceeded only by the human fascination for filling it up with useful functions."

Two trends in aircraft/avionics systems design will place even more importance on software fault avoidance and fault tolerance. These are:

- Increased use of relaxed static stability.
- Integrated avionics and control functions.
 - Integrated flight control/navigation.
 - Integrated flight propulsion control.
 - Flight path management for low altitude missions.

Consideration	Implication
Control Law Algorithm Driven Design	Strict Top-Down Development Not Possible
Airplane/System Design Iterations	Increased Incidence of Unavoidable Requirements Changes
Crucial Front-End Analytical Tasks	Suitable Models Required, Analyses Impact Early-On Design
Significant Post-Integration Development	Built-In Modifiability Provisions and Support Essential
Specialized Hardware and Software	Substantial Tailoring of Methods Often Necessary
Safety Criticality	Optimization and Complementarity of Methods Required
Customer/Regulatory Review	Visibility and Credibility of Evolving Design Necessary

TABLE 2
DIGITAL FLIGHT CONTROL DESIGN CONSIDERATIONS

Figure 1 shows the trend in reducing static stability in unaugmented fixed wing aircraft. Displayed as "time to double amplitude", the data also directly illustrates the decreased failure recovery time available in such vehicles. The most extreme case shown is the X-29 forward swept wing demonstration aircraft. The extremely short "time to double" for the X-29 results from the fact that the airframe was designed to optimize performance throughout its subsonic-supersonic flight envelope. The payoff in performance is significant. The challenge to flight control fault tolerance is likewise significant. In addition to military aircraft, the Boeing Commercial Airplane Company is studying the IAAC (Integrated Application of Active Controls) concept which represents a new, highly integrated commercial jetliner control concept. This new concept also will allow a high level of optimization of aircraft performance with equally stringent demands on the flight control system dynamic response and fault tolerance.

1.2.5 Integration Issues

Integration has a different, perhaps even more challenging, impact on software design and software fault tolerance. Fixed wing aircraft technology is driving towards integration in a number of areas. Table 3 summarizes some of these areas and the related impacts on fault tolerance. These trends could get even more exciting in the helicopter world. Although performance payoffs for highly unstable rotary wing vehicles are not envisioned, single seat light helicopters using a significant level of flight system automation are under development. Such helicopter designs could place every element of the guidance and control suite into the flight critical function category. This includes all those functions listed in Table 3 for fixed wing aircraft plus inertial navigation. Furthermore, the helicopter nap-of-the-earth mission is extremely difficult, requiring complex algorithms and fault recovery response times which are very short.

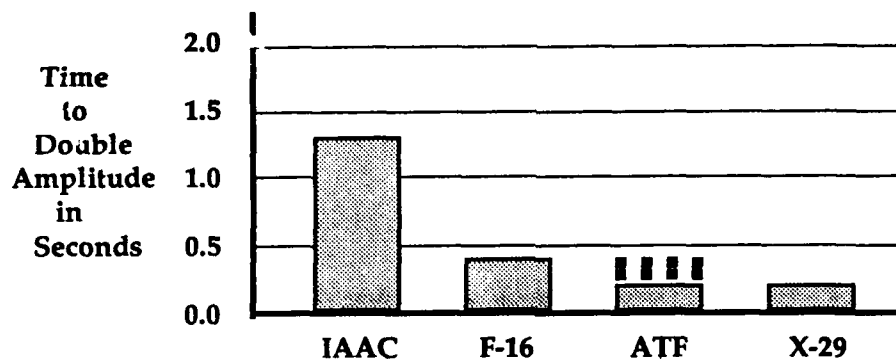


FIGURE 1

TRENDS IN RELAXED STATIC STABILITY

Concept	Description	Crucial Function Impact
Dispersed, Integrated Flight/Navigation Sensors	Sharing of Strapdown Navigation Sensors with Flight Control	More Complex FCS Algorithms for Sensor Normalization, Redundancy Management, and Survivable Dispersion
Integrated Flight/Propulsion Control	Sharing of Information Between Engine and Flight Control Systems, Vectored Thrust	Flight Crucial Engine Control
Flight Path Management	Real Time Optimal Control for TF/TA, Complex Route Decisions at Low Altitude	Low Altitude Automatic Flight Management is Crucial, Sensor Blending Concepts are Flight Crucial, Huge Terrain Data Bases (e.g., DTM) are Flight Crucial

TABLE 3

EXAMPLES OF IMPACTS OF INTEGRATION CONCEPTS ON COMPLEXITY

1.2.6 Flight Control System Software Complexity Issues

Currently implemented flight control software is not particularly complex. This fact would provide us with some comfort if the situation were static. However, interesting and potentially very useful new technological developments such as real-time, on-board optimization and intelligent controls could make flight control software much more complex in the near future. The impact of some of these trends in new technology are summarized in Table 4. Mission critical software tends to be more complex and larger than flight critical

software and may require larger development teams. While the levels of integrity required for mission critical software are lower than those for flight critical software, many of the problems, particularly those associated with real-time operations and crew interaction, remain. Thus the development process for mission critical software is also long and complex. In both cases, an integrated software engineering environment which contains tools specific to the needs of real-time operation is required to support both the initial development of the software and its postdevelopment support.

1.2.7 General Requirements for Tools and Support Environments

In considering software development needs for guidance and control, it is obvious that major improvements are needed both in the tools required and in the general support environment. The main body of this report addresses these needs.

Function	Program Size Complexity	Program
Inner Loop		
• Relaxed Static Stability • Gust Load Alleviation • Ride Quality • Flutter Mode Control • Integrated Control	Modest Increase Modest Increase Modest Increase Higher Sample Rates Modest Increase	Modest Increase Modest Increase Modest Increase Modest Increase Modest Increase
Outer Loop		
• TF/TA/OA • AI Based Decision Making • Integrated Control • Optimal Flight Path Control	Significant Increase ? Significant Increase Significant Increase	Significant Changes Significant Changes Significant Changes Significant Changes
Redundancy Management		
• Analytic Redundancy • AI Based techniques	Modest Increase ?	Modest Increase Significant Changes

TABLE 4

FUTURE FLIGHT CONTROL SOFTWARE TRENDS

It is essential that software development methods not be completely divorced from the development methods used for the total system. What is needed is a set of total system development methods from which the software needs can be defined and developed. The methods needed should:

- Be problem driven and applications based.
- Deal conveniently with combined hardware/software interactions in concurrent mechanizations.
- Address complexity and reliability concerns explicitly and quantitatively.
- Effect component integration during design.

- Include provision for quality assurance.
- Provide a natural transition from system requirements to software design to hardware/software implementation and integration.

Any support environment developed should be oriented towards practitioners in various application areas and provision should be made for the future incorporation of user developed tools. It is also likely that the software development environment will need to be incorporated into a system level prototype facility.

High level programming languages are powerful tools, but they are essentially only implementation tools. What is needed is a variety of appropriately integrated tools working at the different levels of abstraction appropriate for the different phases of the development and postdevelopment support process. This need for higher levels of abstraction will limit the use of ADA as a program design language except at levels immediately above the detailed coding level.

It should be remembered that the problem of software reliability is one of the most important problems facing the real-time system and software designer. To assist in achieving high levels of software reliability, formal methods of design development and proof of design correctness have been advocated as a way of improving the software development process. Currently, such techniques are in their infancy and their inflexibility and obstructiveness, combined with their inability to handle anything but the smallest programs, will preclude their use in the near term. In the long term, however, formal methods could be of great value and any support environment developed specifically for guidance and control needs should have the capability of incorporating such methods when they are eventually available.

The introduction of such formal methods will probably require:

- Stricter definition of requirements.
- System-level formulations and methods.
- Supplementary forms of documentation that provide highly visible and highly accessible information on the complete design and development process.
- Capacity to handle concurrent real-time tasks.

1.3 Structure of the Report

The remaining chapters of this report examine the current technology and future needs as outlined above in more detail. In particular,

- Chapter 2 deals with the development cycle used in guidance and control systems.
- Chapter 3 examines the current status of tools to support this development cycle.
- Chapter 4 discusses the limitations of current tools and system development support environments.
- Chapter 5 examines the needs for additional new and/or improved tools and software engineering environments in the near term.
- Chapter 6 considers the longer term requirements for tools and support environments.
- Chapter 7 contains an overall summary of the report.

CHAPTER 2

PHASES AND METHODS

FOR SYSTEM AND SOFTWARE DEVELOPMENT

2.1 Introduction

The development and postdevelopment support of flight guidance and control systems, avionics systems, and armament systems includes all measures from initial system studies and design to final system integration of a number of complex subsystems. The complex subsystems are, in turn, designed and implemented from many hardware and software components in order to provide the required system functions.

System design involves definition of the overall system functions and their methodical decomposition into well defined subfunctions. The detailed requirements for all hardware and software implemented subfunctions and their performance have to be derived, and the interfaces between the (hardware and/or software) components used to implement the various subfunctions have to be defined.

The development is neither purely hardware nor purely software oriented. The separation of the overall system functions between those to be implemented in hardware and those to be implemented in software occurs during the development process as a consequence of a series of design implementation decisions. Therefore, design criteria for both hardware and software have to be considered in the overall system development process.

The overall flight guidance and control system development and postdevelopment support processes can be usefully viewed in terms of a hierarchical system development model which is based on a set of phases and which makes use of related design and test concepts which are supported by necessary project documentation and appropriate tools. Implicit in the use of this hierarchical system development model is the definition of flight critical functions from the very beginning of system development through all phases of the hierarchical development process. Moreover, the issues of fault tolerance and strong time dependency of the various functions to be implemented are very important in guidance and control system design, and they must be considered and continuously monitored during the development process; i.e., they must be part of the decomposition process itself, be well documented, and be explicitly considered during formal reviews and walk throughs.

2.2 Flight Guidance and Control Systems Specifics

For flight guidance and control systems (FGCS), the software life-cycle is dominated in all phases by reliability and integrity considerations. There are two major aspects to be considered: achievement of the high reliability and integrity needed for safety critical systems such as flight control, and the safety audit or validation of reliability and integrity required for flight certification. Thus, although the FGCS software life-cycle includes all of the phases shown below which are common to most software projects, special requirements, methods, and interfaces between phases generally have to be developed for guidance and control systems. This need for special requirements, methods, and interfaces is strongest for systems most critical to flight safety such as flight control systems which must satisfy stringent requirements on

- Configuration
- Performance (fast real-time; master frame duration from 1-50 msec)

- Reliability
- Survivability
- Supportability/Ease of Modification

and which are constructed from a unique combination and integration of

- Data transmission (exchange) systems
- Sensor systems
- Flight control computers
- Actuation systems
- Back-up systems

into a multiply-redundant overall system implementation.

Typically, guidance and control software consists of

- Flight resident (airborne) software for
 - Executive functions (scheduling, I/O control, redundancy management)
 - Air data and incident sensor computations
 - Inertial attitude and navigation computations
 - Command and stability augmentation computations and automatic flight control mode computations
 - Back-up monitoring and built-in-test computations
- Non-flight (ground based) software for
 - Test and simulation
 - Overall system development support including design and development support software

Both flight resident software and non-flight software are used in the development process. They are usually developed independently but they interact strongly during the system development process.

Because of the specifics mentioned above, the following most important requirements exist for guidance and control software:

- Satisfaction of fast real-time constraints
- High reliability
- High availability/survivability
- Supportability/ease of modification

2.3 The System Development Process

As noted above, the overall system development process can be usefully viewed in terms of a hierarchical model of the various phases of the process. A specific example of such a model which is well-suited for guidance and control system development is proposed and discussed below. This particular model was used as the basis for all of the detailed deliberations of the working group.

2.3.1 The Hierarchical Phase Model

The various phases of system development included in the hierarchical phase model each require definition and explanatory discussion. These are provided in the next several sections of this report.

2.3.1.1 General

According to general and widely accepted system development concepts, the system development process can be logically divided into the set of relatively distinct phases listed below.

- Study phase
- Concept phase
- System design phase
- System development phase
- System integration phase
- Production phase
- Postdevelopment or in-service phase

Due to the criticality of system and safety aspects and the presence of significant hardware/software interactions, the general phase model described above has to be modified for guidance and control systems as described below and as shown in Figure 2.1.

- Study phase
- Concept phase
- System requirements phase
- System design phase
- Subsystem requirements phase
- Subsystem design phase
- SW requirements phase ————— HW specification phase
- Basic software design phase
- Detailed software design phase
- Coding/module test phase ————— HW development phase
- SW integration on host computer phase
- SW integration on target phase ————— HW integration phase
- Subsystem integration phase
- System integration phase
- Flight test phase
- Production phase
- Postdevelopment support or in-service phase

Depending on the size of a project, the system/subsystem requirements phase and the system/subsystem design phase can either be combined or can be split up as shown.

This more detailed breakdown of the phases of system development as given above can be used to define a general hierarchical system development phase model for specifying

- The steps of system development from the customer requirements to final system integration
- The format and contents of the documentation to be produced in the individual phases
- The verification and validation measures controlling the system development from both technical and procedural points of view

Such a general hierarchical system development phase model is given in Figure 2.2.

For every project, detailed project plans and a project handbook should be produced. The project plans and the project handbook should specify all steps, documents and reviews precisely and in conformity with the general system development phase model and with the customer requirements. They should also define all other documents associated with and required for project execution (e.g., quality assurance plan, data processing management plan, etc.).

The actual process of conducting a guidance and control system development project is determined by:

- Customer requirements. These are defined in a requirements document representing the technical part of a contract.
- Constraints due to usage of "off the shelf" equipment and available software (customer furnished software, reusable software, etc.).
- Analysis of the requirements/constraints. To obtain a complete structured design it is appropriate to perform the analysis at several levels such as
 - System level
 - Subsystem level
 - Functional decomposition down to module level
 - Software requirements, equipment specifications
- Analysis of real-time behavior
- Hardware and software development based on the software requirements and the hardware specifications.
- Test of hardware and software, integration tests, acceptance tests, and clearance tests.

Each of the phases of the hierarchical system development model described in Figure 2.2 is discussed in some detail in the following sections of the report.

2.3.1.2 The Study Phase

The task of the Study Phase should be to

- Evaluate the guidance and control possibilities and limitations of the flying vehicle for which the system is being developed.
- Investigate the feasibility of satisfactory vehicle guidance and control system behavior, with special emphasis on feasible real-time behavior.
- Analyze physical configuration and layout possibilities and any related performance implications.

- Investigate existing equipment and technologies.
- Estimate
 - Risks
 - Costs
 - Development duration

Phase	Software Development	Hardware Development
Study		
Concept		
System design • System Requirements • System Design • Subsystem Requirements • Subsystem design • Software Requirement/ Hardware Specification		
System Development • Basic Software Design • Detailed Software design • Coding/Module Test • Software Integration on Host Computer		
System Integration • Software Integration on Target Computer/Hardware Integration • Subsystem Integration • System Integration • Flight Test		
Production		
In-Service		

FIGURE 2.1

**SYSTEM DEVELOPMENT PROCESS SHOWING COMMON AND SEPARATE
HARDWARE AND SOFTWARE DEVELOPMENT PHASES**

2.3.1.3 The Concept Development Phase

The task of the Concept Development Phase should be to

- Define and evaluate alternative system concepts.
- Select the best overall system concept.

The basis for the evaluation should be:

- Performance
- Design and development risk
- Cost
- Reliability and fault tolerance characteristics

2.3.1.4 The System Requirements/System Design Phase

The task of the System Requirements/System Design Phase is to produce a System Specification which incorporates the requirements of the customer and the integration of any further known requirements. All customer requirements influencing the entire system either directly or indirectly should be covered in the System Specification, along with a careful specification of the criticality of each of the required system functions.

A failure mode and effects and criticality analysis (FMECA) is strongly recommended as a requirement for any guidance and control system development, and it should be listed as a requirement in the System Specification. Test philosophy and test strategy also have to be defined and documented in a System Test Specification. *These FMECA and test aspects have to be refined in each of the following phases up to and including the detailed software design phase.*

A general model for a System Specification document for the development of guidance and control systems is given in Appendix 1.

The System Specification document should be evaluated and verified against the customer's Statement of Work in a formal System Design Review by a group consisting of the project management, the specialty field leaders, and, normally, by a representative of the customer. After the System Specification is verified, it becomes the binding basis for all subsequent development effort. At this point, the document is placed under configuration control, and all changes to it have to be evaluated and re-verified against the customer's Statement of Work. Moreover, all subsequent technical details of the development process should be verified against this document.

In addition to the System Specification, a System Architecture document should be produced which describes the entire system architecture, the connections among all subsystems, and preliminary essential installation data (e.g., sizes and weights). These data are revised and supplemented at appropriate points throughout the process of system development. This document is not separately verified and controlled because it contains only a summary of information available in other verified and controlled documents.

2.3.1.5 Subsystem Requirements/Subsystem Design Phase

The System Specification defines the requirements for all subsystems in a comprehensive functional form. The task of the Subsystem Requirements/Subsystem Design Phase is to produce Subsystem Specifications for all subsystems defined in the System Specification.

Each of the Subsystem Specifications should contain:

- A functional description of the subsystem including a copy of the specific subsystem functional requirements contained in the System Specification

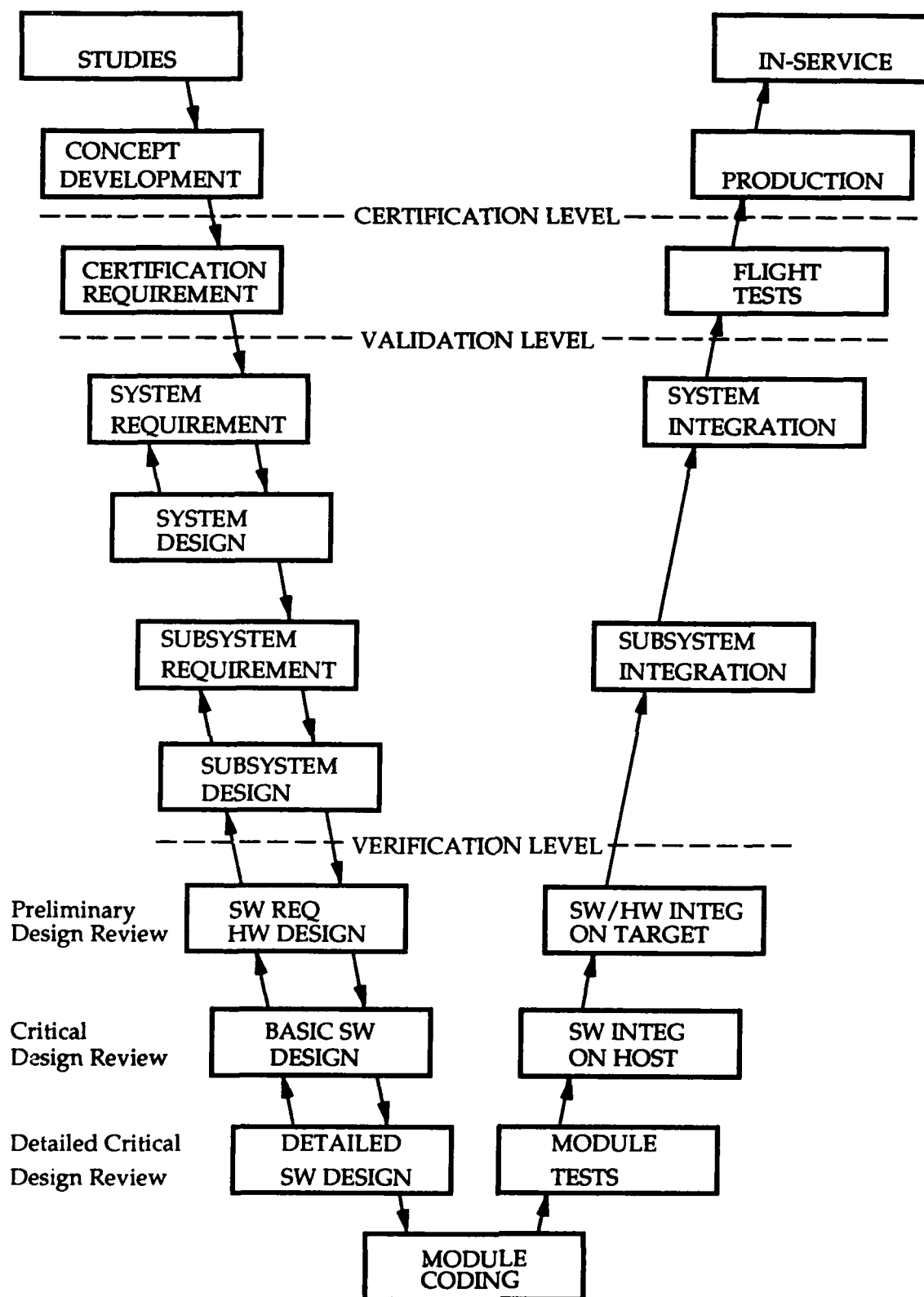


FIGURE 2.2
HIERARCHICAL SYSTEM DEVELOPMENT PHASE MODEL

- The subsystem design concept and subsystem architecture
- Subsystem criticality classification
- Requirements on
 - Sensors and effectors
 - Processors
 - Displays and controls
- A detailed description of the subsystem's interface with all other subsystems and with the system as a whole

A general proposal for a model Subsystem Specification document is given in Appendix 2.

The Subsystem Specifications form the basis for the Subsystem Design Phase. Detailed solutions are not expected in these documents even if they are known. However, the necessary real-time behavior requires detailed analyses of the required sequential execution and timing dependencies among the subsystems and such analyses should be performed during this phase. Also, all requirements relating to

- Allowable dead/delay times
- Allowable and/or required parallel activities
- Synchronization of control loops
- Bus traffic

have to be investigated.

If they are available, derivations of algorithms and constants should be listed in the appendix. Also all requirements on hardware to be used in implementing the particular subsystem being designed should be contained therein.

The Subsystem Specification documents are submitted to a technical review, the Subsystem Review, at which time they are verified and placed under configuration control.

In parallel to the detailed subsystem design activities, test requirements for equipment, software, and subsystem and system integration have to be derived and recorded in a Subsystem Test Specification. This document should be used as basis for deriving all tests and test procedures.

2.3.1.6 The Software Requirements/Hardware Specification Phase

Detailed subsystem design is completed through further stepwise refinement of the subsystem requirements as given in the individual Subsystem Specification documents. The detailed requirements for the hardware and software components to be used to implement the subsystems must be derived from the purely functional requirements during this phase.

Some of the requirements for the hardware components may be determined by use of certain software components and by the design environment; e.g., by

- The computing power (speed and memory) required to execute the software within the time allowed.
- The availability of a particular software development environment.

Also, certain functions might be required that can be realized only through equipment specific software; e.g.,

- Built-in test (BIT).

- Data format conversion.
- Specific sensor functions (e.g., Inertial Navigator, Display).

The final separation of subsystem functions between hardware and software components has to be carried out for each subsystem during the Software Requirements/Hardware Specification Phase because the precise boundaries between hardware and software cannot be determined in an effective way prior to completion of the Subsystem Requirements/Subsystem Design Phase. The results of the Software Requirements/Hardware Specification Phase should be recorded in the following appropriate design documents.

- Software Requirements Documents (including descriptions and derivations of all algorithms to be implemented)
- Equipment Specifications
- An Interface Definition and Control Document

All of these documents should be verified against the appropriate Subsystem Specification documents during a Critical Design Review and placed under configuration control.

2.3.1.7 The Basic Software Design Phase

The objective of the Basic Software Design Phase is to transform the software requirements for a subsystem into a software module design and into a software module test design. The program structure, priorities, program flow, module functions, and interfaces with other modules all have to be determined during this phase.

With respect to timing considerations, a framing concept defining the module calling sequences should be worked out based upon experience from existing similar systems. Also, the results of the bus load analysis (2.3.1.5.) should be confirmed by rapid prototyping tests.

The results of the Basic Software Design process should be contained in the following documents:

- A set of Software Module Specifications.
- A set of Module Test Requirements Documents.

At the end of the Basic Software Design process, both sets of documents should be verified against the appropriate Software Requirements Document during a formal review and placed under configuration control.

2.3.1.8 The Detailed Software Design Phase

During the Detailed Software Design Phase, the internal module structure, the algorithms to be implemented, and the data structures to be used are completely defined. The internal module structure is usually described in pseudo-code. Also, a Module Test Specification has to be completed.

The following information developed during the Detailed Software Design Phase should be documented:

- Module descriptions
- Pseudo-code of the modules /13/
- Detailed interface and data description
- Cross Reference Listings
- Module Test Specifications

Formal technical reviews should be conducted to confirm that

- The results of the Detailed Software Design Phase are consistent with the Software Module Specifications.
- The required software quality attributes are achieved.
- All other software and system requirements are met.

After the release of the software design, the documents generated during the Detailed Software Design Phase are placed under configuration control.

2.3.1.9 The Module Coding/Module Test Phases

The documents generated during the Detailed Software Design Phase form the basis for all module coding and related module test activities. During the Module Coding/Module Test Phases, the modules are coded and commented; syntax errors are debugged, and code inspection and code walk throughs are performed. Also, all previously specified module tests are prepared. Test stubs and drivers are established. Static and dynamic module tests are executed. Detected errors are corrected and documented.

The following information summarizing the results of the Module Coding/Module Test Phases should be formally documented:

- Program/module descriptions
- Data descriptions
- Interface descriptions
- Source code with comments
- Module link information
- Test data
- Sources of stubs and test drivers
- Test protocols
- Test reports

At the end of the Module Code/Module Test Phase activities, a formal technical review should be conducted to verify that

- The requirements contained in the documents generated during the Detailed Software Design Phase have been met.
- The programming standards adopted for use by the project team have been followed.
- The modules have been tested in accordance with the specified module test specifications.
- The required software quality attributes have been achieved.

Then the documents summarizing the results of the Module Coding/Module Test Phases are put under configuration control.

2.3.1.10 The Software Integration on Host Computer Phase

After the software modules have been coded and tested, they are integrated as far as possible on the host computer used to develop the modules. The inputs and outputs to/from each software module are simulated

for use during the integration process. Next, the individual modules are successively integrated one at a time into software subsystems and into the complete software system for the target computer. Then, all functions of these growing software packages are systematically tested according to Software Integration Test Protocols developed during the Software Integration on the Host Computer Phase.

The completed and tested modules to be linked for program execution are then placed under configuration control. Configuration control should also keep track of any corrections and/or modifications to the software modules.

The documentation which should be generated during the Software Integration on the Host Computer Phase and subsequently placed under configuration control consists of detailed descriptions of the Software Integration Test Protocols.

2.3.1.11 The Software Integration on the Target Computer Phase

If the integrated software executes all functions correctly on the host computer, the code for the target processor is generated and the software is loaded into the target computer. Next, the software components that handle the hardware interfaces to the target computer are integrated one at a time. Systematic functional tests are required to validate this software integration on the target computer.

The specified execution times of the modules and the overall program should be confirmed by direct measurement on the target computer and/or by code analysis. The external environment used for these tests is normally simulated.

The results of the integration tests on the target computer should be documented in a Target Computer/Software Integration Report. The software modules to be linked for execution on the target computer should be placed under configuration control. The configuration control system should keep track of any modifications and error corrections made to the software modules during software integration on the target computer and should insure that the appropriate verification activities are properly performed.

2.3.1.12 The Subsystem Integration Phase

When all individual equipment has been sufficiently tested, the total subsystem is tested on a rig. The total subsystem behavior for all flight critical functions is investigated during this phase in as realistic an environment as can be provided in a rig. Also, to the maximum extent possible considering the limitations of rigs, the performance of all subsystems should be validated against the subsystem requirements as specified in the Subsystem Specifications.

2.3.1.13 The System Integration Phase

In direct analogy to the Subsystem Integration Phase, the task of the System Integration Phase is validation of the fact that the overall system satisfies the system requirements as specified in the System Specification.

Any required integration of the flight resident software and the non-flight software is also performed during the System Integration Phase.

2.3.1.14 The Flight Test Phase

Starting with basic functions, more and more complex functions are integrated step by step during the System Integration Phase until complete system performance is demonstrated to the maximum extent possible. The task of the Flight Test Phase is to carry out all flight tests required to certify the performance and safety characteristics of the overall guidance and control system.

2.3.1.15 The Production Phase

During the Production Phase, the airborne flight resident software has to be copied and transferred to the target computers. Tests have to be performed during this phase to assure the integrity of the copying process.

2.3.1.16 The In-Service or Postdevelopment Support Phase

Postdevelopment support activities, including the correction of errors discovered during system operation and the incorporation of additional system capabilities after the guidance and control system enters its operational phase, have to be carried out using the same methods and processes which were used in the original development of the system. That means the same phase model has to be applied to each of these activities.

2.3.2 General Methods for Guidance and Control System Development

In principle, the hierarchical model of the phases of system development described in detail above is independent of special methods or tools. However, good methods can alleviate the difficulties and improve the efficiency of the system development process significantly. Some general requirements for methods which could improve the overall system development process and some of its important phases are discussed below.

2.3.2.1 General Requirements for System Development Methods

Some general requirements on methods to support the system development process can be defined as follows.

- The methods should support the structured decomposition of the system and of the system software according to the following criteria:
 - Data flow
 - Data descriptions
 - Process description
 - Time dependencies
 - Criticality of functions
 - *Parallelism of tasks*
 - Test strategies
- The methods should be consistent and use a single semiformal language.
- The methods should support changes.
- The methods should be supported by an integrated set of tools which enforce strict adherence to the methods themselves.

A proposed set of methods for the individual phases of system development proposed above can be found in Figure 2.3. They are discussed below relative to the various phases to which they apply.

2.3.2.2 Requirements for Methods from the Study Phase through the Software Requirements/Hardware Specification Phase

Methods for the Study Phase through the Software Requirements/Hardware Specification Phase should:

- Generate consistent, complete, easy to read, and well-structured comments.
- Support the parallel work of teams on different parts of the system.
- Be adaptable to the problem area.

According to the principle of structured decomposition, the required system functions should be partitioned into basic functions in several steps. The data flows, the data characteristics and structures, the functions, the criticality of the functions, and the time dependencies of the functions should be described in appropriate detail at each step of the process. Then the basic software functions should be allocated to software components, input/output devices, and processors. The resultant allocation of the basic functions to

software components, input/output devices, and processors, together with their individual requirements and their necessary interactions, provides the structure for the Software Requirements Document.

The diagrams commonly used in such documents to provide functional descriptions of the system functions should be supplemented by additional information. Thereby, a unique allocation of basic system functions and their associated Software Requirements Documents/Equipment Specifications would be obtained which would support configuration control of the total system documentation.

A well known and established approach to structured decomposition is the Structured Analysis and System Specification (S.A.) method developed by Tom d. Marco /1/. This method is well suited to the construction of

- A functional model of the system consisting of
 - A network of functions (including elementary functions and data streams)
 - A list of interfaces (data dictionary)
- A state transition model expressing
 - The possible system states
 - The conditions for changing the system states

However, some desirable features which are not provided by the Structured Analysis and System Specification method are:

- Description of the dynamic behavior of the system by formal constructs which are more powerful than normal text and tables.
- Provision of a formalism to represent safety critical parameters, functions, and time dependencies.
- Incorporation of the commonly used symbolism of the relevant technical disciplines which is often more expressive and easier for specialists to use and understand than the symbols of Structured Analysis (e.g., control law symbolism).

These features could and should be incorporated into the Structured Analysis and System Specification method and into the corresponding tools which support this method.

2.3.2.3 Requirements for Methods for the Basic Software Design Phase

For the Basic Software Design Phase, the method of "Structured Design" (S.D.) /2/,/3/, /4/ based on information hiding and abstract data types is state of the art. This method allows a clean modularization of the software to be developed if the following most important considerations are observed during the system software decomposition process:

- The degree or type of coupling between different modules, with loose coupling being good. The possible types of coupling between modules include
 - Data coupling
 - Stamp coupling
 - Control coupling
 - Common data coupling
 - Content coupling

METHODS PHASES	STRUCT ANAL	STRUCT DESIGN	PSEUDO LANG	STRUCT PROG	STATIC ANAL	DYNAMIC ANAL	SPECIAL METH
Studies							
Concepts							
System Reqs							
System Design							
Subsystem Reqs							
Subsystem Design							
SW Reqs HW Specs							
Basic S/W Design							
Detailed SW Design							
Coding Module Test							
SW Integ on Host							
SW/HW Integ on Target							
Subsystem Integ							
System Integ							
Flight Test							
Production							
In-Service							

FIGURE 2.3

MATRIX OF METHODS VERSUS PHASES

- The degree or type of cohesion within a module with strong cohesion being good. The possible types of module cohesion include
 - Functional cohesion
 - Sequential cohesion
 - Communicational cohesion
 - Procedural cohesion
 - Temporal cohesion
 - Coincidental cohesion

where module cohesion is a measure of the degree of functional association between the elements of a module. Functional cohesion is the best form of module cohesion for postdevelopment support and ease of program modification, while coincidental cohesion is obviously the worst kind of module cohesion for postdevelopment support and ease of program modification.

2.3.2.4 Requirements for Methods for Detailed Software Design

For detailed software design, the application of pseudo language and structograms a la Nassi and Shneiderman is state of the art. The semiformal language used in the antecedent phases has to harmonize with the pseudo language of the detailed software design phase with the capability of downward and upward cross-referencing. The pseudo language also should contain almost all important elements and structures of the programming language to be used to implement the software.

2.3.2.5 Requirements for Methods for Coding/Module Test

For coding, the principles of Structured Programming (S.P.) should be applied because it is always much easier to make a structured working program efficient than it is to get a structurally efficient program working properly.

Since testing of software is a very sophisticated and time consuming process, it is difficult to define the general requirements for a module test method which can be recommended for verification of software modules. Nevertheless, based on direct module test experiences on different embedded system projects, two procedures for module testing have proven to be particularly useful. They are:

- Code walkthroughs (static analysis)
- 100% coverage tests of C1-criteria (dynamic analysis)

For a 100% coverage test of C1-criteria, structure analysis is required including control flow analysis, data flow analysis, and generation of test cases and test data. If dissimilar software development is used to achieve software fault tolerance, cross testing of the dissimilar software is also required.

2.3.2.6 Requirements for Methods for Integration and Testing

All modules have to be grouped into subsystems. The unit tested modules of the subsystem are brought together incrementally (one by one) using stubs and dummies which are replaced successively by the real modules. Then unit tested subsystems are integrated into the whole system in a similar way.

2.3.2.7 Resume

Use of powerful design and development methods will result in systems and related software components which are:

- Highly reliable and survivable.
- Available a large percentage of the time.
- Easy to support during the postdevelopment phase.
- Easy to modify to incorporate additional capability.

However, the current state-of-the-art in software engineering tools and environments is not sufficient to support the existing and desired new methods for guidance and control system development. Therefore, efforts are urgently required to improve the current state-of-the-art of such environments.

Examples of the current development of new software engineering environments are to be found in the STARS program (Software Technology for Adaptable Reliable Systems) /7/ and, more specifically, the Joint Services Software Engineering Environment (JSSEE) development /8/ now underway in the U.S. All of these developments will realize their full performance potential only if the tools contained in them support clear and effective methods for each phase of the development activity. The working group believes that, in order to build high quality software engineering environments, it is necessary for the tools to enforce the methods used. This aspect is especially important if development is done by large project teams. Thus, for example, the Methodman requirements /9/ should be implemented in the JSSEE and similar SEEs.

One important point not explicitly mentioned in current JSSEE documents is the need for a well-defined, standardized interface to databases used to support the activities within each phase of the development process. If such an interface is provided, it is not necessary to have a single continuous set of tools covering all phases of software development. Rather, it is then possible to use the tool which best matches the specific needs of any particular system development.

It is worthwhile to emphasize the ultimate requirement for all tools generated in JSSEE to use the CAIS (Common APSE Interface Set) /10/ standard when they are implemented on computers. Otherwise the required portability of tools is not guaranteed.

A final open issue in the area of methods for the development of guidance and control systems and related software is the need for a methodological approach to the development of test, verification, and validation strategies. One suggestion is given in /11/, but further research and development is necessary to achieve proven results.

2.4 Project Documentation

In this section, a general schema is proposed for all phase dependent technical documentation and all phase independent documentation required to support guidance and control systems development. This schema is designed with the goal that documents be easily managed by a computer supported library system.

According to the proposed schema, phase dependent technical documentation is separated into:

- Overall system development documents
- Software engineering documents
- System integration documents

The additional phase independent project management documentation is separated into:

- Standards documents
- Quality assurance documents
- Organization documents

These various types of documents are discussed below.

2.4.1 System Development Documents

At the highest level of the hierarchy of system development documentation are the customer requirements (e.g., operational and tactical requirements, Statement of Work, etc.).

Beyond this, the phase dependent technical documentation should be split into:

- Analysis documentation, which keeps all information of the decomposition process down to the function module level.
- Official documentation for contractual purposes.

The analysis documentation should include:

- An analysis tree, which contains the decomposition of the system into functions, the functions into subfunctions, etc., down to the system hardware and software components.
- Minutes of reviews of analysis steps.

As a first step in the process, the system is decomposed into basic functions with the following documentation:

- Technical specifications for the basic functions.
- Documentation of the process of further decomposition of the basic functions into modules including specifications for the resulting modules.
- Minutes of walkthroughs of the specifications and documentation of all resulting basic functions and modules.

The first level of the "official" documentation for contractual purposes should contain:

- The system specification
- A system description
- An interface control document (ICD)

In view of the decomposition of the system into several subsystems, the following additional documentation is also required:

- Subsystem specifications
- Software Requirements Documents for all software subsystems
- Equipment Specifications for all hardware subsystems
- Subsystem test requirements documents

There should be a one-to-one identity between the technical contents of the "analysis" documentation and the "official" documentation. Furthermore, this one-to-one identity should be enforced by a configuration control tool.

2.4.2 Software Engineering Documents

Software development is divided into two areas:

- Tool and software engineering environment development
- Operational software development

The software engineering documentation can be subdivided accordingly. The top-level documents are the definitions of the run time system on the host and target computers.

The documentation of the tools and the software engineering environment used (if any) should include:

- Compiler specifications
- Debugger specifications
- Descriptions of all other development tools used
- Documentation of all test tools
- Acceptance documentation for all tools and environments

Details have to be specified for every individual case.

Independent of whether the operational software has to be developed for one or several processors, the operational software documentation should consist of the following documents for every processor:

- Software requirements documents for each processor's software
- Software design documents
- Test requirements and test documentation

During the software design, the software for each processor is divided into a certain number of tasks. At least the following documentation should be produced for each task:

- Detailed task design documentation
- Description of every module in the task
- Source listing for every module in the task
- Test requirements for each task
- Test documentation for each task

2.4.3 System/Subsystem Integration Documents

According to the steps required during the structured decomposition of a system, the system/subsystem integration documentation should be split into at least three levels:

- The system integration level
- The subsystem integration level
- The module integration level where individual modules are integrated into software subsystems.

For each level, at least the following documents are required:

- A test specification document
- A test data base document
- A test procedure document
- A test report document

All findings (modifications) elaborated during integration tests or flight trials must be documented in the appropriate design documents in a fashion to display the original design and the final one. Special attention shall be given to the contents of databases or constants used in the course of computation. Theoretical derivation and reasons for later adjustments must be incorporated. Additionally, all necessary release documents are to be considered part of the integration documentation

2.4.4 Standards Documents

The standards which are applicable, for instance, MIL-STDs, ARINC, etc., have to be specified for each project. Additionally all relevant company level, customer level, etc., regulations have to be documented and included in the project's governing documentation.

2.4.5 Quality Assurance (QA) Documents

Independently, but parallel to the system development, quality assurance documents must be produced which are applicable within a project.

2.4.6 Other Project Documents

Any additional documentation used by any project team members during a project should be structured for ease of access and made available in order to provide all relevant information to all project development team members.

2.5 General Project Support Issues and Measures

In addition to the above, there are certain general project support issues and measures which are important for all guidance and control system development projects. These issues and measures are discussed below.

2.5.1 Organizational Issues

The following particular individual organizational groups which should be involved in software development projects can be identified:

- Development group(s)
- Change and configuration control group(s)
- QA group(s)
- Test group(s)

From an organizational point of view, the organizational relationships which should exist between these groups and the project management are as follows:

- The project management should control the development groups and the change and configuration control group(s).
- The QA and test group should operate independently of the project management.

2.5.2 Project Planning Issues and Measures

Project planning should be a common effort of

- The development and testing groups
- The QA group

The development and testing groups should be responsible for project strategies, and the QA group should be responsible for project tactics. This means that the development and testing groups should decide on:

- The specific methods to be used to support given activities.
- The deliveries resulting from the given activities.
- Planning.

Here, planning is taken to include:

- Development of a complete project activity network
- Development of the list of deliverables
- Development of detailed descriptions of all methods to be used on the project
- Estimation of the time required for completion of all project activities

- Estimation of the manpower requirements for each activity
- Network planning (critical path planning and scheduling)
- Allocation of other required resources.

The QA group should be responsible for tactics; that is, for:

- Packaging activity templates.
- Packaging deliverable templates.
- Collecting sample deliverables.
- Developing prototype project models.

2.5.3 Project Review Issues

Here, one has to distinguish between:

- Formal reviews (with or without customers)
- Internal project reviews

Formal reviews should take place at the end of each major design step in order to verify the results of the design step both technically and formally, and they should be supported by the tools in use by the project to the maximum extent possible.

Authorized official approval signatories (or their deputies) participating in formal reviews should include:

- Project leader
- Customer
- Responsible project members
- Specialists of technical departments who are not members of the project team
- Management of supporting technical departments
- Quality assurance personnel

All formal reviews should be organized by the project leader, and he should distribute the following documents before the review:

- Invitation, agenda, checklist
- Documents to be verified in the review
- Verified referenced documents to be used in the review

The review should verify:

- Consistency of documents being reviewed with the basic requirements documents.
- Internal consistency of documents being reviewed.
- Technical correctness and completeness of the documents being reviewed.

- Formal correctness and completeness of the documents being reviewed.
- Consistency of the documents being reviewed with other validated documents of the same level.
- Compliance with safety critical requirements.

At the end of the review, a checklist is used to insure that all important points have been considered in the review and to record whether or not the results given are acceptable. The list is signed by all of the authorized official approval signatories as formal acceptance of the review.

Internal reviews (e.g., walkthroughs) are held at the discretion of the project leader. Participants are project members, quality assurance personnel, and additional invited specialists. The results of each internal review should be summarized by detailed minutes of the review. These minutes of informal reviews should be available for use in all subsequent formal reviews.

2.5.4 Change and Configuration Control Issues and Measures

The creation and use of a special change and configuration control group which is provided with a computer aided toolset is recommended. All change requests are to be submitted to this group. The potential for external consequences of a change request for a particular hardware or software component is exhibited by this group by producing a cross reference listing for each component directly affected by the change request.

Based on the cross reference listings, the consequences of any change are evaluated and, together with the responsible staff members, a decision is made to either accept or reject the change request.

Within the configuration control of a structured project, the following configuration control documents are necessary:

- Software configuration documentation, which includes all relevant software documents.
- Hardware configuration documentation similar to the software documentation described above but related to the hardware configuration.
- Hardware/software compatibility documentation which describes the compatibility requirements between the hardware and the software configurations.

The configuration control within a project should be supported by the following measures:

- Version control of each document, stepwise clearance of each document as a function of status of the document (i.e., under preparation, internal clearance, external clearance), and access control for authorized people to change documents.
- A reference chapter in each document listing and pointing to the master documentation on the next hierarchical level. These references should be verified in all formal reviews and walkthroughs.
- A machine processable collection of all these references to serve as a basic directory for documentation control. The project member responsible for this directory should also be responsible for insuring that all necessary changes in documents are properly carried out.

All mentioned measures should be controlled as far as possible by supporting management tools.

2.5.5 Quality Assurance and Measurement Issues /5/,/6/

The application of methods alone is not sufficient. How they are used has to be controlled and how efficient they are should be monitored. Therefore, a QA team should be established which collects and evaluates all information and data generated by the project team which is relevant to the overall performance of the system in its intended application, along with the performance of the project team itself.

For example, during the requirements phase, the number of elementary functions, interfaces, and data

elements is computed and their complexity is evaluated. This allows very good control of the success of subsequent applications of the S.A. Method, for example.

In a similar manner, application of the S.D. Method during the design phase can and should be controlled. In this case, for example, the modules, intermodular connections, pathological connections, and control tokens are counted and their complexity is evaluated.

Furthermore, the implementation process should be similarly controlled. A simple, but very effective, instrument to detect divergence of results from requirements during the software implementation phase is the automatic recording of the compilation/assembly rate. A steep descent of this rate signals nearness of the successful completion of the implementation. Such process implementation control measures directly support and facilitate

- Minimization of project cost
- Achieving the project milestones on time
- Maximization of project member performance
- Product quality.

All deviations in performance, cost, and milestone requirements uncovered during implementation process control activities should be reported to the project manager.

2.5.6 Certification Requirements Issues

(See Section 3.5.2.)

REFERENCES

- /1/ Structured Analysis and System Specification, N.Y., Yourdon Press, 1978, de Marco
- /2/ Structured Design Fundamentals of a Discipline of Computer Program System Design, 2nd Ed., Englewood Cliffs, N.J., Prentice-Hall, 1979, Yourdon-Constantine
- /3/ A Practical Guide to Structured Systems Design, N.Y., Yourdon Press, 1980, Page-Jones
- /4/ Reliable Software Through Composite Design, N.Y., Petrocelli/Charter, 1975, Myers
- /5/ Elements of Software Science, N.Y., Elsevier North-Holland, 1977, Halstead, M.H.
- /6/ Controlling Software Projects, N.Y., Yourdon Press, 1982, de Marco
- /7/ Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy, Department of Defense, National Technical Information Service, Springfield, VA., Stock No. AD A128981, March 1983
- /8/ JSSEE, "Plan of action and milestone for definition and preliminary design of a Joint Services Software Engineering Environment," Report DOD, 1984
- /9/ Methodman Requirements, MISA-FM 591-81
- /10/ Draft Specification of the Common APSE Interface Set (CAIS) Version 1.1, 30 September 83
- /11/ J.T. Shepherd, D.J. Martin, R.B. Smith: "Some Approaches to the Design of High Integrity Software," Report Marconi Avionics Ltd., 1984
- /12/ NATO Report, "Distributed Design Methodology," AC 243-D925, AC 243 (Panel III), D-228, 6.8.84
- /13/ W.J. Taylor: 'Ada as a design language,' Ada UK News, Vol. 4, No. 1, Jan. 83

Appendix 2.1: MODEL SYSTEM SPECIFICATION DOCUMENT

1 Introduction

- Introductory explanation of the project
- List of references
- Intention of document

2 General Requirements

2.1 Aircraft Roles / System Tasks

- Mission of the aircraft
- Task of the system

2.2 General Concepts Bases and secondary factors of the system design

2.3 System Architecture Bases and secondary factors of the system architecture

2.4 System Criticality Classification

3 System Requirements

Description of the known implicit system requirements subdivided according to subsystems:

3.1 Cockpit/Display and Control Requirements

3.2 Requirements for Navigation

3.3 Requirements for Communication

3.4 Requirements for Recording

3.5 Requirements for Weapons Aiming

3.6 Requirements for Defensive Aids

3.7 Requirements for Armanent Control

3.8 Requirements for FGCS

3.9 Requirements for general A/C Systems

4 Additional Requirements

System Requirements with regard to "-ilities", e.g.,

4.1 Testability and Maintainability

4.2 Harmonization

4.3 Environmental and Weather Conditions

4.4 Reliability and Redundancy

5 References

Appendix 2.2: MODEL SUBSYSTEM SPECIFICATION DOCUMENT

1 Introduction

- System Content
- Subsystem task
- Intention of Document

2 User Requirements

2.1 General Concepts

- General Requirements which are meaningful to the subsystem; e.g., bus system, HOL, target computer, warnings, operation, redundancy

2.2 Specific Subsystem Requirements

- In depth derivation of subsystem requirements from system requirements

2.3 Subsystem Criticality Classification

3 Design Concept

- System architecture
- Interface description
- Design concept

4 Sensors

- Description of requirements on sensors and effectors of subsystems and their functions

5 Processors

- Description of target computers of subsystem
- Requirements on target computers with respect to subsystem functions

6 Displays and Controls

- Description of the displays/controls of subsystems
- Description of dedicated or multifunction displays and controls

7 Functional Description of Subsystem

- Description of functional requirements of the subsystem
- Functional context including
 - Switch on/off procedures
 - Control/display functions (man-machine interface)

8 Subsystem Interfaces

- Detailed definition of subsystem interfaces to other subsystem

9 Subsystem Failure Modes

- Description of error types and consequences
- Subsystem redundancy

APPENDIX: Derivation of Equations and Algorithms

CHAPTER 3

CURRENT STATUS OF TOOLS, TOOL SETS, AND SUPPORT ENVIRONMENTS FOR EMBEDDED COMPUTER SYSTEMS SOFTWARE

3.1 CURRENT SOFTWARE PRACTICES

3.1.1 Introduction

The term "support software environment" refers to a collection of methods, tools, and procedures for the development, test, and life cycle support of embedded computer software applications like Flight Guidance and Control Systems (FGCS). The acquisition, management, and support of FGCS software is conducted according to internal defense policies and, thus, differs between nations. The environments are generally organized and built in such a way to support or assist compliance with these policies.

Section 3.1 of this chapter first discusses the acquisition, management, and support policies of selected nations. Section 3.2 contains a discussion of software requirements and design methods; the methods are listed in Appendices 3-1 and 3-2. Section 3.3 describes current programming languages, tools, and environments; technical details are presented in Appendices 3.3 and 3.4. Section 3.4 discusses software and system integration/testing and Section 3.5 discusses tools for the support of verification and certification. This chapter concludes with Section 3.6 on software project management.

3.1.2 Current Acquisition, Management, and Support Approaches

3.1.2.1 Federal Republic of Germany

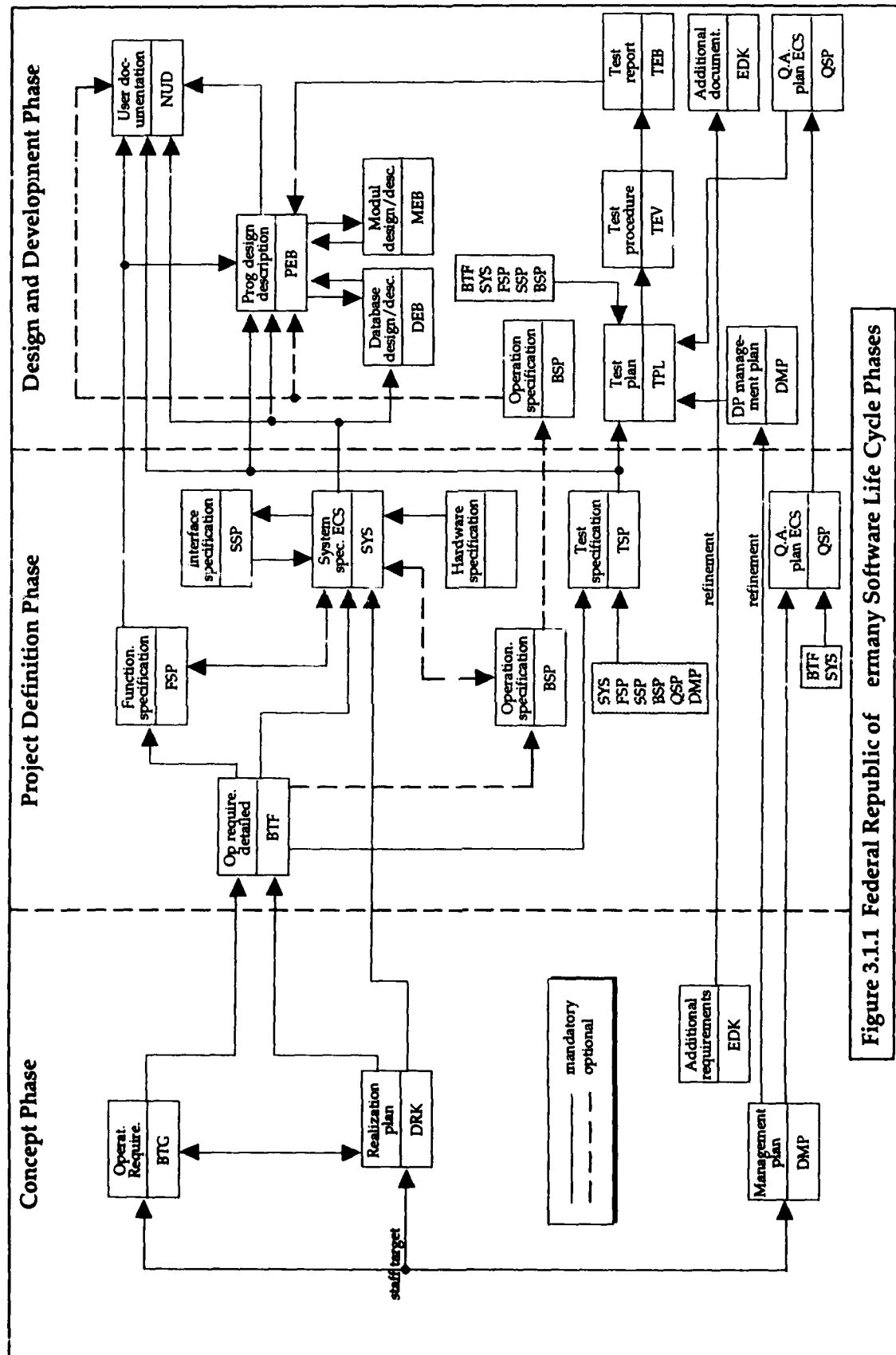
The Federal Republic of Germany has a Software Documentation Standard that applies to all phases of the Software-Life-Cycle of Embedded Computer Systems (ECS). The Standard is based on the "Ruestungsrahmenerlass" (Basic Armaments Directive) and provides a supplement for the software without establishing procedures for the organizational flow of software development. The Standard is mandatory for the documentation of ECS software and achieves the following objectives:

- Improved cooperation and communication between the user, the procurement and development agency, and the contractor.
- Improved control of the progress of the project.
- Support of configuration control.
- Support of software maintenance during the in-service phase.
- Improved effectivity and economic efficiency.

Additionally, the Standard establishes basic requirements that must be met by the documentation of the ECS software. It specifies the documents to be submitted in the individual phases (see Figure 3.1.1), their purpose, their formal structure, and a commented table of contents.

3.1.2.2 France

In July 1984, the French Ministry of Defense established a standard for the development of integrated software systems. This standard, GAMT17 "Methodologie de Developpement de Logiciels Integres," defines the



development cycle of integrated software systems to make their maintenance easier and to control their cost and development planning. Giving a general development scheme, GAMT17 specifies the contents of each phase of development by defining the different participants, the input and output documents, and the products for each phase.

In the beginning of 1983, the French Ministry of Defense specified LTR3 as the standard programming language for embedded software systems, replacing the previous standard, LTR. The Ministry has developed a LTR3 programming environment called "ENTREPRISE". The ENTREPRISE environment is portable on computers running under the UNIX * operating system and is publicly available in France. Details on both the LTR3 language and the ENTERPRISE environment are presented in the Appendices of this chapter.

3.1.2.3 United Kingdom

The United Kingdom Ministry of Defence (MOD) has recently issued a compendium of MOD policy on most aspects of procurement and management of computer based military systems. This document entitled "The IECCA Guide to the Management of Software-Based Systems" is issued under the auspices of the Inter-Establishment Committee on Computer Applications by the Royal Signals & Radar Establishment at Malvern.

It incorporates the substance of earlier policy statements contained in DUS POL PE Statements to Industry dated 11 April 1980 and 31 July 1984, and includes a definitive list of relevant defence standards that form an implicit part of MOD's policy. Plans are in hand to make the Guide into a defence standard in its own right.

3.1.2.4 United States of America

The United States of America Department of Defense (DoD) established a uniform policy for the acquisition, management, and support of mission critical computer systems on 4 June 1985. The new policy is stated in DOD-STD-2167 DEFENSE SYSTEM SOFTWARE DEVELOPMENT and related documents.

DOD-STD-2167 defines the life cycle for defense systems as consisting of four major phases, Concept Exploration, Demonstration and Validation, Full Scale Development, and Production and Deployment. The software development cycle occurs within the system life cycle and consists of six phases as shown in Figure 3.1.2. Whenever computer software is developed, the corresponding activities, reviews, products, baselines, and developmental configurations are applicable. The software development activities can occur sequentially, can overlap in time, or can proceed concurrently.

DOD-STD-2167A is an acquisition policy, specifying the contractually deliverable products for defense systems. The technical methods used to create the products are intentionally not specified; instead, they are allowed to be selected to best suit the particular system.

3.2 SOFTWARE REQUIREMENTS AND DESIGN METHODS

3.2.1 Introduction

Software requirements definition and software-design are sub-activities of system design. They begin after a system architecture has been defined and after the subsystem requirements have been identified. The term "requirement" refers to the clear and precise statement of a need, in function or performance, that must be provided. It is a term that reflects the desire to clearly state the problem that is to be solved, a desire to precisely state what the system is required to do. The term "design" refers to the process of defining the structure, components, and interfaces of a system to satisfy the specified requirements. The term "method" refers to a set of principles and guidelines for carrying out an engineering process. The term "methodology," which is frequently misused as a synonym for "method," refers to the science (or study) of method and is not used in this section.

* - UNIX is a registered trademark of Bell Laboratories, Incorporated.

When computers were first used as integral components of systems there was considerable success; however, these early applications were all small in scale. When large scale applications were attempted, many problems resulted. To a certain extent, these computer and software problems were part of the general problem of how to build large complex systems at affordable costs. At a deeper level, however, the process of specifying and designing software lacked basic engineering principles and methods; the process, therefore, broke down when large scale systems were attempted.

For example, the software requirements were stated exclusively in a natural language, e.g., American English. The software design was a totally manual process with little attention to design validation, risk analysis, and later, product verification. Early experience with large scale systems showed that many software errors were not discovered until after system deployment. These errors included both software discrepancies (bugs) and improper functions. Further, it was discovered that most of the errors could be traced to the requirements specification and the design phases. It was not only difficult to correct these errors but also expensive; the later in the life-cycle the errors were found, the more difficult and expensive they were to correct.

The causes of the errors were identified as poorly stated or ambiguous requirements and the lack of a formal way to specify software requirements. The difficulty of stating the requirements for the user's interaction with the computer system was particularly important. The design problems exhibited poor program structure, little consideration of interfaces, and poor integration approaches. Testing and other methods of product verification were absent from some of the life-cycle phases. Most important, it was not recognized that large scale systems must continually evolve and, therefore, it must be easy and practical to change their software.

The magnitude of these software problems stimulated the growth of technology for both software requirements methods and software design methods. This led to investigations of formal requirements languages with the goal of achieving a complete, consistent, and unambiguous specification of software requirements. In parallel, there have been attempts to develop effective software design methods; i.e., systematic approaches to creating software designs. Technology has, in both cases, produced some results. However, despite this progress the results currently available are incomplete and nonconclusive. There are some partial results in the form of methods and tools that can be applied, but, more importantly, there are also many new rapidly emerging technology developments.

3.2.2 Software Requirements Process Description

Chapter 2 described a Phase Model for the activities involved in FGCS System and Software Development. According to this model, the software requirements phase begins after the Subsystem Requirements/Design Phase. In the subsystem specifications, both the hardware and software are identified as configuration items (HCI's and SCI's). The software requirements process is an activity that specifies all of the requirements of every SCI in detail. The result of this process is a software requirements specification.

There are two categories of software requirements; namely, functional and nonfunctional (also called constraints). A functional requirement refers to the specific purpose or characteristic action of a software component in its environment. For a software component embedded in a weapon system, the environment includes the target computer hardware, other software components, other subsystems, and the external real-world environment. A constraint, or nonfunctional requirement, imposes conditions on the implementation of the software component. Typical constraints are performance, reliability, fault detection, interfaces, safety, security, and ease of change.

Development of software functional requirements involves modeling all the SCI's, including the interaction with their environment. First, the software requirements are stated at a high level. Then, using a particular software requirements method, the requirements are specified in stages of greater detail until a satisfactory level of completeness is reached. None of the methods available today comprehensively cover development of software functional and nonfunctional requirements; rather, they usually cover functional requirements and only some of the nonfunctional items.

Certain steps are particularly important during the process of specifying FGCS software requirements. Every function that an SCI provides must be specified in detail, including the input, output, and processing requirements. The timing, precision, and accuracy of all data must be specified. Critical equations and algorithms for logical and mathematical computations are specified. All requirements for storage size, execution speed, and input/output bandwidth are also specified. Further, the interface requirements

between configuration items must be analyzed and specified. This includes interfaces between SCI's and between a HCI and SCI.

Additional nonfunctional requirements like traceability to the system specification, quality, reliability, interoperability, and validation also should be included in the software requirements specification.

Throughout this process it is important that the requirements remain independent from the design. The emphasis is on what information is input, what processing is required, and what information is output.

3.2.3 Software Design Process Description

Chapter 2 provided an overview of the two software design phases, basic (or preliminary) design and detailed design. The software design process transforms the requirements specification into detailed design specifications that allow direct coding in a particular programming language. The design specifications state how the end-product software will be built.

Design can be viewed as a process of modularization, a process of determining the structure of the computer software. The most widely used process is top down design. Here, a high level design is performed first. Then, using a particular design method, the high level design is successively subdivided into smaller units until a level is reached at which the units can be directly, easily, and correctly programmed. The design process, regardless of the method used, must also specify the correct execution sequences of the program units. This is an especially critical aspect of all real-time software.

3.2.3.1 Design Methods

There are many software design methods in use, representing both formal and heuristic approaches. Some methods use only diagrams and are, therefore, aimed at communication and documentation of the design. Other methods use principles and guidelines that are more formal.

Design methods can be grouped into four categories: functional decomposition, data flow design, data structure design, and object oriented design.

Functional decomposition views the system as a black box that must perform a set of functions on a set of inputs to produce a set of outputs. The method begins with an attempt to identify an algorithm, transformation, or function (in the mathematical sense) that relates the outputs to inputs. If this cannot be done the functions are further divided into subfunctions that are more amenable to the identification of satisfactory computer algorithms. Thus, the original black box is decomposed into a set of smaller boxes, each with inputs and outputs, some between each other, and others being the original inputs and outputs. The decomposition is repeated until a box size is reached that can be defined as a program unit.

In data flow design the first step is to draw a data flow graph of the system, identifying inputs, outputs, and transforms. Each data flow is successively decomposed, retaining the black box concept that transforms an input data stream into an output stream. This process results in a network of programs; however, a hierarchical program structure chart can be derived by following a simple procedure.

In data structure design the first step is to draw a network diagram of the system. Next the data stream structures are defined for the inputs and outputs, resulting in a data structure hierarchy. A second corresponding hierarchy is then defined for the program structure. The resulting program units are connected in a data flow network and a scheduling procedure called program inversion is used to generate a program-execution hierarchy.

In object-oriented design the first step is to draw a data flow diagram of the system. Object-oriented decomposition extracts objects (items or components) from the data flow diagram, noting data dependencies between the objects. The actions on each object and required of it are then identified; next, the visibility of each object toward other objects is established. The interface to each object is defined and then each object is programmed. Object-oriented design works best when the programming language can embody and enforce the properties of an object; with appropriate programming conventions, Ada and LTR3 are languages that can support object-oriented design.

3.2.3.2 Other Factors in the Design Process

The design process must also consider the sequence in which the software is built. An important development has been the idea of "incremental" construction where the software is built in stages, each stage adding functional features to the previous stage.

Testing is an important consideration during the design process because testing remains the primary method for determining the error-free performance of software. The tests must be specified at the software unit level (where modules are individually tested), the software integration level (where groups of modules are integrated and tested), and the system level (where the software is integrated on the target computer). Additional information on certification and acceptance criteria for software is contained in Section 3.5 of this chapter.

3.2.4 Tables of Current Methods, Tools, and Tool Sets

Appendix 3-1 contains a list of current methods and shows the applicable life cycle coverage. The life cycle phases shown are the same as those given in Chapter 2. Additionally, coverage of project management and configuration management is shown and, if appropriate, any related methods are indicated. Appendix 3-2 gives the full name and/or a description of all the method acronyms.

The large number of methods causes confusion when selecting a method for a particular application like FGCS. Further, one method usually is not compatible with the other methods. As a consequence the different methods cannot, in general, be combined.

For FGCS, it is of primary importance to recognize that methods that embody only functional hierarchy and data flow are insufficient because FGCS are time and event dependent.

3.3 PROGRAMMING LANGUAGES, TOOLS, AND ENVIRONMENTS

3.3.1 Introduction

In early computer systems, support for the programmer was extremely limited. Instructions were loaded into the computer by hand toggling the machine code into the computer one bit at a time. The earliest aids for the programmer were translation programs called assemblers. These programs, or tools, translated symbolic statements of the instructions, addresses, and data into the internal machine code. High order languages, most notably FORTRAN, appeared in the mid 1950's. As a result, more sophisticated translation programs, called compilers, translated high order language source programs into machine code. A typical characteristic of a compiler is that one high level source statement is translated into several machine instructions. Thus, compilers were a significant tool development that decreased the labor needed to produce code. Since that time, computer engineers and programmers have desired, and tried to produce, increasingly more powerful compilers and other tools.

Today, software tools have grown significantly in complexity and capability. Most of the early tools performed a single function and were easy to describe, e.g., assemblers, compilers, flowcharters, linkers, and editors. These early, simple tools have since given way to more complex tools that have significantly increased functionality and power. A particularly important development has been the recognition that software tools are a natural and convenient way to enforce a method, whether technical or managerial. For example, the life cycle management of software implies breaking the work down into phases, as described in Chapter 2, with each phase delivering a specific set of products. In the development of a particular product, it may be required that a certain software engineering method be followed. In such cases, a software tool can be constructed to insure that the specific engineering method to be used is correctly followed to produce a particular life cycle product. Similarly, software tools can be constructed to enforce configuration management and other project management functions. Software tools can be categorized as:

- Stand-alone Tools

- Tool Sets
- Software Engineering Environments (SEEs)

A stand-alone tool is a single program that assists the user in developing a specific life-cycle product. The tool typically enforces or guides the user in following a particular method, e.g., a testing method. The tool is characterized by a single user interface, command processor, data base manager, and reporter.

A tool set is a collection of tools provided to a project team to support the development and maintenance of software. Tools are usually available in the toolset for several different life cycle phases, providing varying degrees of assistance to the users. Tool sets are usually not integrated; that is, they usually have different command languages and the output from one tool usually does not conveniently form the input to a second tool.

A SEE is an integrated set of methods, tools, and procedures for software development and maintenance. A SEE normally addresses more than one phase of the life cycle, although limited forms such as programming support environments also exist. A SEE is characterized by a common user interface, a common command language, and a common database management system for all tools. Also, a SEE usually provides high level commands that invoke long sequences of tool and data base interactions.

3.3.2 Generic Tool Descriptions

There are a small number of generic tools that are used in almost every FGCS application. This section provides general descriptions of these tools.

Editor

An editor is a tool for text manipulation. In early computer days, source program text was entered by paper tape or punched card. Today, editors are sophisticated interactive screen/window-management tools. Modern editors are used not only for the creation or modification of source text but also for the viewing or modification of files produced by other tools.

The primary use of an editor is the creation or modification of source program text. The product of the editor is a file, containing the source program statements. Since these program files will always have to be compiled it was recognized that there is an advantage to having a language specific editor, a tool that has some of the specific language requirements built in. These editors simplify the entering of program text and sometimes perform online error checking.

In its most elementary form, a language specific editor may have special options to assist the formatting of the source text. An example for FORTRAN would be automatically starting a line in column 7 whenever the first character was alphabetic, thus preventing text from being placed in the field reserved for line numbers.

Another form of a language specific editor is the "syntax directed editor", which is tightly coupled to the programming language. Most modern languages require opening and closing statements for structured programming constructs. A syntax directed editor can provide templates for these constructs. For example: the template

```

if <condition>
  then
    ....;
    ....;
    ....;
  else
    ....;
    ....;
endif;

```

could be rapidly placed on the screen after typing "if". Additionally, the syntax directed editor can check the structure of the source text for compliance with the rules of the language. Thus, the efficiency of the edit

compilation process is improved because many programming errors are eliminated before compilation.

Compiler

A compiler is a program that translates a high order language source program into its relocatable code equivalent. The term "host" refers to the computer that translates the source program into the compiled code and the term "target" refers to the computer that will execute the compiled code. The term "cross-compiler" refers to the case where the target computer is different from the host computer. In FGCS and avionics systems applications, a source program is cross-compiled on a host computer (generally a commercial machine) for execution on a militarized target computer that is embedded in the system.

Compilers are usually multiple pass programs that may process the source program or some intermediate form several times before completing the compilation process. The output of the earlier stages is referred to as intermediate code. In some host computer systems the intermediate code is used by other tools.

Compilers for real-time applications, particularly FGCS, must produce code to fit in limited storage space. Also, the execution speed on the target computer must be such that all of the required functions can be computed in the assigned time.

Assembler

An assembler is a program that translates an assembly language source program into relocatable code; there is usually a one-to-one correspondence between an assembly language source statement and a machine instruction. Assemblers allow the programmer to use relative addressing and then specify a starting location rather than having to specify each address in absolute terms. Most assemblers also allow the use of labels and other defined values and locations.

Assembly language has been used frequently in FGCS and avionics systems applications because the visibility it provides at the target machine execution level allows the programmer to optimize storage space and execution time. However, assembly language programs are difficult to test and expensive to maintain. Today, the use of assembly languages is generally restricted to routines with exceptionally high performance requirements, to special I/O routines, and to hardware diagnostic software.

Linker

Source program modules, whether in assembly or a high order language, are usually translated separately. Once translated, the modules must be linked together before execution. A linker is a program that creates a load module from one or more independently translated modules by resolving the cross-references among the modules.

Relocating Loader

Relocatable code contains relative addresses of machine instructions and data; this defers the assignment of absolute addresses until the program is ready for execution and allows the flexibility of placing the program in any contiguous block of storage. The linker creates a load module that leaves all addresses in relative form, although it has the cross-references between modules resolved. The relocating loader is a program that executes on the host computer and translates the relative addresses into the absolute addresses; its output is an execution module. A bootstrap loader executes on the target computer and copies the execution module into its storage.

Run-Time Executive

The Run-Time Executive (RTE) resides on the target machine and provides a variety of services for application programs. Typical RTE functions for a complex language such as Ada are dynamic storage management, exception processing, input and output, and task scheduling. Since this RTE is used for all application programs, it should be small and fast to minimize the overhead. The RTE is usually modularized according to the particular services it provides and automatically configured when the execution module is created. Thus, if an application program does not need a particular service then that module is automatically omitted from the RTE.

Simulator/Emulator

When producing code for a target computer that is different from the host, one has to consider the problem of how to test the code. Testing on the target computer is usually difficult because it may be still under development, it may be being integrated with other embedded subsystems, or the number of target machines may be insufficient to support all the programmers. Also, most embedded target computers have poor tools to support testing.

One solution to this problem is to build a software simulator or emulator of the target computer that executes on the host computer. A software emulator accepts the same data, executes the same instructions, and achieves the same results as the target machine. A simulator imitates selected features of the target computer but is not required to achieve identical results. The best tool is a target computer emulator that can operate in either batch or interactive mode. Although the execution speed of an emulator may be significantly slower than the target computer, it has many advantages. First, it can be time shared and used by everyone on the host computer. Second, since the emulator is on the host computer it is easy to generate test data, load the module and test data into the emulator, and monitor the test while in progress. Third, long tests can be run in batch mode during off-peak hours.

In-Circuit Emulation

An additional technique useful in debugging microprocessor hardware and software is called In-Circuit Emulation (ICE). ICE requires a workstation, control device, monitor and debug software, and a physical interface that allows a test rig to be connected to the workstation. The test rig allows the microprocessor CPU chips, memory chips, etc. to be directly inserted in the circuits. The software is copied from the host computer by the workstation and passed to the microprocessor. The workstation monitors the test and allows operation of the system in single step, burst, or continuous mode, or until a specific predefined occurrence. Some systems allow the recording of data bus operations. This feature allows the test engineer to capture the previous N events before a failure or predefined occurrence.

Symbolic Debugger

A symbolic debugger allows a programmer to test a module by controlling the program execution on a target computer emulator or the target computer itself. With the symbolic debugger, the programmer can address the variables using their source program symbols or names. The facilities generally provided include: stop execution at selected locations, single step in increments of source statements, watch the value of specified variables, trace execution, examine the contents of variables, evaluate expressions, display the current sequence of routine calls, display the source corresponding to any part of the program, execute debug command procedures at breakpoints, and call procedures that are program parts.

Pretty Printer

A pretty printer is a program that automatically applies standard rules for formatting program source code. It will accept an input file and format the text to match a style guide. As an example, a pretty printer for a block-structured language will produce a listing where the indentation level of each block shows its nesting level. A pretty printer helps the programmer read and comprehend the program. For example, after extensive program modifications, it helps eliminate confusion about the program structure and nesting levels.

Host to Target Exporter

If the target machine is different from the host machine, it is necessary to have a tool to transmit the execution module from the host to the target. Standard communications software and hardware may be used but these are rarely available for embedded machines.

It is desirable to have a flexible, high bandwidth communications link between the host and target. If the link has a "pass-through" capability, then an interactive user of the host computer can run tests on the embedded computer from the same terminal. High bandwidth is desirable because there is a lot of data to be exchanged between the host and target; for example, diagnostic software is typically sent to the target to test the target hardware, and test data and test results are also exchanged.

3.3.3 Current Languages, Tools, and Environments

Flight guidance and control systems are now programmed in a variety of languages. In general, each member country has selected one or more languages as the standard for this application area. The languages in current use are CMS-2 and JOVIAL in the U.S., CORAL66 in the United Kingdom, LTR3 in France, and PEARL in the Federal Republic of Germany. For each language there exists either a tool set or an environment, or both. The environments currently in use for FGCS do not cover the entire life cycle but some do address more than one phase. Appendix 3-3 contains a brief description of each language in current use and the associated tool set or environment, as appropriate.

3.3.4 ADA

Ada was developed by the US DoD as a high order language for programming large scale and real time systems. By policy, it is now the preferred language of the U.S. DoD.

Ada was the result of a collective effort that began as the DoD common high order language program in 1974. The language requirements were defined in final form in 1978 in the Steelman specification. The language definition was developed by CII Honeywell Bull and later Alslys, and by Honeywell Systems and Research Center, under contract to the DoD. The Ada Language is formally defined in ANSI/MIL-STD-1815A-1983, Ada Programming Language, 22 Jan 1983. Ada has been approved by the American National Standards Institute (ANSI) and it is in the standardization process of the International Standards Organization (ISO).

NATO will implement the Ada language in all military systems, practically exclusively, beginning in January 1986. To aid in the implementation of the Ada language as the standard NATO programming language, an Ada Support and Control Capability organization will be established in Brussels, Belgium.

3.3.4.1 ADA Certification

A major goal of DoD is to achieve a high degree of portability of Ada programs to many computers and operating systems. To achieve this goal the DoD maintains control over the language definition and also has established a certification process for compilers. To use the trademark Ada, a compiler must pass a set of Ada Compiler Validation Capability (ACVC) tests and receive a certificate from the Ada Joint Program Office. Also, in order to continue to use the trademark the compilers must revalidate every year using the latest version of the ACVC tests.

3.3.4.2 ADA Run-Time Environment

Early experience with current Ada compilers has been encouraging regarding the portability of Ada programs; however, there are some potential problems with the Ada run-time environment. The term run-time environment refers to those services required during program execution, e.g. program faults, I/O, external device interrupts. The Ada run-time environment is not formally defined like the language. Therefore, different implementations of the run-time software, particularly on widely different computer architectures, can produce different results. There is currently a DoD working group addressing this issue.

3.3.4.3 Ada Programming Support Environment (APSE)

Attempts to define a support system, or programming support environment, for Ada did not begin until 1978. The level of effort was considerably less than that of the Ada language. The STONEMAN document (produced in 1980) identified the general requirements for an Ada Programming Support Environment (APSE). This document recommended a layered architecture for the environment but was weak in the area of life cycle support. The recommended layers were:

- **Level 0:** Hardware and host computer software as appropriate.
- **Level 1:** Kernel Ada Support Environment (KAPSE) that presents a machine-independent portability interface and provides data base, communication, and run-time support functions to

enable the execution of an Ada program.

- **Level 2:** Minimal Ada Programming Support Environment (MAPSE) that 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.
- **LEVEL 3:** APSEs that are constructed by extensions of the MAPSE to provide fuller support of particular applications or methods.

The most significant layer in this approach is the KAPSE. The assumption is that this layer can be standardized across many different computers and operating systems, allowing portability of all tools in the higher layers. Unfortunately, the KAPSE layer was not formally defined before the DoD began development of two Ada compilers and MAPSEs; not surprisingly, the resulting Ada tools are not portable between the two DoD environments.

3.3.4.4 Common APSE Interface Set (CAIS)

In 1982, the DoD formed a KAPSE Interface Team (KIT) with a corresponding industry and academia team (the KITIA) to address the problem and produce a standard definition of the KAPSE. The teams, to date, have produced the Common APSE Interface Set (CAIS) Version 1, which was issued in 1985 as Proposed MIL-STD-CAIS. The KIT/KITIA are currently working on CAIS Version 2, which addresses the topics deferred in Version 1. Currently, none of the validated Ada compilers or environments use CAIS although several prototype efforts are in progress.

3.3.4.5 Portable Common Tool Environment (PCTE)

The ESPRIT program, sponsored by the Commission of the European Communities, supports several Ada activities. One activity has developed a set of functional specifications to form a basis for a Portable Common Tool Environment (PCTE). PCTE is not identical to CAIS Version 1 but was developed to meet comparable requirements. PCTE has already addressed some of the topics (distributed processing, window management) that will be addressed by CAIS Version 2. Several prototype implementations are currently in progress in Europe.

3.3.4.6 ADA Summary

Currently, Ada is not being used for FGCS software; however, it should be clear that the language and its support environment will achieve wide usage in the future. ADA is now emerging as a language of choice in each of these nations and it is expected to become a future standard. Appendix 3-4 contains a brief description of the Ada Language, information on all the validated compilers, and some details on the state of the associated support environments.

3.4 SOFTWARE AND SYSTEM INTEGRATION/TESTING

3.4.1 Introduction

After the software is coded and module tests are completed, the program enters the integration/testing phase. The testing phase is an important element in the validation process and must be designed to demonstrate that the software, combined with the hardware, operates with the goal of detecting errors. The degree of testing will heavily depend on the performance, complexity, and criticality of the functions being performed by the system. After the integration of modules and subsequent integrity testing, the software is ready for qualification testing. Table 3.4-1 summarizes the information required, tasks to be performed, and expected outputs.

In critical systems, such as flight control, testing must encompass the interactions with the dynamics of the vehicle, consideration of the operational environment, and the effects of disturbances on the system. The behavior of the system in terms of flying qualities and pilot control functions must also be included.

INPUT	TASKS	OUTPUT
Program module source and object listings	Run diagnostic tests on computer and peripheral equipment	Operational software object code
Detailed specifications of each module	Validate deliverable support software	Software Specifications
Checkout procedures for each module	Integrate modules into the operational code	Detailed Design Spec
Integration test plan	Check out total operation Analyze test results Initiate modifications to program modules and recheck results Document test results Update, maintain documentation Begin program to train customer and plan for post-delivery support Conduct the critical design review of the software	System Documentation Test reports

Table 3.4.1
SOFTWARE INTEGRATION AND TESTING

3.4.2 Test Stages

There are five major stages of testing during the software integration and test phase of a system development. They include:

- Execution on the host computer.
- Execution on an emulator of the target computer.
- Execution on the target computer.
- System simulation testing.
- Flight test.

The five stages represent an idealized process, they are not necessarily employed on every system development. However, the last three stages are always necessary for flight control system development.

A variety of tools are used to aid in the software verification and system validation. A list of these tools is shown in Table 3.4.2. The static tools are useful during the development phase of a program, especially at the design, specification, and coding levels. The static tools are grouped into those that examine a specific property compared to those that are generic or more extensive. These tools have been constructed in specific languages limiting their general usage; however, with a high order language like Ada, the tools should be more portable. A dynamic tool aids in the testing of the software during real-time execution and can greatly systematize the testing procedures. Instrumentation and devices to monitor and execute tests are usually customized by the flight control analyst to check the particular program. Again, present tools are language specific, which limits general applicability. Before these tools are applied, they must be validated and proven.

3.4.2.1 Execution on Host Computer

The execution of code on the host computer in a static environment provides the opportunity to examine the software in an analytical sense. Current methods and techniques include control flow, data flow, structural performance, and failure effects. Control flow enables expected functions and allows examination of module invocations and boolean manipulations. Data flow methods allow examination of the propagation, consistency, and range of variables through the system. The structure of the code can be tested to determine that the proper interface and coupling between modules is observed. Embedded test software is included to simulate the operation of the software to evaluate the I/O compatibility and functionality of the system. Failure effects can also be introduced to test error detection and recovery mechanisms and search for out of bounds conditions.

3.4.2.2 Execution on an Emulator

When the target machine is not available, the target computer can be emulated on the host computer to provide a capability for examining the interactions of the hardware and software. Such an emulator can be an effective tool in uncovering errors. This environment can provide interactive and multiuser capability to develop and test software for the target. A test generator can provide the stimuli to conduct module and end-to-end testing of the software. By interfacing the host with the emulator, aircraft, sensor, interface, and actuator effects can be investigated in a closed loop simulation. Both static and dynamic performance data can be gathered and a mechanism for the analysis and evaluation of faults and their effects on the system can be provided. By using appropriate time-scale factors, real-time operation can be emulated. This environment can also be used to validate modifications that are made to the system. One limitation with this method is the difficulty of simulating the redundant execution environment that exists for a flight critical program so that certain portions of code cannot be tested. Also, the interfaces between hardware and software cannot be thoroughly checked out.

3.4.2.3 Execution on Target Computer

During this phase of the effort, the software is incrementally loaded into the target machine to verify operation. Individual modules are tested and then linked to provide an end-to-end testing. The focus of this testing is system performance and computer/software compatibility. The environment will usually consist of the target machines, the host (properly interfaced to the targets), and loader and diagnostic hardware/software. Only a few selected components are normally available at this phase of effort (e.g., control panels, controller) so that simulation of the real execution environment is used to a large extent. The host contains the models of the aircraft, sensors, actuators, atmosphere, disturbances, special interfaces and noise generators. Testing to be performed consists of open-loop type system checks and closed loop simulation. The testing addresses such areas as power up/down sequences, operating system, interrupt recognition and servicing, control structure, timing, and I/O signal processing and conversion. Proper recording capability must also be provided for capturing streams of data for analysis. The test environment must also allow the examination of internal states (e.g., reads/writes into selected addresses and registers), stopping of the program, stepping of the program, tracing of program execution paths, and tagging of data and it must allow inclusion of embedded test software.

In a flight control system, each software module including filters, schedules, limiters, mode logic, and fault/reconfiguration logic are first individually verified for correct operation. End-to-end tests are then performed to verify logicals, timing, static and dynamic behavior (transient and open loop frequency response) over a full range of inputs, synchronization of redundant processors, inter-processor communications, and mode

control logic. Closed loop tests are performed to verify stability, maneuver and step response verification, response to environmental disturbances, and the effects of failures on error detection and fault tolerance features.

At the completion of this phase, there should be a high level of confidence that the software is operating as specified.

Specific Static Tools	General Static Tools	Dynamic Tools
Circular reference checker	Accuracy analyzer	Simulations
Code comparator	Assembly code verifier	Test bed (iron bird)
Consistency checker	Assertion checker	Monte Carlo
Cross-reference checker	Documentation and construction systems	Test data generator
Data base analyzer	Formal languages with syntax analyzers	Test driver
Interface checker		
Program flow analyzer	Requirements Specifications Program design	Test execution monitor
Set/use checker	Program code	
Standards checker	Sneak Path Analyser Symbolic evaluator	Test record generator Timing analyzer
Units consistency checker	Theorem prover	
Unreachable code detector	Verification condition generator	
	Failure Mode Effects Analyzer	

Table 3.4.2
VERIFICATION AND VALIDATION TOOLS

3.4.2.4 System Integration

The full range of operational environmental conditions are crucial to thorough validation. Incomplete and/or invalid test results can result from failure to include all the actions or effects present in the external or user environment. In this phase of the project, prototype computers are brought into a hot bench, iron-bird, or test rig. The environment is made to represent the operational environment (real world) as closely as is practical. The environment includes a real-time high fidelity model of the aircraft, real or simulated sensor models, models of the operational environment (winds, turbulence, etc.), actuators, cockpits, test instrumentation and as much of other real hardware as possible to provide a high level of realism. A complete simulator capability

would also include a motion base for the cockpit as well as out of the window capability to simulate external visual effects. Use of such simulators during the System Integration Phase focuses more on overall functional operation and interface compatibility.

Tests encompass system performance over a range of flight and mission conditions (including marginal or worst cases). The tests are designed to confirm functional performance and robustness, to evaluate the effects of failures, to confirm fault survivability, to evaluate pilot-in-the-loop operation, to evaluate crew confidence, to confirm proper sequencing of modes, and to verify proper functional dynamics.

Since high fidelity simulation represents a complex and costly environment, it should be possible to reconfigure the environment for the conduct of increasingly realistic specific tests. For example, initial testing of the flight control system would use the flight control computers with simulated sensor inputs to accomplish end-to-end tests. The aircraft simulation with simulated actuators would be brought in for closed-loop testing, with real actuators brought in to study the effects of their dynamics on the operation. After this stage, the cockpit would be integrated to assess human factor issues.

The environments discussed in previous sections are then used to correct software design errors or deficiencies resulting from the simulation testing.

As a last step, the simulation environment is used to certify the system before commitment to flight test.

3.4.2.5 Flight Test

The flight test program is directed toward confirmation of specified performance levels as well as demonstration of safety. It places the vehicle in an environment where the coupling effects of the vehicle dynamics, aerodynamic environment, and flight guidance and control system can be examined. Considerable instrumentation is required to collect data that is correlated with analytical and laboratory prediction. Data can be stored on-board or telemetered to ground stations to ascertain current performance and aircraft status. Carefully controlled faults can be inserted to test the effects of fault tolerance techniques on the performance of the aircraft.

Any software errors or discrepancies discovered in flight are investigated in the ground based facilities discussed in the previous section. The information collected can also provide the opportunity to correct simulation models.

3.5 TOOLS TO SUPPORT CERTIFICATION, VALIDATION, AND VERIFICATION

3.5.1 Introduction

Figure 2.2 of Chapter 2 shows the certification, validation, and verification levels that are applicable to FGCS. A simple definition of system certification is that it is the process of establishing the fact that the FGCS is safe for flight. Validation refers to the process of evaluating the system, at the conclusion of each subsystem phase and at the conclusion of the system integration phase, to assure that it complies with the system requirements. Verification refers to the process of determining whether the products of a given phase fulfill the requirements established during the previous phase. A pertinent and strongly related issue is software quality assurance (SQA), which refers to the planned and systematic actions necessary to provide confidence that the software conforms to established technical requirements.

3.5.2 Certification

There are no separate certification criteria for FGCS software. A certification is given by a certifying authority for the operational system, which contains both the relevant hardware and the associated flight resident software.

Certification requires demonstration with documented evidence that operation of the FGCS with the aircraft

complies with the functional and safety requirements and that no unacceptable risk for the aircraft, its crew, and/or a third party exists if a system malfunction (software error or hardware failure) occurs.

Evidence that is used in the certification process to assure proper functional and safety characteristics of the FGCS and its associated system software consists of:

- Applicable documents and regulations
- Software development method
- Verification and validation procedures
- Quality assurance and configuration control
- Documentation

These must be identified in a certification plan and depend on the criticality of the system. Therefore, the certifying authority is involved in the development, test, and validation of the system and software according to the certification plan. This involvement requires the approval of specifications, test plans and procedures; participation in audits, design reviews, tests; and the submission of reports.

For FGCS the software must be included in the system safety analysis. This analysis has to provide measures for a Failure Mode and Effect Analysis (FMEA) to identify faults or deficiencies relating to safety critical functions.

A software release document describes the configuration of the software and the appropriate hardware, together with any known remaining discrepancies and deficiencies. The software release documentation is delivered to the certification authority.

3.5.3 Validation

Validation refers to a demonstration that a system correctly does its intended function in its entirety. For embedded software this requires demonstration of correct combined hardware and software performance. This can be done at various levels in the simulated environment and uses many of the testing tools discussed in the previous section. Validation can only be accomplished to the degree of completeness and accuracy of the input requirements. A complete validation, therefore, includes requirements validation.

Appropriate validation tests are highly related to the designers' goals. For FGCS, such tests are usually conducted in the available environment that most closely approximates real flight. For program validation, the minimum cost environment required would consist of hardware-in-the-loop simulation. In addition to generation of response time tests implicit in FGCS hardware-in-the-loop simulation, frequency response tests that relate directly to good control law design practice are also used. In addition, tools for describing function analysis also provides high coverage when highly nonlinear functions or flight conditions are involved (bang-bang controls for example). Finally, redundancy management must be thoroughly examined under all flight extremes.

To be complete, validation must be carried forth to iron bird simulation and flight testing. For redundancy management software/hardware, flight validation may well interact with functional design; i.e., comparison monitoring thresholds may be determined during this phase.

3.5.4 Verification

At various points in the design process, a given software development module (or integrated collection of modules) must be verified against the module design specification. The designer has numerous tools at his disposal to accomplish this function. The tools have evolved from a set of "hand" operations into a substantial set of verification computer programs available today. A partial list of the available tools that can be used to assist in the verification process is given in Table 3.5.1.

Functions	Design	Coding	Integration
General Description	Program Design Language and HIPO	High-Order Language	High-Order Language
System Consistency	Data Flow Analysis		
Static Analysis	Syntax Checks Input/ Output Verif	Flow Chart Gener Standards Check Units Consistency	Set/Use Checks Unreachable Code Interface Checks Circular References
Tests	Decision Tables	Instrumented Tests Test Data Set Record Generator	Instrumented Tests Test Data set Record Generator
Control Functions Executive Synchronization Logical Control Laws Self-Tests	Symbolic Evaluation Table of Events Symbolic Evaluation Review Symbolic Evaluation	Exhaustive Tests Exhaustive Tests Symbolic Evaluator Simulation Symbolic Evaluator	Timing Analyzer Monte Carlo Sim Monte Carlo Sim Simulation Monte Carlo Sim

TABLE 3.5.1
V & V WITH A SET OF TOOLS

The emerging directions in the development of computer tools for assistance in verification are two fold. First, software organizations are rapidly expanding their own set of tools that, while serving to automate a given group's verification capability, aggravates the independent V & V and certification processes. The second direction consists of interfacing individual tools into integrated CAD packages containing tools for functional design (preliminary and detailed) through hardware and software V & V and certification.

3.5.5 Software Quality Assurance (SQA)

3.5.5.1 Purpose

The purpose of software quality assurance (SQA) is to assure that delivered software conforms with the established system requirements and performance levels. The main theme is that quality in the product enters through the design process and not through testing. The SQA process verifies the correct implementation - during the design process - of the software standards defined in the specification. Verification at the end of each software phase is achieved by use of software inspections and reviews, providing a high degree of confidence that the development results are consistent with the requirements. Finally, the SQA organization shall, in cooperation with the software engineering organization, accept the software for delivery.

3.5.5.2 Tasks of SQA

The tasks of SQA are to insure that:

- The phases of the software life cycle as defined in the project guidelines are properly respected and implemented.

- The software reviews as defined in the project standards for each of these phases are properly performed.
- Appropriate inspections are performed before each formal review of each software life cycle phase.
- Required software audits are performed.
- The software is acceptable for delivery.

3.5.5.3 Responsibility

Each participant should have an identified organizational element responsible for the specific software produced under a given project.

3.5.5.4 Document Review and Checking

Each of the documents generated during the development phase must be checked by SQA for adequacy against the requirements and software documentation quality characteristics such as:

- Consistency
- Traceability
- Clarity, readability
- Structure
- Completeness with respect to hardware and software interfaces

For each of the documents, a report is established prior to each formal review of the results of each phase of the software life-cycle.

3.5.5.5 Standards, Practices, and Conventions

The software standards, practices, and conventions to be used in any development project are identified in a project contract document. These standards, practices, and conventions are requirements to which contractors and subcontractors are obliged to conform.

3.5.5.6 Reviews, Software Audits, and Software Quality Inspections

Reviews:

Software reviews are planned consistent with the project milestones and shall be explicitly shown in the relevant development plans. SQA is a part of all software reviews and its role is to:

- Assure that the requirements are adequately stated in the software requirements document.
- Assure that all requirements are implemented in the design and the discrepancies (if any) identified in the reports are solved/tracked.
- Assure that all the software tests during the specific life-cycle phase have been successfully completed by witnessing/review of test results.

- Assure that the subsystems have been formally verified by review of the data package prior to software system verification. Successful integration into the software system shall be verified by witnessing of integration.
- Assure that system tests are carried out according to approved procedures.

All software data packages are inspected/reviewed by SQA before every project review and any discrepancies noted against the requirements are identified for processing and resolution.

Software Audits:

Software audits are performed by SQA consistent with the general audit rules and an audit report is prepared.

Software Quality Inspection (SQI):

The objective of SQI is to compare the achieved quality of the products under contract with the specified quality, and to record the results of the inspection in the inspection report. Each participant in an SQI prepares a SQI report, which is submitted to the prime contractor for review. This is typically submitted to the prime contractor one week before the submission of formal review documents. The SQI report contains information about the following points:

- Formal inspection process
- Results of inspection of the general content
- Results of inspection of the specific content
- Decisions on further processing
- Release of documents
- Discrepancies identified and proposed corrective actions

A document may be subject to SQI several times until all points are clarified and/or issues resolved. For each review sequence, a special SQI report is generated

3.5.5.7 SQA of Configuration Management

SQA of configuration management assures that configuration management is performed consistent with the configuration management plan.

3.5.5.8 Tools, Techniques, and Methods for SQA

Special software tools, techniques, and methods that support the quality assurance objectives, are described in the respective software development plan. Their purpose and use are also described.

In the United States, a series of contract developments are currently underway at the Rome Air Development Center (RADC) to formalize a set of quality assurance procedures and metrics.

3.5.5.9 Quality Factors and Criteria

Table 3.5.2 displays some user quality concerns. The concerns for performance, validity, and adaptability of software are correlated with many of quality factors. Each of the user quality concerns can be broken down into several criteria as shown in more detail in Table 3.5.3.

ACQUISITION CONCERN	USER CONCERN	QUALITY FACTOR
PERFORMANCE - HOW WELL DOES IT FUNCTION	HOW WELL DOES IT UTILIZE A RESOURCE? HOW SECURE IS IT? WHAT CONFIDENCE CAN BE PLACED IN WHAT IT DOES? HOW WELL WILL IT PERFORM UNDER ADVERSE CONDITIONS? HOW EASY IS IT TO USE?	EFFICIENCY INTEGRITY RELIABILITY SURVIVABILITY USABILITY
DESIGN - HOW VALID IS THE DESIGN	HOW WELL DOES IT CONFORM TO THE REQUIREMENTS? HOW EASY IS IT TO REPAIR? HOW EASY IS IT TO VERIFY ITS PERFORMANCE? HOW EASY IS IT TO EXPAND OR UPGRADE ITS CAPABILITY OR PERFORMANCE?	CORRECTNESS MAINTAINABILITY VERIFIABILITY EXPANDABILITY
ADAPTION - HOW ADAPTABLE IS IT	HOW EASY IS IT TO CHANGE? HOW EASY IS IT TO INTERFACE WITH ANOTHER SYSTEM? HOW EASY IS IT TO TRANSPORT? HOW EASY IS IT TO CONVERT FOR USE IN ANOTHER APPLICATION?	FLEXIBILITY INTEROPEREABILITY PORTABILITY REUSABILITY

TABLE 3.5.2
QUALITY CONCERNS

.....;
endif;

could be rapidly placed on the screen after typing "if". Additionally, the syntax directed editor can check the structure of the source text for compliance with the rules of the language. Thus, the efficiency of the edit

3-22

ACQUISITION CONCERN		PERFORMANCE				DESIGN		ADAPTATION						
FACTOR	CRITERION	EFFICIENCY				INTEGRITY		RELIABILITY						
		SURVIVABILITY				USABILITY		CORRECTNESS						
		MAINTAINABILITY				VERIFIABILITY		EXPAND ABILITY						
		FLEXIBILITY				INTEROPERABILITY		PORTABILITY						
		REUSABILITY												
PERFORMANCE														
ACCURACY				X										
ANOMALY MANAGEMENT				X										
AUTONOMY					X									
DISTRIBUTEDNESS					X									
EFFECTIVENESS-COMMUNICATION		X												
EFFECTIVENESS-PROCESSING		X												
EFFECTIVENESS-STORAGE		X												
OPERABILITY					X									
RECONFIGURABILITY					X									
SYSTEM ACCESSIBILITY			X											
TRAINING				X										
DESIGN														
COMPLETENESS							X							
CONSISTENCY							X	X						
TRACEABILITY							X							
VISIBILITY								X	X					
ADAPTATION														
APPLICATION INDEPENDENCE										X				X
AUGMENTABILITY											X			
COMMONALITY												X		
DOCUMENT ACCESSIBILITY						X								
FUNCTIONAL OVERLAP												X		
FUNCTIONAL SCOPE														X
GENERALITY									X	X				X
INDEPENDENCE											X	X		X
SYSTEM CLARITY											X		X	X
SYSTEM COMPATIBILITY											X			X
VIRTUALITY									X					
GENERAL														
MODULARITY					X			X	X	X	X	X	X	X
SELF-DESCRIPTIVENESS								X	X	X	X	X	X	X
SIMPLICITY				X				X	X	X	X		X	X

TABLE 3.5.3
SOFTWARE QUALITY FACTORS AND CRITERIA

3.5.5.10 SQA of Code Control and Media Control

The SQA function ensures that code and media control as documented in the development plans for individual software packages and carried out within the scope of a development project is adequate to guarantee identification of tested and delivered code with its accompanying documentation. The SQA function regarding code control is designed to specifically prevent any unknown accidental or malign modification of any relevant code.

3.5.5.11 SQA Standards

The United States DoD has started the development of a SQA standard, DOD-STD-2168. This standard, which is currently in draft form, establishes the requirements for software quality evaluation and related activities to be performed during software development. The main theme is that quality in the product enters through the design process and not through testing.

The SQA standard describes a process for evaluating the software products as they are being developed. It is accomplished as a parallel and independent effort from the development of the software product. To be effective, DOD-STD-2168 is to be used with DOD-STD-2167, Software Development. The latter standard provides the requirements for planning and building quality into the product. Fig. 3.5.1 illustrates the relationships between the two standards.

The requirements for individual SQA evaluations are specific to a particular phase of the software development cycle. During any given phase of the cycle, the developer must verify that: (1) all required activities are performed, (2) required approaches, tools, and techniques are utilized, (3) the process is in compliance with customer procedures and standards, and (4) the process is adequate to develop and document all requirements. To assess quality, software quality factors that the software is required to possess may be assigned and specified. Examples of these factors are: efficiency, integrity, maintainability, reliability, reusability, and testability. The extent to which SQA assesses the degree of incorporation of such quality factors, along with the evaluation of the adequacy of all other requirements, constitutes an important part of the SQA process.

3.6 SOFTWARE PROJECT MANAGEMENT

3.6.1 Introduction

The first task of software project management is to define the approach to be used in managing the project. The phase model of Chapter 2 of this report presents an approach that is tailored to military mission critical software (and systems) in general and to flight critical software in particular. Figure 3.1.2. shows activities, documents, reviews, and baselines needed for the software development, test, and integration process required by DOD-STD-2167 (Reference 3.6.1). It represents the current state of practice within the United States. Details vary from project to project and country to country but the products, activities and reviews are typical of FGCS software projects. After selecting the general approach, the detailed methods, procedures, and tools must be selected to support the development of each product, to complete each activity, and to conduct management and technical reviews.

Table 3.6.1 shows some of the functions needed in the day to day management of all software projects. Currently, the management functions are supported by stand-alone tools or small tool sets; there are a few examples where the management functions are integrated with the technical activities. Today, it is primarily up to the people performing these functions to complete the needed integration. In the following paragraphs each management function will be discussed.

3.6.2 Management

3.6.2.1 Project Planning

Project planning includes the generation and maintenance of a work break down structure so that duties and

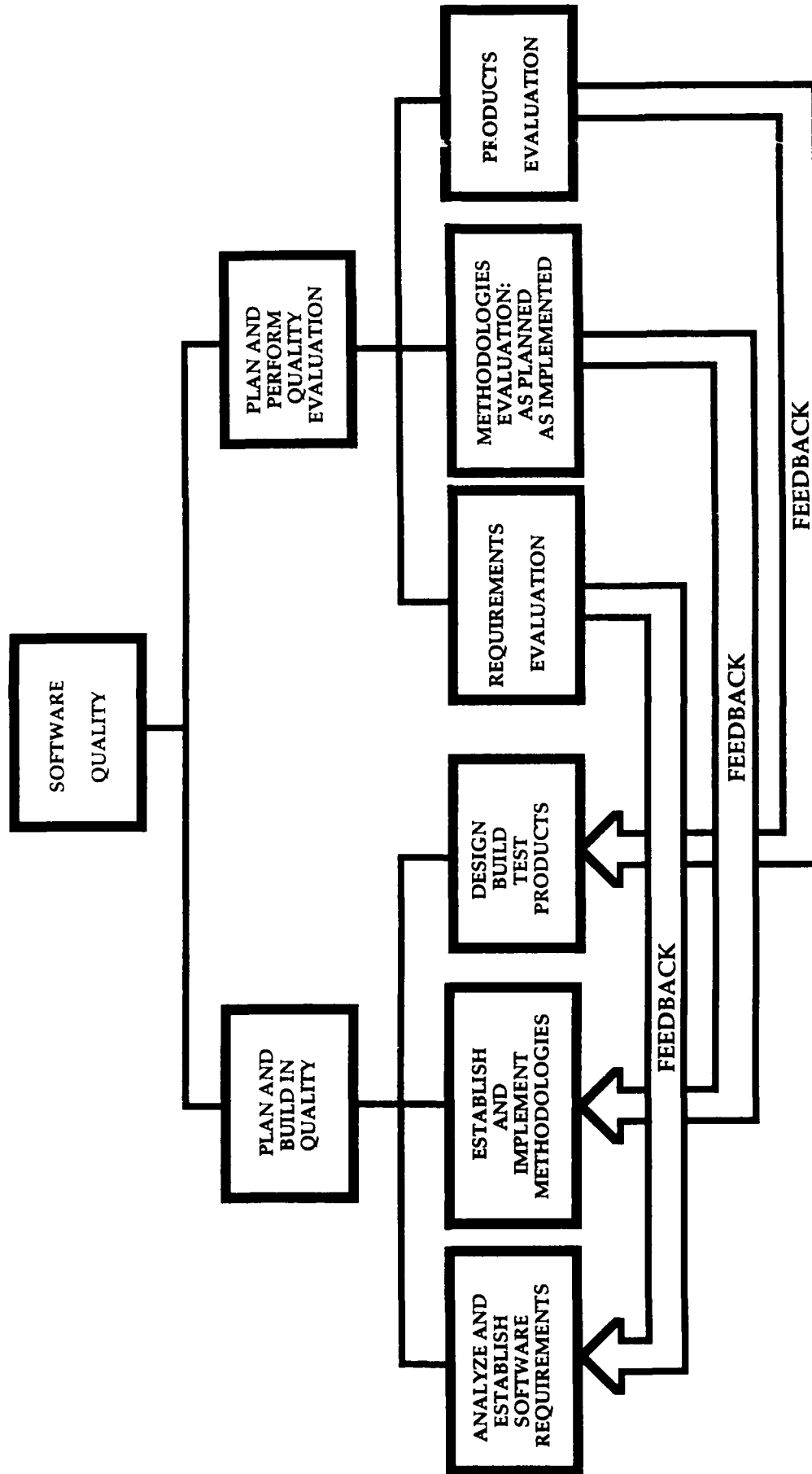


FIGURE 3.5.1 SOFTWARE QUALITY FRAMEWORK

MANAGEMENT Project Planning Project Control Project Communications
DOCUMENTATION Engineering Documentation Formal Management Documentation Informal Documentation
CONFIGURATION MANAGEMENT Baseline Identification Control and Tracking of Access and Change Control of Software Releases

TABLE 3.6.1
PROJECT MANAGEMENT FUNCTIONS

resources can be assigned to the various project teams. Planning the initial schedule and tracking the planned versus actual schedules must also be done. Stand-alone methods and tools for performing these activities are widely available and used. However, quantitative measures of progress or of productivity are still generally primitive and imprecise.

At the project management level, resource estimation is primarily directed toward the amount and skills of labor required. Estimating the time phasing of personnel is also required. Methods and tools (for example the COCOMO Model, Reference 3.6.2) are available and are in use. Significant improvement is needed in this area to improve the accuracy of the estimating process. Further, it is desirable to be able to start the process using the rough parameter values that are readily available at the beginning of the project.

Large projects require contracting and subcontracting for some of the work. Stand-alone methods and tools are available to help in the preparation of these procurement actions.

3.6.2.2 Project Control

There are three aspects to project control that are amenable to improvement through use of appropriate methods and tools. The first is measuring the current status of planned funding, expenditures, manpower expenditures, and activities (e.g., state of completion of software design). The second is insuring that project standards and conventions are enforced. The third is to insure that project information is accessible to those and only those who need it. Some methods and tools are available in these areas, however, considerable improvement is needed. Today's software engineering environments give limited automated support to collecting data to support project status tracking.

3.6.2.3 Project Communications

There are many aspects to project communications. They include the communication of information among project personnel and between project personnel and external support or control personnel. Also included is the communication between people and project data bases (for example, system specifications, source code) and communication of information or data from one data base to a different one (for example, transferring a loadfile from the software engineering environment host computer to the target computer). In current practice most communication still is paper oriented and requires manual handling. Electronic mail, various

electronic bulletin boards, and automated calendar systems are coming into use. Electronic storage and retrieval of documents is beginning to be practiced in some projects. Networking of computers used for software engineering environments and for testing is sometimes used and is becoming more prevalent.

3.6.3 Documentation

3.6.3.1 Engineering Documentation

Documents such as the System/Segment Specification and Software Detailed Design Document (SDDD) of DOD-STD-2167 (Figure 3.1.2) are produced as part of the software development process. They record the functionality and design of the software at varying levels of detail. In today's practice the more detailed documents starting with the SDDD are frequently produced as a by-product of the development process. The SDDD is frequently produced from the PDL (Program Design Language) description of the computer program. Less frequently the top level documents are derived directly from specialized requirements and design methods supported by tools such as PSL/PSA (Reference 3.6.3) and the System Specification Language of DCDS (Reference 3.6.4).

3.6.3.2 Formal Management Documentation

In addition to documents such as schedules and reports to track performance on day-to-day operations, there are many formal management documents required in a complex software development process. DOD-STD-2167, for example, defines a Software Development Plan, Software Configuration Management Plan, and a Software Quality Assurance Plan. In today's practice these are developed and maintained using general purpose word processing tools. They are stored and catalogued using manual or computer assisted library techniques.

3.6.3.3 Informal Documentation

Many special purpose documents are generated during a large software development process. These include reports on special studies, presentations at reviews, minutes of meetings, and guidance and policy papers to cover special management and technical issues. In today's practice these are developed using word processing or manual methods and stored and catalogued for historical and reference purposes using manual or computer assisted library techniques.

3.6.4 Configuration Management

3.6.4.1 Baseline Identification

The software baselines must be identified and defined. Baselineing consists of providing technical descriptions of all software items and dependent hardware items that make up a system. This includes all the documentation, all the forms that the program might take (source, object, load image), and all forms of the related support software (translators, binders, job control, etc.). The identification function includes mechanisms that permit changes in identification only when authorized. This function is generally partly automated. Data Base Management Systems are used to store both the software items themselves and all related information.

3.6.4.2 Control and Tracking of Software Access and Change

Configuration Management includes mechanisms for requesting and approving change to configuration controlled items. Specific functions include documentation of trouble reports, change proposals, and configuration control board agenda and minutes. This function is generally partially automated by data base management systems and word processors.

3.6.4.3. Control of Software Releases

Configuration management includes the process of building and collecting sets of software for both formal and interim releases. This includes all pertinent data and documentation. As in the two previous functions this function is generally partially automated.

REFERENCES

- 3.6.1 DOD-STD-2167, Military Standards Defense System Software Development, Draft of 5 Dec 1983, prepared by United States of America Joint Logistics Commanders.
- 3.6.2 Software Engineering Economics, B. Boehm, Prentice Hall, 1981.
- 3.6.3 Problem Statement Language (PSL) Language Reference Summary, ISDOS Project, University of Michigan (August 1981).
- 3.6.4 Architectural Description of the Distributed Computing Design System (DCDS), 3 December 1984, TRW Defense Systems Group, Huntsville, Alabama 35805.

APPENDIX 3.1

[illegible]

List of Methods vs. Life Cycle																			
	Concept																		
	System/Subsystem Req. and Design		Software Requirements		Software Design		Coding and Module Tests		Software Integration		Hardware/Software Integration		Subsystem/System Integration		Flight Tests		Project Management		Configuration Management
METHODS	1	2	3	4	5	6	7	8	9	PM	CM	Related Methodology							
MAESTRO				X	X					X	X	JSP							
MASCOT				X	X	X	X					n/a							
mbp-tool-sy			X	X	X							n/a							
MODUS			X									n/a							
MOSES				X	X	X						n/a							
MOTOR			X	X	X					X	(X)	ERA Models							
NASSI				X								n/a							
NET	X	X	X									Petri Nets							
OOD				X	X							n/a							
PAISLey												n/a							
PAPICS											X	n/a							
PRACTIPLAN										X		n/a							
PRADOS	[X]	[X]	X	X	X	X				X		SAD1							
PROMOD	[X]	[X]	X	X								SA, SD							
SADT	X	X	X	(X)								n/a							
SAFRA	X	X	X	X	X	X	X					CORE,MASCOT,PSL/PSA							
SARA	X	X	X	X	X							Petri Nets							
SARS	[X]		X								(X)	n/a							
SCR			X	X	X	X	X					n/a							
SDS			X	X	X					X		n/a							
SEM	X	X	X									ERA Models							

List of Methods vs. Life Cycle												
	Concept	System/Subsystem Req. and Design	Software Requirements	Software Design	Coding and Module Tests	Software Integration	Hardware/Software Integration	Subsystem/System Integration	Flight Tests	Project Management	Configuration Management	
	1	2	3	4	5	6	7	8	9	PM	CM	Related Methodology
SET				(X)							X	n/a
SINDUS				X	X							N/S Diagrams
SLAN-4			X	X	X							n/a
SREM												n/a
SSSA		X	X	X	X							n/a
STRADIS	X	X	X	X	X	X	X	X	X			n/a
USE			X	X	X							n/a
WESIR											X	n/a

ABBREVIATIONS:

X - applicable

(X) - partially applicable

[X] - applicable but not especially developed for this purpose

n/a - not applicable (method not related to any others)

ERA - Entity, Relationship, Attribute

LCP - Logical Construction of Programs

LCS - Logical Construction of Systems

N/S - Nassi/Shneiderman

SA - Structured Analysis

SD - Structured Design

Appendix 3-2

DEFINITION OF ACRONYMS	
Acronym	Full Name or Description
ACM/PCM	Active and Passive Modelling
ADORA	Module Test Tool (product name)
AIM	Ada Integrated Methodology
AMDES	Application Development System for Software (product name)
BOIE	Tool System for System and Software Development (product name)
CADA	Computer Aided Design of Ada Software
CADOS	Computer Aided Design for Organizers and System Engineers
CAMIC/S	Computer Aided Design and Test for Micros (product name)
COMPASS	Specification tool using DSL (product name)
CONTEXT	(product name)
CORE	Controlled Requirements Expression
CSTS	Cross Software Test System
DADES	Data Oriented Design
DAISEE	Decentralized Architecture of an Information Systems Engineering Environment
DARTS	Design Aid for Real Time Systems
DCDS	Distributed Computing Design System
DEDOC	Design and Documentation System
DIPROTOR	Diagram Program Generator
DLAO	Definition de Logiciel Assistee par Ordinateur
DSDA	Disciplined Software Design Approach

DEFINITION OF ACRONYMS	
Acronym	Full Name or Description
DSSAD	Data Structured Systems Analysis and Design
DSSD	Data Structured Systems Development
EDM	Evolutionary Design Development
EPOS	Entwurfsunterstuetzendes Prozess-orientiertes Spezifikations- system (Design-supporting process-oriented specification system)
ESPRESO	Erstellung der Spezifikation von Prozessrechner Software (Generation of the Specification of Process Computer Software)
FLOW	(product name)
GEIS	Gradual Evolution of Information Systems
HDM	Hierarchical Development Methodology
HOS	Higher Order Software
IBMFSO-SEP	IBM Federal Systems Division Software Engineering Practices
IDAS	Integrated Data Acquisition and Simulation System
IE	Information Engineering
IORL/TAGS	Input/Output Requirements Language Technology for the Automated Generation of Systems
ISAC	Information Systems Work and Analysis of Changes
ISDOS	Information System Design and Optimization System
PSL/PSA	Problem Statement Language/Problem Statement Analyser
JSD	Jackson System Development
JSP	Jackson Structured Programming
LISKOV-ZILLES	Named for Barbara Liskov and Stephan Zilles
LITTON-DSD	Litton DSD Software Development Method

DEFINITION OF ACRONYMS	
Acronym	Full Name or Description
MACH	Methode d'Analyse et de Conception Hierarchisee
MAESTRO	Software Development System for Philips MAESTRO Computer
MASCOT	Modular Approach to Software Construction, Operation, and Test
mbp	Tool Set from mbp (Mathematischer Beratungs und Programmierungsdienst GmbH)
MODUS	Methodische Organisation und Dokumentation von Steuerungsablaeufen (Methodical Organization and Documentation of Control Flows)
MOSES	Methoden Orientierted Software Entwicklungs System (Method Oriented Software Development System)
MOTOR	Tool set based on ORGWARE-M (product name)
NASSI	A tool producing Nassi/Shniedeman diagrams (product name)
NET	Petri net modelling tool
OOD	Object Oriented Design
PAISLey	Process-oriented Applicative, Interpretable Specification Language
PAPICS	Product and Project Information Control System
PRACTIPLAN	Project management support tool (product name)
PRADOS	Project Abwicklungs und Dokumentation System (Project Management and Documentation System)
PROMOD	Project Model (product name)
SADT	Structured Analysis and Design Technique
SAFRA	Semi-Automated Functional Requirements Analysis
SARA	System Architect's Apprentice
SARS	System for Application Oriented Requirements Specifications

DEFINITION OF ACRONYMS	
Acronym	Full Name or Description
SCR	Software Cost Reduction Methodology
SDS	Software Development System
SEM	System Encyclopedia Manager
SET	System Engineering Tool
SINDUS	Structure Interpretation and Display Utility System
SLAN-4	Software Language-4
SREM	Software Requirements Engineering Methodology
SSSA	Structured Software Signalling Architecture
STRADIS	Structured Analysis, Design, and Implementation of Information System
USE	User Software Engineering
WESIR	Wartung und Erzeugung von Sourceprogrammen fuer Implementierung und Releasepflege (Maintenance and Generation of Source Programs for Implementation and Release)

Appendix 3.3

Current Languages and Environments

CMS-2/MTASS

CMS-2

CMS-2 is a hardware independent, problem oriented, high-level programming language. It is used to generate programs for real-time command and decision applications as well as scientific applications. CMS-2 cross compilers exist for US Navy 16-bit and 32-bit standard computers. The 16-bit version, CMS-2M retains the features of CMS-2 but adds new features that enhance clarity and versatility for dividing programs into semi-independent modules.

CMS-2 uses familiar English words and algebraic expressions to define the logical operations and data manipulations to be done by the computer. It is structured to provide a means of describing the various data arrangements and processing instructions as an orderly set of statements.

A CMS-2 program has two parts:

- Data Designs - A series of data description statements that define all data structures.
- Procedures - A series of executable statements that explicitly define the computer operations.

MTASS

The US Navy Machine Transferable Support Software (MTASS) is designated as the standard tool set for the development of CMS-2 software for Navy standard computers. MTASS/L is used to support the 32-bit computers and MTASS/M is used for the 16-bit machines. Revision is now in progress that will allow both series of machines to be supported from a common improved MTASS.

The principal features of MTASS include:

- Concurrent use of the system through time shared access.
- Transportability to many host machines.
- Adaptability to each step of the design and maintenance life cycle.
- Project control including project status reports for changes, cost, and schedule.

MTASS includes the following tools:

- CMS-2 cross compiler
- FORTRAN cross compiler
- MACRO cross assembler
- SYSGEN loader/system generator
- SIM simulator of the 16 and 32-bit Navy standard computers

MTASS HOST COMPUTER SYSTEMS

VERSION	OPERATING SYSTEM	PROCESSOR
1	OS 1100	UNIVAC 1100
3	OS-MVT, OS/VS1, OS/VS2	IBM 360, 370, et al.
4	KRONOS, NOS (except BE)	CDC 6000, CYBER 70, 170
5	UNIX-BERKELEY	DEC VAX-11
8	GCOS	HONEYWELL DPS/66
10	VM/CMS	IBM 370, et al.
11	VAX/VMS	DEC VAX-11
13	SCOPE, NOS/BE	CDC 6000, CYBER 70, 170
14	TOPS-20	DECSYSTEM-20
16	UNIX-BELL	DEC VAX-11

FASP (Facility for Automated Software Production)

Introduction

FASP was constructed at the U.S. Naval Air Development Center (NADC), Warminster, Pa. to assist software engineering for weapon system software. Two types of facilities were built: software production facilities and software integration facilities. The software integration facilities were built for each project and consist of laboratory hot-mockups of the embedded computers with realistic simulation of external inputs. FASP, the software production facility, is an integrated software environment hosted on a large scale commercial multicomputer configuration. The host configuration consists of five Control Data Corp. mainframes, a CYBER 175, CYBER 730, two CYBER 760's, and a CYBER 875. FASP became operational in July 1975 and was the first integrated environment to be used for weapon system software development and among the first integrated environments. FASP supports the life cycle activities from Mission Requirements to Code and Test; however, only the Code-and-Test and later phases are supported by an integrated environment. In the earlier life cycle activities, the support is provided by loosely-coupled sets of tools. FASP is described in references [1] and [2].

Usage

FASP is used extensively. It is specified as government-furnished equipment on contracts, and the availability and performance has been more than adequate for the development, test, and support of weapon system software. During the period July 1975 through Sept, 1985, about 1,400,000 jobs were processed and about 20,000,000 lines of source code were online at the end of the period. The total source lines developed during the period was several times larger.

Benefits

By using FASP for weapon system software development and support, NADC experienced improved quality (less than 2 errors per 1000 lines of source), greater productivity (over 400 lines/mm; software reuse between projects), faster technology transfer (CYBER FASP given to corporations; rehosted to VAX UNIX), and better management visibility and control (NADC and its contractors were able to support several large projects and many smaller ones).

Technical Features

1. Integrated System

FASP provides dual functions; to the software engineer it appears as an advanced programming system, to the project manager it appears as an information and control system.

FASP permits the use of several development methods, allowing simultaneous use by several projects and their contractors.

FASP allows incremental software development.

FASP supports structured programming with "include" segments.

FASP supports configuration management, including identification, control, status accounting, and the establishment of baselines.

FASP provides three types of management control: control over software tools, control over access to the software end products, and control over access to the host computer system.

2. The Data Base

FASP uses a centralized data base with a fixed number of libraries that are encapsulated and managed as a whole rather than distinct parts. Each data base contains the following libraries:

- The source library.
- The object library.
- The test library containing test input data, previous test results, test directives, and system generation directives.
- The interface data library containing information such as linkages to external object programs or to shared source code.
- The production data library containing modification histories, and a variety of management information.

The software for a particular weapon system is contained in several "data bases". User commands allow software to be shared between data bases, allow data bases to be divided into smaller ones or combined into larger ones; other commands allow the data bases to be copied.

Source and object code libraries are synchronized; likewise, for test data and test results. At any time in the development schedule the management data is consistent with the technical data.

FASP automatically recompiles dependent modules when software is modified.

3. Procedures, Tools, Commands, and Processing

A set of procedures is defined that are invoked by parameterized user commands. A procedure is a set of computer directives that automates a particular work task, invokes the proper tools in the proper sequence, provides all data base manipulations and correspondences, and automatically records statistics of all activities. A tool or set of tools are automatically invoked as part of the execution of a procedure. A benefit of procedures is repeatability, use of a standard tool set, and a commonality of how work is done.

4. Testing

FASP supports four distinct types of testing; they are:

- Progression testing that evaluates new or modified software operation.
- Regression testing that identifies changes to previously attained software operation.
- Automated test analysis that measures the effectiveness of a test by identifying the software source code paths exercised.
- Trial testing that provides for testing proposed software changes without modifying the data base.

Regression testing is used once proper operation is achieved. Test data, test results, and test directives are accumulated in the data base during the life of a module; tests are automatically run whenever the module is modified.

A tool called ATA (Automated Test Analyzer) scans the source code and inserts software probes at program decision points. The instrumented code is executed with the test data on a software emulation of the target computer. The system reports how many times each path was executed, flagging those not executed. The percentage of total paths tested is available along with indications of "dead code" and code paths most frequently executed.

5. Multi-languages and Multi-target Computers

FASP provides CMS-2M, CMS-2Y, SPL/I, and corresponding assemblers for the Navy standard computers; a FORTRAN compiler and assembler are provided for the host computer.

The languages and other tools for a particular target computer are presented to the users as an integrated string of tools.

6. Distributed Computing

FASP is implemented as a partially distributed system on the multicomputer host configuration; all computers have access to the FASP data base. At NADC users and project integration facilities are interconnected with the host computers by a high speed local area network. Remote users, primarily contractors, access FASP by commercial communication services; additionally, ARPANET and satellite communications are provided.

7. Security

The multicomputer host configuration allows the computers to be electronically partitioned so that one or more computers can run in secure mode. The use of cryptographic equipment allows remote users to execute FASP while in secure mode.

References

- [1] Stuebing, H. G.; "A Modern Facility for Software Production and Maintenance"; Published in: NATO AGARDograph No. 258 Guidance and Control Software, May 1980, pp 3-1, 3-14. Also published in Military Electronics Defence Expo '80 Conference Proceedings, Wiesbaden, Germany, Oct 1980, pp 828-845 and in Proceedings IEEE COMPSAC '80, Oct 1980, pp 407-418.
- [2] Stuebing, H. G.; "A Software Engineering Environment (SEE) for Weapon System Software"; Published in: NATO AGARD Conference Proceedings No. 330, Software for Avionics, January 1983, pp 45-1, 45-16. Also published in IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, July 1984, pp 384-397.

CORAL-66

Coral 66 is a general-purpose programming language based on Algol 60, with some features from Coral 64 and Jovial, and some from FORTRAN. It was originally designed in 1966 by I. F. Currie and M. Griffiths of the Royal Radar Establishment in response to the need for a compiler for a fixed-point computer in a control environment. In such fields of application, some debasement of high-level language ideals is acceptable if, in return, there is a worthwhile gain in speed of compilation with minimal equipment and in efficiency of object code. The need for a language that takes these requirements into account, even though it may not be fully machine-independent, is widely felt in industrial and military work. The official definition of Coral 66 was formalised taking advantage of experience gained in the use of the language. Under the auspices of the Inter-Establishment Committee for Computer Applications, technical advice was sought from staff of the Royal Naval Scientific Service, the Royal Armament Research and Development Establishment, the Royal Radar Establishment, the Defence ADP Training Centre, from officers serving in all three services and from interested sections of industry.

The present definition is an inter-service standard for military programming which has also been widely adopted for civil purposes in the British control and automation industry. Such civil usage is supported by the Royal Signals and Radar Establishment and by the National Computing Centre at Manchester, on behalf of the Department of Industry. The NCC has agreed to provide information services and training facilities, and enquiries about Coral 66 for industrial application should be directed to that organization.

The controlling authority for Coral 66 is the Inter-Establishment Committee for Computer Applications. There are currently 36 Coral 66 compilers approved by IECCA.

Part of the above is extracted from "The Official Definition of CORAL 66" HMSO 1976.

JOVIAL

JOVIAL J73 (MIL-STD 1879B) is used today as the U.S. Air Force standard language for avionic applications. It is also used in several armonic (armament electronics) applications. JOVIAL is controlled by the Language Control Agent, ASD-AFALC/AXT at Wright- Patterson AFB, Ohio 4533-6503.

JOVIAL compilers exist for many embedded computers. The following list has been compiled from information supplied by the JOVIAL Language Control Facility:

Developer	Host	Target
SEA	DEC-10	DEC-10, 1750A MAGIC 362F, AN/AYK-15
	IBM 370	IBM 370, 1750A
	IBM 4341	TI-990,9900
	DEC-10	Z-8002
	DEC-20	Z-8002, Z-8001
	VAX-11/780	Z-8002
PSS	VAX-11/780	VAX-11/780, 1750A, Z-800X
	DEC-10	INTEL 8086
	IBM 4341	CP-2EX, 1750A
SofTech	IBM 370	IBM 370, DIS/Z-8002

Several other compilers with similar host/target combinations are in development.

THE LTR3 PROGRAMMING LANGUAGE

Introduction

LTR3 is the standard language within the French Ministry of Defense for programming embedded software systems. The LTR3 language is defined in the following document: LTR3 Manuel Officiel de Reference (Indice 2) 30 June 1985 edited by the French Ministry of Defense.

Technical Presentation

1. Lexicography:

- Character set: ASCII.
- Identifier size: Limited to 1 line of LTR3 code.
- Comments: Extend from the character “%” to the end of the line.

2. Syntax:

The syntax adopts the PASCAL style with the following important differences:

- No mandatory order in declarative parts for types, variables, constants.
- No nesting of blocks.
- Compound statements are of the style:

```
if. . . end if;
case. . . . end case;
do. . . . . end do;
```

without any internal declaration (except for a loop parameter).

3. Types:

- LTR3 is a strongly typed language.
- The programmer may declare his own types: Examples are enumerative, array, record, reference (to dynamic memory structures) types, and subtypes for all predefined or declared types.
- Integer, real, character, string, bit chain (“logical”) types are predefined.
- Declarations of implementation clauses are possible only for record types, array type element size and the integer representation of enumerative type values.

4. Subprograms:

- Subprograms are recursive, reentrant, overloadable.
- Three categories of subprograms: procedures, functions, operators.
- Six parameter passing modes allowing one to control both the logical use of the parameters by the subprograms and the implementation (by address or value) of parameter passing.
- No generic subprograms.

- No nested subprograms (subprograms are only encapsulated directly inside modules and their interface may or may not be visible to the external users of the module encapsulating them).

5. Modularity - Overall structure of a LTR3 program

- A LTR3 program is a collection of Modules (similar to Ada Packages) having an INTERFACE visible to external users and a BODY hidden to external users.
- The notion of MODULE together with the possibility of declaring OPAQUE (similar to Ada limited private) types, which can only be used outside the module through their name and the operations declared in the module interface, enables one to implement objects and abstract data types structures directly in LTR3.
- The main program is a unique module, the PROGRAM, with a specific starting block but no interface.
- Modules are elaborated entirely at compile time and the main program can be started directly without any previous run-time elaboration.
- No generic modules.
- No nested modules (all modules are library units).
- Separate compilation of a module interface or a module body.

6. Tasking

- Concurrent tasks may be created and activated (we say "scheduled" in LTR3) on PROCESSES the syntactic structure of which is similar to subprograms.
- A first task is scheduled automatically on the main program starting block.
- A task is scheduled by a specific SCHEDULE statement and then generally processes completely asynchronously to its scheduling task. Its priority is defined by the SCHEDULE statement. Parameters can be passed to the newly scheduled task.
- INTERRUPT PROCESSES can be linked to hardware interrupts. Hardware interrupts activate tasks on interrupt processes which have a priority higher than the priority of any scheduled task, cannot be put in a waiting condition (they may have to wait for a processor to become available if other higher priority interrupts are processed, but they cannot execute a statement implying a "logical" wait).
- Synchronization points between tasks can be declared only in modular declarations; they are purely static (they cannot be local to a process or subprogram and they cannot be referenced by a REFERENCE type).
- The following synchronization point types are available:
 - Protected data sections (simultaneous access by a task to several of them is possible)
 - Semaphores
 - Events (when an event has arrived, all the waiting tasks resume activity; a selective wait primitive on events is available).
 - Mail boxes: they can be of any size (number of elements) and their elements can be of any type (but the type of all the elements in a given mailbox is the same).

7. Exceptions

- Exceptions can be declared by the programmer (there are also predefined exceptions).
- All the exceptions that a subprogram may raise are signalled in its interface.
- An exception can be raised only in a subprogram.
- When an exception is raised in a subprogram, it is never handled locally but propagated to the caller.
- If the caller has a handler for this exception, it is processed by this handler and normal execution flow resumes after the call.
- If the caller has no handler for this exception, then the task executing the caller is immediately terminated.

8. Input/Output

- A FILE constructor allows one to declare logical, selective or sequential access, file types.
- A logical file may be connected/disconnected to a physical file which can be a system file or a peripheral device.
- Classical GET, PUT, READ, WRITE and service operations are available.
- GET, PUT, READ, WRITE, file positioning operations can cause a task to be suspended during the operation.

The ENTREPRISE LTR3 Environment

Company - Information on the LTR3 environment and its distributors can be obtained from the French Ministry of Defense at the following:

CELAR (attention: Mr. Hervouin)
35170 BRUZ FRANCE
Phone number 99429349

Host Machines and Systems

- APOLLO under UNIX
- Other hosts may be made available in the near future:
 - VAX-UNIX
 - BULL/SP37/UNIX

Kernel LTR3 Environment on the Host Machine

- The kernel LTR3 environment essentially manipulates LTR3 objects which are "Modulotheques" (Domains of Libraries), "Contextes" (Program libraries), (main) "Programs", Module interfaces, and Module bodies.
- It is built on the UNIX file management system and the data base "ATOME" (one instantiation of "ATOME" for each "Modulotheque").
- The kernel LTR3 environment is not directly visible to users. It is expected that it will be made visible to customers wanting to add new tools.

Text Editors

- WED (ED plus an interface to LTR3 objects and to the LTR3 parser)
- Other more sophisticated editors will be available in the near future (Vi, WINNIE, . . .).

LTR3 Program Library Management

- A complete program library management system is provided. It is built on two main concepts:
 - "Modulotheque": A name space for LTR3 objects in which several libraries (contextes) can be created.
 - "Contexte": Program library. A LTR3 program object may be shared by several "contextes".
- Modulotheques are "flat" (there is no concept of hierarchy).
- It is possible to define a version name and an edition number for each LTR3 object. It applies then to both the source and binary information relative to this object.

LTR3 Compilers

- A separate parser is available and has a direct interface with some editors.
- Two compilers were validated by summer of 1985:
 - A compiler generating binary objects that are usable for interpreted execution on the host machine
 - A compiler generating 68000 binary code executable under the "MOP3" real time run-time executive

- Other compilers may be made available in the next few years for the following target machines:
 - CMF (The future French standard architecture for medium sized military embedded computers)
 - Electronic Serge DASSAULT 2084 and subsequent embedded computers
 - ROLM/HAWK 32

Run-Time Executives

- Interpreter under UNIX on the host machine
- "MOP3" real time run-time executive with two layers:
 - The kernel "SCEPTRE" (French standard real time run-time executive) which must be implemented in assembly language on each target.
 - The MOP3 routines ("agencies") using the kernel "SCEPTRE" written in PASCAL and therefore portable.

LTR3 Symbolic Debugger - Runs on the host machine in conjunction with the interpreter.

Host to Target Exporter - The "GENAP" tool provides complete facilities to build a LTR3 executable program from user's modules and MOP3 agencies.

Other Tools

- Tools to export/import LTR3 objects from/to UNIX files
- Source compression tools
- Indenter

PEARL (Process and Experiment Automation Real Time Language)

The PEARL language is widely used in the Federal Republic of Germany. PEARL was developed as a HOL for real-time programming of process and experiment automation systems. The basic idea of PEARL is a separation of the application program into two different parts: a Problem-Part and a System-Part.

The Problem-Part includes the data processing functions that are required for the technical process operation, i.e., the Problem-Part is an abstract model of the technical/ industrial process. The System-Part describes the interface of the program with the process environment. It combines the Problem-Part with the existing real HW, e.g., peripheral-devices, interrupt-ports or signal-sources.

PEARL is designed for an "Abstract Machine," assuming a specific PEARL operating system. The target computer system can consist of a single processor or a processor network. The structure of a PEARL-Program is comparable with a HW-System, whose components are associated via a Bus-System.

PEARL offers a comfortable module structure and supports the development of complex real-time programs. A module consists of a System-Part or a Problem-Part or both. The Problem-Part may be subdivided in a Subproblem-Part etc. Each module can be developed, compiled and tested isolated from the total SW-System. The program modules are integrated into the total system at the end of the development process. This PEARL feature allows the program system to be developed by teams working in parallel.

Attributes of the language:

- Modular program structure (permits complex programs),
- Good portability (by means of the division in system-part and problem-part),
- Suitable for distributed microprocessor-systems,
- Special language elements for process-i/o and time-sharing,
- Algorithmic language elements according to the standard of a hol,
- Abstract data-types and operators for use in real-time programming,
- Language standardization by din 66253 (submission to iso),
- Easy to learn.

PEARL-Compiler Overview

Company	TARGET-System	HOST-System	Remarks
ATM Computer GmbH Buecklestr. 3-5 7750 Konstanz	AEG 80-20 ATM 80-30 ATM 80-05 HD ATM 80-10	AEG 80-20 ATM 80-30 ATM 80-05 HD	
BBC AG Postfach 130 6800 Mannheim	DP 1000 DP 1500	DP 1000 DP 1500	PEARL Subset
Dornier System GmbH Postfach 1360 7990 Friedrichshafen	AEG80-20 MUDAS 432 INTEL 8086	AEG 80-20 PDP 11	Subset for Avionic and Distributed Systems
ESG GmbH Vogelweideplatz 9 8000 Muenchen	LITEF LRI 432 SIEMENS 7xxx		Subset for Avionic
Krupp-Atlas-Elektr. Seebaldsbruecker- Heerstr. 235	EPR 1300 EPR 1500 SDR 1300	EPR 1300 EPR 1500 SDR 1300	

2800 Bremen 44

Fraunhofer-Institut
(IITB)
Sebastian-Kneipp-
Str. 12-14
7500 Karlsruhe

GWK GmbH
Postfach 1360
5120 Herzogenrath

Norsk-Data
Deutschland
Abraham-Lincoln-
Str. 30
6200 Wiesbaden

SEL
Hellmuth-Hirth-
Str. 42
7000 Stuttgart 40

Siemens AG
Rheinbruecken-
Str. 50
7500 Karlsruhe

MPR 1300

RDC
SIEMENS 310
SIEMENS R30
(8086)

c't 68000

NORD 10
NORD 100

LSI 11

SIEMENS 330
SIEMENS R10
SIEMENS R20
SIEMENS R30
SIEMENS R40

MPR 1300

SIEMENS 3xx
SIEMENS R30
SIEMENS 310
SIEMENS 7760

c't 68000

NORD 10
NORD 100

PDP 11

SIEMENS 330
SIEMENS R10
SIEMENS R20
SIEMENS R30
SIEMENS R40

co-operation:
Fa. WERUM
Subset for
Distributed
Systems

co-operation:
Fa. WERUM

co-operation:
Fa. GPP

WERUM PEARL Environment

Company

WERUM
Datenverarbeitungssystem GmbH
Glogauer Str. 2A
2120 Lueneburg
tel. : 04131/53066
telex: 2182141 weru d

Host/Target Machines

The WERUM-PEARL System is installed on the following computers:

Host	Target
VAX11/7xx (VMS)	VAX11/7xx(VMS), HP1000F (RTE IV B), INTEL 8086 (iRMX 86), LSI 11, PCS CADMUS 9000 (MUNIX), MOTOROLA 68000, ZILOG Z 8000
SIEMENS 7 . xxx (BS 2000)	SIEMENS 7 . xxx (BS 2000), HP1000F (RTE IV B) INTEL 8086 (iRMX 86), LSI 11, PCS CADMUS 9000 (MUNIX), MOTOROLA 68000, ZILOG Z 8000, RDC, SIEMENS R-SERIE (ORG 300 PV)
SIEMENS R-SERIE (ORG 300 PV)	SIEMENS 7 . xxx (BS 2000) HP1000F (RTE IV B), HP3000 (MPF IV), INTEL 8086 (iRMX 86), LSI 11, ND 100 (SINTRAN), MOTOROLA 68000, ZILOG Z 8000, RDC, SIEMENS R-SERIE (ORG 300 PV)
PCS CADMUS 9000 (MUNIX)	RDC, HP1000F (RTE IV B), INTEL 8086 (iRMX 86), LSI 11, PCS CADMUS 9000 (MUNIX), MOTOROLA 68000, ZILOG Z 8000,
ND 100 (SINTRAN)	ND 100 (SINTRAN), HP1000F (RTE IV B), INTEL 8086 (iRMX 86), LSI 11, ZILOG Z 8000
INTEL 86/310	INTEL 8086 (iRMX) (iRMX 86)
IBM 4341 (CMS)	SIEMENS R SERIE (ORG 300 PV) HP1000F (RTE IV B), LSI 11, MOTOROLA 68000, ZILOG Z 8000, PCS CADMUS 9000 (MUNIX)
HP3000 (MPF IV)	HP3000 (MPF IV), HP1000F (RTE IV B), MOTOROLA 68000, ZILOG Z 8000, PCS CADMUS 9000 (MUNIX)
HP1000F (RTE IV B)	HP1000F (RTE IV B)

Kernel Environment on the Host Machine

None

Text Editors

No special PEARL-Editors are required. The standard editors of the HOST can be used.

PEARL Database-System BAPAS-DB

The real-time database system BAPAS-DB administrates files on background storage or in memory. The data can be accessed concurrently by PEARL tasks and users via terminal. BAPAS-DB includes a relational user interface. Different files can be accessed by different access strategies (hash, B-tree, index-sequential or sequential).

PEARL-Compiler(Cross-Compiler)

The compiler (respectively cross-compiler) translates PEARL programs according to DIN 66253, including data types and operators, pointers, local procedures and nested structures defined by the user. The functions of the PEARL Compiler can be divided into two fundamental parts:

- Preprocessor, to fit in other program-texts or changeable System-Parts,
- Parser, scanner, symbolic analyser, syntax analyser, semantic analyser, generator of symbol tables and cross reference list;
- Code-generators, according to the further processing.

PEARL Runtime-System BAPAS (Basis fuer Prozessautomatisierung)

The PEARL operating system is provided by the target operating system together with the runtime-system BAPAS. The specific PEARL language elements are translated into System- Calls for the TARGET operating system.

PEARL Test-System/Debugger

The PEARL test system allows the testing of PEARL programs interactively on source level. In addition to usual functions (TRACE, DISPLAY, BREAKPOINT) it allows the displaying and changing of the status of tasks and synchronization variables. The debugger is activated by code extensions implemented by the PEARL Compiler.

Host to Target Exporters

The code produced by the PEARL Compiler is further processed on the TARGET Computer (compiling, linking, locating, program start).

Other Tools

GPP PEARL ENVIRONMENT

Company

GPP
Gesellschaft fuer Prozessrechner-Programmierung
Kolpingring 18a
3024 Oberhaching
tel. : 089/61104229

Host/Target Machines

The GPP-PEARL System is installed on the following computers:

Host	Target
SIEMENS 7.531	MICRONOVA, INTEL 8086, PDP-11/03/23
SIEMENS 330	MICRONOVA, INTEL 8086, PDP-11/03/23
PDP 11/34	PDP-11/03/23, INTEL 8086
INTERDATA 7/31	INTEL 8086
DATA GEN. NOVA	MICRONOVA
INTEL 86/330	INTEL 8086

Kernel Environment on the Host Machine

None.

Text Editors

No special PEARL Editors are required. The standard editors of the HOST can be used.

Program Library

None.

PEARL Compiler (Cross-Compiler)

The compiler (respectively cross-compiler) translates PEARL programs according to DIN 66253. The adaption to the special process periphery is provided by the compiler option PRELUDE. The functions of the PEARL Compiler can be divided into two fundamental parts:

- Syntactic and semantic analysis
- Code generators (example: ASSEMBLER)

Run-Time Executives

None.

PEARL Test-System/Debugger

The PEARL test system allows one to test PEARL programs interactively on source level. The debugger is activated by code extensions implemented by the PEARL Compiler.

Host to Target Exporters

The code produced by the PEARL compiler is further processed on the target computer (compiling, linking, locating, program start).

Other Tools

APPENDIX 3.4

THE ADA LANGUAGE AND CURRENT COMPILERS

THE ADA * LANGUAGE

Ada is a powerful, general purpose language with integrated facilities supporting many modern programming practices. Among the requirements for the language were features that would reduce software costs by increasing maintainability, evolvability, reliability, and portability. Ada is suitable for a variety of applications, including systems programming, computational programming, general programming, and especially real time programming. Ada provides language features for multi-tasking such as tasks, rendezvous, priorities, and entry calls. In addition to such powerful programming language features, Ada is able to reduce software life cycle costs by providing for modularization and separate compilation using packages, scope rules, and a compilation data base.

Ada Language Features

Strong typing: Objects (variables) of a given type may take on only those values that are appropriate to that type, and only certain predefined operations may be performed on those objects. Since type checking is done at compile time, strong typing aids in program development by insuring that any errors associated with data types are detected at compile time.

Data Abstraction: Also known as information hiding, data abstraction hides the details of implementation while providing users with mechanisms for using the implementation. Abstraction allows the user to focus on important characteristics while ignoring underlying details. Ada provides various levels of abstraction through such features as data typing and the package mechanism.

Concurrent Processing: For many applications it is important that a program be conceived as several parallel activities rather than a serial sequence of actions. Most high order languages provide little or no support for handling such concurrent activities; they rely on facilities of the host operating system. Ada uses tasks to support parallel activities directly within the language.

Separate Compilation: Ada's separate compilation feature lets a programmer divide a large program into compilation units that may be compiled at different times. When a unit is being compiled, the compilation library provides information about other relevant compilation units. This differs from other languages where independently compiled modules have little information about other modules.

Generic Definitions: A generic unit is a template of an algorithm that can be tailored to particular needs at compile time. Often the logic of a program is independent of types of the values being manipulated. In a strongly typed language such as Ada, all types must be defined at compilation time. A generic program unit can be created by adding a prefix to an existing program unit. This prefix is called the generic part and it defines any generic parameters. A generic routine is called like a subroutine, and becomes non-generic when it is called because it is passed parameters that are defined to have a certain data type.

Exception Handling: In many operations, especially embedded computer systems, it is critical that a system be able to recover quickly and efficiently from error conditions. Ada includes predefined exception conditions, and also permits the user to define exceptions. It provides the ability to "raise" exception conditions and the ability to actually handle conditions. When an exception occurs, normal processing is suspended and control passes to the exception handler.

* ADA is a registered trademark of the United States Government, Department of Defense

ALSYS COMPILER

Company

Alsys S. A.
29 Avenue de Versailles
78170 La Celle Saint Cloud
FRANCE
Phone : (1) 3918.12.44

Alsys, Inc.
1432 Main Street
Waltham, Massachusetts 02154
U.S.A.
Phone : (617) 890-0030

Alsys Ltd.
Partridge House
Newtown Road
Henly-on-Thames
Oxon RG9 1EN, United Kingdom
Phone (491) 579090

Host Machines and System/Status

Native Compilers

HP9000 series 200 and 300 under HP/UX Validated 1.6
SUN2 and SUN3 series under UNIX 4.2 Validated 1.6
Apollo workstations under AEGIS and IX Validated 1.6
PC/AT under MS/DOS Validation test in progress (1.7)
IBM370 under CMS and MVSLate 86

Cross Compilers

VAX/VMS to 68000/UNIX Validated 1.6
VAX/VMS to 80286/MS/DOS Validated 1.6

Kernel ADA Environment on the Host Machine

- ALSYS does not presently adhere to the CAIS.
- ALSYS has a standard (not operating system-dependent) interface set used by all tools. Users will be able to add new tools easily.

Text Editors

- Standard text editors provided by the host system are used.
- ALSYS is also developing a syntactic editor that will be available at the end of 1986. This editor will offer two windows (Syntactic and Text) in order to allow easy insertion of Ada Text within a program structure.

ADA Program Library Manager

Available now.

ADA Compilers

All the Alslys software is written in Ada. Emphasis has been put on code quality and compiler capacity. Third party benchmarks show that the generated code is better than C or Pascal code on the same machine; moreover, the compiler and related tools (amounting to more than 3,000,000 lines of Ada) have been successfully compiled through the Alslys compilers.

Run-Time Executives

Available for bare 68000 and bare 80286 in second half of 1986.

ADA Symbolic Debugger

Available with Version 2.0 of the compilers by mid-86.

Host to Target Exporters**Other Tools**

Other sophisticated Ada oriented tools will be available during 1986. These include:

- Pretty Printers
- Cross Reference Generator
- Library Browser

AIR FORCE ARMANENT LAB (AFATL) ADA CROSS COMPILER

Company

Dept of Computer Science
Florida State University
Tallahassee, Florida 32306
(904) 644-5452

Host Machines and Systems/Status

HOST: CDC Cyber 170 series NOS or NOS/BE operating system
TARGET: Bare Z-8002 embedded microcomputer
Validated to ACVC version 1.6

Rehost/Retarget for VAX/VMS to AN/UYK- 44 in progress by:
Harris Corp GISD,
Melbourne, Florida 32935

Kernel ADA Environment on the Host Machine

Runs as a time-share job on NOS operating system. Largest partition required for compiler operation is 41K words.

Requires standard Pascal environment and Z-8002 cross-assembler on the host machine.

Text Editors

Uses any of the standard text editors on the host.

ADA Program Library Manager

ADA Compilers

- The compiler was designed and coded for the specific purpose of producing code for an embedded microcomputer used for experiments in guidance and control applications. Small size and high execution speed of the produced code were the principal design considerations, along with the requirement to pass validation.
- The compiler is composed of four separate passes that process succeeding representations of the program. At the end of the fourth pass, assembly code is produced. The object code is produced by the assembler. The use of assembly code as one of the intermediate languages allows the interface to modules which are written exclusively in assembly language.
- The first pass performs lexical and syntactic analysis using an LALR(1) parser based on an "S" attributed grammar. Predicates are used to resolve ambiguities and perform preliminary semantic analysis.
- The second pass performs semantic analysis and produces Intermediate Language 1 (IL1) driven by the parse action sequence of the first pass (shift reduce). IL1 is a linked graph representation of the semantic content of the source.
- The traverser pass builds IL2 and storage binding by a top down recursive descent traversal of IL1. IL2 is prefix code (Ada language-independent) for a primitive virtual stack machine.
- The code generator, a table-driven program using the Ganapathi and Fischer technique, performs an LALR(1) parse, similar to pass one, to produce assembly code for the target.

Run-Time Executives

The run-time executive (RTE) is self contained to run on a bare Z-8002 microcomputer. Validation testing was performed on a single board development module. The RTE with debugging (trace) code occupies 8K bytes, and 6K bytes without debugging code.

ADA Symbolic Debugger

Host to Target Exporters

Separate program. Checks for changes to modules and has an option to only load new code.

Other Tools

- Pretty Printer
- Cross Reference Generator.

DANSK DATAMATIC CENTER (DDC) ADA COMPILER SYSTEM

Company

DDC International A/S
Lundtoftevej 1 C
DK-2800 Lyngby
Denmark
Tel: +45-2-87 11 44
Telex: 33704 ddc dk

Host Machines and System/Status

Native Compilers

DEC VAX/VMS Validated 1.6
Honeywell DPS 6(GCOS6 Rel 3.0) Validated 1.4
Honeywell DPS 8(GCOS6 Rel 3.0) Validated 1.4
DEC VAX/VMS (UNIX V) Dec 36
Nokia MPS10 - Nokia Electronics, Finland Late 86
CDC Cyber 180 - CDC, USA In progress

Cross Compilers - Hosted on VAX/VMS

National NS 32016 - Tech Univ of Lund, Sweden Partial
MIL STD 1750A - ACT Inc., USA Late 86
SDS80/A - Swedish Defense Ministry In progress
MARA (8086 & 80286 based) - Selenia, Italy Late 86
Intel 80286 (bare + RMX) - ACT & Intel, USA Late 86
Motorola 68020 (bare + UNIX V) Airtech, Israel Late 86
Christian Rovsing CR 80 Partial

Kernel ADA Environment on the Host Machine - The DDC Ada compiler System has a standardized, well-defined and well-documented KAPSE interface. In rehosting the DDC Ada Compiler, one only has to adjust to the name and parameter convention besides adjusting the KAPSE interface to the new host environment.

Text Editors - Uses editor of host system.

ADA Program Library Manager - All Program Libraries and their manipulations are controlled by the Program Library Manager. Hierarchically tree-structured subordinate libraries are called Domains. A current program library consists of a string, from leaf to root, of Domains. The features are:

- Centralized Program Library Management
- Portable User Interface for Domain Names
- Handling of Concurrent Compilations/Linkings etc.
- Hierarchical Library Structure (Project Oriented)
- Automatic Clean-up Facility
- Deletion of Compilation Units from a Domain
- Creation/Deletion of Domains
- Connect/Disconnect Domains (move from one tree to another)
- Domain Consistency Checking
- Protection of Domains (Parent Domains)
- Protection of Compilation Units (Locking)
- Centralized Container (File) Management
- Tracing Capabilities
- Fast Compilation Unit Access on Name
- Display Information about Compilation Unit

ADA Compilers

- Target-independent Front End with well-defined and well-documented interfaces.
- Code generator middle part reusable for most target architectures.
- Multipass compiler to allow hosting on smaller machines.
- Entire compiler written in Ada, supporting rehosting on targets previously completed.
- Front End provides a number of target independent code optimization algorithms.
- The Code Generator provides traditional code optimization algorithms such as the peephole optimizer.

Run-Time Executives

- Ada tasking kernel written in Ada.
- Large part of I/C system written in Ada.

ADA Symbolic Debugger - Written in Ada. Special feature allows status display of the current Ada tasks.

Host to Target Exporters

Other Tools

DEPARTMENT OF THE ARMY ALS (Ada Language System)

Company

SofTech, Incorporated
460 Totten Pond Road
Waltham, Massachusetts 02254-9197

Host Machines and System

- Validated: VAX 11/780, 785, 8600, MicroVAX II, VMS Version 4.0 and later
 - Retarget development:
 - Intel iAPX86 and iAPX186, validated November 1985
- Intel iAPX286, scheduled for Summer 1986
- AN/UYK-44, scheduled for a prototype delivery in Spring, 1986.

Kernel ADA Environment on the Host Machine

- All host-resident ALS tools access the ALS Environment Database using the KAPSE (Kernel Ada Programming Support Environment) interface. Some of the ALS tools are VMS tools that have been adapted to run in the ALS, such as EDT and RUNOFF.
- The KAPSE appears to the user as an Ada package ALS KAPSE, containing procedures and function for controlling I/O, for managing the database, and for controlling Ada programs.

Text Editors - ALS uses the VAX/VMS EDT text editor and RUNOFF text processor.

ADA Program Library Manager

- The ALS Environment Database is one of three basic components of the system.
- The basic characteristics of the ALS Environment Database are:
 - A hierarchical file structure
 - An access control mechanism that restricts read, append, write, and execute permission.
 - Program libraries that provide private work spaces.
 - Two kinds of sharing so that multiple identical copies will not be required.
 - The ability to freeze files so that they may never be changed in place.
 - Automatic maintenance of derivation histories so that differences between object modules or load images may be rapidly discovered.
- ALS has an interactive program library manager tool, LIB, that allows the user to acquire, delete, examine and manipulate the entries in a program library. LIB accomplishes strict enforcement of module interface relationships.

ADA Compilers

- The ALS compiler can be characterized by the following features:
 - A target machine independent section and code generators for VAX/VMS and Intel 86 families.
 - Machine independent and machine dependent optimizations.
 - Calling sequence trace-back on program aborts and execution frequency monitoring at block level.
 - Machine-independent intermediate language is based on DIANA.

Run-Time Executives

- Run-time support for each target environment is provided by routines and data structures in the run-time support libraries.
- Functions include memory management, I/O request handling, task management, interrupt handling, run-time diagnostic support (VAX only), and exception handling.

ADA Symbolic Debugger

- The debugger provides interactive, symbolic debugging at the source level for programs written in Ada and executing in the ALS environment. These facilities aid the user in localizing malfunctions by providing controlled incremental execution of a program and the symbolic display of its states at various points.
- The program analysis tools can monitor both the time and frequency characteristics of an executing program. The Statistical Analyzer samples the target program to determine, roughly, the amount of time spent executing each subprogram. The Frequency Analyzer records how often each subprogram is executed.

Host to Target Exporters

- The ALS allows exporting of linked containers from a program library to target environments by exporter tools. An exporter is developed for each target environment.
- The exporters create, on an image medium, an executable image for transferral to the target.

Other Tools

- The ALS contains a number of other tools (over 70) including:
 - Configuration controls that assist an administrator or project manager in establishing and controlling baselines.
 - Maintenance tools, such as backup/recovery of files.
 - File oriented tools, such as a bit-by-bit file comparator.

DIGITAL EQUIPMENT CORPORATION VAX ADA COMPILER

Company

Digital Equipment Corporation
Boston, Massachusetts

Host Machines and System/Status

- MicroVAX I; VAX-11/725,730,750,780,782,785 Validated 1.6
 - VAX Venus 8600
- Operating Systems
- VAX VMS Version 4.2 or later
 - MicroVMS Version 1.0 or later

Kernel ADA Environment on the Host Machine

- The VAX "Ada Programming Support Environment" (APSE) consists of the VAX common language environment and the Ada program library management utility (ACS).
- VAX Ada conforms to the VAX Calling Standard, which provides the ability to call and be called by code written in other languages.
- VAX Ada is also able to handle exceptions from non-Ada code, generate exceptions to be handled by non-Ada code, and share data with non-Ada code through global variables and common blocks.

Text Editors

Standard host text editors, including EDT are used. A Language Sensitive Editor (LSE) is also available for Ada.

ADA Program Library Manager

Allows shared use of a program library by multiple programmers; the ability to share compiled Ada code either by reference or copy; the use of individual libraries as sublibraries of team libraries; and automatic recompilation of obsolete units.

ADA Compilers

The compilers are oriented toward medium- to large-scale software development. The intermediate representations used by the compiler are an abstract syntax tree and a code generator intermediate language. The abstract syntax tree representation is similar to DIANA. However, there is no intention to be compatible with DIANA. The compiler is written in several languages including BLISS, PL/1, VAX MACRO, and Ada. The format of the final object program is a standard VAX/VMS object file. The compiler generates a DEBUG symbol table (DST) as part of the object module for use with VAX DEBUG to allow full symbolic debugging.

Run-Time Executives

Storage management is handled by a subheap allocated for each collection. Fixed-size blocks with bit map allocation techniques are used for access types whose denoted type is fixed-size, and variable length blocks with first fit allocation techniques are used for access types whose denoted type is not fixed-size. Collections are deallocated when leaving the scope of the parent access type. Storage for tasks is allocated in two parts: a task control block and a task stack. An Ada run-time kernel implements multitasking within a single VAX/VMS process. The method of passing data parameters in a rendezvous is the same as for subprogram parameters.

ADA Symbolic Debugger

Fully symbolic debugging capability through the VMS debugger is provided for the Ada programmer. This capability includes support for multitasking, packages, and mixed Ada and non-Ada code. All general facilities of a symbolic debugger described above are supported.

Host to Target Exporters

Other Tools

Tools and utilities provided with VAX/VMS include:

- VAX Symbolic Debugger (DEBUG)
- Record Management Services (RMS) routines that aid in designing, managing, and sharing files across languages and applications
- Common Run-Time Library (RTL)
- VAX/VMS file system
- DIGITAL Command Language (DCL)

Optional tools and utilities include:

- Source Code Management System (CMS)
- Module Management System (MMS)
- Language Sensitive Editor (LSE)
- Test Manager
- Performance and Test Coverage Analyzer (PCA)

INTERMETRICS, Inc. ADA COMPILER

Company

Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

Host Machines and System

- System is hosted on 370-architectures (IBM, 370, 43XX, 308X, Amdahl, etc.) running the UTS (UNIX) operating system. Validated in December 1985.
- A rehost to the VAX under VMS and a retarget to the MIL STD 1750A embedded computer is in progress.
- Other hosts are being developed.

Kernel Ada Environment on the Host

The system incorporates a subset of CAIS version 1.2 called the HIF. This implements the CAIS "node model" including attributes and relations. All Tools use the HIF, as do many other Intermetrics Ada tools. The HIF presents an identical (Ada package) interface to the tools on 370/UTS and on VAX/VMS. (The same HIF has been developed for three other operating systems).

Text Editors - The system is compatible with any host-system editor.

ADA Program Library Manager

- Program library manager (PLM) tool
- HIF (CAIS subset) forms the basis
- Dependency information stored in the library by the compiler (human input is not required)
- Version/revision tracking and support
- Specific features include:
 - Automatic recompilation
 - Status listing
 - Compilation order computation
 - Shared sub-libraries
 - Locking of revisions (freezing)
 - Create, delete, examine, modify
 - Control links to other libraries
 - Perform partial compilation
 - Resume compilation which did not complete

ADA Compilers

- Uses/generates DIANA
- Global optimizer
- Table-driven code-generator
- Optimization after INLINES, address calculations

- Look-back/look-ahead register allocator
- Table-driven syntax error recovery
- Separate, independent listing generator (with interspersed source and assembler, cross reference)
- Most programs and rep specs

Runtime Executives

- Written primarily in Ada
- Run on UNIX (on 370), bare machine (on 1750A)
- No overhead to set up exception handlers
- Haberman-Nassi tasking optimization (entry calls use caller's stack)
- Storage management uses "multi-queue, first-fit" algorithm — no "garbage collection"

Symbolic Debugger

Debugger being built under separate effort.

Host to Target Exporter

The 1750A compiler produces load modules in the correct download format.

Other Tools

UNIX and VAX/VMS Tools can easily be used with the system. The Intermetrics Byron Ada PDL, and 23 additional tools are compatible. The current effort also includes a stub generator, a linker, a preamble generator, a listing/ cross-reference generator. Configuration management and library-related functions are provided by the HIF and PLM. The 1750A compiler is compatible with VAX/VMS-hosted, government-owned simulator, linker, assembler, and Jovial compiler.

RATIONAL ADA ENVIRONMENT

Company

RATIONAL
1501 Salado Drive
Mountain View, CA 34043 - USA
Phone: 415 340 4700

Host Machine - RATIONAL R1000

Kernel ADA Environment - On the Host Machine

The RATIONAL environment provides the following facilities:

- Predefined object management
- ADA as a command language plus many commands on function keys
- An interface to the DIANA tree intermediate language of Ada programs

Text Editor

The text editor is fully integrated into the environment; it can behave as well as a text formatter and an Ada syntactic editor.

ADA Program Library Manager

The user may create/delete Ada program libraries. Program libraries keep only the DIANA tree representation of Ada library units and the object code. Ada source code is not preserved when a library unit has been successfully compiled.

The coherence of an Ada library is strictly enforced, but the necessary recompilations are minimized by the fact that dependencies are tracked down to individual declarative items.

ADA Compilers

In June 1986 only the host compiler for the R1000 machine is available. Compilers are being developed for the 1750A and VAX target computers. The development of a compiler for the 68000 family of processors is due to be started in the summer of 1986. The compiler provides DIANA trees, object code and good quality compilation diagnostics.

The different compilation phases - parsing, semantic analysis, and code generation can be processed separately. The compiler provides also a very interesting capability of incremental compilation and semantic completion of an incomplete unit which can be very efficiently used in conjunction with the text editor.

Run-Time Executives

The R1000 architecture was specifically designed to provide an efficient implementation for the Ada run-time executive. The two main consequences of that are:

- Type checking takes almost no extra time (it is processed concurrently with expression evaluation).
- Task handling is much faster than on conventional monoproductors.

The Ada standard I/O packages are implemented in a very basic way (for example, they cannot be shared by several concurrent tasks of the same program trying to address the same peripheral) but the environment provides some I/O packages that satisfy a large majority of needs. Nothing is known in June 1986 about run-time executives on machines other than the R1000.

ADA Symbolic Debugger

A powerful Ada symbolic debugger is available. It currently debugs program on the R1000 machine only, but it is designed also to be a crossed debugger capable of controlling from the R1000 machine a debugging session for a program running on a target machine.

This debugger has two specific interesting features:

- Breakpoints and traces can be positioned as well in declarative section elaborations as in imperative statement sequences
- On the R1000 machine, no specific compilation and link editing is required to execute a program under the debugger control

Host to Target Exporters - No documentation available in June 1986.

Other Tools

- Pretty Printer: this is provided by the DIANA tree decompiler
- Configuration Management

In addition to the possibility of giving version and edition attributes to Ada units, the RATIONAL environment provides two specific features:

- **Subsystems:** They provide a level of decomposition above the Ada packages. A subsystem gathers several Ada packages and library subprograms and presents to external users (other subsystems) an interface with a subset of the specifications of its packages and library subprograms.
- **Activities:** An activity is a choice of version-edition for each subsystem of a whole system. The environment is able to automatically build a system corresponding to a given activity and use the detailed tracing of inter-unit dependencies to minimize the required compilations.

ROLM AND DATA GENERAL ADA ENVIRONMENT

Company

ROLM/MSC (a subsidiary of LORAL Corp)
One River Oaks Place
San Jose, CA 95134 - USA
Phone: 408 942 8000

Host Machines and System/Status

Host machines: ECLIPSE 32 bits architecture:
DATA GENERAL: MV20000 - MV10000 - MV 8000 - MV 4000 - MV 2000
ROLM: MSE 800
Host operating system: AOS/VS

Kernel ADA Environment on the Host Machine

Built directly upon the AOS/VS file system and standard services. AOS/VS standard services remain available under the Ada environment. It is possible to add new tools to the environment but ROLM and DATA GENERAL do not provide the necessary documentation as a standard.

Text Editors - Standard AOS/VS editors: SED or SLATE

ADA Program Library Manager

- The user may create/delete Ada program libraries.
- Before compiling or link editing, the Ada program coherence is checked automatically.
- Library manager routines can be called by the user to inspect a library or delete library units.
- A library does not allow multiple versions for a given subprogram or package body.

ADA Compilers

Ada compilers are available for the following target machines:

- ECLIPSE 32 bits architecture: Host machines plus the ROLM HAWK32
- ROLM MSE 14
- ROLM 1666B
- 1750 (not yet available in the beginning of 1986)
- 68000 (projected by DATA GENERAL - We have no information about possible availability date)

The compilers produce the following objects:

- DIANA intermediate representation of ADA programs
- Optional Assembly code which is related by comments to the ADA source code
- Binary code
- Optional symbol tables for use by symbolic debugger
- A listing with good quality compilation diagnostics

Run-Time Executives

Host machines run-time executives:

- They are implemented under AOS/VS
- Ada tasks are implemented as coroutines on a single AOS/VS task. However, a few extra AOS/VS tasks are provided to handle inputs, which allows Ada tasks to be suspended while their inputs are handled by these AOS/VS server tasks.

Mil-spec machines run-time executives:

- They are tailored to real-time needs.
- Each Ada task is implemented as an operating system task.
- Interrupts can be handled in Ada.

Ada Symbolic Debuggers

A powerful Ada symbolic debugger is available to debug programs executing on the host machine. It provides good facilities to debug multi-task Ada programs. Target machine debugging monitored from the host machine is also possible.

The only serious limitations of this tool are:

- AOS/VS does not allow for multi-windowing, therefore, the debugger normal output is the single window of the user terminal
- The debugger is driven statically by the program control flow, it cannot be driven by dynamic assertions.

Host to Target Exporters

Each target machine has its specific linker to constitute program files from user packages and subprograms, standard libraries and run-time executives.

The exporters only transfer program files via data links or magnetic tapes.

Other Tools

Configuration Management - It is handled at the Ada source code File level by the standard AOS/VS configuration management tool: TCS configuration and library management are therefore completely separated.

Pretty printer - A complete Ada source code pretty printer is available. It also provides general statistics and cross-references on one Ada source file (if these data are wanted for a whole Ada program, a file containing the whole Ada program must be submitted to the pretty printer).

SYSTEAM KG

Company

SYSTEAM KG
Am Entenfang 10
D-7500 Karlsruhe 21
West Germany

Host Machines and System/Status

- VAX/VMS currently available (validated)
- SIEMENS/BS 2000 currently available
- VAX/UNIX available late 1984
- MC68000/UNIX available late 1984
- 370/MVS available early 1985

Kernel ADA Environment on the Host Machine

Text Editors

ADA Program Library Manager

The library user system allows inspection of the current status and the contents of a project library. The effect of a recompilation can be tested in advance.

ADA Compilers

- The compiler is designed for easy rehosting and retargeting with tools available to help third parties in the effort.
- Uses the DoD-approved DIANA Ada Intermediate language
- To guarantee stability and maintainability, important parts of the compiler are specified by formal methods (Ex.: Attribute grammars for analysis and transformation and formal target descriptions for the code generators. From these, specifications parts of the compiler are generated automatically.)
- The compiler and all other development tools are written in Ada
- The compiler can compile itself.

Run-Time Executive

- **Complexity Analyzer:** Provides a measure of program's complexity.
- **Completeness Tester:** Shows where the execution flow has gone through a program and how many times it passed there.
- **Visibility Tool:** Provides an automatic print of all the identifiers used within a unit and declared outside.
- **Automatic Recompilation:** After the recompilation of a given unit, this tool automatically recompiles all the dependent units.
- **Configuration Management Tools:** The CM concepts are only in the specification/design stage. Therefore they will not be available before the end of 1985. It will be integrated into the library management.

ADA Symbolic Debugger

Host to Target Exporters

Other Tools

SYSTEMS DESIGNERS SOFTWARE

Company

Systems Designers Software, Inc.
Suite 1201
5203 Leesburg Turnpike
Falls Church, Virginia 22041

Systems Designers
Pembroke House
Pembroke Broadway
Camberley
Surrey
Gu15 3XH (+276) 686200

Host Machines and System

- VAX/VMS Host Compiler is available now.
- Motorola 68000 Cross Compiler available third quarter of 1985.
- MIL-STD-1750 Cross Compiler available third quarter of 1985.
- Intel 286 Cross Compiler available first quarter of 1986.

Kernel ADA Environment on the Host Machine

Perspective/Ada is a Project Support Environment (PSE) for Ada. It incorporated a multi-user, multi-access database and provides support for the Project Manager, Designer, Programmer and the Maintenance team.

Features include:

- Project management facilities
- Design method support
- Change control
- Version control
- Configuration Management
- Host development in Ada
- Cross development in Ada

Text Editors

ADA Program Library Manager - Ada Library User System

ADA Compilers

Features include:

- Extensive compile-time diagnostics
- DIANA Ada Intermediate language
- Separate compilation
- Implementation of Representation clauses
- 16 and 32 bit integer defined types
- 3 levels of floating point precision

Run-Time Executives

- The run-time system is organized in two parts
- Tasking support provides the routines to control the synchronization and interactions between Ada tasks (target independent).
- Target dependent software, the run-time support necessary to provide heap management, exception handler identification, block data move, block data compare, low-level semaphores, context switching, etc.

ADA Symbolic Debuggers - Full host/target source-level debugger

Host to Target Exporters - Full support of Downline loading and source to memory mapping

Other Tools

- Pretty Printer
- Cross Reference Tool
- Name Expander Tool
- Configuration Management Tools - available through Perspective/Ada

TELESOFT

Company

TeleSoft
10639 Roselle Street
San Diego, California 92121
(619) 457-2700

Host Machines and Systems - VAX/VMS, VAX/Unix

Kernel Ada Environment on the Host Machine - Portable KAPSE interface

Text Editors - Uses host text editors

ADA Program Library Manager - Fully integrated with host file system

Ada Compilers

Run-time Executives

Ada Symbolic Debuggers

Host to Target Exporters - Embedded Systems Kit (ESK): Cross compilation from VAX, IBM 370, MC68000 and 8086 hosts to MC68000 embedded systems

Other Tools

- Computer Based Learning Materials
- Embedded Systems Development Kit

VERDIX ADA DEVELOPMENT SYSTEM (VADS)

Company

VERDIX Corporation
14130-A Sullyfield Circle
Chantilly, VA 22021
Phone: (703) 378-7600

GEC Software, Ltd.
132-135 Long Acre
London, England, WC2E 9AH
Phone: 01-240-7171

Host Machines and Systems/Status

- VAX 11/750 (UNIX 4.2 BSD) Validated 1.5
- VAX 11/785 (ULTRIX) Validated 1.5
- SUN 2/120 (UNIX 4.2 BSD) Validated 1.5
- VAX 11/780 (VMS) Spring 86
- SEQUENT BAL/8000 (UNIX 4.2 BSD) Spring 86
- CCI Power 6/32 (UNIX 4.2 BSD) Spring 86
- Tektronix 6130 (UNIX 4.2 BSD) Spring 86
- Harris H60, H700, H800, H1000, & H1200 (by Harris Corp) Summer 86

Cross Compilers

Host	Target
VAX 11/780 (VMS)& all UNIX 4.2 hosts	MC 68000 family Fall 86
VAX 11/780 (VMS)& all UNIX 4.2 hosts	MIL-STD 1750a Fall 86
VAX 11/780 (VMS)& all UNIX 4.2 hosts	Intel 80x86 Winter 86
VAX 11/780 (VMS)& all UNIX 4.2 hosts	NS32032 Winter 86

Kernel ADA Environment on the Host Machine

VADS is an integrated product providing a comprehensive compiler, symbolic debugger, Program Library Utilities and Runtime System in one package for self-targeted as well as cross-targeted embedded system application development. VADS was designed not to replace an existing programming environment. Rather, its tools complement and integrate with existing environments to allow the Ada user to concentrate on the Ada application and not on a new operating system.

Text Editors

VADS is integrated with the system editors on the UNIX and VAX/VMS based systems.

ADA Program Library Manager

VADS includes a set of integrated utilities to organize, manage, manipulate, and display Program Library information. The VERDIX Ada Program Library utilities provide for library creation and deletion; library dependency link creation, link removal, and cross-referencing. In addition, VADS permits Ada Program Libraries to be hierarchically organized, so that units not local to one library can be found in other libraries. Users can elect to separate the modifications of developers so that their changes do not interfere with others. Changes are then updated formally at which time the nature of the changes, who did them, when, etc., can be recorded.

VADS is designed to meet the needs of a distributed development environment, by supporting version control over not just one but MANY different distributed computers. A user can update a new version on one machine and then log onto other machines and have the changes automatically available there.

ADA Compilers

The VERDIX Ada Development System is centered on a high-performance, production quality Ada compiler which is fully compliant with ANSI/MIL-STD-1815A. The System includes the Compiler, Pre-Linker, Debugger, Program Library Utilities, and Runtime System.

The VERDIX Ada Compiler uses innovative syntactic error-recovery techniques to maximize the number of actual errors found per compilation while minimizing spurious error messages. Each message provides not only a concise description of the error, but directs the user to a specific paragraph of the Ada Language Reference Manual for a more detailed explanation.

The VADS compiler is designed to support full optimization both for space and for time. Local values are assigned locations by the register allocator and may be registers or memory locations. Global values are assigned memory locations. VADS uses a global graph-coloring register allocator with local priorities to accommodate small target register sets. A set of global optimization experts is applied to the low-level linear intermediate code repeatedly until no further change is found or until optimization time expires. On a typical 1 million instructions per second (MIPS) virtual memory computer system, the Compiler processes an average of 1,000 Ada source statements per minute, and utilizes previously compiled Ada specifications at the rate of 20,000 statements per minute.

Run-Time Executives

The Runtime System is designed to be tailorable and to handle special realtime, scheduling, and I/O requirements. The runtime system controls subsequent program execution and provides the principal interfaces to the operating system. The Runtime System provides comprehensive support for tasking, debugging, exception handling, and input/output. Retargetability is a prime requirement for the runtime system, and the interfaces have been designed to accommodate multiprocessing targets. The Runtime System also interfaces with the symbolic debugger to provide comprehensive program check-out facilities. Initially, the VADS for the MIL-STD-1750A is using the Hunter & Ready VRTX Operating System as the underlying kernel.

ADA Symbolic Debugger

The VADS debugger provides a fully symbolic debugging facility. The user can access Ada entities by name, and the debugger utilizes its understanding of Ada types to display appropriately formatted values, allowing other subtasks to continue processing. Breakpoints may be conditional based on the value of expressions relating to the running program. In addition, single-step execution is provided, optionally stepping over procedure invocations. The debugger provides access to the Ada source code during a debugging session. This facility is integrated with the breakpoint and stepping capabilities so the user has a source "window" into the program at all times. The user interface to the debugger will accommodate downline as well as host system debugging. The debugger uses the same separate compilation information produced and used by the compiler. The object code is thus free of symbolic information and is not modified for debugging purposes.

Host to Target Exporters

VADS will support the Tektronix 8540 down-load vehicle, through ICOM 40 software. The linked application image would be down-loaded using ICOM 40 software into the 8540 and then into the target machine. This operation is under the control of the debugger. The user interacts with the debugger in the normal way and uses the debugger as a controller. All of the normal debugger features are available to the cross-development user.

Other tools

Additionally, VADS provides a number of productivity and ease-of-use features as part of the standard VADS tool set. This includes VADS online help commands, an Ada source code formatter and a set of libraries covering standard Ada, VERDIX supported extensions and public domain packages.

CHAPTER 4

GENERAL LIMITATIONS OF CURRENT DESIGN SUPPORT ENVIRONMENTS

4.1 Introduction

While Chapter 3 was able to give extensive tables listing current software design support environments and noting the attributes of each, no attempt will be made to follow suit in Chapter 4. Appendix 3.1 shows that there is no environment in current use which makes any pretence to completeness, and hence the limitations of specific current environments are not relevant. The point at issue is the generic limitations which apply to a greater or lesser extent to all of the currently available software design support environments.

These limitations can be classified into a number of types as described below:

Limitations of Completeness

Appendix 3.1 lists sixty-nine software tools, including those solely applicable to project management and control and configuration management, and it divides the system development task into nine phases, excluding the parallel management tasks. Only six tools are applicable to more than six of these phases, and on average only three phases are covered.

Limitations of Integration

Currently there are a large number of tools covering individual parts of the life cycle. What is needed are a few sets of powerful methods and their related tools to cover the complete life cycle.

A well integrated tool set will retrieve the data for each phase of the development from the output of previous phases with the minimum of operator intervention, and it will permit the designer to work backwards and forwards in iterations of the development process with a minimum of constraint. Perhaps the design of such a well integrated tool set—and, in particular, the selection of default procedures to be more helpful than tiresome—is as much an art form as a technological problem. If so, it is unlikely that any such set of tools will please everybody. However, the current state of development in this respect is that no existing set of tools pleases anybody.

Limitations of Availability

An essential part of any design support environment is an operating system on the host machine. A high level of integration between the tool set and the operating system can give good efficiency, but militates against portability. On the other hand, if the tool set is superimposed on a generally used operating system, the portability is determined by the availability of the host operating system.

Limitations of Extensibility and Interchangeability

The specification, design and implementation of a major airborne software package is a lengthy process during the course of which there can be significant progress in tool development and availability. Generally, it has been the case that once a project has started using a defined environment, it may subsequently be difficult to change or to integrate additional tools into the environment. The availability of a basic tool set to which additional and more capable tools could be added as they are delivered would help in this regard. It is possible to define tool interface standards which would allow tools to be added or changed during the

software life cycle. A start has been made in this respect by the introduction—under the STARS (Software Technology for Adaptable and Reliable Systems) program—of the concept of “Information Interfaces” which are at a higher and more general level than any particular development environment. Although the need is quite likely to arise in the future, no work is known to have taken place on the interfacing of different development design support environments in cooperative projects, or on the transferability of a project to a more up to date design support environment.

The requirement of extensibility and interchangeability places strong demands on the design and structure of the database management system upon which the design support environment is based.

The design support environment should be able to support a number of high level languages and a number of different methods in multiple combinations in the same system. Each project would expect to define its own particular combination of languages and methods.

Limitations of Performance

The potential capabilities of a design support environment will not be realized in practice unless its performance is adequate for the application, both in terms of usability and of quality of the end product.

“Adequate” is not definable in the general case. What is adequate for some users may be grossly inadequate for others. But unless the speed of compilation, the timeliness and accessibility of the error messages, the size, speed, reliability and integrity of the object code, and all the other measures of usability and handiness are such as to ease the stress on the human parts of the design support environment, the environment will not be realizing its full potential.

These measures are not entirely covered by consideration of the ergonomics of the design of the environment. However, the human interface is a vital factor in determining the handiness and, hence, the potential benefits that the environment can provide.

For guidance and control applications, integrity and run-time speed and efficiency will always be of prime importance. Target architectures and run-time executives may need to be designed specifically for particular projects, and hence the environment must support the development and exercising of efficient and dependable run-time executives on a wide variety of target architectures.

Limitations of Application

The foregoing limitations are all of a general nature, and will relate to any user. There are also limitations which, although they do not affect every user, will nevertheless be of sufficiently widespread influence to warrant inclusion in a general discussion. For example, there is currently no programming support environment explicitly designed for use in the development of safety critical systems.

For flight applications where integrity is of paramount importance, it is common practice to use restricted instruction sets, to place even more emphasis than usual on the use of small and simple modules, and to give extra attention to module and system testing. At the moment, little or no safety critical software is written in a high order language. If the use of a high order language is to improve integrity, special attention must be given to all of the programming tools in the environment. Currently, in safety critical applications only the simplest of tools are used because these can be adequately validated. If wider use is to be made of tools, this will place greater demands on validation procedures for those tools.

The defense user may also be concerned with considerations of security to a greater degree than the average user. As yet, and very properly so, security considerations have not taken a very large part in the thinking of programming environment designers. In due course there will be users undertaking multi-site and even multi-national development of large military systems who will need their system development environment to provide multi-level security to the most rigorous military and commercial standards.

It is unrealistic to suppose that all of these limitations will be swept away with one master stroke; the improvements are much more likely to come about by an evolutionary development process.

The remainder of this chapter charts the routes by which this evolution may come about, and discusses the

considerations which must be borne in mind during the process.

4.2 The Interface between Machine and the Environment - The Operating System

Even the smallest of modern general purpose computers is provided with an operating system. In general the operating system is unique to a particular machine or range of machines. It gives the means of controlling the machine(s), and may be regarded as the fundamental support environment.

Such operating systems typically provide means to handle text and to produce executable code for the machine which hosts the operating system. However, little support is provided for target software development, nor does a vendor-supplied system normally handle configuration control procedures well.

A successful development support environment will undoubtedly be required on a variety of different host machines, but it may be uneconomic to rewrite every operating system from scratch. Therefore the developer of a more comprehensive support environment has two alternatives:

- To base his toolset on an existing vendor-supplied operating system, superimposed by an integrating layer which presents an adaptable interface to the tools which can be modified to fit several different operating systems. This is a direct implementation of the Stoneman philosophy.
- To base his toolset on a more widely available operating system, where it becomes practical to accept a direct integration between tools using the operating system as an integration mechanism.

The first of these alternatives can be arranged to be independent of an operating system to a considerable degree, but may pay for its ease of transportability by a concession to efficiency. The second must be based on a generally available operating system, such as UNIX. The level of integration between the tools at the operating system level may be greater, but in this case a loss of efficiency may arise from the generality of the base operating system.

In the future it may become commercially viable to design an operating system to have an outer tool integration layer superimposed upon it without the necessity to perform in a stand-alone mode. For example, if CAIS and PCTE achieve the status of widely accepted standards, this approach may offer a higher potential efficiency than the other approaches discussed above.

4.3 The Development Process

The real-time software development process involves many phases, special design and implementation problems, extremely stringent requirements for verification and validation, and several kinds of development control. The limitations of current support environments with regard to these considerations are discussed below.

4.3.1 The Phases of Real-Time Software Development

The phases through which a software system development passes are fairly well understood, although they are not always expressed in exactly the same way. Chapter 2, Section 2.3.1 discusses this in some detail.

The reason that this breakdown is not unique is that the development process is, in reality, almost continuous, and the steps between phases are nothing like as clear cut as it is necessary to suggest in order to illustrate the underlying systematic structure. Moreover, most descriptions of the phases of software development do not take account of the fact that the design process is one of continuous iteration, sometimes looping back within a phase, sometimes refining a phase, and sometimes going back to adjust the work of several of the previous phases.

Although simplified breakdowns of the design process can usefully neglect both the quasi-continuous nature and the iterative nature of the process, it is vitally important that the design support environment neglect neither of these factors.

Both factors place considerable strain on the configuration control function for a complex system development, and it is for this reason that configuration control has been in the past, and still is today, a notoriously fault prone aspect of system development.

The contents of Appendix 3.1 have been analyzed in the histogram of Figure 4.1 which indicates that the tools currently available are concentrated mainly in serving the software design and coding phases. This is not because these phases are the most important—all phases of the development are equally important—but because these phases were the most labor intensive and error prone. The provision of tools in this part of the life cycle has allowed the development of larger and more complex systems, with the result that other life cycle phases now reveal major problems.

For the purpose of discussion of the functional structure of the development environment, it is instructive to distinguish carefully between the more-or-less sequential phases through which the system development must pass and the development functions which must be exercised to some degree throughout the entire development process.

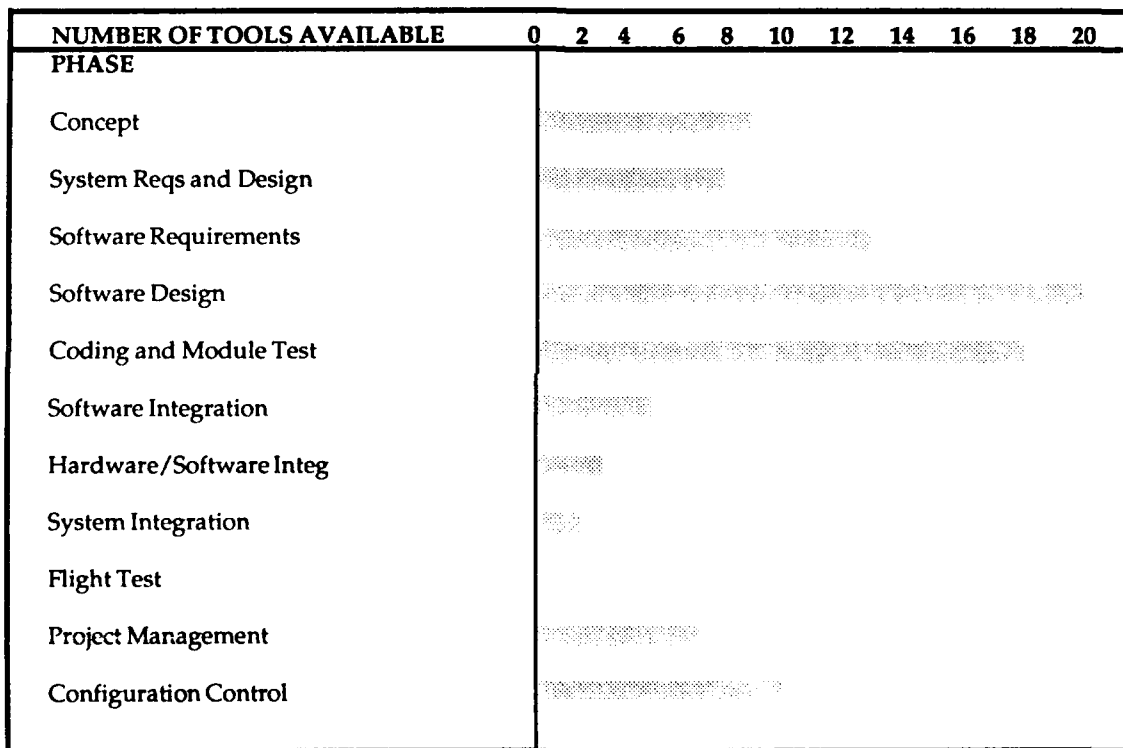


FIGURE 4.1

NUMBER OF TOOLS AVAILABLE BY PHASE

4.3.2 Requirements, Design, and Implementation

An unambiguous formalized method of expressing software requirements is needed. The method must be usable and understandable to both system and software engineers and to management so that it becomes a tool of the trade while still being rigorous. There have been developments in this area, e.g., CORE, but as yet there has been no wide acceptance of these formal methods. The possibility of automatically producing code from the expressed requirement has some attraction, particularly since the penalties in code size are likely to

be offset by advances in computing technology. Perhaps in the longer term new approaches to automatic code production may reduce the overhead.

A methodology for real-time executives is necessary if they are to be logical, understandable, easily modified, and efficiently produced. The development of MASCOT methodology and associated tools like Context have helped to fulfill this requirement. It is necessary that MASCOT or an equivalent methodology be developed so as to make it compatible with the facilities that ADA provides. A common real time executive, supported by an appropriate design methodology which can be used on many projects, will clearly result in savings in both software production and postdevelopment support, as well as in improved software integrity.

A major activity in real-time systems is the provision of an operator interface. An enormous amount of effort has to be expended in designing display formats and keying sequences and in their subsequent software implementation. There is a need for tools to increase the efficiency of the production of display and control software and its subsequent modification. Tools that automatically produce the source code from a relatively high level specification of the display and control requirement are necessary, even though they may have to be hardware dependent. Such tools should also provide an analysis in terms of detailed keying sequences so that the software engineer can understand what has been done and verify that it is correct.

4.3.3 Verification and Validation

Verification is common to every phase of the development process. Because there are no mechanistic test methods available for the early phases of the design process, verification at these points takes the form of "design reviews" or "illustrated walk-throughs". As used here verification means taking whatever is known of the design so far and comparing it with what is known of the requirement.

Verification and validation have traditionally relied on testing, which has involved the software designer in detailed manual work required to develop test harnesses and test parameters for each software module and each collection of modules. The success of this testing depends largely on the engineer's expertise and insight for answering such questions as: "Did the test give sufficient coverage?" and "Were the tests correct?". There are a number of tools (or tool functions) which would greatly assist in this area.

- A package to analyze source code, perhaps as a built-in function of an advanced compiler. This should address both self-consistency and consistency with the requirements.
- A tool to develop test requirements by direct evaluation of the software. This could be used in conjunction with a tool to monitor actual test coverage in an iterative process, the result of which would be a fully tested software package.
- A method of checking target code for data sensitivity. This could be particularly useful in checking for overflow and underflow in fixed point machines. It would determine the acceptable range of input parameters and check these against the requirements specification to ensure that the software response to out of limit data was tested.
- Current methods of real time debugging and tracing are slow and difficult, and generally mean that the system under test is not running in real time. Micro-processor development systems use hardware emulation and monitoring techniques for debugging and trace, but they are not user friendly. There seems to be scope for new development tools which use micro-processor development hardware techniques in combination with a powerful software package in the host computer to provide a more comprehensive and useful real-time debugging tool.
- Failure Mode Effect Analysis (FMEA), an established technique in hardware, is a more difficult concept in software since software failure is a much less well defined notion. There is certainly a requirement for system FMEA's, and a tool which could analyze software effects resulting from hardware failure modes could be of benefit. Software failures are essentially design errors. However, a tool which could predict the result of postulated errors would also be useful.

Most systems provide some form of symbolic debugging tool, but none of them are as powerful as the language in which the program was written. In principle, one would want to be able to communicate with the program through the debugging tool in the program implementation language. In particular, it would

be desirable to be able to specify actions while the program under test is halted at a breakpoint. Such actions should be defined in the same language using the visible identifiers of the program.

For debugging real-time systems, it is often essential that the debug facility leave timing relations unaffected. Many of the problems occurring in such systems are due to timing problems, and observations of the program behavior should change timing relationships as little as possible. Little effort has been devoted to general ways of achieving this goal. One approach is through specialized, dedicated hardware that allows inspection of the inner workings of a program. Such equipment is so strongly dependent on the target computer that there is very little chance of achieving commonality in the tools used. Nevertheless, establishment of principles on which to base such equipment would certainly help.

4.3.4 Development Control

It is useful to consider the addition of the following four development control functions to the usual phases of system and software development:

- Management control
- Quality control
- Configuration control
- Documentation

These functions are of quite a different nature from those of the quasi-sequential development phases. In a well managed development, they are all active to some degree from the start to the finish. Moreover, most of these functions make some contribution to the deliverable parts of the system. For example, handbooks, test reports, etc. used during the development phase can also be deliverables. They are all, however, essential controls, and a lapse in any one at any time during the development could have far reaching consequences.

Management control comprises a continued reiteration of three functions:

- Task estimation
- Task allocation
- Task monitoring

Estimation is too often regarded only as a magical incantation or initiation rite necessary to launch a project. In fact, just as the design process is essentially iterative, so should estimation also be performed iteratively. Without this activity, project management is flying blind. Although there are tools available which can help in assigning man hours and costs to a particular amount of code, use of such tools begs the question to a certain extent. However, developing accurate estimates of the size and execution time for real-time software has proved notoriously difficult and development of better methods/tools is necessary. It is important that the estimates be accurate enough to allow the selection of a machine with proper capabilities. A low estimate results in the need for code optimization, expansion of the target hardware or even a new target machine, all of which are very costly exercises.

Quality control is a function of project management whose purpose is to ensure that the end product meets the user's/customer's quality requirements. This control will normally be exercised by verifying that an acceptable set of software engineering codes of practice or standards are being adhered to. Quality assurance consists of a planned and systematic series of actions necessary to provide adequate confidence that the quality control procedures are working properly. No tools are widely available to assist these functions, but there appears to be some possibility for computer assistance in these areas.

Configuration control is made particularly difficult by the necessarily iterative nature of the development process. It is especially prone to the more insidious forms of human error, the sins of omission, the careless slips, and lapses of memory. Good configuration control is the key to successful integration. It should be noted that configuration control must apply to the system hardware as well as to the system software, which

exacerbates the problem. It is not easy to see how a completely foolproof configuration control system could be developed.

Software configuration control during the specification, design and in-service phases of real-time software development is also a vital ingredient for software quality control and project management. A number of tools are available, but they only cover part of the requirement. Also, new and more comprehensive tools are coming into the market, but their capabilities have yet to be proven. The tools need to be able to capture the software engineers' knowledge, in terms of what happened, when it happened, and why it happened. This will ensure that a full and detailed history of the project is available throughout its life cycle. Integrated management tools of these kinds covering all phases of the life cycle are needed.

Documentation is part of the delivered product as well as being a development function. But, since it can play a vital role in communicating design changes during the development process, it is best considered as part of the software development process. This important activity is very often regarded as necessary but unpleasant and costly. Good documentation support tools could stimulate software developers to produce useful documentation early enough to make a contribution to the design process and help keep associated costs acceptable.

It is necessary to provide detailed and comprehensive documentation for software systems so that they can be supported during their in-service life. Often the documentation must be to specific standards defined by the customer. Documentation can require a significant percentage of the total effort needed to produce the software. If increases in efficiency in software production are to be obtained, then the area of documentation must be attacked. Tools are required that enable documentation to be produced in a far more advanced way as part of the software design, coding, and testing process. The tools should produce the documentation to the standards required by the customer to which end the widespread adoption of a common documentation standard would be highly desirable.

Current documentation practices give rise to large amounts of paperwork which are costly to produce and maintain and which make it difficult to access the information contained therein. Consideration needs to be given to computer-based information retrieval systems for use in the development and support of documentation.

4.3.5 A Development Process Overview

The development phases and the project management functions together constitute the sum total of the activities which must be exercised during the development process, the floor plan, as it were, of the system development space. The system bricks which are built up on this floor plan are the actual elements of the system itself. In detail, these are different for every system, but it is possible to make some generalizations.

The application will usually allow a fairly natural decomposition into separable functions, and associated with each of these functions will be a set of algorithms which determine the transformations between a set of data structures. These elements of the design can be expected to be defined or almost defined, in the software requirement specifications. To these elements the software designer must add further design elements. Associated with each function or, perhaps more accurately, with each pair of functions must be an inter-functional interface which will comprise message structures and control structures. Overall there must be added the control elements, i.e., the message access handling and the control flow.

These elements are the heart of a design. Their correctness ensures the successful functioning of the system in service. However, they do nothing whatever for the control of the development process itself, and without adequate control there can be little chance of achievement of correct design elements.

The complete breakdown of the entire system development process is encapsulated in Figure 4.2. The floor plan represents the activities which have to be carried out: Along one axis the quasi sequential design phases are listed, and along the other the continuous and unceasing design control functions are listed. Adding depth in the vertical plane are the actual elements of the delivered system.

The system development gradually "fills up" this space, advancing generally along the design phase axis, but with frequent iterations back over completed work for revision, and occasional leaps forward where, for example, an opportunity is seen to reuse some existing code or design. This is the way that system designers

work, and an environment which does not acknowledge such methods or support them will be found wanting at least in usability and handiness, if not in more fundamental ways.

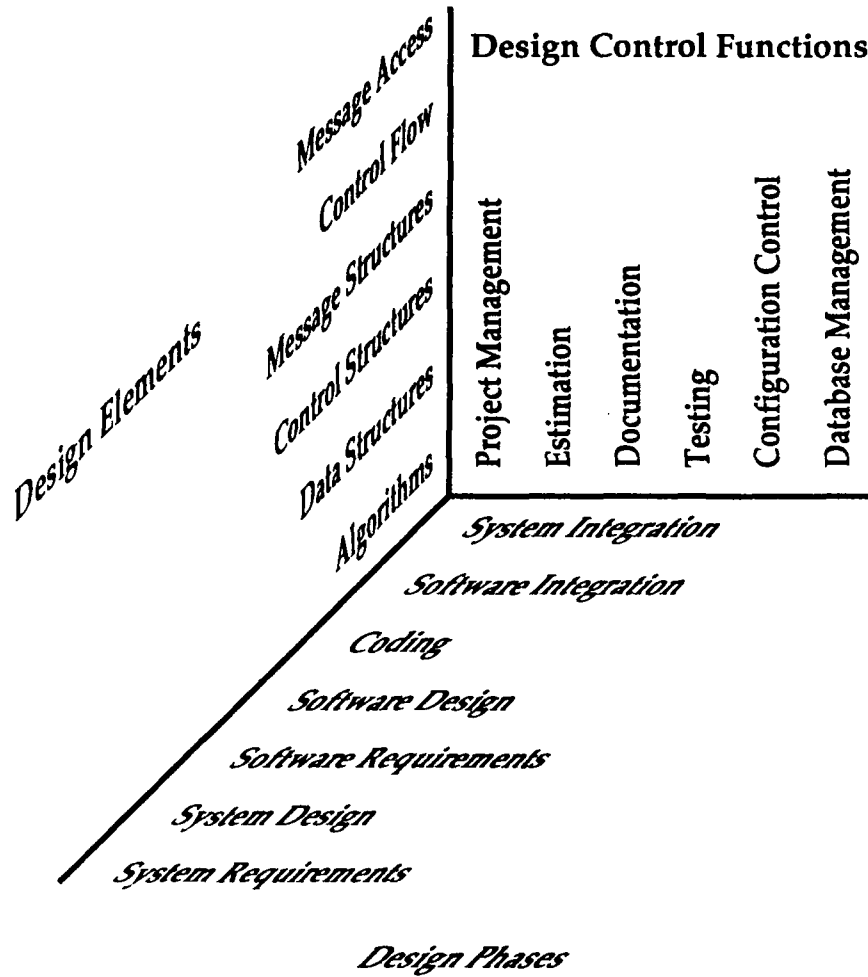


FIGURE 4.2
SYSTEM DEVELOPMENT SPACE

CHAPTER 5

REQUIREMENTS FOR SOFTWARE ENGINEERING ENVIRONMENTS AND TOOLS

5.1 Introduction

Chapter 3 gave an overview of currently available software engineering environments and tools, and Chapter 4 pointed out some of their limitations. The present chapter is an attempt to establish requirements for a software engineering environment providing tools to cover the entire software life cycle reasonably well. By "reasonably well" is meant an environment that is realistically achievable in a few years time (say before 1990) as opposed to an ideal or mythical environment. This means that requirements for the environment must be already fulfilled by some existing tools or at least be the goal of some research and development project already in progress.

Section 5.2 presents some requirements which are specific to the guidance and control domain. Section 5.3 states the need for an "integrated" engineering software environment (as the terms of reference for this Working Group suggest), defines what is meant by "integrated", and gives consequent requirements for the kernel for such an environment. Section 5.4 gives a few requirements which apply to the whole set of tools in the environment. Each of the remaining sections deals with tools supporting a specific phase of the software development cycle with the last section devoted to general support tools.

5.2 The Guidance and Control Environment

In considering the software engineering environment for guidance and control systems, it is necessary to consider an environment which has to cope with a wide variety of requirements. At one end of the scale are high integrity, safety critical systems and at the other end are extremely large and complex mission systems. However, the design techniques used in flight critical systems are very different from those of mission critical systems, and a much more restricted approach to software design must be taken because of the safety critical parameters.

The ideal guidance and control software engineering environment would be based upon an integrated data base and would contain a compatible tool set usable throughout the software life cycle. The environment should also be extensible and allow for incremental enhancements as new tools become available. The overall environment should have a rigorous and robust configuration control tool incorporated into it which can provide traceability of the final code through its previous versions back to the original design requirements. The environment should enforce standards established for use within the project and prevent the use of non-standard design methods. The environment should also contain documentation tools capable of handling full text and graphics in an integrated manner.

5.3 What Makes a Software Engineering Environment "Integrated"

A properly integrated software engineering environment must support the whole software life cycle and must offer to its users:

- The possibility of easily tracing the difference between intermediate products resulting from the different phases of software development.
- A uniform command language and user interface to all tools included in the environment.

- The possibility of adding new tools to support the evolution of methodologies or the need to address new target machines without losing the characteristics listed above. Such a possibility is greatly eased if the kernel environment provides convenient standard facilities for the tools to communicate with their environment (other tools and users).
- Extensibility in power (e.g. number of users, number and size of supported tools).

These are the main characteristics of an "integrated" software engineering environment advocated by CAIS and PCTE. Technically, to obtain these characteristics it is necessary to define a "kernel" environment which is a software layer located between the (classical) host operating system and the tools of the environment.

Also, the last characteristic (extensibility in power) is much more easily reached if the environment can be distributed on a local network connecting individual work stations to each other and to common servers, with each work station being able to offer a significant individual processing power and memory storage.

5.3.1 The Kernel Environment

The kernel environment should include:

- An Object Management System (OMS) built on a data base giving adequate support for an attributed entity relation model. It should be possible to distribute this OMS on different work stations or servers and to hide this distribution from the end users.
- Program execution and I/O primitives.
- Protection mechanisms.
- A uniform command language to communicate with the tools in the environment. This command language should also allow the users to interface with the host operating system. That is, the services of the host operating system should remain available to the end users provided they do not use it to attempt to destroy the integrity of the tools and the OMS.
- Program (tool) communication facilities and standard user interface specifications.

Appendix 5.1 summarizes the PCTE view of these elements. The PCTE view appears to match the needs of guidance and control systems well. However, guidance and control systems design has more specific needs than the PCTE user interface provides and they are detailed below.

5.3.2 User Interface Requirements

A good user interface enables the user to view multiple sources of information on the screen of his workstation at any moment of his choosing. Such a user interface can be formed by objects which are windows, icons, and tool representations. The following definitions are proposed in order to specify these desirable user interface operations a little further:

- **User Interface Processes:** There are two kinds of user interface processes:
 - **User agent processes:** The functionalities of user agent processes include the display and management of windows, the control of input from the keyboard and pointing devices, and the management of commands and menus.
 - **Applications processes:** In the context of software engineering environments, application processes represent tools available to the users.
- **Frame:** A frame is a virtual device that any process can use immediately for its output. It contains the logical description and representation of a virtual terminal session.
- **Viewport:** A viewport is the portion of a frame which is to become visible in a window. Several viewports may be established on one frame.

- **Window:** A window is a rectangular area of an actual display device which serves as the physical device from which viewports may be visualized. A window may contain more than one viewport, in which case the viewports are then placed vertically on top of each other with no overlapping.
- **Cursor:** The cursor is the visible symbol on the screen that reflects the position of the pointing device.
- **Caret:** The caret is the position where, for each application, the next input from the keyboard or from another application is to be inserted.
- **Current Application:** The current application is the application which directly interacts with the user through one of its open windows. An application may have several open windows.

Beyond the above definitions which specify desirable user interface requirements and/or features, the user interface should also include the following operations:

- **Command Language Operations:** A uniform command language should be provided for all tools and used to start/terminate an application or interactively send commands to an application. The objects which constitute the parameters of commands should normally be inserted in the caret. However, the following alternate command mechanisms also should be provided to speed up the input of commands to an application:
 - It should be possible to start typing a command and ask the system to semantically complete it with the full name and default parameters in the caret. This leaves the user only with the task of replacing irrelevant default parameters.
 - It should be possible, in some cases, to designate objects as parameters for a command by the cursor. The user should have the capability to map commonly used commands to available function keys or to short strings of characters.
 - It should be possible to use the last two command mechanisms in conjunction.
- **Basic Editing:** The user interface should have a structural view of the object being edited rather than considering it as "flat" text (CONC 86.1, CONC 86.2, RAT 85). The basic functionalities provided by each frame definition should include:
 - The ability to select an object at the cursor position. Thanks to the user interface's view of structured objects, it will know where to stop selection on both sides of the cursor position.
 - The ability to extend such a selection to a more general item.
 - The ability to copy a selection to the caret position of the current application.
 - The ability to delete the selected object.
 - The ability to move the selected object.
- **Menu Management (NIEV 85):** The user should be able to view where he is in the system and where he can go from his current position. He should also be able to view the current object he is working with and to extend this view in both directions. That is, he should be able to expand the view of the current object or go up to a more global view of the object or to the encompassing object.

5.4 General Requirements for Software Engineering Tools

Certain general requirements for software engineering tools can be defined. For example, such tools must be integrable into an integrated software engineering environment as defined in the previous section.

Software engineering tools also must be able to use and generate permanent objects in the OMS as well as use and generate displayed, printed, and/or temporary objects. Moreover, the activation of software engineering tools must generate histories of the use of permanent objects or parts of permanent objects with sufficient detail to be able to replay previous work sessions using these objects. Finally, on-line documentation and help facilities must be provided for each tool, and the on-line help facilities must be driven by the current state of use of the tools.

5.5 System Requirements Tools

System requirements are derived from the results of the study phase and the concept phase of the system development cycle as described Chapter 2. System requirements have two main facets; namely, functionality and constraints.

In order to support the development of system requirements, system requirements tools should incorporate at least the following capabilities:

- A good means for representing system decomposition into subsystems.
- An effective means for verifying the coherence of the system decomposition, of the data flow between subsystems, and of subsystem constraints. For each subsystem, the system requirements tools should provide:
 - A means for description of input and output data together with the relations that may have to hold between them.
 - A means for easily expressing complete descriptions of the functionality of the subsystems.
 - A means for formally describing the constraints that apply (environment, implementation, time, safety constraints) to the subsystems.

Different methods can be used to support the development of system requirements. However, additional and, perhaps, different tools may have to be provided for the different methods in order to cover all the above-mentioned capabilities. Nevertheless, the tools provided in a given environment should be able to enforce a unique, project selected, system requirements development method.

There is not a great deal of experience in using system requirements tools except in a few organizations such as Teledyne Brown's experience with IORL/TAGS and the US Army Ballistic Missile Defense Advanced Technology Center's (BMDATC) and TRW's experience with DCDS (see Appendices 3.1 and 3.2). However, many different methods and languages have been developed to support the system requirements phase. A good overview of these may be found in (GOLD 85).

5.6 Software Requirements and Prototyping Tools

In addition to systems requirements tools, there is a need for software requirements tools. Moreover, prototyping tools are required to support the software requirements process as described below.

5.6.1 Software Requirements Tools

A set of tools should be provided which enable the system designer to take the system requirements and produce the initial software requirements. A number of potentially useful methods exist for the software requirements phase and for the subsequent software design phases, and it is essential that the environment provide flexibility to support tools for these different methods. Also, it is desirable at this level that both graphical and textual descriptions of the system be possible, and that provision be made for the incorporation of formal methods as they become available.

5.6.2 Prototyping

Before proceeding to the actual software development process, it is often necessary to gain assurance that the functional part of the requirements are sound and actually reflect what the users who established them intended. The best way to fulfill this need is to build a prototype based on the software requirements which will be able to demonstrate properties and behavior of the software to be developed.

The cost of developing such prototype should represent only a small percentage of the total expected software development cost (10-20%), and the prototype should aim at rapidly implementing, maybe only partially, the functional part of the software requirements with no consideration to their constraint part. Different kinds of prototyping techniques and tools may be used.

5.7 Basic Software Design Tools

At the software design level, a schematic capture scheme is suitable for capturing the software design if a schematic design approach is adopted. However, there should be a textual representation tied to this graphical representation, and there should be a one-to-one correspondence between the graphical and textual approach. This is analagous to hardware design where a schematic representation is backed by a cell library describing the logic of the devices and a net list describing their interconnections. A number of organizations are working on such systems. For example, Advanced Systems Architectures in the UK have a compatible schematic and textual language. It should be possible to move freely between text and graphics so that different phases of the design can be conducted in the most appropriate form.

In general, the composition of the design should be carried out on screen, and it should be possible to undertake hierarchic decomposition in this mode. The software components thus defined should be ultimately decomposable into ADA packages or LTR3 modules or equivalent structures if other languages are used.

At the basic software design stage, tools are also needed to specify the dynamic semantics of these packages or modules. Unfortunately, ADA and LTR3 provide only syntax and static semantics for their specification.

The particular tools that are required are dependent on the specific design methods used and on this issue there is no general agreement. Therefore, it is not useful to try to elaborate here on particular tools. Instead, some examples of such methods/tools are provided as appendices to this chapter.

5.8 Detailed Software Design Tools

Tools are needed to help in transforming the rather declarative graphical and textual output of the basic software design phase into an imperative, implementation oriented form during the detailed software design phase. Again these tools are methodology dependent and examples are given in appendixes of Chapter 3 for different methodologies.

5.9 Software Implementation Tools

There are many types of software implementation tools. They are all very important and each is discussed in turn in the following subsections.

5.9.1 Program-Constructor Tools

If the detailed design phase is carried out as advocated in Section 5.8, it will produce some mixture of formal, non-compilable constructs and fragments of compilable high level language (HLL) code. The formal non-compilable constructs will be semantically equivalent to HLL constructs and the programming phase will

therefore be very mechanical. Nevertheless, it may be opportune to leave some choice in style during this phase in order to improve readability and efficiency of the HLL code.

Therefore, it is recommended that a program-constructor tool be used during this programming phase which will produce HLL code from the detailed design documents interactively with a human programmer. All of the mechanical work should be automatically done by an appropriate tool which will only ask the programmer's help to resolve relevant choices. An example of program construction rules that can form the backbone of such a program-constructor tool can be found in (bOIND 31).

5.9.2 Syntactic Editor

A language specific editor should be provided so that the system only accepts syntactically correct statements. Once again, a number of companies have such products in development and some are now becoming available.

5.9.3 Compilers

Compilers are obviously essential tools for the software implementation phase. To be of maximum value, they

- Must be validated.
- Should implement the whole set of representation clauses.
- Should implement all the predefined ADA pragmas which have an influence on code generation. LTR3 compilers should incorporate equivalent compilation directives.
- Should support separate execution of parsing, semantic analysis, and code generation.
- Should never attempt to use any external compilation units during separate compilations which are not coherent.
- Should have the capability for performing "incremental compilation"; that is,
 - For compiling a program fragment or an incomplete program unit (which is "semantically complemented" as a result of the operation).
 - For including new items in a program unit A without having to recompile the whole of A and the other program units that use A.
 - For suppressing items in a program unit A without having to recompile the whole of A and the other program units outside A that do not use the suppressed items (even if they use the remaining part of A).
- Should produce:
 - Compilation listings with very explicit error diagnoses pointing to actual error locations.
 - Intermediate language objects constituting good source objects for generating code to any target machine. These intermediate language objects should retain most of the semantics of the high level language and be machine architecture independent. Such intermediate language objects should provide a good interface with other tools; i.e., their representation should have an attributed tree structure which could be manipulated by a general attributed tree handling facility and be reusable by other tools.
 - Symbol tables.
 - Memory mapping.

- Assembly symbolic code with convenient references to the high level language source code and the memory mapping.
- Conveniently optimized binary code (some elements of methodology to evaluate its efficiency may be found in BAFIGA 85).
- Should provide formal description of the intermediate language and the interface with the run-time executives.
- Should provide decompilers together with compilers in order to be able to reconstitute formatted source text from intermediate language objects. The presence of such decompilers avoids keeping source text and source text handling except in the initial editing phase.

5.9.4 Program Library Manager (PLM)

The PLM should support creating new objects in a library by the compiler and link editor and should automatically link these objects with:

- Other objects related to the same compilation unit where a compilation unit consists of the Source Code, the I.L. Object, the symbol Tables, and the Binary Code.
- Other compilation units which are used by or encompassed within the newly created Object.

The PLM also should support the creation of multiple libraries and provide adequate protection mechanism for concurrent use of them.

The compiler/link editor should be able to retrieve objects from libraries other than that within which it generates objects. The compiler and link editor also should be the main tools interfacing with the PLM, but the PLM should facilitate interfacing with other tools (debugger, pretty printer ...).

Additionally, the PLM should allow the user to:

- Create/delete libraries.
- Inspect the contents of a library and trace its internal links.
- Delete objects in a library.

Finally, the PLM should prevent any tool or user interfacing with it from performing an operation which will destroy the consistency of a library (leave a pending link in the library).

The requirements discussed above are intended to characterize a minimal program library manager. More details can be found in (RIP 84).

Much more elaborated library managers can be developed which will provide support for:

- Configuration management (see Section 5.10).
- Reusing software components included in the libraries. In this case, support is needed to retrieve components matching one's needs from the library. Retrieval of reusable components can be based on:
 - Using their semantic specification. Some interesting R and D works are in progress in this area (GO 86).
 - Establishing a taxonomy of components based on their functionality, their implementation characteristics (when relevant to the users), and the constraints they impose on their users (BOOCH 86).

5.9.5 Predefined Package/Module Libraries

Beyond what is specified in the high level language reference manual, a software implementation tool set should:

- Contain a comprehensive mathematical library.
- Have good interfaces to the operating systems when relevant.
- Provide graphic input/output.
- Provide a comprehensive interface to the assembly symbolic language.

5.9.6 Run-Time-Executives (RTE)

The RTE provides an interface between the application program and the operating system or the bare machine architecture *when no operating system is provided*. The main functions for which such a run-time interface is required are:

- Dynamic memory management
- Exceptions
- Tasking
- High Level Input/output
- Interrupt handling

Timing

Some programming languages (e.g., ADA, LTR3, LTR2, and PEARL) incorporate a large part of the RTE functionality as built-in language facilities. Other languages (CORAL, JOVIAL ...) specify only dynamic memory management and I/O facilities and leave the implementor free to provide whatever facilities he needs to implement the other RTE functions which are listed above. Also standard RTEs have been given an interface in some of these languages (e.g., the MASCOT RTE interfaced in CORAL).

There is currently (in 1986) no agreement in the software community on the general specification of such RTE facilities, and this remains true even in the restricted domain of flight guidance and control systems. However, it is likely that the use of ADA and LTR3 in the near future will focus much more attention on this issue and will enforce at least the following common requirements on RTE specifications:

Dynamic memory management

- Dynamic data should be strongly typed. Strong-typing should be relaxed only under exceptional circumstances by using specific features, the access to which is under the control of the technical management.
- It should be possible to allocate dynamic data of any type.
- Safe, explicit deallocation is too costly to implement. Both ADA and LTR3 provide an unsafe explicit deallocation, the use of which is discouraged. Of course, the RTE is always free to deallocate dynamic data when such data can no longer be reached by the program for visibility reasons.

Exceptions

- An exception mechanism should be provided to flag run-time error conditions and to enable the programmer to factorize the handling of an exceptional run time condition.

- Both predefined and user-defined exceptions should be provided.
- As a result of an exception being raised in a program unit by the RTE or by in-line code, execution of the program unit should be terminated and a special sequence, called an "exception handler", should be executed if such handler is provided. If an exception is raised in a task which provides no handler for it, then this task should be immediately terminated.

Tasking

- The RTE should provide facilities to explicitly activate concurrent tasks during program execution, together with tools enabling such concurrent tasks to synchronize and communicate. It is not enough to provide facilities to activate concurrent tasks only when program execution is started.
- It should be possible to prioritize tasks and use preemptive scheduling to enforce the assigned priority of execution of the tasks. That is, no task should be granted resources for execution while another higher priority task is eligible and not granted resources for execution.
- The RTE scheduling should enforce the "finite delay property". That is, no task or branch of a task should remain indefinitely eligible but never executed.

High level input/output

- The RTE should provide an interface to a file management system and make files visible to the high level language programmer through "logical files".
- The RTE should provide the capability to map physical external devices and peripheral equipments to logical files. The RTE should insure that when an input/output operation takes a significant amount of time, it should be "non-blocking"; i.e., the task requesting the I/O should be put in a waiting position by the RTE until the I/O operation is completed in order to avoid the possibility that the task will retain resources while other tasks are eligible but short of resources. This requirement is not necessary in order to legally comply with ADA or LTR3 but it is mandatory for nearly all multi-task applications making use of high level I/O facilities.

Interrupt Handling

- It should be possible to attach tasks to external events (interrupts). In order to be able to take "immediate actions" upon the arrival of such interrupts, the task attached to an interrupt should be granted a priority which is higher than that of any task explicitly activated by the program.

Timing

- The RTE should provide an interface to read the real-time clock.
- It should be possible for a task to explicitly wait for at least a given amount of time.
- When a task initiates a blocking synchronization/communication action, it should be able to put a time-out on it in order to become eligible again when the time-out has elapsed even if the synchronization/communication action could not be performed.

The RTE for the Safety Critical Systems

- In general, every modern RTE should incorporate the above described facilities. However, it may be dangerous to use some of these facilities in safety critical systems. For example, in areas such as flight guidance and control, it may be necessary to have a run time system which is essentially a simple scheduler which schedules tasks on a strict time sequence. Such a scheduler can be a reduced and secure version of the more general purpose RTE described above, which is obtained by excluding those features having an unacceptably high or unreliable response time.

5.10 Validation and Test Tools

Validation and testing are extremely important activities in guidance and control system software development. Some requirements for tools to support these activities are discussed below.

5.10.1 Symbolic (High Level Language) Debugger

- A symbolic debugger should be available and it should provide an interface to the user at the level of the programming language (ADA, LTR3...) and not the machine code.
- Symbolic debugging should be possible both when executing a program on the host machine and when executing a program on the target machine.
- Symbolic debugging of a program executing on the target machine may require that a part of the debugger execute on the host and monitor and interpret program execution on the target machine.
- The symbolic debugger must be able to work both interactively (the operator issues a command to the debugger from his terminal and the debugger displays data on the terminal or on the printer) and in "batch mode" (the debugger executes commands given in input files and provides data into output files). When working interactively, the symbolic debugger should make full use of the multi-window capability of the user terminal.
- The general functions that the symbolic debugger should provide are the following:
 - Setting and removing "breakpoints", on different kinds of events. (A "breakpoint" is a program execution stop in order to perform debugging actions.) Events on which a breakpoint may be set should at least include:
 - Execution of a given statement based on its position.
 - Raising of an exception.
 - Occurrence of asynchronous interrupts from the operator or from external devices.

In addition to setting breakpoints based on the above "static" events, newer "event driven" symbolic debuggers can set breakpoints based on more dynamic kinds of events (MAU 85) such as:

- Realization of an arbitrary relation expressed in the high level programming language.
- Modification (assignment, destruction) of an object in the program.
- Nth execution of a loop.
- Execution of a given class of statement with given parameters.
- Event expressions built with the above elementary events and the following operators:
 - AND : Simultaneous conjunction of events
 - OR : Union of events
 - THEN : Sequential occurrence
 - TIMES : Repetition
 - IN : A given event in a given context

- Performing the following actions on a breakpoint:
 - Display
 - Values of objects and their attributes
 - Task status
 - Inspect the program status:
 - Current statements
 - Call stacks
 - Modify:
 - The value of a variable.
 - The priority of a task.
- Inspect traces and histories.
- Validate/unvalidate traces and histories.
- Set/remove breakpoints.
- Resume execution (unconditionally or step by step).
- Terminate the debugging session.
- "Fork" to other environment tools without leaving the debugging session.
- Tracing statement execution
- Building histories of program objects and task status.

5.10.2 Concurrency Property Analyser

Concurrency is an important aspect of an increasing number of real-time software systems. Being able to analyze its properties is important as further described below.

Static Analysis

Some properties of general interest, such as liveness and safeness, can be analyzed in many cases. Furthermore, it may be possible to systematically compute all the invariants on the states of a system. This allows one to try to match these invariants with related desirable properties extracted from the specifications. Unfortunately such computations are not always possible and usually require large and complex tools. In spite of continuous progress of research in this field, no serious attempts have yet been made (by mid-1986) to commercialize such tools.

Dynamic Analysis

Some scientists (GERM 82, TAY 84) have proposed techniques to transform a program P into a program P' which presents the same deadness errors as P , but which lends itself to detection of the imminence of a deadlock before it actually occurs. Other researchers have proposed extensions to the debugger concept in order to perform dynamic analysis (LED 85, MILANO 85). However, the practical utility of such techniques is yet too poorly established to consider their actual use at this time.

System Simulation: The "Software Rig" Concept

Current practice in real-time systems design is to develop the software on a host machine and compile it to a target. The target machine is then run in a hardware rig and the total system tested. By analogy it would be desirable to use a software rig prior to use of a hardware rig to test or to otherwise prove that the compiled code meets the software requirements. To do this, it is necessary to provide a simulator of the rest of the system including (if necessary) the aircraft, missile, or space vehicle dynamics so that the response of the software can be examined at an early stage. The output from such a software rig should be given in a form appropriate to the user; i.e, for a flight control system, for example, such things as time response, frequency response and root locus charts should be produced. Interfaces should be provided to the hardware rig so that the software can be tested and evaluated independently of the hardware and prior to hardware/software integration. Finally, provision should be made for flight test data to be fed into the software rig so that the response of the software to actual flight test data can be established.

5.10.3 Failure Modes and Effects Analysis

In the longer term, it would be desirable to have failure modes and effects analysis tools incorporated into the support environment along with the rest of the validation and test tools so that the effects of failures in the software could be analyzed.

5.11 Support Tools

A wide variety of general support tools are needed. Requirements for some of the more important tools of this kind are discussed below.

5.11.1 Text Editor

- There should be one general purpose text editor in the environment which is used for all kinds of text to be edited.
- This editor should allow one to create, modify, or inspect a text file, or to merge several text files.
- The editor should have at least one working buffer in which the text is processed. Permanent text files in the environment should never be altered interactively.
- It should be possible to move the current position cursor quickly to anywhere in the working buffer by a continuous move (without being obliged to indicate the final destination of the cursor move) in order to read, modify, insert, suppress, or move text.
- The visual image of the working buffer should be assignable to any window on the terminal screen.
- This editor should use a general attributed tree handling facility which should be used also to support intermediate language objects independent of which high level language they come from (see Section 5.8 - Compilers). This will give to the editor the capability of working directly on the syntactic and semantic structure of a program unit and will provide other functions such as:
 - Automatic building of frames for compound syntactic elements.
 - Automatic and programmable indentation within such frames.
 - Provision of an interface with language analyzers to provide incremental compilation without leaving the editing session.
 - Provision of an interface with language compilers allowing an automatic merge of source

text and error files and provision for capability to navigate from error to error in the resulting text file.

5.11.2 Configuration Management

- It should be possible to handle multiple versions and successive editions of each version for all objects in the environment.
- It should be possible to constitute subsystems composed of related objects. A subsystem should have a name, version, and edition and from that triple (subsystem name, version edition) it should be possible to retrieve all the objects composing the subsystem together with their respective name, version, and edition.
- Precise specifications for configuration management are very dependent on the specific projects to be managed and the desires of management. The main options for such specifications are based on:
 - Syntax and semantics of versions, editions, releases of source objects.
 - The whole global set of objects generated during the software development cycle.
- Configuration management tools can and probably should be built as an extension to the program library manager (NARF 85).

5.11.3 Project Management

- Project management tools have three main functions:
 - To estimate all the resources required for a software system project.
 - To provide histories of the different facts surrounding the project development.
 - To compare histories with corresponding estimates in order to see how well the project development is going. This comparison could indicate where the project is behind schedule or above estimated costs in order to allow project management to take corrective actions or at least derive lessons for future projects (i.e., improve estimation models...).
- The complexity of project management tools can vary widely. Very simple tools may take account only of task duration and sequencing while more sophisticated tools may use a knowledge base containing elements on development methodologies, resource assigning rules, histories, etc.
- Cost, development time, and project personnel estimation tools should be provided. In general, such resource estimation tools must be able to:
 - Support a project development diagram showing the different phases of development of the different subsystems of the project, including their dependencies and resource allocation rules. The main lines of the development methodologies must be used by the tools to elaborate such a project development diagram.
 - Use past project histories and/or predefined models, together with the project development diagram, to estimate cost, development time, and personnel requirements.
 - Take account of existing constraints (availability of resources, desired completion time...) to adjust the estimates.
 - Undertake simulations to see how sensitive the project development plan is to various difficulties that may arise and to revise estimates to allow for reasonable resource margins to cope with such anticipated difficulties.

- Tools for collecting and organizing historical information relating to project development must take account of both human resource consumptions and effective use of the environment. There is a requirement here for histories of the use of the major environment tools. The historical information collected should include:
 - The date of creation and modification of environment objects.
 - The reasons for creation and modification of environment objects when they are relevant.
 - The resource consumptions for such creation and modification.

5.12 References

- ANNA 84** **Anna - a language for Annotating ADA Programs-Preliminary Reference Manual-** D.C. LUCKHAM - F.W. von HENKE - B. KRIEG-BRUCKNER - O. OWE Stanford University June 1984
- ARP 84** **Proposition Au Projet National de Genie Logiciel "ARPEGE"** - Ministere Francais de l'Industrie- ADI - July 1984
- BAFIGA 85** **An Approach for Evaluating the Performance Efficiency of ADA Compilers** - M.J. BASSMAN, B.A. FISHER, A. GARGARO, Computer Sciences Corporation- "ADA in Use": Proceedings of the ADA International Conference - Paris May 1985 - CAMBRIDGE University Press
- BKP 84** **Now You May Compose Temporal Logic Specifications** - H. BARRINGER- R. KUIPER - A. PNUELI - University of MANCHESTER (GB) and WEIZMANN Institute of Science, REHOVOT, ISRAEL - 16th ACM STOC 1984
- BKP 85** **A Really Abstract Concurrent Model and its Temporal Logic**
- H. BARRINGER - R. KUIPER - A. PNUELI - University of MANCHESTER (GB) and WEIZMANN Institute of Science, REHOVOT, ISRAEL - Working Paper
- BOND 81** **Methode de Conception et de Programmation d'Applications Multi-Taches** - P. de BONDELI, CR2A and AEROSPATIALE/ Space Division Docteur-Ingenieur Thesis - P et M. CURIE (PARIS 6) University - July 1981
- BOND 83** **Models for the Control of Concurrency in ADA Based on Predicate Transition Nets**- P. de BONDELI, CR2A and AEROSPATIALE/SPACE Division - Second Joint ADA EUROPE/ ADATEC Conference, BRUSSELS 1983
- BOOCH 83** **Software Engineering with ADA** - G. Booch- Benjamin/Cummings Publishing Company 1983
- BOOCH 86** **Software Components with ADA** - G. Booch- Benjamin/ Cummings Publishing Company 1987
- CAIS** **Proposed Military Standard Common APSE Interface Set (CAIS)** 31 January 1985 - Department of Defense - USA
- CONC 86.1** **Concerto - Atelier Logiciel - Presentation Technique** - French Telecom. Ministry - CNET/LANNION - February 1986
- CONC 86.2** **Une session sur le poste de travail CONCERTO** - A. CONCHON - J. CAMACHO - A.M. RASSER - included in CONC.86.1 but also presented in the Proceedings of the "3eme congres de genie logiciel" AFCET VERSAILLES May 1986

- GERM 82** **Monitoring for Deadlocks in ADA Tasking** - S.M. GERMAN, D.P. HELMBOLD, D.C. LUCKHAM, STANFORD University - Proceedings of the ADATEC Conference on ADA, Arlington, October 1982
- GO 86** **Reusing and Interconnecting Software Components** - J.A. GOGUEN, SRI International - IEEE Computer February 1986
- GOLD 85** **ADA for Specification: Possibilities and Limitations** - Edited by S.J. GOLDSACK, Imperial College of Science and Technology LONDON CAMBRIDGE University Press 1985
- HDM 79** **The HDM (Hierarchical Development Method) Handbook** - SRI International 1979
- HILL 84** **Asphodel - An ADA Compatible Specification and Design Language** - A.D. HILL, CEGB LONDON - Proceedings of the Third Joint ADA EUROPE/ADATEC Conference, BRUSSELS June 1984 - CAMBRIDGE University Press
- LISK 74** **Programming with Abstract Data Types** - B. Liskov and S.N. Zilles, Mit - Proceedings of ACM/SIGPLAN Conference on Very High-Level Languages - Sigplan Notices April 1974
- MACH 85** **Guide du Concepteur MACH** - L. BOIDOT, M. BOURGAIN, M. LISSANDRE - IGL 1985
- MAU 85** **An Event - Driven Debugger for ADA** - C. MAUGER and K. PAMMETT, ALSYS - ADA in Use: Proceedings of the ADA International Conference PARIS May 1985, CAMBRIDGE University Press
- MILANO 85** **Execution Monitoring and Debugging Tool for ADA Using Relational Algebra** - A. DIMAIO, S. CERI, S. CRESPI REGHIZZI, -Politecnico di MILANO - ADA in Use: Proceedings of the ADA International Conference PARIS May 1985, CAMBRIDGE University Press
- NARF 85** **Extending the Scope of the Program Library** - K.H. NARFELT and D. SCHEFSTROM, University of LULEA and TELELOGIC AB - ADA in use: Proceedings of the ADA International Conference, Paris May 1985 CAMBRIDGE University Press
- NIEV 85** **Dialogue Design: Principles and Experiments** - J. NIEVERGELT, C. MULLER AND H. SUGAYA - BBC International Symposium on -Computer Systems in Process Control - BADEN (AARGAU SWITZERLAND) September 1985. Also reprinted in a Tutorial on Aspects of Program Development Environments given at the "3eme congres de Genie Logiciel" AFCET VERSAILLES May 1986
- PCTE** **A Basis for a Portable Common Tool Environment - Functional Specifications** - 2nd Edition January 1985 - European Strategic Programme for Research and Development in Information Technology (ESPRIT)
- PET 77** **PETRINets** - J.L. PETERSON, University of TEXAS at AUSTIN - ACM Computing Surveys, September 1977
- PET 80** **Net Theory and Applications** - Lecture Notes in Computer Science n° 84 SPRINGER VERLAG 1980
- RAT 85** **Reference Manual for the Rational ADA Environment** Rational Inc. 1985
- RIP 84** **The ADA "Program Library": its Meaning and its Implementation in an Existing Generic Environment in Industry** - K. RIPKEN, Laboratoires de Marcoussis - CGE - Proceedings of the Third Joint ADA Europe/ADATEC Conference, Brussels June 1984 CAMBRIDGE University Press
- RSL 1** **A Requirements Engineering Methodology for Real Time Processing Requirements** - M.W. ALFORD, TRW - IEEE Transactions on Software Engineering January 1977

- RSL 2** **Software Requirements Engineering Methodology (SREM) at the Age of Two - M.W. Alford, TRW - IEEE 3rd International Conference on Software Engineering 1978.**
- UNIX** **UNIX System V Users Manual - ATT/Bell Laboratories**

APPENDIX 5.1

THE KERNEL ENVIRONMENT

A.5.1.1 Introduction

This appendix summarizes the PCTE view of the kernel environment which appears to match well with the needs described in this chapter. The object management system (OMS) of PCTE is also semantically close to the CAIS document. However, the user interface part of the PCTE is not summarized because flight guidance and control systems design places special requirements on user interfaces which are better satisfied by a special user interface designed specifically for such applications.

The following sections address:

- The object management system (OMS).
- The program execution primitives.
- The inter-process communication facilities.
- The protection mechanisms to preserve data integrity.
- The I/O primitives.
- The distribution mechanisms.

A.5.1.2 Object Management System (OMS)

An object has the following characteristics:

- Attributes (defining its properties).
- Links referring to other objects. A link can be primary (creation link) or secondary. Links can have attributes.
- Contents (optionally): the contents are handled directly by the specific tools, not by the OMS.

Objects, attributes, and links are typed.

Object types are hierarchical: from a "parent" object type it is possible to derive a new object type which inherits its parent's type characteristics and adds new specific characteristics to them.

Objects are referred to by a path: a path starts from a root object (the current process or the top root of the system) and defines a chain of links, the last of which points to the object to be referred.

The following attributes are predefined and apply to all object types:

- Access control attributes: owner, group, mode
- Physical volume
- Identifier
- Number of links starting from objects of the type

- Date-time attributes:
 - Creation date
 - Last access date
 - Last modification date

A schema is a collection of definitions of objects, attributes, and link types. Schemas are used to define the scope which is visible to:

- The whole set of users (system schema).
- A given project.
- A specific user.

A.5.1.3 Program Execution Primitives

The term "program" is defined as the description of a potential activity and the term "process" is defined as the actual execution of such an activity. The static context contains the following elements:

- Executable or interpretable code
- Conditions of execution (type of processor or interpreter)
- General execution information
- On-line documentation

A process is associated with:

- A system-wide process identity.
- The identity of the user having created the process.
- A working schema that determines the visible scope in the data base.
- An execution environment which is initiated by the parent program.
- Open files (initially inherited from the parent program).

Two primitives are necessary for starting the execution of a new process; namely,

- A primitive for creation of a new process that executes a program indicated as a parameter and blocks the calling process until completion of the invoked program. The calling sequence contains the input parameters, and an indication of the execution context (working schema, execution environment). A return value indicates whether the operation was performed properly or not.
- A primitive identical to the above except that the calling process does not wait until completion of the invoked program. The operation returns a value indicating the process-identity of the new process.

For terminating the execution of a process, two primitives are necessary:

- A primitive for termination of the current process and of the "son" processes.
- A primitive for termination of a designated process and of its "son" processes.

Both will set the terminated process status code to an indicated value.

Processes can be temporarily suspended/resumed. A suspended process has its files still open, can be terminated, and can receive a signal (which will cause the suspended process to automatically resume its execution). The following operations are provided for manipulation of suspended processes:

- Examination and modification of a field in memory.
- The setting and removal of a breakpoint at a specified address.

A.5.1.4 Inter-Process Communication Facilities

The PCTE view of the kernel environment supports the following four inter-process communication facilities.

- **Pipes:** Analogous to UNIX pipes (UNIX). Pipes provide a simple means of communication between processes with only very loose synchronization. A pipe is an inter-process communication channel which can be visualized as a buffer with a limited capacity.

Four primitives are provided for operating on pipes: open, close, read, write. They are analogous to the corresponding primitives on files with some semantic differences:

- A call to open for reading is illegal if the pipe is not yet opened for writing (i.e., no "consumer" is allowed if there is no "producer"). If a call to close for writing is issued on a pipe which was opened for both reading and writing, then the next call to read will get a "broken pipe" error.
- If the pipe is opened for writing, a call to write will put a block of data into the pipe if there is still enough room for it.
- If the pipe is opened both for reading and writing, a call to read will take from the pipe, on a FIFO basis, a block of data if it was present in the pipe.
- Read and write primitives may be blocking or non-blocking when the pipe is empty/full.
- **Message Queues:** Analogous to UNIX System V Message Queues. Processes can Send / Receive messages to/from a message queue.

Messages are typed, but the type is under control of the user processes; a message-queue can contain messages of different types and can be limited in size.

Send and Receive can be either blocking or non-blocking. The send primitive indicates the contents and the type of the message to be sent. One can specify whether the primitive "receive" should receive:

- Any message.
- Only messages of a given type.
- Only messages whose type has at least a given value (type identifiers are ordered).

Other primitives allow a queue to be scanned, a copy of the type and contents of a given message to be obtained, and a given message from a queue to be removed. Only one process can perform these operations and the receive operation; namely, the process which is "connected" to the queue. (A pair of connect/disconnect primitives is therefore also provided.) The send operation can be performed by a process which knows the message-queue identifier. Message-queues can be saved and restored from mass-memory.

- **Shared-Memory:** It is possible to dynamically create memory segments which can be shared by several processes. A pair of primitives Reserve/Release provides the necessary minimum

primitives to allow for exclusive access to these memory segments.

- **Signals:** These are asynchronous signals that can be sent to a given process. Two primitives are provided:
 - Send a signal.
 - Handle a received signal.

When a process receives a signal, it is interrupted if it was active and it immediately executes this primitive.

A.5.1.5 Protection Mechanisms to Preserve Data Integrity

Protection mechanisms are required for access to resources by concurrent processes.

A "resource" can be:

- An OMS object with its contents and its attributes.
- A link and its attributes.

An activity is the framework in which a set of related operations takes place. An activity always has a set of acquired resources and associated locks which are held on these resources.

A process is always initiated in the context of an activity. A process can start and control new activities which are then considered as nested to the one in which the process was started.

When a process terminates, the activity in which the process was started and any nested activities also terminate. Activities can also be terminated explicitly, but the effective termination of an activity must wait for the termination of any process which was initiated in its context.

An activity locks the resources it needs access to. Locks can apply to read operations, write operations or both. A lock can be:

- Unlocked, in which case the activity indicates it is going to access the resource without any need for protection.
- Exclusive (read exclusive or write exclusive) in which case write operations are immediately applied to a write exclusively access resource.
- Write protected which is the same as write exclusive except that modifications to the resource are only applied globally at the end of the access or not applied if the operation for which the access was performed fails.

If a resource is already locked by some activities, an activity trying to use the resource is delayed if this lock is incompatible with other existing locks on the resource. A lock can apply selectively to accesses by other external activities or other nested activities. In this case, it must be equal or weaker against internal activities as compared to external activities.

A lock applies to the indicated resources and its "concerned domain". The "concerned domain" is defined as follows.

- If the resource is an object, then the concerned domain is the object and the set of links leading to and originating from this object.
- If the resource is a link, the concerned domain is the link and the object from which it originates. If the link is a "primary" link, the concerned domain also includes the concerned domain of the destination object.

A.5.1.6 I/O Primitives

Two types of objects have contents:

- Files.
- Devices.

These objects are manipulated by the OMS as other objects, but additional primitives are provided to manipulate their contents. These primitives are described below:

- Given its pathname, the object contents can be opened/closed for reading, writing or both. The result of the open operation is a "File-descriptor". File descriptors are local to a process but are inherited by its son processes.
- Once the contents of an object is open, one can write any data to it, or read any data from it. Read and write operations can be specified to be blocking (causing the calling process to wait until completion of the operation) or non-blocking.
- A file object can be seen as a stream or as a randomly accessible sequence of bytes. In addition to those already mentioned, the following functionalities are available on files:
 - Get the current size of the file.
 - Position for the next (read or write) operations either relative to the present position or absolute.
- A device object (I/O device) is only accessible sequentially. A control primitive is provided to read or write the status of a device; its semantics is dependent on the specific class of device.

A.5.1.7 Distribution Mechanisms

The environment can be distributed on a local net connecting several workstations and common servers. This distribution is in general transparent to users. However, users have the facility to control it (locate an object or have a process initiated on a given station...) when necessary.

The main principles that the implementation of the kernel environment must comply with to allow such a distribution are the following:

- Provide a standard low level communication mechanism (such as the OSI "transport") to insure independence from the physical network topology and communication media.
- Have each workstation or server be an object of the OMS which has status attribute (connected, connection in progress, disconnected, or failure).
- Uniquely identify all users, processes and all objects in the distributed system.
- Group OMS objects together on "volumes". An object is created relative to a volume and not to a workstation or server. Some volumes must be replicated on each connected workstation and server.
- Information on volumes replicated on each connected workstation, schema definition sets, and activity description data are objects which are shared by all workstations and servers and consequently are managed by the distribution.
- An "Administration Volume" contains, among other things, the OMS root objects, the workstation and server objects, the password and group files, and the directories of existing volumes.

Each workstation and server has a "projection" of the administration volume, and modifications to it are broadcast to all connected workstations and servers.

- The decision as to where a process should run is made at program execution time after the pathname of all the objects requested for the process execution has been evaluated. This decision depends on different possible criteria and may be forced by the user.

APPENDIX 5.2

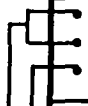
SOFTWARE DESIGN TOOLS TO SUPPORT THE "OBJECT ORIENTED" METHODOLOGY

A.5.2.1 Basic Software Design Tools

In Chapter 2 (Section 2.3.2.3) it is stated that structured design based on information hiding and abstract data types is the state of the art for basic software design. In order to see what tools are needed to support such structured design, the basic concepts involved will be described in somewhat greater detail than given in Chapter 2.

Most basic is the concept of an "OBJECT" (LISK 74, BOOCH 83). An object is a data structure presenting to its users a set of operations which constitutes its INTERFACE or specification. The detail (implementation) of the data structure is not visible for external users. Rather, it is HIDDEN or ENCAPSULATED in the BODY (implementation) of the object.

The following diagram illustrates this object structure and an example of such an object in the ADA package for INTEGER_QUEUE is given below.

Abstract view of the object for users		
	Type declarations	OBJECT INTERFACE
	Data declarations	
	Interface of visible operations	
	Detail of types and data declared in the interface	OBJECT
	Body of visible operations	BODY
	- Internal types and data - Internal operations	(hidden from the users)

Object Structure

Package INTEGER_QUEUE is

SIZE: constant POSITIVE: = 100;

- This package encapsulates a queue of INTEGER elements.
- The maximum size of queue is SIZE.
- Initially the queue is empty.

Procedure ENQUEUE (EL: INTEGER);

—/* PRECONDITION: the queue is not full (the caller waits until this is true).

—/* POSTCONDITION: EL is inserted as last element into the queue.

Procedure DEQUEUE (EL: out INTEGER);

—/* PRECONDITION: The queue is not-empty (the caller waits until this is true).

—/* POSTCONDITION: The first element is taken out from the queue and its value is assigned to EL.

Procedure TRY_ENQUEUE (EL: INTEGER; IS_FULL: out BOOLEAN);

—/* POSTCONDITION:

- If the queue is full then IS_full;
- else ENQUEUE (EL) is executed;
- not IS_FULL
- end if;

Procedure TRY_DEQUEUE (EL: out INTEGER; IS_EMPTY: out BOOLEAN);

—/* POST CONDITION:

- If the queue is empty then IS_EMPTY;
- else DEQUEUE (EL) is executed;
- not Is_Empty;
- end if;

End INTEGER_QUEUE;

Package body INTEGER_QUEUE is

- Implementation of the queue,
- Bodies for procedures ENQUEUE, DEQUEUE, TRY_ENQUEUE, TRY_DEQUEUE.

End INTEGER_QUEUE;

INTEGER_QUEUE: An example of an object represented by an ADA package

The second important concept is that of an ABSTRACT DATA TYPE (LISK 74, BOOCH 83). An abstract data type (adt) is a template which allows one to define a class of objects sharing common properties.

The name "ABSTRACT DATA TYPE" has the following origin:

ABSTRACT

The INTERFACE or SPECIFICATION of the abstract data type gives the properties of the adt visible for the users.

These properties can be viewed as an ABSTRACTION of their implementation.

DATA TYPE

One can define on such a type a class of objects sharing common properties.

An ADA example of an abstract data type is given below.

An object `INTEGER_QUEUE`, similar to that we just saw, can be created on the adt `QUEUE` as shown by the package described below:

```
package INTEGER_QUEUE is new QUEUE (100, INTEGER);
```

Generic

```
SIZE: POSITIVE; -- maximum number of elements in the queue
type ELEMENT_TYPE is private; -- type of elements in the queue
```

Package `QUEUE` is

```
-- This generic package defines an abstract data type QUEUE.
-- Object queues can be created on this adt by instantiating this generic package.
-- A newly instantiated queue is initially empty.
```

```
Procedure ENQUEUE (EL: ELEMENT_TYPE);
```

```
--/* PRECONDITION: The queue is not full (the caller waits until this is true).
--/* POSTCONDITION: EL is inserted as last element into the queue.
```

```
Procedure DEQUEUE (EL: out ELEMENT_TYPE);
```

```
--/* PRECONDITION: The queue is not empty (the caller waits until this is true).
--/* POSTCONDITION: The first element is taken out from the queue and its value is assigned to EL.
```

```
Procedure TRY_ENQUEUE (EL: ELEMENT_TYPE; IS_FULL: out BOOLEAN);
```

```
--/* POSTCONDITION:
--   If the queue is full then IS_FULL;
--   else ENQUEUE (EL) is executed;
--   not IS_FULL;
--   end if;
```

```
Procedure TRY_DEQUEUE (EL: out ELEMENT_TYPE; IS_EMPTY: out BOOLEAN);
```

```
--/* POSTCONDITION:
--   If the queue is empty then IS_EMPTY;
--   else DEQUEUE (EL) is executed;
--   not IS_EMPTY;
--   end if;
```

```
End QUEUE;
```

Package body `QUEUE` is

```
-- Implementation of the queue,
-- Bodies for procedures ENQUEUE, DEQUEUE, TRY_ENQUEUE, TRY_DEQUEUE
```

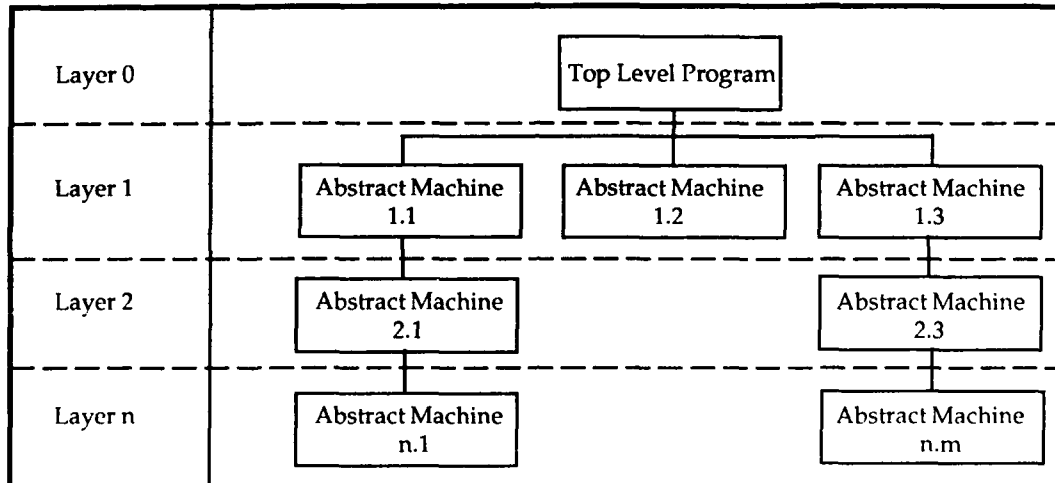
```
End QUEUE;
```

Large software subsystems are generally developed top down as a succession of hierarchical layers. In order to build such a layer structure, an additional structuring concept is needed beyond the concepts of objects and data types. This additional structuring concept will be referred to as the **ABSTRACT MACHINE**. This concept is less commonly used than the two others described above (objects and abstract data types) because it is relevant only for large pieces of software. However, it is quite well known in the USA through its role in the HDM methodology (HDM 79), and in FRANCE through its role in the MACH methodology (MACH 85).

An abstract machine can be an object, an abstract data type, or a set of objects and abstract data types having

some functional or logical links between them. In the context of large, hierarchically structured software systems, an abstract machine is a piece of hierarchical layer structure the type of which is illustrated by the diagram shown below.

According to the ABSTRACT MACHINE structuring concept, an abstract machine on layer N can use for its implementation only abstract machines on layer N + 1 (This is a consequence of the "Well-Formedness" property of the hierarchy). The development of such a hierarchically structured software system using ABSTRACT MACHINES is processed iteratively top down: Iteration N consists in developing the basic design of layer N together with the detailed design of layer N-1.



A Hierarchy Of Abstract Machines

Note: The symbol  means that the implementation A uses B where "uses" means uses operations on the objects of B and/or creates objects on the acts of B

Now that the main concepts used by state-of-the-art basic software design methodologies have been captured, the kinds of tools that are needed to support an efficient use of these concepts can be usefully described as follows.

From the two examples provided above (object and abstract data type), it can be seen that the modular structure of ADA is a convenient frame to support such concepts. The same is true for LTR3, but not for earlier programming languages such as JOVIAL, PEARL, LTR2, CMS2, and CORAL 66 which do not provide this modular structure.

A library manager is essential to maintain the "use" links between these modules (objects and adt). The discussion of this tool is expanded a bit further in the section devoted to software implementation tools.

ADA "Package Specifications" or LTR3 "Module Interfaces" provide only syntactic and static semantic specifications; that is, no dynamic semantic specifications are included in ADA "Package Specifications" or LTR3 "Module Interfaces". In the examples, dynamic semantic specifications are provided by comments defining:

- State conditions such as "empty" or "full"
- Preconditions and post conditions for each operation.

For the reader's convenience, these dynamic semantic specifications were written as free text comments. If a formal annotation system was used instead, such as algebraic notation or ANNA developed by the

University of BREMEN, SRI and STANFORD University (ANNA 84), or ASPHODEL (HILL 84) developed by the British Central Electricity Generating Board, then tools should be provided to handle this annotation system and check the completeness and coherence of specifications written in the annotation system.

An object, such as "INTEGER_QUEUE", built on the adt QUEUE, may well be used by several concurrent tasks. In this case temporal conditions must be added to the specifications in order to specify the synchronization of the concurrent users on the operations on the object. In the examples, such temporal conditions are the waiting of a caller on ENQUEUE until the queue is not full or the waiting of a caller on DEQUEUE until the queue is not empty.

The two best understood possibilities for specifying temporal conditions are temporal logic (BKP 84, BKP 85) and PETRI nets (PET 77, PET 80). If temporal logic and/or Petri nets are used, then tools should be provided to handle and manipulate them and to analyze interesting temporal properties (e.g., find all the invariants, check for deadlock freeness, safeness...) on temporal specifications expressed in this language.

No tools have yet been proposed to handle temporal logics. The most complete set of tools that has been proposed to handle PETRI nets was defined in the French project "ARPEGE" (ARP 84).

Finally, another category of tools must be provided to define an abstract machine hierarchical structure and to check for its "well-formedness".

A.5.2.2 Detailed Software Design Tools

For each layer of the hierarchical structure, detailed software design consists of implementing the abstract data types and objects, whose specifications were defined during the basic software design phase, in terms of abstract data types and objects of the next layer.

Ideally, tools would be available that are able to automatically produce private specification parts (ADA) or "opaque" interface parts (LTR3) and bodies from ADA package specifications or LTR3 module interfaces when the latter are supplemented by formal semantics specifications. This is not within the state of the art today, and this operation is therefore still essentially "manual". Nevertheless, it is important that the software designer proceed in a very methodical way in developing the design from the specifications in order to produce verifiable software.

For sequential software fragments, assertions will be systematically inserted as annotations within declarative parts (making the domain of values of variables precise) and imperative parts (giving conditions that must hold when the program execution reaches their location). Such program fragments are then verifiable in two steps:

- Proof that post conditions for the program fragment hold as a result of:
 - Preconditions for this program fragment.
 - The abstract semantics of the operations used on the lower layer.
 - Assertions inserted in the program fragment.
- Transformation of the assertions inserted in the program fragment into check functions raising exceptions if the assertion they check is not satisfied and checking that these exceptions are never raised when testing the program fragment.

Tools should be provided to mechanize both of these steps.

To develop the concurrency control structure of a module (object or adt), the best starting point is a PETRI net specification of the synchronization rules to be enforced by the module. Temporal logic specifications may have some advantages over PETRI net specifications when only specifications are considered, but they do not provide much help in developing an implementation from the specification.

A good general approach to follow is:

- To consider a PETRI net grammar of the implementation language (such a grammar exists for ADA [BOND 83] or LTR3) and the PETRI net specifications of the lower layer.
- To try to develop the PETRI net specifications of the module to be implemented into more detailed PETRI nets using PETRI net morphisms, until a level of PETRI nets is reached which are directly implementable.
- To try to prove that the PETRI nets finally obtained still possess the properties defined by the original specification PETRI nets.

A considerable set of tools to handle PETRI nets and mechanize proof of algorithms is needed, but, although for a large part they are within the state of the art, they are not available at the beginning of 1986.

CHAPTER 6

FUTURE TRENDS

6.1 Introduction

Software for embedded systems such as flight guidance and control systems has—for various reasons—always been considered different from other types of software, such as simulation or scientific computation. The reasons for this distinction can be summarized as:

- Embedded systems software deals with a great variety of devices which are application dependent. Programs for embedded systems have to deal with events occurring in the environment in real time.
- Memory is often at a premium, so great care must be taken to make as efficient use of it as possible. This requirement often conflicts with real-time needs.
- The cost of producing the software is not a dominating factor.
- System integrity is the single most important issue in both design and postdevelopment support of the software for safety critical systems.

The situation is becoming increasingly distressing as a result of the rapidly growing complexity of embedded systems software due to growth in the number of interacting functions to be realized and the amount of their interaction, and the complexity of each of them grows bafflingly fast. The term complexity is used here in a rather loose sense; its purpose in this context is merely to indicate that a difficulty threatens to get out of hand completely if no adequate way is found for dealing with it.

The increase in complexity of embedded systems software is not so much the result of the control mechanisms getting more sophisticated. Rather it is due to the fact that there is a strong tendency to integrate control functions and tactical functions, like weapon control, strategic planning, etc. This tremendous increase in complexity cannot be handled simply by writing more code because the number of details as well as their diversity is beyond human capacity to be dealt with directly.

Therefore, new methods are desperately needed to help control and handle the complexity. Note that there is a difference between controlling and handling. One aims at keeping the complexity as low as possible, whereas the other provides mechanisms to deal with whatever complexity is necessary.

Fortunately, history shows that with good theories, methods, and related tools, the human abilities to deal with complexity grow significantly. For example, when first designed, the T-Ford was an incredibly complex piece of machinery. Today its design would be regarded as a simple exercise for a somewhat advanced student.

6.2 Control of Complexity

There are several ways in which one can hope to gain control over the increases in complexity. They are listed and described below.

- Integrating the requirements for and the specification of the function to be realized by a system into the development cycle. This cycle consists of alternations between high level decisions about

system and program structure and lower level implementation of those decisions.

- Documenting the various design decisions in an easily accessible way so that design activities can effectively use insights gained in previous work.
- Development of systems in a number of stages, each of which adds precision to the results of the previous stage. This technique is often referred to as "top-down design" or "stepwise refinement". Note that this refinement strategy pertains mainly to the software in a system, although in large systems with many functional hierarchies, the technique can certainly be helpful in structuring the overall system design process.
- Implementing the software in a higher level language which alleviates many of the problems of machine coding.
- Use of standard techniques and standardized components whenever such components are available and wherever such techniques and their associated tools can be applied. This has always been done for hardware parts of systems, but is used far less for the software components. The design and postdevelopment support costs of system software being as high as they are, it would certainly be beneficial if software "components" techniques for controlling complexity could be used more extensively.

Note that this list gives no clues as to how to implement the techniques. Moreover, any special requirements for real-time embedded systems are largely ignored. The remainder of this chapter will present some ideas that, taken together, may help in finding a solution.

6.2.1 System specification

Probably the most important aspect of all design activity—whether for real-time application or not—is the careful specification of the intended functions and behavior of the system. The difficulty of specifying a system component is partly dependent on the contribution of that component to the total system behavior. The more indirect an influence a component has on that behaviour, the less clear is the component's specification structure and elements. In particular, timing and interface requirements for such a component can be very difficult to establish, even though the system as a whole is completely specified. This is particularly true for software in real-time systems. The task of specifying software for a complex system is therefore far from trivial and yet essential for the successful design of the necessary software.

Demands for software have grown enormously since processing power and memory have become available in huge quantities. Designs are generally documented merely because it is required by customers, and the documentation produced does not normally contain sufficient information to extract experience from the design process documented. It seems likely that this is a major factor in the lack of engineering standards and techniques for software design.

Software has spread to almost all imaginable application areas long before a science of programming was available to guide the designers. Scientific understanding has therefore never existed in the field of programming. Statements like "the program is almost ready" or "I have now found the last bug" are clear examples of this.

Some progress has been made in the understanding of what programming really is. A number of researchers have undertaken to study the program at a more fundamental level than was hitherto practiced. Results of these efforts all seem to indicate that a more mathematical attitude towards software design will lead to better and probably more correct programs. (Here, correctness must be understood as the semantics of the program being equivalent to the specifications.) According to these results, programming activity boils down to a series of correctness-preserving transformations, where the choice of transformation is guided by efficiency considerations (sometimes design efficiency, sometimes execution efficiency; the latter being the typical real-time case).

The general approach to specifying a program's behavior is to specify the inputs it is allowed to operate on and what the ensuing outputs should be. A powerful and, in principle, very general notation for such specification is first-order predicate calculus. Unfortunately, predicate calculus is difficult to read for most

people, and it is therefore discarded as unpractical. It must be noted, however, that intrinsically difficult problems—like the specification of complex functions—can never be made simple by notation. On the other hand, it also is well known that bad notations can render trivial problems totally intractable. As an example consider arithmetic in Roman numerals.

It is conceivable that textual manipulation of expressions in predicate calculus can make these specifications more palatable without loss of precision. Such expressions could very well form a basis for complete specification and derivation of programs. The derivation process as well as the specification activity itself, would be computer assisted in that a machine could check the specifications for internal consistency.

Great care is required to specify as little as possible in order not to preempt design decisions. In particular, non-determinism must be used whenever arbitrary choices have to be made. The specification "language" must allow such non-determinism. In terms of predicates, this means that (in constructs where choices are involved) the various alternatives must each fully specify the conditions under which the alternative is open. No inter-dependence between such alternatives can be allowed.

The example of predicate calculus was given mainly to illustrate the possibility of computer assisted program generation. Other techniques that offer similar precision and formality could equally well be used. In fact, it could be advantageous to allow designers to choose their preferred representation. Graphical representation of system specifications seems to appeal very much to a large group of designers, whereas others prefer more textually oriented algebraic notations. If the semantics of the different notations is well defined, it should be possible to easily convert from one to another.

6.2.2 Documentation

Probably the most neglected aspect of programming activity is documentation of the design and, in particular, of the decisions taken during the design phase. Documentation serves not only postdevelopment support programmers. More importantly—it serves to improve designs. Unfortunately, most documentation merely provides the information available in the program sources in a slightly more readable form. Little, if anything, about the design process can be learned from such descriptions even though the design of new related programs and other information about existing designs is needed. In fact, for understanding of the actual program text, no reference to purely descriptive documentation should be necessary. The commented text together with documentation that provides insights as to what the program is supposed to achieve, why it is structured as it is, and what the various identifiers stand for should be enough for any competent person to understand the program. Design tools can be envisaged that suggest decisions on the basis of what can be learned from documentation of previous designs. Such tools obviously require rather extensive knowledge about both the design at hand and the relevant existing ones. The tools themselves can assist the program designer in establishing what the required knowledge is and how to formulate it. In this way, the design tools will both assist the designer in the difficult selection between apparently equivalent alternatives (which may be vastly different in the context of the entire design) and help document such decisions.

The documentation generated by the design tool can be made much more accessible than is normally done by providing an access mechanism through automatically extracted keywords, much like automatic indexing of technical literature. An efficient access mechanism of this sort will prove beneficial even though no use of it can be made by tools for design assistance yet. An information retrieval system that can be used to provide such access mechanisms can also be used to integrate project oriented information such as test results, planning estimates, and actual effort, all of which depend on the complexity of the system/software and which is reflected in the documented design history. Providing access to all relevant informational data in a structured, cohesive way should lead to better understanding of the entire design process.

6.2.3 Structured Software Development

The approach toward developing complex software has progressively drifted from nonstructured coding of functions thought to be useful to a strict top-down decomposition. Although the top-down approach is far better for both controlling and handling of complexity in principle, it has the potential disadvantage of leading to low-level implementation modules that are either difficult to implement efficiently or which foreclose the use of existing software or both.

Considerable efforts are devoted these days to the design of languages in which it becomes feasible to produce

modules that lend themselves to reuse (Ada, LTR-3). However, it is very distressing to find that the current design methods and tools fail to take advantage of the availability of such reusable building blocks. Assuming the availability of applicable software components for a given task, there is no method currently available that guides the decomposition of the total system's task in such a way that the components are made usable by providing the correct computational environment; and this is true even though such a decomposition is often possible without having to compromise too much in the design of the software. A good example is the success of mathematical libraries in scientific software where, more often than not, procedures are deliberately designed to meet interface requirements of these library routines. It seems desirable, and within the possibilities of the near future, to devise and implement machine assisted design strategies that will effectively construct applications software around existing software building blocks.

For a tool to be able to assist in such designs, it is essential that certain information about the available components be accessible by the tool. In particular, syntactic and semantic information about the interfaces is needed. The syntactic part is relatively simple and is already handled quite adequately by languages like Ada, LTR-3, and CHILL. The major problems lie in the semantic domain. Very few approaches have thus far resulted in effective use of formal specification methods. The interpretation of less formally described semantic properties of program components should be possible, however, with the aid of suitably constructed expert systems. Rather hopeful developments in the use of semi-formal methods for program development have been reported by various institutions; e.g., the US Naval Research Laboratory's Software Cost Reduction Methodology. Note, however, that this effort is not aimed at re-use of existing software. It seems very likely that tools based on expert systems technology can be built to interpret the available information with sufficient precision to be helpful in the design of complex software systems.

Equally unsatisfactory is the situation regarding the influence of the target architecture on the design process. Most programs are developed as if the target were a single processor, resulting in emphasis mainly on the sequential nature of the program and some occasional interactions between tasks whose execution may conceptually proceed in parallel. If the target happens to consist of multiple processors, the assignment of processes to processors is done in a rather ad hoc manner fairly late in the design process. Ideally, there should be either total disregard for the target architecture—in which case the configuration tools will take total care of distributing the various parallel activities either at load or at run time—or the design should specifically address the distribution of processes at a very early design stage and subsequently exploit the resulting concurrency. An approach in which tools are expected to fully cope with the problem of distribution of processes over processors is not viable for lack of appropriate tools. It is feared that a long time will pass before such tools will actually become available. It is suggested, therefore, that design methods and tools should specifically address the concurrency problem from the very early stages on.

6.2.4 High-Level Language Implementation

Programming languages provide a mechanism to separate the algorithmic details of a program from the machine architecture the program is to be executed on. Moreover, they allow the abstraction of data formats from those implemented in the machine hardware. However, these abstractions, helpful as they may be in programming, lead to a significant reduction in machine utilization. Both memory and processor cycles tend to be used more freely by compiler-generated code than by hand crafted programs.

The advantages of abstraction are so evident that, on many occasions, designers have simply accepted the apparent loss of machine performance. On the other hand, great improvements have been made in compiler technology. Modern compilers do an excellent job in optimizing the code generation to such an extent that it requires a gifted programmer to improve on the computer generated code. Of course, if the design were done with reference to the actual machine facilities rather than those provided by the language, better code could easily be produced. The powers of these modern compilers lie mainly in their outstanding ability to cope with expression evaluation.

The solution to the problem of the facilities available in most languages being too limited has drawn considerable attention. Several new language designs provide far more flexible data structuring facilities than the previous generation of the languages. Equally impressive are the advances made in language design with respect to control over execution of the algorithms implemented. Of particular importance is the ability offered by some modern languages to precisely control various aspects of memory layout of data structures and to associate program entry points with hardware interrupts and similar machine dependent program attributes.

However, these modern languages—notably ADA—are so complex that compiler technology suddenly proves incapable of dealing with them efficiently. The complexity of languages such as ADA results from the desire to integrate program design aspects with implementation aspects. Constructs like those for separate compilation are typically useful for implementation, whereas the packaging of related objects strongly suggests a design strategy. It is precisely the conjunction of these features that makes compilation of ADA so difficult. It seems certain, nevertheless, that even for such complex languages, compilers will eventually be produced that generate good quality code.

Far greater improvements in software development may be expected when it becomes possible to make use of a program's intended behavior as documented in a detailed formal specification. Given such a specification, it seems possible to control the code generation and memory allocation schemes of a compiler for the best translation from source code to machine executable code. In particular, the best compromise between execution speed and memory requirements could be selected in any given situation. Recent developments in processor design on the other hand indicate that both memory and processing power are becoming cheaper. Therefore, the need for efficient code generation may not be so pressing in the near future. In fact, the current generation of micro-processors is significantly more powerful than mainframes of only 10 years ago (e.g. 68020, 32032).

6.2.5 Standardization

There are several aspects to standardization. Some of them have to do with the design process, while others are related to the implementation technique. Standardization will first be considered in the design phases of program development and then a discussion of standardization in the software implementation phases will be given.

6.2.5.1 Standardization in the Software Design Process

The design of real-time software, and in particular mission critical real-time software, consists of a long series of decisions, each one bridging part of the gap between the high-level specification (or possibly even the requirements document) and the desired implementation with the required properties of resource utilization and performance. However, since this gap is too large to be directly bridged by a straightforward implementation of the requirements, there is a strong likelihood that a decision will be found to have unwanted consequences that could not have been foreseen at the time it was made. Therefore, there must be a mechanism to identify a wrong decision (in the light of problems occurring much later) and to backtrack to the point in the design where that decision was taken. However, this backtracking should be of a very subtle kind; i.e., whatever was learned after that decision was taken should not be discarded; rather, it should be used to guide the design activity from then on.

The efficiency of carrying out the various phases of the software development cycle is strongly dependent on the availability of tools to support the various steps. The use of a standard style of documenting (note that documenting is meant here in a very wide sense) will allow such tools to be designed and used for a wide variety of applications. It is an absolute necessity that there be a wide class of applications because of the high development cost of such very complex tools. Tools of this kind will prove very effective in verification and validation procedures, the successful completion of which is essential for flight critical software. A documented design history, including negative decisions, will undoubtedly simplify the correctness proof obligations significantly.

Another aspect of design standardization is the reuse of previously developed software components. Such components will have been proven correct in a given context. The proof of correctness obligation of the new design with respect to that component reduces to the proof that the context in which it is going to be used is indeed in agreement with the context used in the original proof of correctness of the component.

The use of standard components thus simplifies both the design, in the sense that a potentially large number of decisions need not be taken, and the verification of its correctness.

6.2.5.2 Standardization in the Implementation

A somewhat grey area between design and implementation is the use of facilities provided by a run-time kernel in the target configuration. If such facilities can be standardized, it will affect both the design stage,

by providing primitives in terms of which the design must be realizable, and the implementation stage. The implementation actually benefits from both the predefined availability of these functions, just like the use of predefined components eases the overall implementation burden, and the absence of a need to implement (similar) functions in a new kernel.

A more direct benefit may be expected from standardized software modules that can be used directly in an implementation, either through a configuration process, or by means of an architecture that is capable of automatically providing the right environment for such modules.

6.2.6 Software Re-Use

An obvious approach to reducing the cost of software production and the problems in demonstrating program correctness is to make use of software that has been constructed in a previous project. If it can be guaranteed that the interface requirements of such a software module are met both syntactically and semantically, then there is no need to further demonstrate the correctness of at least that module. However, currently there is generally not enough information available about existing modules to determine whether interface requirements are indeed fully met.

The potential benefits of re-use are so large—probably the single most effective cost reduction factor in all of software engineering—that modern programming languages should provide facilities for control of the syntactic interfaces with some minor help for the semantic correctness of module use even though compilation speed is negatively affected.

Unfortunately, much research is still needed before methods and related tools for software design will become available that effectively exploit available modules in designing new software. The fundamental problem is determination of the suitability of modules from a library for the design. Even if a module were perfectly adequate to solve a particular problem, it might be difficult to recognize it. Much more problematic is the case where a module is somewhat less than ideal, but could nevertheless be used through careful design of its environment. Essential in this determination of a module's suitability for a design is the availability of computer readable documentation on the module that is sufficiently complete. The documentation tools mentioned in (6.2.2) could conceivably assist in the generation of such documentation.

In this connection, the practice that has been established at Toshiba in Japan is very interesting. There, designers receive an award if modules submitted by them are accepted for inclusion in a library. The next step may be to encourage actual use of modules from that library. Even though only recently implemented, the scheme has already led to significant re-use of software modules at Toshiba.

6.3 Management of Complexity

All efforts, methods, and tools to limit the complexity of the programming task will not prevent the actual programs from being more complex than people can handle. Better design methods and tools will undoubtedly lead to greater demands on system performance and functionality, and thus, implicitly, to greater system complexity. Hence, even if the design complexity can be kept to an absolute minimum, methods are needed to deal with the remaining non-negligible amount of complexity. The previous chapters quite adequately cover these aspects as far as tools currently available or known to be feasible are concerned.

In a similar vein, some powerful new tools for the management of complex designs can be expected, mainly in the area of assisting management in estimating design and implementation progress. Very few methods currently exist to accurately assess the status of a design and implementation at any given moment.

There is a great potential in employing expert systems to draw (tentative) conclusions from the available information in the project library. A crucial element is that the documentation be developed in a computer manageable way; e.g., as hinted at above. The development of such (complex) expert system based tools should eventually lead to reliable estimation of project cost and turn-around time at the early stages of a project.

Related to this is a prototype building tool that could be used in several ways as described below:

- To reach agreement with customers on system functionality. In particular, difficult areas like man-machine interfaces could be studied much more thoroughly than is currently possible before implementation of the system.
- To determine difficult areas in the design and, at later stages, outright design errors. Errors in the design generally manifest themselves in difficulty of implementation. A prototyping tool—having knowledge of the various notations used in the different design stages—should be able to spot such problem areas. It should then be possible to backtrack and make alternative design decisions and, under guidance of the tools, be able to deduce whether the sort of problem that caused the backtracking is likely to occur again.

The use of prototyping in spotting design problems in as early a design stage as possible involves rather more than the application of a number of tools. In particular, a designer may want to specifically study certain aspects of a design, like control structures, data flow, or distributability of functions. Such emphasis on particular aspects of an incomplete design can only be realized if prototyping tools are available that take care of other aspects in such a way that the resulting prototype faithfully models the aspects to be studied. The designer may be required to make decisions for the sole purpose of creating the prototype. This can be almost as difficult a task as the actual design unless the prototyping tool has sufficient knowledge to at least suggest possibilities.

Prototypes are normally used for either of two purposes; namely, the analysis of system requirements or the study of design problems. Their use in requirements analysis has been rather successful and seems likely to grow in popularity. For example, by using them at a very early stage in a project, essential problems with the system requirements are identified and can be solved prior to the actual system design stage in which changes are far more costly. Although prototyping has recently become fashionable for design, the experience gained in building a prototype is not normally made available in such way that the design activity directly profits from it. Neither is there accumulation of experience and knowledge which would help to improve future designs. Expert systems should provide a mechanism for achieving such "corporate learning".

Such accumulation of design knowledge is not only possible in the context of prototyping, but can also be based on careful study of good design documentation. An accessible, preferably computer accessible, database of case histories will prove an immeasurable asset for both the technicians involved in a new design (or a redesign of an existing system) and their program managers. The latter can base their appreciation of the status of a project on histories of similar projects with much more confidence than in current practice where the essential details of progress and problems are simply not accessible (if at all available).

Testing of (partially) implemented designs is often cumbersome and therefore done in only limited depth. Major problems are the design of adequate tests for a given module and the creation of the required environment for the module.

The design tools proposed in the previous section will make the generation of both the environment and the tests to be performed amenable to (partial) automation. The completeness of tests performed, as well as more standardized procedures for test evaluation, will result in more reliable software. Note that the computer assisted design itself assures better designs with greatly reduced risk of hidden semantic errors. Note, however, that a test can never guarantee a module's correctness. The only approach to truly secure software lies in the application of mathematical rigor in proving that transformations from specification to implementation preserve correctness. Unfortunately, even if such correctness can be proven there remains the simple typographical error that leads to neither syntactic incorrectness nor semantic nonsense, and thus can never be automatically detected (e.g., the use of "-" instead of "+"; note that "/" instead of "*" might be detected through some variant of dimension analysis).

6.4 New Techniques

The nature of both the software and its development will quite drastically change in the future because of these developments:

- The cost of hardware is declining steadily.
- The size of a standard amount of processing capability is decreasing.
- The techniques, collectively known as Knowledge Based Systems (KBS), are becoming feasible for real-time applications.

6.4.1 Very Fast Machines

One obvious way to increase the power of embedded systems is to increase the power of the computing elements in such systems. The straightforward approach is to develop faster, more powerful machines, so that programming embedded systems basically remains what it used to be, the advantage being that no new techniques of dealing with multiple processors need to be developed. There are two techniques that can be used to significantly increase the power of a processor, each of which is discussed briefly below.

6.4.1.1 High Speed Single Processors

A single processor can be made faster in essentially two ways:

- By increasing the basic processor speed through good design and decreasing the size of the processor. The increase in speed through reduction of size is probably not going to be very impressive, certainly not orders of magnitude. The increase that may result from designing for speed can be more impressive, but has the drawback that the design complexity, and thus the design cycle, will grow beyond acceptable limits, at least for the current generation of powerful processors (e.g., Motorola, Intel, National Semiconductor). However, it is becoming clear that simple processors (Reduced Instruction Set Computers, RISC) can be made to execute code so fast that the greater number of instructions needed for a typical operation is more than offset by the increase in speed. Thus, RISC machines may play a significant role in the not too distant future. It is perfectly possible to produce compilers that will generate efficient code for these new RISC machines.
- By incorporating language support directly in the silicon. This has not proven to be very successful yet. The NS32000 series very nicely implements certain language features, but is not faster than e.g., the MC68020. The famous iAPX432, which was the almost ideal Ada machine, totally failed. Rational's R1000 is a very fast machine and has an architecture that is optimized for the execution of ADA programs.

In general there seems to be a relation between the architectural features of a machine and the structure of programming languages to be supported. The more dynamic the features of the language, the more complex is the machine. This seems to indicate that languages with late binding should execute fast on special machines. This is indeed the case. LISP machines execute programs much faster than general purpose machines. The current trend in RISC machines, which often equal the performance of special machines in execution of languages with only static language features, will probably be balanced by faster implementation of these special machines.

6.4.1.2 Parallel Processors

The interesting question "How to build parallelism into a processor that effectively results in faster program execution" has not been answered in any conclusive way yet. Many attempts have been reported, each of which has advantages for a particular class of problems, but no general solution has been found yet. The essential architectures of such parallel processors can be summarized as:

- **Dataflow Machines**

The processor consists of a great number of very simple processing elements, each of which is activated by the arrival of all necessary data to perform its particular (programmable) function. The result of that computation is sent to other processing elements for further processing. It appears that, although dataflow machines are a very powerful principle, there always is a scheduling problem in the interconnection network that severely limits the performance achievable by these machines.

- **Reduction Machines**

A very particular type of machine that is mainly geared to the efficient execution of applicative languages is the reduction machine. The absence of side effects in these languages permits very elegant transformations to be performed that are exploited by such reduction machines. At present, however, no successful machines of this class exist.

- **Array Processors**

These machines are particularly good at tasks in which large arrays of data need to be processed in exactly the same way. Typical examples of such tasks are vector operations where, for example, the addition of two vectors consists of performing the same operation a great number of times. Array processors are ideal machines for solving problems of this sort, but far from ideal for all other types of problems. It seems possible that array processors may be used as special computing devices for very special tasks in real-time systems (e.g., the numerical integration of differential equations). Well known examples of array processors are so-called supercomputers such as the Cray-1, DAB and Cyber.

6.4.2 Multi-Processors

A special class of architectures is formed by machines that consist of large numbers of very intimately coupled processors. Generally, these systems require very special programming styles in order to exploit the parallelism. Thus far, relatively little experience has been gained with programming machines of this sort. Some examples of multiprocessors of this sort are:

- **Systolic Arrays and Similar Arrangements**

A number of such processor arrangements have been studied over the past few years that have interesting properties. It appears that the performance of these systems depends critically on the application; i.e., problem solutions that map well onto the processor structure will gain enormously over the single processor implementations, whereas others may not exploit the possibilities at all. It is not quite clear yet which problems are best suited for a particular processor arrangement.

A general problem with these architectures is the availability of programming languages that fully exploit their advantages. Not only is it very difficult to automatically allocate functions to processors in the architecture, it is also very difficult to prove (even in a loose sense) the correctness of the compilation system. Note that even with traditional single processor systems, there is as yet no way in which language compilers can be demonstrated to be either correct or incorrect. The best example is the ADA validation procedure. Many validated compilers will, given a non-trivial program, fail to produce code at all—which is not a great problem—or produce erroneous code—which may be a great problem. Compilation for complex multi-computer architectures is more difficult by an order of magnitude. By the same token, programming these machines in assembly language is not desirable either, as it is already doubtful in the single processor case whether correct programs of some complexity can be constructed using notations at a very low semantic level.

Examples of multi-computer architectures are the Hypercube, the various topologies that can be built using the Transputer, and other similar architectures.

- **Multiple Processor to Memory Mapping**

The assumption underlying multi-processor machines is that, although all processing elements must be able to access all the available memory, it is very unlikely that two such elements will require access to the same (group of) memory locations simultaneously. It is then advantageous to interconnect the processing elements and the memory by means of a complex switch that is capable of routing a great number of memory accesses simultaneously. Whenever a conflict arises, the network autonomously schedules the accesses. The best known examples of such machines are the Ultra Computer of New York University and the HyperCube. Problems with the Ultra Computer seem to indicate that what works well on a small scale (up to 64 processors, say) may totally fail with large numbers of processors.

6.4.3 Distributed Architectures

The availability of cheap and powerful processors will permit systems designers to structure the target processing system in the most suitable way for a particular (class of) application(s). Such architectures will have different characteristics, depending on the topology of their interconnection network, on the communication strategy, etc.. The selection of the most suitable architecture is far from trivial because, apart from considerations about the real-time behavior of the architecture, certain aspects of developing software systems for these architectures must be taken into account. For example, if special tools must be developed for distributing functions over a number of processors in a distributed system, then that architecture and the associated tools must be useful in a fairly large number of applications to offset the tremendous cost of their development.

Assumed in this discussion about architectures is the possibility of exploiting concurrency in the execution of the application program(s). If a system's program consists of a single cycle in which all activity is done, uninterrupted by unpredictable events, then there is little to be gained from a parallel architecture. The only way to achieve an increase in performance is to use the most powerful processor available and to code it as efficiently as possible. For more elaborate systems, e.g., those that need multiple tasks to execute in parallel (at a conceptual level at least), it is necessary to design the program as a parallel program, rather than design it as a sequential program to which, at some stage, concurrency is added. At the moment few, if any design methods exist that directly support the development of distributed parallel programs. A number of architectural possibilities for distributed processing will be discussed in the following sections.

6.4.3.1 Tightly Coupled Processors

An important characteristic of tightly coupled processors is their inter-dependence. This dependence can result from sharing of memory as well as from synchronization requirements between processes in the various processors. Shared memory makes it possible for parallel processes to reside in different processors and yet communicate directly through the shared address space. At the same time this facility makes programming the communication between processes as difficult as if they resided in a single processor. Moreover, the often necessary synchronization between processes desiring to access the same memory locations can severely limit the gain in processing speed that would otherwise be expected from the use of distributed multiple processors. Tight coupling also implies that a great similarity between the processors being used is desirable, at the very least. Vastly differing processors cannot simply be attached to the same bus, and furthermore, may have an entirely different view of how the memory is used. This would prevent effective sharing of data between such processors, and would therefore limit the possibilities of shared data access severely. It is not generally clear whether shared data is a good feature to use for communication between processes. Unless it can be convincingly demonstrated that the sharing of data will under no circumstance lead to illegal use or access of such data, it is probably better not to use it. The major way in which tightly coupled processors will be useful, then, is in their ability to very rapidly pass messages from one process to another.

6.4.3.2 Loosely Coupled Processors

There are many ways in which processors that share only a communications network can be made to cooperate. Most techniques are a variant of the object oriented paradigm in which processes access external procedures and data only through the use of messages. This approach does not make optimal use of the separation that is implicit in distributed architectures. In fact, with this approach a process must, very much like traditionally implemented systems, have global knowledge in order to ask for services and/or data. It is true that the physical location of the required services may not be known to any of the processes, but that only solves a configuration problem and does not reduce the complexity of process interaction.

It is possible to completely isolate the various processes through an associative communication mechanism (such a strategy might be characterized as "procedure abstraction"). The processes then need not know where services are performed or where data is used or produced. This isolation of processes may lead to totally standardized processing components that can be used in different configurations without any change. In this case, the development of a system then becomes a matter of configuring components into a useful system. For that purpose, design tools will have to be built that will be very different from the sort of tools that are needed for traditional system design. In general, this configuration problem was described by DEC for configuring VAX computers.. However, it seems well within the current possibilities. This use of standard functional

units (small, "technical" rather than "operational" functions) should greatly enhance system reliability in a number of ways.

- The functions themselves will be small and therefore well understood.
- The functions will need to be programmed and proven correct only once and used thereafter without having to modify them.
- The architecture will allow spare processing capacity to be included in very elegantly.
- System monitoring will be integrated with the standard communications mechanism. This should allow adequate handling of hardware errors.

6.4.4 Languages

Alternative implementation languages such as functional and declarative languages will change implementation strategies as soon as sufficiently powerful special processors become available. Already announced, and expected in 1986, are extremely fast LISP chips that will easily surpass the performance of today's most powerful LISP machines. Implementing parts of systems in LISP will then become feasible and, for certain functions, will be very attractive. However, little, if any, work has been done in developing design strategies and implementation methods for functional languages. Now that hardware is becoming available for the real-time use of functional and possibly declarative languages, there is a serious danger of these languages being used without sufficient methodological background. This would, in a sense, take us back to the dark ages of programming! Nevertheless, it would be totally irresponsible to ignore such developments because quite a few problems can be solved much easier using functional or declarative languages than with traditional, imperative languages. Examples of features offered by a number of functional languages which are not found in imperative languages are:

- Lazy evaluation which allows infinite objects to be manipulated as well as partial functions to be used.
- Higher-order functions that produce functions as a result of their invocation.
- Very powerful list processing through which relations between objects can be modeled very effectively.

Another aspect of the use of functional languages is the inherent parallelism in evaluating functions. This is mainly due to the absence of side effects in ideal functional languages. Side effects are those effects of evaluation of a function that are not passed as a result of invoking the function. Thus any change to a global variable or any input/output is a side effect in the context of functional programming. Such side effects have a detrimental effect on the parallel evaluation of functions.

Unfortunately, so far as is known no language can be used for real systems without facilities to produce side effects (at least those due to I/O). It is therefore questionable to what degree the parallelism of functional languages can be effectively exploited in real systems.

6.4.5 Artificial Intelligence

The steadily increasing requirements with respect to handling of complex situations can, even today, hardly be met with the traditional implementation techniques. A very loosely coupled distributed system as indicated above will not help very much in this respect, although implementing a system once conceived may be virtually impossible with traditional architectures while remaining possible in this style. It appears that a possible solution to these problems may be expected from the application of knowledge based systems.

In general, knowledge based systems allow partially known solutions to particular problems to be implemented in an effective way, a property not normally attributed to algorithmic solutions. The penalty paid for this feature is a tremendous reduction in processing speed (if a KBS implementation of a given problem is compared to a traditional one).

Use of Knowledge Based Systems can be envisaged for both the system design phase and for real-time execution as part of the target system.

6.4.5.1 Use in the Design Phase

The many aspects of designing a program (and possibly even a total system) are traditionally handled by experts that have great skills and knowledge, experience, or insight. This situation is not likely to change dramatically with the introduction of integrated support environments. The various tools in an integrated support environment will all operate on the same database, but most likely they will not provide an integrated view of the entire design process. It seems likely that a great increase in productivity and quality could be achieved if all the design aspects can be integrated in a support environment in such a way that all the ramifications of decisions in one area are automatically derived and reported to the other areas. In such a system the design of tests for software modules would be directly linked both with the design decisions taken for those modules and their intended environment. Such an integration could also support the derivation of planning requirements for equipment to be available when testing must begin.

Obviously, a powerful environment like this cannot be realized with the current style of programming. Far too many dependencies between design and management aspects are vague rather than clear-cut, and therefore they escape traditional implementation. The declarative style of programming that is emerging seems to provide an alternative that, in due course, may be sufficiently powerful to usefully implement at least a significant fraction of the desired functionality. The main reason for this increase in power over traditional methods lies in the higher semantic content of the primitives used for implementing functions and relations. Fundamentally, there is nothing that can be done in this style that cannot be done with a traditional approach. The difference is one of human limitations. The situation is somewhat similar to the difference between machine language coding and the use of a high level language.

This observation is important in that expectations of what can be achieved using these techniques should be realistic, rather than overly optimistic. It is almost certain that the current generation of expert systems will prove to be rather more limited than is claimed by many vendors. In particular, the reliance of these systems on very simple reasoning schemes will prove totally inadequate for the modeling of uncertainty and ignorance in a truly useful way. Nevertheless, quite interesting results can be expected from the use of expert systems in project support environments if only because it will focus attention to the more global problems in designing complex systems.

6.4.5.2 Real-Time Use

The current state of the art in artificial intelligence and knowledge based systems is such that certain types of problems in not too dynamic environments can just be executed in real-time. For faster changing contexts and for more complex problems, the discrepancy between what is currently possible and what is needed is still enormous: an order of magnitude rather than a factor of two. Nevertheless, current progress is rapid, and expected future understanding of where most of the time is actually spent in the execution of KBSs should lead to a drastic reduction in the processing time through clever exploitation of parallelism and more ingenious basic algorithms used in the KBS execution machine.

The problems related to the use of KBSs for real-time systems are manifold, however, and not limited to those of execution speed. At present no general methods are known to deal with any of the following classes of KBS problems:

- **Consistency of Knowledge Bases**

All KBSs have the knowledge to be used for the solution of a class of problems explicitly made available to them in a so-called knowledge base. Generally, these knowledge bases consist of numerous rules relating circumstances to activities. Large knowledge bases tend to become rather difficult to control in the sense that nobody can be certain that new rules added to knowledge base will not invalidate others already there. It seems possible that tools can be developed to check internal consistency, although this is a rather vague area if rules-of-thumb are added that are known to have a limited validity but have proven to be practical. Extensive research is still needed in this area. Note that there are similar problems with more traditional implementations, but such problems hardly ever get to the surface because use of such rules is hidden in the algorithms used.

- **Completeness of Knowledge Bases**

This is an even more phenomenal problem than consistency. To a much greater degree than in traditional implementations of systems, in knowledge based systems it is possible to build systems for which essential details of how to solve the problem are still missing. This raises questions as to what the actual power of such a system might be. Also, there is a danger that such a system will be used to solve problems that are not actually within the boundaries of its knowledge, especially since those boundaries are only vaguely known. The interpretation of results produced by a system in such circumstances is a difficult matter and may be utterly impossible! Note that in current practice decisions are often based on intuition rather than provable deduction.

- **Deletion of Old Data**

In any real-time system it is necessary on the average to delete data at the same rate the environment supplies it. The problem is the determination of which data will not be needed at a later stage for re-interpretation in the context of changed world models.

Keeping around great amounts of old data has the disadvantage of clogging up the system to the point where performance losses may be expected. Typically, sets of certain sorts of data have to be maintained at all times in these systems, and the larger the sets, the more processing has to be done.

- **Dealing with Data that Cannot Satisfactorily be Explained Using the Available Models of the Environment**

Models always embody simplifications with respect to the real world. This means that certain phenomena observed will escape explanation in terms of the model. In that case, the traditional approach is to discard the observation as probably erroneous. However, a far better approach would be to conclude that the model is inadequate and consequently needs to be improved (cf. the situation with scientific theories and experiments). Neither the necessary model building knowledge required nor the methods of using it are likely to be available in the near future. A system capable of improving its own performance through enhancement of the underlying model of the environment is truly a learning system. Although such learning systems are a possibility in theory, they are not likely to become practical for a long time. However, it is well possible that through interaction with the system's users, such model enhancements can be realized without having to completely rebuild the system or major parts of it.

6.5 Conclusions

Three main streams of developments will have a significant effect on the design of systems and in particular, on the software component of those systems.

- **The Strong Tendency Towards Distribution of Processing**

This tendency has both potential benefits and potential dangers. The problem of communication between the distributed processes requires very careful consideration. If taken too lightly, the problems of designing such systems will be greater than they used to be in single processor systems, rather than smaller. Problems associated with parallelism that never arose in traditional single processor systems in comparable degrees are:

- **The Meaning of Time**

Programs cannot rely on predetermined ordering of events in the same way as they used to.

- **Deadlock**

Deadlock is a much more serious problem in a parallel system, and can only be prevented by a formal proof of its absence.

- **Distribution of Functions Over the Various Processors**

Should there be a static, dynamic, or semi-dynamic assignment of processes or processors?

- **Testing and Verification**

Testing and verification of a parallel system are largely unexplored domains. However, the potential benefits of parallelism are such that careful study of the problems and their possible solution is well warranted. To list only a few such benefits:

- **Technological Innovation**

Technological innovation is greatest in the area of integration. This implies that processing modules should be small in order to selectively replace modules with newer, more suitable ones.

- **Distribution**

Distribution, if handled well, permits reliability in a very natural way.

- Provided processes are not too tightly coupled, parallel systems can easily be modified and extended.
- Demonstration of correctness can be significantly easier.

There will undoubtedly be a clear impact of parallelism on design strategies. Most important will be the change in thinking about programming. To date, parallelism has been considered as an extra feature, with the basic assumption being sequential execution. This will definitely have to change if the benefits of parallel processing are to be collected. For example:

- Formal methods for systems and program specification will have to become available. Tools for semi-automatic transformation of those specifications to constructable hardware and executable programs may be expected in the not-too distant future. Such tools will greatly reduce the implementation effort and will also significantly simplify the validation procedures. Until such methods and tools are available, good tool-assisted documentation will help reduce the cost of building complex systems through both learning from documented experience and re-use of available software modules.
- Finally, application of techniques developed in research in artificial intelligence will lead to dramatic changes in system concepts. To a much greater extent than is currently done, explicit models will be used as a basis for a system's activities. The result will be more flexible systems which are capable of autonomous behavior in situations where current systems only offer advice to users. The software complexity of these systems is still so great that it is not likely that the techniques will be used for flight critical software for a long time to come. Note however, that next year the first field trials of simple AI systems in a defense context are expected.

EXECUTIVE SUMMARY

Introduction

This Executive Summary provides an overview of the formal report of Working Group 08 of the AGARD Guidance and Control Panel. The Sections of the summary correspond to the chapters of the report for ease of reference to the material being summarized. It is hoped that this Executive Summary will serve those individuals who might not have time to read the full report or who wish to preview the report prior to reading it in its entirety.

The Terms of Reference for the efforts of Working Group 08, as approved by the National Delegates Board of AGARD, were:

- (i) To develop and consider a set of requirements for a high level language support environment from a guidance and control system point of view.
- (ii) To evaluate the characteristics and capabilities offered by advanced language environments, either existing or in the course of development, with respect to the requirements defined in (i) above.
- (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i).

Item (i) of the Terms of Reference is addressed in Chapters 1 and 2. Chapter 1 discusses the key issues which must be considered in overall guidance and control system design and implementation with special emphasis on those issues which are unique to high integrity, flight critical software. Chapter 2 completes the development begun in Chapter 1 and considers requirements for high level language support environments for guidance and control systems software design and implementation by providing a detailed elaboration of the phases, tasks, and methods associated with current guidance and control system design and implementation practice.

Chapters 3 and 4 address Item (ii) of the Terms of Reference. Chapter 3 provides a descriptive summary of software design and development technology which existed or was known to be under development at the time the report was prepared (circa 1986), with special emphasis being given to those software technology developments being carried out in connection with the development of the ADA and LTR3 high level languages and their related support environments. Chapter 4 discusses the general limitations of the software tools and software engineering environments described in Chapter 3 from the perspective of the requirements described in Chapters 1 and 2.

Item (iii) of the Terms of Reference is addressed in Chapters 5 and 6. Chapter 5 provides a description of the desired characteristics and capabilities for a class of new software engineering environments which exploit the use of ADA or LTR3 and the modern object-oriented software design techniques which they support. An environment incorporating these desired characteristics and capabilities would represent a major improvement over earlier tools and environments as described in Chapter 3. Finally, Chapter 6 considers future trends in computer hardware and software technology and their promise and limitations relative to the design and implementation of high integrity, flight crucial guidance and control systems.

General Requirements for Guidance and Control System Design and Implementation

A set of general requirements for design and implementation of guidance and control systems, including requirements for software, software tools and software support environments, is provided in Chapter 1.

Probably the most important and challenging of these requirements derives from the flight safety critical nature of guidance and control systems. This leads to requirements for extremely high integrity of major parts of the overall guidance and control system, including major parts of the software, which lead, in turn, to related requirements for both fault avoidance and fault tolerance in guidance and control system software design and implementation. Software fault tolerance is required because the best fault avoidance techniques

available for software design are unable to guarantee the integrity required using fault avoidance techniques alone.

There are two trends occurring in aircraft/avionics system design which will place even more importance on software fault avoidance and fault tolerance in the future: Increased use of relaxed static stability and increasing integration of avionics and control functions. The latter include integrated flight control/navigation, integrated flight/propulsion control, and integrated flight path management for low altitude missions. Integrated flight control/navigation will lead to use of dispersed sensors and sharing of navigation sensor data with flight control, and it will result in use of more complex flight control algorithms for sensor normalization, redundancy management, and survivability through dispersion. Integrated flight/propulsion control, including use of vectored thrust, will result in sharing of information between engine and flight control systems and will make such integrated control systems flight crucial. Integrated flight path management at low altitudes will require real-time optimal control for terrain following/terrain avoidance and for complex real-time route decision, making low altitude flight management, sensor blending, and (huge) terrain databases used flight crucial.

The extremely high integrity required of guidance and control software leads to the need for visibility of the guidance and control system design in the flight software itself and for corresponding high integrity of software tools and environments used in the design, implementation, and testing of the flight software. Both of these requirements are important to the tasks of verification, validation, and certification of flight safety critical guidance and control software. Because of deep concerns for these issues of visibility of design and integrity of tools and environments, as well as the need for high performance in terms of processor loading and memory utilization, most digital guidance and control systems developed to date have had their software written in assembly language. Clearly, any future consideration of the use of high level languages and their related tools and software engineering environments must include careful evaluation of these two important requirements for design visibility and tool and environment integrity.

In addition to requirements for extremely high integrity, there are other important requirements that derive from the fundamental nature and characteristics of guidance and control systems. These include:

- Stability of feedback loops which require hard constraints on computational latency.
- Multimode and cyclic multiloop operation which require multiple rate task scheduling with hard real-time execution constraints.
- Achievement of specified performance in the face of uncertainties and complex variations of vehicle dynamics which require use of robust algorithms.
- Flight near vehicle operational limits which require real-time monitoring of critical flight variables and automatic invocation of algorithms necessary for safe recovery of the vehicle.
- Interfacing with crew stations and various required sensors and actuators which require time critical and complex I/O operations with obvious implications regarding specialized operating system/executive design and performance and the run-time environment in general.

Currently used guidance and control system design techniques and practices also lead to special considerations and requirements. For example:

- Flight control design is control law algorithm driven, which precludes strict top down development.
- The overall airplane/system design is an iterative process which leads to an increased incidence of unavoidable guidance and control system requirement changes.
- There is significant post-integration development which implies a need for built-in modifiability and support.
- Guidance and control systems typically use specialized hardware and software which often leads to requirement for tailoring of methods.
- Safety criticality requires optimization and complementarity of methods.
- Customer/regulatory review requires visibility and credibility of the evolving design.

As regards program size and complexity, it is expected that incorporation of relaxed static stability, gust load alleviation, active ride quality control, flutter mode control, and integrated control into a basic guidance control system design will lead to only modest increases in the size and complexity of software for inner-loop control. The use of terrain following/terrain avoidance, AI decision making, integrated control, and optimal flight path control can all be expected to lead to significant increases in software size and complexity and size whereas AI based redundancy management techniques can be expected to lead to significant increases in both

size and complexity of flight crucial software.

Finally, it can be inferred from the above that software development methods should not be completely divorced from the methods used to develop complete guidance and control systems. Specifically, it is important that the software development methods used:

- Be problem driven and application based.
- Deal conveniently with combined software/hardware interactions in concurrent mechanizations.
- Address complexity and integrity concerns explicitly and quantitatively.
- Effect component integration during design.
- Include provision for quality assurance.
- Provide a natural transition from system requirements to software design to hardware and software implementation and to their integration.

Phases of and Methods for Guidance and Control System Design and Implementation

The design and implementation of flight guidance and control systems is an exacting and complex process which is best carried out in phases using appropriate methods for each specific phase. Chapter 2 describes a set of phases and methods which are representative of current practice.

As described above, the life cycle of guidance and control systems (including software) is dominated in all phases by considerations of reliability and integrity, hard timing constraints, and the need for flight certification of the overall system including the software. Thus, while guidance and control systems software design and development includes all of the life-cycle phases common to most software projects, special requirements, methods, and interfaces between phases have to be developed in order to cope with these special considerations. As a result, guidance and control systems - which are constructed from unique combinations of redundant data transmission (exchange) systems, sensor systems, guidance and control computers and associated software, and actuation systems - must satisfy stringent requirements on timing, configuration and interfaces, integrity, performance, reliability, survivability, and supportability and ease of modification.

Typically, guidance and control systems software consists of flight resident (airborne) software, commonly referred to as the operational flight program and the flight control program, and non-flight software are important in the overall systems development process.

It is pointed out in Chapter 2 that the overall process of development of any system can be logically divided into seven relatively distinct phases, referred to as the study phase, the concept development phase, the system design phase, the system development phase, the system integration phase, the production phase, and the in-service phase. The system design, system development, and system integration phases of this seven phase model must be modified and expanded in the case of guidance and control systems because of the criticality and safety aspects of such systems and the presence of significant hardware/software interactions. In this modification and expansion, the system design phase is replaced by five phases, referred to as the system requirements phase, the system design phase, the subsystem requirements phase, the subsystem design phase, and the software requirements phase. Similarly, the system development phase is replaced by a basic software design phase, a detailed software design phase, a coding/module test phase, and a software integration on the host computer phase; and the system integration phase is replaced by a software integration on the target computer phase, a subsystem integration phase, a system integration phase, and a flight test phase. Except for the flight test phase, this expanded phase model is appropriate for most systems implemented using both hardware and software.

It is shown in Chapter 2 that the modified and expanded set of phases described above is representable in two particularly useful forms. In the first, strictly top-down, hierarchical form, those phases which are common to both the hardware and software aspects of overall system development and which must be carried out essentially sequentially are arrayed in a strict vertical hierarchy, while those that can be carried out concurrently and independently for the software and the hardware parts of the system are arrayed in two parallel paths in the vertical hierarchy. A commonly used representation of the life-cycle phases of software design development and implementation which is isomorphic to the first form described above is the well known "waterfall chart" in which the phases are arrayed hierarchically in descending ranks to the right. In a "waterfall chart" representation, the parallel paths referred to above would be arrayed as parallel waterfalls

in an overall system of cataracts.

In the second, even more useful form, the phases are arrayed in the form of a "V". Those phases which correspond to top-down decomposition, design, and implementation are arrayed in descending ranks along the left or descending side of the "V" and those phases which correspond to successively higher levels of integration and test are arrayed in ascending ranks along the right or ascending side of the "V". The usual iterative nature of requirements generation and of design and implementation is reflected in the "V" chart as shown on the chart given in Chapter 2.

The "V" chart has several distinct advantages over the waterfall chart. First, those phases which are associated with top-down design and implementation are represented in the usual top-down form used in waterfall charts while those phases that are more properly viewed bottom-up, such as integration and test, are represented in a bottom-up form which is not possible in a waterfall representation. Second, those bottom-up phases which relate to specific top-down phases appear at the same level on the two sides of the "V" permitting direct visual association of related phases. This is particularly useful in indicating the levels at which verification, validation, and certification occur in the bottom-up process of integration and test. This is because the top-down phases which generate software/hardware specifications to which the results of software/hardware integration phases are to be verified are on the same level in the "V" chart. Similar direct horizontal associations hold for validation and certification. Finally, by including the higher level phases of studies, concept formulation, and certification requirements on the left side of the "V" and the flight test, production, and in-service phases on the right side of the "V", software development for guidance and control systems is placed in a proper context and not isolated out of context as is the case with the usual waterfall chart.

Chapter 2 also describes the essential tasks to be performed during each particular phase on the "V" chart and outlines the essential contents of appropriate specifications and reports to be generated. These tasks are summarized below.

The essential tasks of the Study Phase are to investigate the characteristics of the vehicle and the feasibility of achieving satisfactory guidance and control system behavior for it, to analyze physical configuration and layout possibilities together with existing equipment and technologies, and to generate initial estimates of costs, risks, and development duration.

The tasks of the Concept Development Phase are to define and evaluate alternate system concepts and to select the best overall concept based on performance, design and development risk, cost, and reliability and fault tolerance characteristics.

Production of a System Specification which incorporates all known customer and other requirements and defines the requirements for all subsystems in a comprehensive functional form is the main task of the System Requirements/Systems Design Phase. The System Specification should include a requirement for a failure mode and effectors analysis together with a statement of test philosophy and strategy. A general model for a System Specification document for guidance and control systems is proposed. In addition to the System Specification, a System Architecture document describing the entire system architecture, the connections and interfaces between all subsystems, and preliminary installation data such as weights, size, and envelopes should be generated during this phase.

Similarly, production of Subsystems Specifications for all subsystems defined in the System Specification is the main task of the Subsystem Requirements/Subsystem Design Phase. Each Subsystem Specification should contain a detailed functional description of the subsystem; a description of the subsystem architecture and design concept; the subsystem criticality classification; all requirements on sensors and effectors, processors, and displays and controls; allowable delay/dead times; allowable and/or required parallel activities; requirements for synchronization of control loops; required bus traffic; a detailed description of the subsystem's interface with all other subsystems and the system as a whole; and test requirements for equipment, software, and subsystem and system integration.

The major task of the Software Requirements/Hardware Specification Phase is to complete subsystem design through further stepwise refinement of the subsystem requirements and to allocate subsystem functions between hardware and software components based on trades between software and hardware implementations of individual functions. The results of this phase are appropriately summarized in Software Requirements Documents, Equipment/Hardware Specification Documents, and an Interface Definition and Control Document, all of which are placed under configuration control.

During the Basic Software Design Phase, the software requirements for a subsystem are transformed into a set software module designs and a related set of module test designs. Priorities, program flow, individual module functions, individual module interfaces with other modules, and timing and module calling sequences are all defined during this phase. The results of this phase are a set of Software Module Specifications and a set of Module Test Requirements Documents which are put under configuration control.

Complete definition of the algorithms to be implemented, the data structures to be used, and the internal module structure are the main tasks to be accomplished during the Detailed Software Design Phase. The internal module structure is usually described in pseudo-code or an appropriate program definition language. The pseudo-code for the modules, detailed interface and data descriptions, cross-reference listings, and module test specifications. All of this documentation is placed under configuration control.

During the Module Coding/Module Test Phases, the modules are coded and commented, syntax errors are debugged, code inspections and walkthroughs are performed, module tests are prepared, test stubs and drivers are established, static and dynamic module tests are executed, and detected errors are corrected. The results of these phases are documented in the form of program/module descriptions, data descriptions, interface descriptions, source code with comments, module link descriptions, test data, descriptions of program stubs and test drivers, descriptions of test protocols, and test reports. All of this documentation is put under configuration control.

The main task of the Software Integration on the Host Computer Phase is to integrate the individual modules one at a time into software subsystems and into the complete software system on the host computer and to systematically test the growing software packages in accordance with software integration and test protocols developed during this phase. The completed and tested modules are placed under configuration control along with the Software Integration Test Protocol Document which describes the protocols used.

During the Software Integration on the Target Computer Phase, the software is compiled into code for the target processor and loaded into the target computer. Then the software components that handle hardware interfaces are integrated one at a time and systematic functional tests are performed to validate the software integration on the target computer using a simulation of the external environment. Also, execution times of the modules and the overall program are confirmed by direct measurement and/or code analysis. The results of this phase are documented in a Target Computer/Software Integration Report which is put under configuration control.

In the Subsystem Integration Phase, the subsystems are tested and the behavior of all flight control functions are investigated on a rig. Also, subsystem performance is validated against the Subsystem Specification requirements.

Similarly, in the System Integration Phase, the task is to validate the fact that the requirements of the System Specification are satisfied. Also, any required integration of flight resident and non-flight software is performed during this phase.

The task of the Flight Test Phase is to carry out all flight tests required to certify the performance and safety characteristics of the overall guidance and control system.

During the Production Phase, the flight resident software is copied and transferred to the target computers and tests are performed to assure the integrity of the copying process.

In the Postdevelopment Support Phase, errors discovered during system operation are corrected and desired additional capabilities are incorporated into the system. The phase model described above should be applied to both of these activities.

Chapter 2 also describes requirements for both general methods and specific phase dependent methods to support guidance and control system design and implementation. Requirements for general methods include: (a) support for structured decomposition of the system and software in terms of data flow, data descriptions, process descriptions, time dependencies, criticality of functions, parallelism of tasks, and test strategies, (b) consistency of methods based on a single semi-formal language, (c) methods for easily incorporating changes, and (d) an integrated set of tools which enforce the strict adherence to the methods themselves.

As regards specific phase dependent methods, it is pointed out that methods for the Study Phase through the Software Requirements/Hardware Specification Phase should generate consistent, complete, easy to read and well-structured comments, should support parallel works of teams developing different parts of the overall system, and should be adaptable to the specific application domain. Furthermore, commonly used methods need to be modified to incorporate additional features. For example, it is noted that diagrams commonly used to provide functional descriptions of systems need to be supplemented by additional information. Moreover, the Structured Analysis and System Specification Method of de Marco - which supports construction of a functional model of the system consisting of a network of system functions, a list of interfaces in the form of a data dictionary, and a state transition model describing system states and the condition for changes in the system states - needs to be modified to include description of the dynamic behavior of the system by formal constructs that are more powerful than test and tables, to provide a formalism to represent safety critical parameters, functions, and time dependencies, and to incorporate the more expressive symbolism commonly used in the relevant technical disciplines.

Chapter 2 suggests that the method of Structured Design due to Yourdan and Constantine is state-of-the-art for the Basic Software Design Phase, that the use of pseudo-language and structograms a la Nassi and Schneiderman is state-of-the-art for the Detailed Software Design Phase, and that the principles of Structured Programming are state-of-the-art for the Coding and Module Test Phase. No additional requirements for improved methods for these phases are suggested, nor are any requirements for methods for any of the integration phases discussed. However, it is pointed out that new, improved domain specific methods are needed and that the current state-of-the-art in software tools and environments is not sufficient to support the existing and needed new methods for guidance and control systems design and implementation. Moreover, it is noted that the newer software engineering environments under development by the STARS program and the JSSEE will only realize their true potential if the tools they contain directly support the design and development methods used! Thus, implementation of Methodman in STARS is important.

The need for well-defined and standardized interfaces to appropriate design-oriented databases remains. Likewise, the area of effective methods for test, verification, and validation is also an open issue which needs to be aggressively addressed.

Chapter 2 proposes a general schema for all phase dependent documentation and all phase independent documentation which can be designed to be easily managed by a computer support library system. The phase dependent documentation is separated into overall system development documents, software engineering documents, and system integration documents. The phase independent documentation is separated into standards documents, quality assurance documents, and organization documents. Each of these six types of documentation is described in some detail. It is clear from the discussion that a proper set of documents to support the design and implementation of guidance and control systems throughout all phases of the process is indeed extensive and complex.

The final set of issues and measures discussed in Chapter 2 relate to project organization, project planning, project review, change and configuration control, and quality assurance. The discussion of organization issues describes the kinds of organizational groups typically involved in software development projects and the necessary organizational relationships between them. The section on project planning describes the nature of the planning process and the responsibilities of the various organizational groups in carrying out the planning process. The discussion of project review describes the various kinds of reviews which should be carried out, who should perform them, and what the reviews should verify. The section on change and configuration control describes the need for and nature of the process, the documents which are necessary, and the measures which should be taken to ensure the integrity of the process. The discussion of quality assurance describes the function of quality assurance, the need for an independent quality assurance team, the data and information they require to perform their function, and the level at which they should report their findings.

Current Status of Tools, Toolsets, and Software Engineering Environments for Embedded Computer Systems Software

The term software support environment refers to a collection of methods, tools, and procedures for the design, implementation, test, and life-cycle support of embedded computer software, applications such as guidance and control systems. The acquisition, management, and support of guidance and control software is

conducted according to internal defense policies which differ between nations, and environments are generally organized and built so as to support compliance with these policies. Thus, the details of environments can be expected to differ between nations in this respect. However, in most other respects, environments being developed by various nations have much in common even though they may vary in terms of coverage and state of implementation.

Chapter 3 provides a brief introduction to the acquisition, management, and support policies of the Federal Republic of Germany, France, the United Kingdom, and the United States. It is observed that there is no real standardization between the Four Powers in this area. Chapter 3 then discusses and describes the current status of: Software requirements specification and design methods; programming languages, tools, and environments; tools for software and system integration and testing, tools for software verification, validation, and certification; and tools for software project management.

As regards software requirements specification and design methods, it has been observed from experience with increasingly complex embedded software systems that many (far too many) software errors in the form of discrepancies (bugs) or improper functionality were not discovered until after system deployment. Moreover, it was noted that most of these errors were traceable to the requirements specification and design phases. The magnitude of these particular software problems stimulated much analysis of the underlying reasons for the occurrence of such errors and provided the basis for a rapid growth in technology for software requirements specification and software design. Chapter 3 concludes that, while significant progress has been made in both cases, there are many important, rapidly emerging new technology developments in this area.

Chapter 3 discusses the nature of the software requirements process and the software design process in some detail in terms of the phase model developed in Chapter 2. It is pointed out that there are two categories of software requirements; namely, functional requirements and non-functional requirements or constraints. Functional requirements refer to the specific purpose or action required of a software component or system in its operating environment. Non-functional requirements impose conditions or constraints on the implementation of software components such as traceability to specifications, performance, reliability fault detection and isolation, interface and interoperability requirements, safety requirements, security requirements, validation and certification, and ease of change. It is observed that, at the current time, none of the requirements specification methods comprehensively cover development of both functional and non-functional requirements; rather, they cover most functional requirements but only some of the non-functional requirements.

As regards software design methods, it is pointed out that there are many different software design methods in current use. Some use only diagrams and are aimed at improved description and documentation of a given design, while others are more formal and use and enforce particular design principles and guidelines. Chapter 3 groups known design methods into four categories referred to respectively as functional decomposition, data flow design, data structure design, and object oriented design and it gives a description of the process involved in each category. These categories are listed above in order of increasingly recent development and application. Object oriented design is discussed in detail in Chapter 5 as the current method of choice when using ADA as the software implementation language together with ADA-based software engineering environments. It is pointed out that methods that employ only functional decomposition and data flow are not sufficient for guidance and control software because of the critical timing and event dependence which is an inherent aspect of all guidance and control systems software.

Appendices 3-1 and 3-2 lists sixty-one current methods and indicates the life cycle phases which they cover. Additionally, coverage of project management and configuration management is shown together with any related methods. This large number of available methods makes selection of a particular method or methods for any specific application such as guidance and control system and software design difficult, especially when one realizes that the methods are generally incompatible and cannot be combined. Since the methods listed are not described or characterized in any detail, these appendices are useful primarily for providing a check list of available methods for the various life-cycle phases indicated. Upon perusal of the appendices, it can be concluded that by far the greatest coverage is available for the software design and module coding and test phases, while there are a few methods which provide some coverage for almost all of the phases indicated.

As Chapter 3 points out, the earliest tools for software development were assembly languages and related programs called assemblers for translation of assembly languages into internal machine code. These were

soon followed by much more powerful higher order languages such as Fortran and Cobol and their related compilers. Since these beginnings in the 1950's, computer scientists and engineers have produced increasingly powerful compilers and other powerful tools such as flowcharters, linkers, loaders, and editors. Most recently, these earlier single function tools have given way to even more complex tools with significantly increased functionality and power. More importantly, it has come to be realized that such powerful software tools are a natural and convenient way to enforce both design and managerial methods. The set of all available software tools can be usefully categorized as stand-alone tools, tool sets, and software engineering environments. Stand-alone tools are typically single function tools which obviously operate independently of other tools and are often unique to a particular phase of the software life-cycle. Tool sets are typically collections of individual stand-alone tools applicable to the various phases of the software life-cycle. Software engineering environments are integrated sets of methods, tools, and procedures for software design, development, and postdevelopment support.

There is a small set of generic kinds of tools that are common to almost all embedded computer system application. A typical set of generic tools for guidance and control system software design and development which use a given high order language and given host and target computers would include an editor, a compiler, an assembler, a linker, a relocatable loader, a run-time executive, a simulator/emulator, an in-circuit emulator, a symbolic debugger, a pretty printer, and a host-to-target exporter.

High order languages currently in use for guidance and control systems include LTR3 in France, PEARL in the Federal Republic of Germany, CORAL 66 in the United Kingdom, and CMS-2 and Jovial in the United States. Appendix 3 of Chapter 3 contains a brief description of each language in current use and available associated tool sets and environments. From Appendix 3 it can be concluded that the available tool sets/environments for languages in current use contain most of the generic tools listed above.

More recently, ADA is emerging as the language of choice in each of the four countries and it is expected to become a future standard. A major goal of ADA is to achieve a high degree of portability of ADA programs to many computers and operating systems. This is to be achieved through control of the language definition and certification of compilers. This aspect of ADA standardization has been quite successful to date in that early experience with current ADA compilers has demonstrated good portability. Also, the compiler certification process is steadily becoming more comprehensive and complete. However, there are three remaining problem areas with ADA and ADA compilers at this point in time which are particularly important in guidance and control applications. The first of these concerns are run time environment which, unlike the language, is not formally defined. As a result, different implementations of the run-time time software on different computers can produce different results. This problem area is being addressed by a DOD working group. The second of these problem areas concerns the ADA tasking model which is now known to be insufficient for real-time embedded systems. It too is being addressed by working groups. The third is the performance of existing compilers both in terms of their own performance and the performance of the code they produce. Both issues are well-known to compiler developers and should improve significantly as the compilers are matured.

Along with the development of ADA came the realization that there would be great value in standardizing on ADA-based software engineering environments as well. The STONEMAN document produced in 1980 identified the general requirements for an ADA Programming Support Environment (APSE) and recommended a layered structure consisting of hardware and host computer system software at Level 0; a Kernel ADA Programming Support Environment (KAPSE) which would provide a machine-independent portability interface with database, communication, and run-time support at Level 1; a Minimal ADA Support Environment (MAPSE) consisting of a minimal set of integrated tools, written in ADA and supported by the KAPSE, which are necessary and sufficient to support ADA applications programs; and ADA Programming Support Environments (APSES) that are constructed as extensions of the MAPSE to provide full support of application specific methods and tools. The most important layer is clearly the KAPSE because the assumption is that this layer can be standardized across most computer systems and operating systems allowing portability of all tools in the higher layers. The pressure to develop compilers before the KAPSE was defined has led to compilers and associated MAPSES with tools that are not portable. However, efforts are underway in the US through two teams (KIT and KITIA) to produce a standard definitions of a KAPSE. To date initial versions of a Common APSE Interface Set (CAIS) have been produced. A similar effort in the European Community has been initiated to develop a Portable Common Tool Environment (PCTE) which addresses the same issues. Thus, there is hope that these efforts will prove fruitful and perhaps even converge and merge to provide an effective standard KAPSE definition. Chapter 3 recommends that these efforts receive strong support by all countries involved.

Currently, ADA is not being used for guidance and control software. However, it appears likely that the language and its evolving support environments will be used in the future for many, if not most, real-time embedded applications including guidance and control systems. For example, recently the US Air Force Flight Dynamics Laboratory has experimentally implemented some parts of flight control systems in ADA and has discovered that the performance penalty for using ADA for inner-loop control may not be very great. Appendix 4 of Chapter 3 contains a brief overview of the ADA language, information on known validated ADA compilers, and some details on the associated programming support environments.

The material summarized above from Chapter 3 has dealt with current status of languages, tools, tool sets, and environments which primarily support the top-down design and implementation phases on the left hand part of the "V" chart. Once the software modules have been coded and tested as individual modules, software and system integration and test begins. This corresponds to the phases on the right hand side of the "V" chart. Chapter 3 describes the software integration and testing process in terms of input to the process, the tasks of the process, and the outputs of the process. The inputs are module source and object code listings, individual module specifications, checkout procedures for each module, and integration test plans. The tasks consist of running diagnostic tests on computer and peripheral equipment, validating deliverables, integration of modules into operational code, checkout of total software operation, analysis of test results, modification of modules to correct errors uncovered, documentation of test results, updating documentation, customer training and planning for post-delivery support, and performing the critical design review for the software. The output is operational software object code, software specification support software, detailed design specifications, system documentation, and test reports.

Chapter 3 defines five major possible stages of testing during the integration and testing phases of a typical overall system development which involve execution on the host computer, execution on an emulator of the target computer, execution on the target computer, system simulation, and flight test. The last three of these stages are always required for guidance and control systems. The nature of these five test stages and the activities that take place during them are described in some detail and a list of static and dynamic tools for aiding software verification and system validation during the software and system integration and test process is provided. Specific static tools listed include circular reference checkers, code comparators, consistency checkers, cross-reference checkers, data base analyzers, interface checkers, program flow analyzers, set/use checkers, standards checkers, units consistency checkers, and unreachable code detectors. General static tools listed include accuracy analyzers, assembly code verifiers, assertion checkers, documentation and construction systems, formal languages with syntax checkers, requirements specification program design, program code, sneak path analyzers, symbolic evaluators, theorem provers, verification condition generators, failure mode effects analyzers. The dynamic tools listed include simulators, iron-bird test beds, test data generators, test drivers, test execution monitors, test record generators, and timing analyzers. The static tools are most useful during the specification, design and coding phases of software development and implementation, while the dynamic tools aid the testing of the software during real-time execution.

The levels of software and system integration and testing at which verification, validation, and certification occur are clearly indicated on the "V" chart. Verification refers to the process of determining whether or not the products of a given phases fulfill the requirements established during previous phase: and, in the case of software, it occurs at the level of software/hardware integration on the target computer and below. Validation refers to the process of evaluating the performance of the subsystems and the system at the end of each subsystem integration phase and at the end of the full system integration phase to assure that the subsystems and the full system meet the specified subsystem and the system requirements. It occurs during the subsystem and the full system integration phases. Certification is the process of establishing the fact that a guidance and control system, including its software, is safe for flight. It is completed at the end of the flight test phase.

Numerous tools exist for support of software verification, most of which have evolved from a set of "hand operations". Chapter 3 lists those described above which are associated with the design, coding and integration phases of software and system design and implementation. Existing validation tools are primarily those listed above which relate to test generation, execution, and analysis. To be complete, validation must be carried through iron-bird simulation and flight testing. Certification is typically a review process as opposed to a testing process, although much of the documentation reviewed is in the form of test results from the integration and test phases of the overall process. Critical to the certification process are the results of special safety analysis and tests including flight tests. The review is usually based on applicable documents and regulations, software development methods used, verification and validation procedures

used and results therefore, quality assurance and configuration control procedures, and overall system documentation.

Chapter 3 also discusses the issue of software quality assurance (SQA). The tasks of SQA are to assure that the phases of the software life-cycle as defined in the project guidelines are properly respected and implemented, that the software reviews as defined in the project standards for each of these phases are properly performed, that appropriate inspections are performed before each formal review of each life-cycle phase, that required software audits are performed, and that software is acceptable for delivery. The discussion covers the issues of responsibility for SQA; document review and checking for SQA; the role of standards, audits, and software quality inspections in SQA; SQA assurance of configuration management, tools, techniques, and methods for SQA; use of quality factors and criteria in SQA; SQA of code control and media control; and SQA standards. Formalization of SQA methods, techniques, and tools is in process.

The final topic discussed in Chapter 3 is software project management. It is pointed out that the primary functions of software project management include project planning; project control; project communications; definition of documentation requirements for the project including engineering documentation, formal management documentation, and informal documentation; and configuration management including baseline identification, control, and tracking of access and change; and control of software releases. Each of these functions is described briefly. Currently available tools for support of project management and configuration control are listed in the appendices.

Limitations of Currently Available Software Engineering Environments

Chapter 3 gives extensive tables listing current software tools and software engineering environments. However, analysis of this information given in Chapter 4 shows that no tools of software engineering environment in current use which makes any pretense to completeness. Hence, it is immediately concluded in Chapter 4 that any discussion of specific available tools or software engineering environments is not relevant to the development of comprehensive software engineering environments in the future. Instead, Chapter 4 discusses the generic limitations which apply to a greater or lesser extent to all currently available software engineering environments. The types of generic limitations identified in Chapter 4 include limitations of completeness, limitations of integration, limitations of availability, limitations of extensibility and interchangeability, limitations of performance, and limitations of application. Each of these types of limitations is discussed in some detail in Chapter 4, together with the critical role of operating systems in the implementation of environments.

Chapter 4 demonstrates the limitations of completeness of tools by analyzing the information given in Chapter 3. Of the sixty-nine tools listed, it is noted that none are applicable to more than five or six of the nine phases against which the tools were analyzed for coverage. Moreover, most of the software engineering environments listed include only the generic tools plus a very small number of the many tools known to be of value to the overall software design and implementation process.

Limitations of integration are obvious to anyone trying to use the large number of existing tools which support specific individual methods addressed to individual parts of the software life-cycle. The individual tools simply have not been designed to interact effectively and synergistically with one another and they often support incompatible methods. It is pointed out that what is needed are few powerful methods and their related integrated sets of tools which cover the entire software life-cycle. Currently, no existing set of tools pleases anybody. Limitations of availability arise from the fact that tool sets and environments have been built on existing operating systems. This restricts their availability on other machines and operating systems.

Limitations on extensibility and interchangeability reflect the difficulty experienced with changing tools or adding additional tools to a project's tool set or environment once the project is underway. These difficulties arise because of the lack of standard tool interfaces in existing tool sets and environments and because of inadequacies in the database management systems upon which tool sets and software engineering environments are built.

Limitations of performance relate to speed of compilation; the timeliness and accessibility of the error messages; the size, speed, reliability and integrity of the object code; and the ergonomics of the human interface to the tool sets and environments. For guidance and control applications, integrity and run-time speed and efficiency also will always be of prime importance along with the need to support a wide range

of redundant, fault tolerant target computer architecture.

Limitations of application refer to limitations specific to a particular class of software as opposed to the other limitations discussed above which refer to software in general. In the case of guidance and control applications, integrity considerations have made the use of restricted instruction sets commonplace and only the simplest tools are used because they can be adequately validated. Wider applicability and use of tool sets and environments will place much greater emphasis on validation of the tool sets and environments. Similarly, security is much more important for designers and implementors of defence related software than for the average software designer and implementor and this lead to the need for multi-level security for tools sets and environments. Also, there is the need for tools for development of software for redundant, fault tolerant, distributed computing architectures typical of existing and emerging guidance and control systems. There are relatively few tools available that address these considerations and almost none of them have been designed to be integrated with available or future tool sets or environments.

Chapter 4 points out that these many limitations are unlikely to be overcome with one master stroke; e.g., with development of the environment. As a result, the chapter charts a course by which the limitations can be overcome through a process of directed evolution. The course is charted by discussing the limitations of some of the tools listed in Chapter 3 and, more significantly, by describing the functions of new tools that are under development or that should be developed.

The chapter concludes with a recasting of the software design and development process into a three dimensional system development space along whose axes are represented the design phases, the design elements, and the design control functions, respectively. The floor plan in this development space is determined by the plane formed from the design phases axis and design elements axis and it represents the activities to be carried out. System development proceeds by gradually filling up the system development space with frequent revisits to previously filled volumes in the form of design iterations.

Requirements for Software Engineering Environments and Tools

Chapter 5 attempts to establish requirements for future software engineering environments and tools which cover the software life-cycle reasonably well, where by "reasonably well" is meant an environment that is reasonably achievable in a few years time as opposed to an ideal environment. The requirements described assume the use of object oriented design with ADA or another modern language of similar power being used as the software implementation language.

A software engineering environment for guidance and control software has to cope with a wide range of requirements. At one end of the spectrum are high integrity, safety critical systems and at the other end are extremely large and complex mission systems. Moreover, the design techniques used for flight critical systems software are much more restricted and conservative than those for mission critical systems.

An ideal environment for guidance and control systems would be based on an integrated data base and would contain a compatible tool set which is usable throughout the software life-cycle. It would also be extendible and allow for incremental enhancements as new tools become available. It would contain a rigorous and robust configuration control tool which would provide complete traceability of the final code through its earlier versions all the way back to the original design requirements. It would enforce standards and design methods specific to guidance and control systems design and implementation. And it would contain documentation tools capable of handling full text and graphics in an integrated manner. Finally, a properly integrated environment would include the main characteristics advocated by CAIS and PCTE. That is, it would support the whole life-cycle and offer its users tools for tracing differences between intermediate products arising from different phases of the life-cycle, a uniform command language and user interface, the possibility of adding new tools to support the evolution of methods and new target computers, and extensibility in number of users and number and size of supported tools. The latter technically require a kernel environment.

Chapter 5 advocates a kernel environment supporting object oriented design. This would include an object management system built on a data base supporting an attributed entity relation model, powerful program execution and I/O primitives, strong protection mechanisms, a uniform command language, program (tool) communication facilities, and standard user interface specifications. It is noted that the PCTE view fits the needs of guidance and control systems quite well and the PCTE view of these elements is summarized in an

appendix to Chapter 5.

Chapter 5 suggests that a good user interface can be formed by objects which are windows, icons, and tool representations, and it specifies the related user interface operations in some detail using the concepts of user agent and applications processes, frames, viewports, windows, cursors, carets, and current applications, each of which is carefully defined. It also discusses desirable command language operations, basic editing operations, and menu management operations.

Chapter 5 concludes with a description of general requirements for a set of software engineering tools including system and subsystem requirements tools, software requirements and prototyping tools, basic software design tools, detailed software design tools, software implementation tools, validation and test tools, and support tools. Each of these sets of requirements as discussed in Chapter 5 is summarized below.

System requirements, which are developed during the Study Phase and the Concept Development Phase as described in Chapter 2, have two main facets; namely, functionality and constraints. In order to support the development of system requirements, system requirements tools should have a good means for representing system decomposition into subsystems and an effective means for verifying the coherence of the system decomposition, of the data flow between subsystems, and of the system constraints. Moreover, for each subsystem, the systems requirements tools should provide a means for description of input and output data together with relations which must hold between them, a means for completely describing the functionality of the subsystems, and a means for formally describing the constraints that apply to the subsystems. Because different methods may be used in the development of system requirements, different tool sets may have to be provided, but, in every case, the tools provided should be able to enforce a unique project selected system requirements development method. There is not much experience in using system requirements tools except for Teledyne Brown's experience with IORL/TAGS, the US Army's experience with BMDATC, and TRW's experience with DCDS. However, many methods and languages have been developed to support the system requirements process and a good summary of these is referenced.

Software requirements tools and software prototyping tools are properly considered together because prototyping is often used to support the development of software requirements. A number of potentially useful methods exist for support of the software requirements phase, and it is essential that the environment provide the flexibility to support tools for these different methods. Also, both graphical and textual descriptions of the system and its software should be possible and provision should be made for the incorporation of formal methods as they become available. In the process of developing software requirements, it is often necessary to gain some added assurance that the functional part of the requirements are sound and actually reflect what the users intended. This can often be best accomplished through development of a prototype aimed at rapidly implementing all or some appropriate percentage of the functional part of the software requirements without regard for the constraint part of the requirements. The prototype can then be used to demonstrate the properties and behavior of the software as specified by the software requirements. To be cost effective, the prototype cost should not exceed some small percentage (e.g., 10-20 percent) of the expected software development cost.

The particular tools required to support basic software design depend strongly on the specific design methods used. Unfortunately, there is no general agreement as to which design methods are best. A schematic capture scheme is suitable for capturing the software design if a schematic design approach is used. However, in this case there should be a textual representation tied to the graphical representation with a one-to-one correspondence existing between the graphical and textual representations. This is analogous to the situation in hardware design where a schematic representation is backed by a cell library and net list. Also, it should be possible to move freely back and forth among the graphical and textual representations for different phases of the design and implementation so that the various aspects of the design and implementation can be conducted in the most appropriate form. In general, it should be possible to carry out much of the design process on the screen including hierarchical decomposition. The software components thus defined should be decomposable into ADA packages or LTR3 modules or similar structures appropriate to other languages. Also, at this stage, tools are needed to specify the dynamic semantics of packages or modules, but, unfortunately ADA and LTR3 provide only syntax and static semantics in their specification.

Powerful tools are obviously needed to help transforming the rather declarative graphical and textual output from the basic software design phase into the imperative, implementation oriented form during the detailed software design phase. Again, such tools are obviously methodology dependent. Examples of such tools are given in Chapter 3.

It is pointed out that several different kinds of software implementation tools are required including program-constructor tools, syntactic editors, compilers, program library managers, predefined package/module libraries, and run time executives. Development of an interactive program-constructor tool which will produce high level language code directly from detailed software design requirements and any ask the programmer to resolve relevant choices appears to be feasible and is recommended. The recommendation for syntactic editors is obvious given their usefulness. Similarly, powerful compilers are required and a detailed set of requirements for more powerful and use compilers, including the need for incremental compilation, is given. The requirements for a minimal program library manager are described in the context of object oriented design along with much more elaborated program library managers which can support configuration management and reusable software components which now appear feasible. The contents of predefined package/module libraries beyond what is normally specified in the ADA language reference manual are described. Considerable discussion of the requirements for run-time executives in the context of ADA and LTR3 is given in terms of the desired functions which include interrupt handling, and timing. It is noted that there is no agreement in the software community on the general specification of such run-time executive facilities and that this remains true even in the restricted domain of guidance and control software. However, the disagreement among guidance and control system designers is less pronounced than is the disagreement among software designers at large. Also, important restrictions on use of some of the desired facilities described for run-time executives for flight critical systems are pointed out.

Chapter 5 describes tools required to support software validation and test including symbolic (high level language) debuggers, concurrency property analyzers, and tools for failure modes and effects analysis. The need for a symbolic debugger which operates at the level of the programming language (e.g., ADA), which operates on both the host and target machines as well as operating on the host machine while monitoring and interpreting execution on the target machine, and which is able to work both interactively and in batch mode is pointed out. The desired capabilities of setting and removing breakpoints and the functions to be available at the breakpoints, for tracing statement execution, and for building histories of program objects and task status are all discussed in appropriate detail. Concurrency property analyzers are discussed in terms of static and dynamic property analyzers and system simulators. It is noted that the former are used mostly in the early stages of development while the latter are usually the workhorses when concurrency analysis is of interest. Finally, it is pointed out that failure mode and effects analysis is especially important in flight critical systems.

Chapter 5 concludes with a discussion of various general support tools including text editors, configuration management tools, and project management tools. It points out that a good environment should use only one general purpose text editor which supports all of the kinds of text to be processed; which supports creation, modification, inspection and merging of files; which has at least one working buffer in which all text is processed; which supports rapid current position cursor movement to anywhere in the buffer in order to read, modify, insert, suppress, or move text; which allows the visual image of the working buffer to be assigned to any window on the terminal screen; and which uses a general attributed tree handling facility. The powerful capabilities of such an editor are described. Configuration management tools should handle multiple versions and successive editions of each version of all objects in the environment, should support constitution and configuration control of subsystems composed of objects and the retrieval of all objects composing a subsystem, and should be built as an extension of the program library manager. Beyond these desired characteristics, it is pointed out that precise specifications for configuration management tools are dependent on the specifics of the project to be managed and the desires of management, with the specifications being based on either the syntax and semantics of versions, editions, and releases of source objects or on the whole global set of objects generated during the software design and implementation process. Project management tools have three main functions; namely, estimation of all resources required for the project, the providing histories of the various facts surrounding the project development, and the ability to compare histories with estimates and measurements of progress to see how well the project is doing. The complexity of project management tools can vary widely, ranging from those that only account for development task duration and sequencing to those that use knowledge bases containing information on development methods, resource assignment rules, histories of experience on similar projects, etc. In general, cost, development time, and project personnel estimation tools should be provided, and a discussion of the desired functionality of such tools is discussed. Finally, the need for tools for collecting histories relative to the project development for use in postdevelopment support and on similar future projects is discussed and a minimal set of information to be collected for these purposes is described.

Future Trends

Chapter 6 begins by summarizing the distinctions between software for embedded systems such as guidance and control systems and software for other applications such as simulation or scientific computation. As examples, it is noted that embedded systems software deals with a great variety of input and output devices which are application dependent, with events that occur in the environment in real time, with situations in which memory is at a premium and must be used efficiently, with situations for which the cost of producing the software is not a dominating factor, and with situations in which system integrity is the single most important issue both in design and implementation and in postdevelopment support. Moreover, the complexity of software for embedded systems is growing rapidly due to growth in the number of functions to be implemented, the increasing complexity of each of the individual functions, and the increasing interaction among the individual functions. This increasing complexity is primarily due to the increasing integration of guidance and control functions with mission functions. As a result, it is pointed out that new methods and tools are needed to help control and handle this increasing complexity.

Chapter 6 asserts that history shows that with good theories, methods, and tools, the ability of humans to successfully cope with complexity grows significantly. Some known techniques discussed earlier for gaining control over increases in complexity include integrating the requirements for and specifications of the functions to be implemented into the development cycle, carefully documenting design decisions in an easily accessible form for use in future projects, development of the system top-down in a number of phases through stepwise refinement, implementing the software in a high level language, and use of standard techniques and standardized components whenever possible. Unfortunately, such a "list" gives no clues as to how to implement these techniques, and, moreover, special requirements for real time embedded systems are essentially ignored. Chapter 6 presents some ideas for each of these techniques which may help in finding a "solution" to the problem of successfully coping with increasing complexity.

The careful specification of the intended functions and behavior of a system is cited as the most important aspect of all system design and implementation activity. Moreover, it is noted that the difficulty of specifying a system component depends on the contribution of the component to the overall system behavior. For example, timing and interface requirements for a component can be very difficult to establish even though the functionality of the system as a whole is completely specified. A trend which is observable in the results of recent fundamental efforts aimed at understanding what programming really is that a more mathematical attitude towards software design will lead to better and probably more correct programs. In this connection, it is noted that a general approach to specifying a program's behavior is to specify the inputs it is allowed to operate on and what the ensuing outputs should be. A powerful and, in principle, very general notation for expressing such specifications is first-order predicate calculus which, unfortunately, is very difficult to read for most people. Perhaps textual manipulations of expressions in predicate calculus could make such specifications much more easy to read and manipulate by software designers without losing precision. If so, such expressions could form the basis for complete and precise specification and derivation of programs, where both the derivation process and the specification activity could be computer assisted. Representations other than predicate calculus that offer similar precision and formality (such as certain kinds of graphical representations) could also be used and designer could choose their preferred form of representation or combinations thereof. If the semantics of the different notations is well defined, it should be possible for a machine to easily convert from one to the other as desired. Whatever the representation used, care should be taken not to overspecify and thus preempt design choices, which, for predicates, means that the various alternatives must each fully specify the conditions under which that alternative is open and that no interdependence between alternatives can be allowed.

Careful and complete documentation of the rationale for the decisions taken during the design phase is cited as the most neglected aspect of programming activity. Such documentation could serve both as a source of information to improve designs by evolution over successive projects and as a source of invaluable information during postdevelopment support. Design tools can be envisaged which would suggest design considerations and decisions based on previous designs. While such tools would require extensive knowledge of the design at hand and previous designs, the tools themselves can assist the designer in formulating and establishing the required knowledge. The documentation generated by such design tools could be made quite accessible by providing automatic extraction of keywords. Moreover, an information retrieval system based on use of such keywords could be used to integrate project oriented information such as test results, planning estimates, and actual effort expended leading to a better understanding of the design process itself.

REPORT DOCUMENTATION PAGE			
1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-AR-229	ISBN 92-835-0538-7	UNCLASSIFIED
5. Originator	Advisory Group for Aerospace Research and Development North Atlantic Treaty Organization 7 rue Ancelle, 92200 Neuilly sur Seine, France		
6. Title	THE IMPLICATIONS OF USING INTEGRATED SOFTWARE SUPPORT ENVIRONMENT FOR DESIGN OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE		
7. Presented at			
8. Author(s)/Editor(s)		9. Date	
Edited by E.B.Stear and J.T.Shepherd		February 1990	
10. Author's/Editor's Address		11. Pages	
See flyleaf.		192	
12. Distribution Statement	This document is distributed in accordance with AGARD policies and regulations, which are outlined on the Outside Back Covers of all AGARD publications.		
13. Keywords/Descriptors			
Flight control Missile control Guidance computers		Computer systems programs Programming languages Design	
14. Abstract			
This report summarises the deliberations and conclusions of Working Group 08 of the Guidance and Control Panel of AGARD, the Terms of Reference of which were: (i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint. (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i). (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i). The Working Group attempted to consider all software design and development technologies which existed or were known to be under development at the time.			

AGARD Advisory Report No.229 Advisory Group for Aerospace Research and Development, NATO THE IMPLICATIONS OF USING INTEGRATED SOFTWARE SUPPORT ENVIRONMENT FOR DESIGN OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE Edited by E.B.Stear and J.T.Shepherd Published February 1990 192 pages This report summarises the deliberations and conclusions of Working Group 08 of the Guidance and Control Panel of AGARD, the Terms of Reference of which were: P.T.O.	AGARD-AR-229 Flight control Missile control Guidance computers Computer systems Programs Programming languages Design	AGARD Advisory Report No.229 Advisory Group for Aerospace Research and Development, NATO THE IMPLICATIONS OF USING INTEGRATED SOFTWARE SUPPORT ENVIRONMENT FOR DESIGN OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE Edited by E.B.Stear and J.T.Shepherd Published February 1990 192 pages This report summarises the deliberations and conclusions of Working Group 08 of the Guidance and Control Panel of AGARD, the Terms of Reference of which were: P.T.O.	AGARD-AR-229 Flight control Missile control Guidance computers Computer systems Programs Programming languages Design
AGARD Advisory Report No.229 Advisory Group for Aerospace Research and Development, NATO THE IMPLICATIONS OF USING INTEGRATED SOFTWARE SUPPORT ENVIRONMENT FOR DESIGN OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE Edited by E.B.Stear and J.T.Shepherd Published February 1990 192 pages This report summarises the deliberations and conclusions of Working Group 08 of the Guidance and Control Panel of AGARD, the Terms of Reference of which were: P.T.O.	AGARD-AR-229 Flight control Missile control Guidance computers Computer systems Programs Programming languages Design	AGARD Advisory Report No.229 Advisory Group for Aerospace Research and Development, NATO THE IMPLICATIONS OF USING INTEGRATED SOFTWARE SUPPORT ENVIRONMENT FOR DESIGN OF GUIDANCE AND CONTROL SYSTEMS SOFTWARE Edited by E.B.Stear and J.T.Shepherd Published February 1990 192 pages This report summarises the deliberations and conclusions of Working Group 08 of the Guidance and Control Panel of AGARD, the Terms of Reference of which were: P.T.O.	AGARD-AR-229 Flight control Missile control Guidance computers Computer systems Programs Programming languages Design

(i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint. (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i). (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i). The Working Group attempted to consider all software design and development technologies which existed or were known to be under development at the time. ISBN 92-835-0538-7	(i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint. (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i). (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i). The Working Group attempted to consider all software design and development technologies which existed or were known to be under development at the time. ISBN 92-835-0538-7
(i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint. (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i). (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i). The Working Group attempted to consider all software design and development technologies which existed or were known to be under development at the time. ISBN 92-835-0538-7	(i) To develop and consider a set of requirements for a high level language software support environment from a guidance and control systems viewpoint. (ii) To evaluate the characteristics and capabilities offered by advanced language support environments, either existing or in the course of development, with respect to the requirements defined in (i). (iii) If necessary, to determine the modifications which would have to be contemplated to meet fully the needs expressed in (i). The Working Group attempted to consider all software design and development technologies which existed or were known to be under development at the time. ISBN 92-835-0538-7