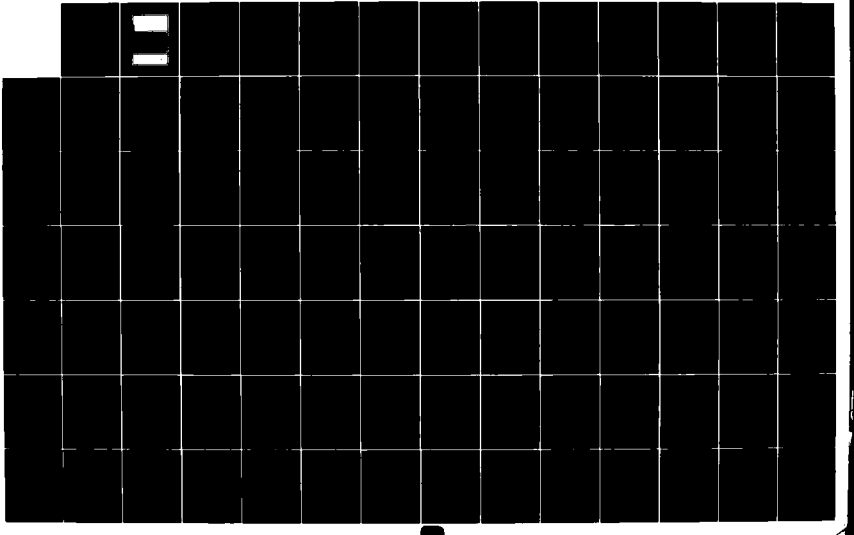
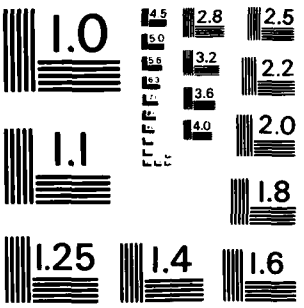
AD-A127 131 SOFTWARE FOR AVIONICS(U) ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT NEUILLY-SUR-SEINE (FRANCE) JAN 83 AGARD-CP-330 1/5

UNCLASSIFIED F/G 9/2 NL





AGARD-CP-330

AGARD-CP-330

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

7 RUE ANCELLE 92200 NEUILLY SUR SEINE FRANCE

AGARD CONFERENCE PROCEEDINGS No. 330

Software for Avionics

NORTH ATLANTIC TREATY ORGANIZATION

DISTRIBUTION AND AVAILABILITY
ON BACK COVER

AGARD-CP-330

NORTH ATLANTIC TREATY ORGANIZATION

ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT

(ORGANISATION DU TRAITE DE L'ATLANTIQUE NORD)

AGARD Conference Proceedings No.330

SOFTWARE FOR AVIONICS

Copies of papers presented at the Avionics Panel's 44th Symposium held at the Atlantic Hotel, The Hague-Kijkduin, Netherlands, 6–10 September 1982.

THE MISSION OF AGARD

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:

Exchanging of scientific and technical information;

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;

Providing scientific and technical advice and assistance to the North Atlantic Military Committee in the field of aerospace research and development;

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;

Providing assistance to member nations for the purpose of increasing their scientific and technical potential;

Recommending effective ways for the member nations to use their research and development capabilities for the common benefit of the NATO community.

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.

Published January 1983

ISBN 92-835-0325-6

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

THEME

The last decade has brought about an explosion-like progress in electronic data processing technology. This can be mainly attributed to the continuously improving performance of semi-conductor devices with an ever-increasing integration density, and the tremendously fast development of digital computers.

While hardware costs for computers of all sizes are decreasing, costs and complexity of the associated software are rapidly increasing. This problem has become even more critical with the implementation and application of advanced microprocessors and microcomputers. With the high degree of digitalization in avionic systems, software determines, to a large degree, the mission-critical performance in navigation, weapon delivery, flight control and defensive aids.

To address the predominant impact of software in avionics, it is opportune for the AVP, and in the interest of the NATO community, to devote a symposium to this discipline.

The objectives of this Meeting are as follows:

- Provide an overview of the software elements associated with embedded computer resources.
- Address current issues related to software requirements, design, development, verification, and validation.
- Identify software life-cycle considerations such as costs, management, and maintenance.
- Discuss trends in software technology and identify the key features contributing to more efficient and more economic software programs in the NATO countries.

* * *

Au cours de la dernière décennie, la technologie du traitement électronique des données a fait l'objet de progrès stupéfiants que l'on peut attribuer essentiellement à l'amélioration continue des performances des semi-conducteurs, accompagnée d'une augmentation constante de la densité d'intégration, et au développement extrêmement rapide des ordinateurs numériques.

Parallèlement à une réduction des coûts du matériel des ordinateurs de toutes dimensions, on assiste à une augmentation rapide des coûts et de la complexité du logiciel qui lui est associé. Ce problème a revêtu un aspect encore plus critique avec la mise en oeuvre et l'application des microprocesseurs et microordinateurs de conception avancée. Etant donné le degré élevé de digitalisation des systèmes électroniques embarqués, le logiciel détermine dans une large mesure les performances, si importantes pour l'accomplissement de la mission, des systèmes de navigation, de tir de munitions, de contrôle du vol et des aides à la défense.

Afin d'étudier l'impact prédominant du logiciel dans les systèmes électroniques embarqués, il est à la fois opportun pour l'AVP et bénéfique pour la Communauté Atlantique de consacrer un symposium à cette discipline.

Les objectifs de la réunion sont les suivants:

procéder à un tour d'horizon des éléments de logiciel associés aux composants incorporés d'ordinateurs,

traiter des problèmes d'actualité posés par les impératifs, la conception, le développement, la vérification et la validation du logiciel,

délimiter les domaines intéressant le cycle du logiciel, tels que les coûts, la gestion et la maintenance,

examiner les tendances de la technologie du logiciel et identifier les éléments clés qui contribuent à rendre plus efficaces et plus économiques les programmes de logiciel des pays de l'OTAN.

LIST OF OFFICIALS

AVIONICS PANEL

Chairman: Mr Y.Brault
Thomson CSF
Division Equipements
Avioniques et Spatiaux
178 Bld. Gabriel Péri
92240 Malakoff
France

Deputy Chairman: Dr F.Diamond
Chief Scientist, RADC/CA
Rome Air Development Center
Griffiss AFB,
N.Y.13440
USA

TECHNICAL PROGRAM COMMITTEE

Chairman: Mr M.Jacobsen
AEG-Telefunken, A14 V3
Postfach 1730, D-7900 Ulm
Germany

Members:
Mr R.O.Mitchell, US
Dr A.A.Callaway, UK
Dott. Ing. L.Crovella, It
Dr W.Ware, US
Mr B.Mirailles, Fr
Dr H.Hessel, Ge

HOST NATION COORDINATOR

Mr E.G.H.Bleeker
Nederlandse Delegatie bij de AGARD
P/A Stichting NLR
Postbus 126
2600 AC Delft
Netherlands

PANEL EXECUTIVE

L/Col. J.B.Catiller
AGARD/NATO
7, rue Ancelle
92200 Neuilly-sur-Seine
France

CONTENTS

TECHNICAL EVALUATION REPORT

ON THE

AVIONICS PANEL SYMPOSIUM

ON

SOFTWARE FOR AVIONICS

PRECEDING PAGE BLANK-NOT FILMED

EXECUTIVE SUMMARY

The following summarizes the significant conclusions and recommendations from the Technical Evaluation.

CONCLUSIONS:

- Software will continue to be a major problem area, since advances in software technology are matched by increases in hardware capability, the complexity of requirements and the rate of growth in avionics software.

- The requirement is to develop affordable avionics hardware and software systems that are more reliable, higher performing, more flexible in configuration and easier to modify than present systems.

- There are too many different standard methodologies, too many different languages with insufficient emphasis on good design techniques and tools. A strong need exists for better coordination of software related research and development activities among the NATO nations to increase overall efficiency and reduce redundancy of efforts.

- A major challenge is to prevent software from limiting the use and performance of new hardware technology, e.g. VHSIC (Very High Speed Integrated Circuits). Software technology needs to be developed in consonance with hardware technology to make effective use of available capabilities.

- Avionics Software is moving into areas of such prominence that its failure might have catastrophic results. Credible methodologies are to be developed to specify, predict and measure software reliability. The achievement of adequate reliability and the knowledge that this level of reliability has been reached are of prime importance. Appropriate funding is required for Software Quality Assurance, Verification and Validation as well as Test and Evaluation.

- Software will remain a dominant factor in the total cost of avionics software over its whole life time. We, therefore, need to consider Life Cycle Cost as opposed to acquisition cost.

- Innovative techniques to permit reusability of application/support software are to be developed. For increased software productivity integrated tools are needed which allow for some degree of automation in software production.

- It appears that the US DOD Standard High Order Language ADA is generally accepted by most of the NATO nations. The importance of the ADA Programming Support Environment (APSE) throughout all phases of the software development cycle must be emphasized. Active European involvement in the definition of APSE/ADA is most essential for an effective implementation process.

RECOMMENDATIONS:

- NATO or AGARD to support study to investigate alternative software development approaches for large avionics computer systems.

- NATO or AGARD to support working group for the definition of a testing environment related to APSE/ADA with promotion of strong European involvement.

- NATO to establish organization for coordinating software related research activities, acquisition policies and standardization.

TECHNICAL EVALUATION REPORT

BY

M. JACOBSEN
AEG-TELEFUNKEN, ULM

1. INTRODUCTION

The 44th symposium of the Avionics Panel on "Software for Avionics" was held in The Hague, The Netherlands, 6 - 10 September 1982. The Program Committee consisted of Mr. M. Jacobsen (GE), Chairman, Dr. W. Ware (US), Mr. R.O. Mitchell (US), Dr. A.A. Callaway (UK), Dr. L. Crovella (IT), Mr. B. Mirailles (FR), Dr. H. Hessel (GE).

The technical evaluation for each session has been performed by the session chairmen and edited into its final form by the program chairman. The evaluation has also taken into account comments and recommendations which were received from authors/participants of the symposium.

This report represents an attempt by the Program Committee to provide a summary of the entire symposium and to draw conclusions and derive recommendations from the presentations and discussions.

2. THEME AND OBJECTIVES

The last decade has brought about an explosion-like progress in electronic data processing technology. This can be mainly attributed to the continuously improving performance of semi-conductor devices with an ever-increasing integration density, and the tremendously fast development of digital computers.

While hardware costs for computers of all sizes are decreasing, costs and complexity of the associated software are rapidly increasing. This problem has become even more critical with the implementation and application of advanced microprocessors and microcomputers e.g. by the introduction of new technologies like Very High Speed Integrated Circuits (VHSIC) and Very Large Scale Integration (VLSI). With the high degree of digitalization in avionic systems, software determines, to a large degree, the mission-critical performance in navigation, weapon delivery, flight control and defensive aids.

To address the predominant impact of software in avionics, it was opportune for the Avionics Panel (AVP), and in the interest of the NATO community, to devote a symposium to this discipline.

The objectives of the meeting were as follows:

- Provide an overview of the software elements associated with embedded computer resources.
- Address current issues related to software requirements, design, development, verification and validation.
- Identify software life-cycle considerations such as costs, management, and maintenance.
- Discuss trends in software technology and identify the key features contributing to more efficient and more economic software programs in the NATO countries.

The symposium was organized in five sessions in which 44 formal papers were presented covering

I	Software Technology (Tutorial)
II	Software and System Requirement Analysis
III	Software Design and Development Process
IV	Software Verification and Validation
V	Software Life Cycle Consideration

3. TECHNICAL EVALUATION

3.1 Overview

In most modern military weapon systems, software is the basis for their purposeful behaviour. It is the means whereby computers and computer-based elements are instructed to achieve desired operational capability; in a sense, software provides the "I.Q." of a military system. By its nature, therefore, software has a major intellectual aspect that reflects the translation of user-specified requirements into detailed specifications that will become computer programs. Production of software must be looked at as a process, only a part of which involves computer-based tools. In particular, the predominant applications of such tools occur during the coding phase; to a lesser extent, during the testing and debugging phase; and to a minimum extent, during the initial requirements specification phase.

Thus, even though significant advances have been made in providing sophisticated, integrated and comprehensive computer-based tools for the development of software, there still remains a large part of it that is not only labor intensive but is also intellectually intensive. As military systems ascend in their level of smartness, the expected performance of the software will become increasingly sophisticated and complex. Therefore, we can look forward to a continuing difficulty in software production for just this reason.

The papers at this symposium of course reflect a microcosmos of the software activity in the NATO countries. Bearing this in mind, it is nonetheless striking that the problems experienced in various countries are remarkably similar, and it is striking that the state-of-art in software production is not significantly different across those NATO countries who reported to the conference. This phenomenon is itself a manifestation of the intellectual content of software and a reflection of a drive by all countries to create appropriate software development environments.

It is true that standardization of approach, of language, of management mechanisms, of documentation, and of other aspects of software will lead to economies. It does not follow that such standardization automatically guarantees a successful software undertaking. The current state-of-art in software seems to be that no matter how competently any organization tries, and no matter how complete and contemporary its management and software development environment, some projects will still be less tractable to a successful completion.
Such is the problem we face.

The military world does not dominate software matters. While military requirements tend to be specialized in the sense that such software must function in a real-time environment, must be proof against anomalous and unexpected behaviour, must not jeopardize flight safety, and similar stringent requirements, there are nonetheless enormous software undertakings in the business and industrial world as well. The track record there seems not to be dramatically different. Since computer technology - both hardware and software - is spread widely across military and nonmilitary activities, there is little Research and Development (R & D) activity one can recommend that would be specifically beneficial to military affairs.

Thus, such desirable advances as improved higher order languages, more comprehensive and powerful validation-and-verification techniques, more thoroughly integrated development tools, more comprehensive requirement specification tools, and all the rest are equally applicable across all fields of computer application.
Certainly, standardization within a country or even within a company is a necessary effort. On the other hand, there is always a possibility that arbitrary enforcement of standardization will of itself inhibit the introduction of new advances; some consideration must be given to this point before embarking on standardization as a goal unto itself. The real issue is not whether to have standardization, but rather to what degree and in what areas it should be imposed.

Thus, the appropriate posture for NATO is to encourage and support as much software R & D efforts as funds and qualified researchers make possible. Correspondingly, NATO and its supporting industrial base must be quick to recognize and exploit software R & D advances that come from industrial organizations, universities, or elsewhere.

3.2 Opening Speeches

After Prof. Gerlach in his capacity as Netherlands' National Delegate and Chairman of AGARD had warmly welcomed the participants of the symposium, General van der Kaa, Director of Material of the Royal Netherlands' Air Force, provided the keynote address on the theme of the meeting.
Main topics raised in this speech became the red line through the conference. He stressed the need for software standardization and commonality, improved supportability and maintainability leading to better cost effectiveness. In particular, he pointed out standardization of software languages, interfaces, architectures, data structures and documentation procedures. He stressed the importance of ADA, the NATO standard high order language for embedded computer systems and the expectation of NATO that ADA will provide the tools to write maintainable and easy to support programs consisting of individually developed and tested, reuseable building block modules, communicating with each other via common interface packages.

General van der Kaa concluded his speech with the hope that the meeting will contribute to more efficient and more economical software programs in the NATO countries.

The opening remarks of the Program Chairman, Mr. Jacobsen, started with a brief review of the evaluation of information technology during the past four decades, stressing the tremendous progress of electronic data processing in various fields of application.

He outlined that hardware developments have by far exceeded software capabilities, resulting in the fact that software now determines the speed of innovation. He mentioned that new technology programs like VHSIC and VLSI, may pay off as force multiplier in NATO if software technology is developed to a similar degree. At present software is the limiting factor, the bottleneck.

By drawing the attention to the rate of growth in airborne software, and its predominant impact on avionics, as well as to the continuing breakthroughs in hardware technology, Mr. Jacobsen concluded that major improvements in all areas of the software discipline are imperative.

SESSION I - SOFTWARE TECHNOLOGY (TUTORIAL)

Introduction

This session was designed as an introduction to some of the techniques which would be discussed in more detail in later sessions, and also as an overview and tutorial to the general problems of software for avionics.

It was originally planned to present six papers in the session. Unfortunately, however, an important paper on bridging the customer-supplier communication gap in weapon aiming software design had to be withdrawn at a late date. Thus, only five papers were presented.

Technical Evaluation

The first paper, by Dr. Willis Ware (1), was entitled "Avionic Software: Where Are We?", and reviewed the progress in avionics software from the early days to the present day. It then considered the dimensions of the modern avionics software task, noting that the implementation and management of software resources has become a major problem area. Following a review of the advances in software technology, Dr. Ware warned of threats against demands of increasingly complex requirements and concluded that software would continue to be troublesome. He foresaw an exciting future as we learn how to exploit information as a resource.

Dr. Sundstrom's paper (2), "Avionic Software Design", discussed avionic software design requirements and methods arising out of General Dynamics' experience with the F-16 aircraft. He described the four part software structure which had been adopted and showed how a relatively simple software design process arose from the use of this structure. This paper was valuable in that it represented an airframe manufacturer's view of the total avionics software design concept.

The following paper (4) was also given from the point of view of the aircraft contractor, this time MBB. It was by Drs von Groote and Schwegler and was entitled "Software Development, A Case Study". The paper was complementary to that given by Dr. Sundstrom in that it considered life cycle aspects arising out of the development of the Tornado operational flight program. In his presentation, Dr. von Groote highlighted those phases which proved to be most critical, and the paper also included a list of tools and techniques which experience had proved to be of assistance in such developments.

One of the modern software development techniques which was discussed in some detail in Session III is MASCOT. In Session I, Dr. H. Simpson, one of the inventors of MASCOT, presented a tutorial (5) on MASCOT developments, particularly those designed to improve the structure and integrity of the resulting application software.

The final paper in the session (6) was presented by M. Gilles Bracon and was entitled: "Towards a Genuine Workshop of Avionic Software". This paper gave details of Electronique Serge Dassault developments leading to a software generation suite known as AIGLE.

SESSION II - SOFTWARE AND SYSTEM REQUIREMENT ANALYSIS

Introduction

The development of a written software requirement specification is the first step in the software development process. Poor requirement specificationsare the largest contributor to software errors. This session therefore was taylored around the advances in requirement analysis and specifications. Transforming user needs into testable requirements is a most difficult task. Good software requirement specifications must be complete, consistent, correct, concise, unambiguous, measurable, traceable and design free. The high error frequencies in analysis and requirements can be reduced by application of structured disciplines, precise language, and automated tools. Improved methodologies and tools for developing software requirement specifications have been presented in the papers of this session.

Technical Evaluation

Dr. T.G. Stanton's paper (7) "Requirements Decomposition and Other Myths" described the problematics of requirements interpretation, emphasizing the most likely wrong decisions that can occur in the design.
In particular, the author suggested a theoretical means for generation of the "perfect" specification, i.e. the complete, formal, non redundant and unambiguous one.

The following paper (8) by Mr. C.P. Price "Practical Considerations in the Introduction of Requirements Analysis Techniques", on the contrary, dealt with a practical application of the requirement analysis techniques by describing methods and tools to aid the engineer particularly in analyzing and expressing system or software requirements of large projects in a controlled and precise manner. The author also pointed out that, as in any computer aided system, advantages will be lost unless adequate training and preparation is made prior to starting adoption of the system.

Mr. D.M. Weiss's paper (9) "The A-7E Software Requirements Document: Three Years of Change Data" presented a detailed analysis of the approach applied and the results achieved by U.S. Navy in producing the A-7E Software Requirements Document, which is a complete and concise description of the aircraft Operation of Flight Program (OFP), and in monitoring its changes.
Success in change cost reduction is based on the strict application of the Software Requirements Document.

Mr. F. Doladille's (10) paper "Un Systeme D'Aide à la Définition de Logiciels Avioniques" described another automated system, under realization in France, for airborne software documentation definition and modification.

Mr. D. Jordan's paper (11) "The Mentor Approach to Requirements Specification" presented a data base system for the maintenance of the technical documentation to be utilized by systems engineering. Trends in its growth potential towards a more sophisticated system were also described.

Mr. N. Christensen's paper (12) "The Computer Aided Specification System EASY", described a specification language developed in Germany and already employed for a non-avionic project, with future use for an avionic system design being deemed possible.

The support tools used and the problematics encountered during the program were also highlighted.

SESSION III - SOFTWARE DESIGN AND DEVELOPMENT PROCESS

Introduction

The attendees of Session III were unanimous in expressing the vital necessity of supervision, rationalization and control of software development. This was not really surprising as in this particular field of avionics and in the field of "classic" informatics the need for rationalization after a phase of wanton and informal "youth" was soon felt. Perhaps it appeared a little bit later in the field of embedded systems because of their initial, very special characteristics : small special-purpose computers with small memory size in a severe and demanding real-time environment without software support (development system, adjustment aids, high-order languages) programmed directly in machine code, then in Assembler, with a sense for completeness and efficiency which was not compatible with readability and maintainability.

The development of computer performance, the multiplication of computers in the systems, in particular, the appearance of microprocessors and the explosion of their possibilities, and the integration of large-sized complex digital systems have administered a fatal blow to the era of "ingenious tinkering" and have shown to the system managers that rigorous principles must be adopted in order to improve the quality of the entire product. This change is greatly facilitated by the existence of powerful host computers capable of ensuring all kinds of simulations of target machines, thus facilitating the tasks of adjustment and enhancing their performance; in the future it is possible to orient oneself toward high-order source languages (HOL) and to make use of sophisticated management tools (filing, development management, test strategy).

This has a favorable effect on the quality of the product delivered, and a certain number of key words quoted repeatedly are revealing: configuration control, reliability, portability, flexibility, maintainability and documentation aid.

The use of high-order languages is really a panacea in this field. The papers show, however, that the simple "language" notion (means of coding + syntax rules) is transcended to arrive at the "state-of-the-spirit" notion, basis of a "software environment" which imposes a series of principles and furnishes a set of tools intended to guide the design, to facilitate, to rationalize, but also to control the development and to satisfy as optimally as possible the specific requirements of avionic software.

Technical Evaluation

It has been demonstrated - and verified by experience - that one software management method must be applied and that it must be applied everywhere in a project, perhaps in several parallel or mutually dependent projects. Thus we arrive quite naturally at the "standard" notion. Mr. Sundstrom (13), "The Impacts of Standardization on Avionic Software", after having rapidly depicted the history of the advent of digital computers, software, and their irresistible explosion, cited several historical or current examples of the utilization of standards in connection with software. He showed the influence of the utilization of these standards on the development of certain programs, and ended with an inventory of the most imminent developments: mass memory interfaces, environment of operating system, digital bus utilization protocols, and, what is improbable according to him, standardization of algorithms.

Lt. COL Kramer's paper (14) "ADA Status and Outlook" presented a historical review of the motivation of the designers of ADA, their approach, their research work, the results and the outlook, showing clearly the desire of the U.S. DOD to promote its universal utilization. He also discussed the means of commissioning and validating it, the time schedule and the initial applications envisaged.

M. de Montgolfier's paper (15) "Standardization of LTR for embedded computers: the present and the future" presented the language which, after a contract from the French Ministry of Defense, has been studied, developed and supported since 1974, and which is utilized by many French organizations. This LTR-V1 is simple in application and is strongly real-time oriented. A recent analysis has caused its originators to initiate the generation of a version No. 2 by the introduction of notions from PASCAL. LTR-V2 is compatible i.e. in the Motorola 68000 to appear almost simultaneously with ADA.

The topic of high order languages was discussed in several papers, but was the essential point of paper No. 9, "Operational Flight Program development Using HOL" read by Mr. Westbrook who showed the necessity of evolution of the Assembler language toward HOL, of execution on a host computer of the requirements of simulation, emulation, environment simulation. He presented applications (restructuring of the AFE, development of the AFE utilizing TRIAL

Mr. Reitz (17) "An Approach to a Portable PASCAL Language for Different On-Board Computer Systems" presented a study on PASCAL, supported by the German Federal Ministry of Defense. The main ideas were portability, application for micro-processors, development on host computer, hierarchic design, flexibility, efficiency. Shortcomings in the basic version - in particular for real-time applications - have made it necessary to write extensions.

Mr. Boardman (18) "Use of HOL on Microprocessors" also dealt with the topic of High Order Languages (HOL) mainly emphasizing the rationalization in microprocessor software resulting from the use of HOL compared to Assembler, feeling, however, that the generation of an equivalent Assembler listing at the time of compilation is indispensable. He also stressed the improvement of maintainability. He underscored the importance of the test phase (on-line, off-line .....) and described the experience with the language "CORAL 66".

In paper No. 19 "Software Design and Development Using Mascot", Mr. Dibble presented a method of analysis and management with its general principles, its rules of application, and several real cases. This rigorous method requires structural modularity and facilitates, according to the author, the containment of the trend towards increasing software costs and the improvement of software reliability.

Paper No. 20 (Dr. Güsman) "Safety Critical Fast Real Time Systems" handled the problem of safety in real-time systems utilizing several processors. The ideas derived from the analysis of real cases have resulted in the definition of a number of criteria for the selection of a high order language (HOL); among four languages examined, the language "C" has been selected and applied in two projects. These applications were described, and the position with respect to ADA was outlined.

Prof. Schneidewind (21) "Usability of Military Standards for the Maintenance of Embedded Computer Software" particularly emphasized the impact of standards on the maintenance of embedded software, with a detailed investigation of three U.S. standards. He concluded that this impact is often weak, and that revisions are necessary; quite generally, he is of the opinion that maintainability must be defined and pursued as early as possible in the development process.

Configuration Management is an essential technique in controlling the software development process. Two papers addressed this topic. The first by Mr. Pedersen (22) "Software Configuration Management at Work", presented experiences gained from Norwegian Defense Programs.

Mr. Fulford then presented an interesting paper on "Configuration Management and the ADA Programming Support Environment (APSE)" (23). Configuration management, quality assurance, better manageability of the development process, reduction of software costs, visibility, portability are the objectives.

Paper No. 24 on reliability "Practical Software Fault Tolerance for Real Time Systems" presented by Mr. Knight was a fine and clear introduction into fault-tolerant systems. Some basic notions were explained, and the principal characteristics of the fault-tolerant software system were defined.

Paper No. 25 cancelled.

Paper No. 26 "Electronic Warfare Software" presented by Mr. Shaw described a case study showing the topicality of the Electronic Warfare software problem and the method applied to solve it (definition of an ideal instruction set, of bench marks, comparison with the application of analysis standards, and performance of existing compilers).

SESSION IV - SOFTWARE VERIFICATION AND VALIDATION

Introduction

With the advent of software intensive weapons systems containing more and more embedded computers and incorporating system architectures of ever-increasing complexity, increased emphasis has been placed on a category of testing which is called software Verification and Validation (V & V).

Verification and Validation is a systematic evaluation of software during development. The concept is that a more orderly and efficient process results when V & V techniques are implemented. Simply stated, verification and validation is an in-process review and analysis of each developmental step to assure that the software being developed adheres to the technical objectives and conforms to the software requirements specified. More precisely, it is a step-by-step examination of program requirements, specification, unit module and system level code during and, as the case might be, very soon after development. The quality and cost of software are the key parameters the technique of V & V purports to control.

Session IV, Software Verification and Validation, in keeping with the overall theme of the conference addresses:

1) current issues in software technology related to the development of quality software

2) emerging disciplines and methodologies designed to provide technical practitioners with tools to control software cost growth.

It is an expanding opinion that because of the intricate nature of software in avionics systems, isolating the "software problem" is most difficult and perhaps no longer practical. Toward this end, to realize the full potential of the advances in V & V techniques, a complete testing environment is required. The testing environment should cover the full spectrum of software development. The papers of this session tended not to do this, but to focus on the current problems at hand. In general, the strategy and testing methods adopted by programming staffs cover only partial segments of the total process due primarily to insufficient resources being allocated for testing. Still, there appeared to be some need to rethink the approach within these resource constraints. For example, data was presented which represented the relating cost of errors based on where they occurred in the development cycle. Software errors discovered in the test phase were shown to cost ten times more to correct than errors uncovered during the earlier phases of requirements and design. The paradox is that there was little evidence shown in the papers presented indicating this as a driving factor in our testing approach. Many authors, however, spoke of the need to research and exploit the concept of a complete software testing environment, using microprocessor technology, and taking more of a system level approach to testing.

It was quite clear from the papers presented in this session that there was no solution to the global problems of software quality and cost growth as a result of software V & V activities. The authors in their discussions and from the data presented did give indications that a testing environment, complete with automated tools, would help significantly in coping with the exploding use of computers in every facet of avionics systems. Yet, there is no widespread mandate to define such an environment, to commit the resources, to conduct the necessary research, or to establish the required universal standards.

Technical Evaluation

The paper by Mr. R.G. Wilson (27) "An Eight Point Testing Strategy for Real-Time Software" proposed a viable strategy for software testing. It represents a disciplined approach to software testing beginning in the development cycle at module coding (build) and proceeding through the integration phase. The testing concept uses both a static and a dynamic environment. The testing strategy is based upon the software being decomposed into modules and top down structure to do useful work. The philosophy of build a little, test a little is employed until a fully integrated system is in place. The paper considered the responsibilities of the programming staff using this strategy and assessed the problems encountered in re-testing due to the errors detected.

Dr. R. Pelissero's paper (28) "Tornado Flight Control Software Validation: Methodology and Tools" discussed the use of a real-time facility with a closed feedback loop for testing critical functions. The basic intent is to be able to accomplish on the ground the testing of real-time avionics functions that are mission critical. The paper described the real-time facility used in testing the Tornado flight control software and elaborated in detail on the results. The paper concluded by offering the feasibility of extending this concept to other critical functions.

Mr. Sundberg's paper (29) "Applications of Network Modeling and Analysis to System Validation and Verification" is a concept of applying system analysis technique and engineering discipline to the software development problem. The system proposed by the paper, using network analysis and Boolean logic techniques, enables the verification and validation of complete software intensive systems or concepts at any stage of the development cycle. The paper concluded that the use of such a technique which provides uniform applicability across all phases of the development cycle will result in a more reliable product.

The paper by Monsieur Taillibert (30) "IDA-Software Test Language and Related Tools" dealt with the idea of a testing language with the objective of defining the resources required for the testing of real-time software. The paper described the study undertaken and presented the definition of test language which resulted. The paper put forth several typical problems (including parallel processing) which are likely to be encountered and addressed how the test language would solve these problems. The point of the paper was the desirability of the use of a macro-language and a library of standard tools from early in the development cycle to test real-time software.

[illegible] paper [illegible] Mr. [illegible] "Software Verification [illegible] Civil Avi[illegible] [illegible]" (AHRS), furthered the concept [illegible] testing [illegible] software [illegible] of the life cycle from the requirements [illegible] testing. His paper described formal procedures to verify and [illegible] software [illegible] step in the process with some insight into the [illegible] task. Although the method was applied to a [illegible] task, [illegible] that these standards could be adapted to any existing [illegible] requirement. The paper concluded with a critical evaluation of the results as applied to the [illegible] task.

Dr. Crocker's paper [illegible], "Progress in Verification of Microprograms" dealt with [illegible] the development of a usable and practical program verification system for microprograms. Dr. Crocker made the point that this effort is particularly [illegible] in avionics systems where complex microprogrammable architectures are [illegible]. The concept being pursued by Dr. Crocker would, in principle, accept as input, formal, machine readable descriptions of the host and target machines and a copy of the microprogram. Then, using proof theorem, declare that the microprogram is either correct or point out the reason why it is not. Since this is not yet possible, a formal statement of the programmer's rationale for believing that program is correct, is required and serves as the proof with the verification system playing the role of the proof checker. The paper demonstrated the applications of verification systems with some insight into the use of such tools in the future.

Mr. McManis' paper [illegible], "Validation of Software for Missile to Aircraft Integration" dealt with the complicated task of integrating missile systems on existing, well established, software intensive aircraft systems. The discussion centered around the criticality of the task from a safety standpoint, the potential differences in technologies employed due to the time lag of the missile and aircraft developments, the organizational differences among others. The presenter stressed the need for a fully automated facility with highly sophisticated tools capable of examining the many modes of the system prior to flight testing. Finally it was emphasized that with the dynamics of the technology and the extended life of aircraft weapon systems, missile integration of this type will be more the rule than the exception. Therefore, a standard definition of the interface along with appropriate facility and tools are essential.

The paper by Mr. Dowling [illegible], "Implementing High Quality Software", like several of the previous papers, recognized the need for software validation at each stage of the development process but concentrated on a particular phase, in this case the implementation phase. The paper examined some accepted mechanics and techniques for software development such as flowcharts, high order languages vs. low level ones, data flow analysis, etc., and offered an analysis and alternative for their use. The paper concluded with a discussion of ADA and its environment on the software development process.

Mr. Delacroix's paper (35), "The Quality of Avionic Software: Specification and Evaluation" dealt with software quality metrics and related metodologies. It was outlined that this discipline is still far from having reached its maturity. The approach described by Mr. Delacroix is an attempt of classification of software quality criteria in all essential phases of development as well as an empirical metrics permitting to historize the experience gained on a project.

Dr. Martin's paper (36), "Dissimilar Software in High Integrity Applications in Flight Controls" stressed the need for a new approach to system design for those components requiring high reliability and offered as a solution the technique of dissimilar redundancy. His concept of a dissimilar redundant solution using two totally different software development teams and equipments (microprocessors) was discussed and the impact of this architectural design on software procedures was then examined.

Mr. Migneault's paper (37), "The Cost of Software Tolerance", proposed the use of fault tolerance techniques as a means of controlling the cost of software. His concept deals with the fact that some errors can and should be tolerated and that the paper develops a model that attempts to balance the system level relationship among the factors of cost, redundancy and reliability. Mr. Migneault's paper proposed the use of fault tolerance techniques as a means of controlling the cost of software while buying only the reliability needed.

SESSION I - SOFTWARE LIFE CYCLE CONSIDERATIONS

Introduction

Much work on Software Life Cycle Considerations has been done in recent years and it was clear from the nine contributions of this session that a general understanding of solving the problems is present:

- improvement of the interface between user and producer
- investigation of technical aspects/solutions related to software reliability and maintainability
- necessity of a system engineering environment which supports all phases of the avionics life cycle
- optimization of management procedures and associated tools.

However, at present we don't have satisfying general solutions to the problems. The near future has to be bridged with what we have now, and much work remains to be done to handle more and more complex systems.

Technical Evaluation

Dr. Carrington's paper (38), "Management of Large Real-Time Military Avionics Software Programs", presented the experiences gained during the development of the digital processing and display system flown in the RAF Nimrod MR. It emphasized the risks induced by additional requirements during the development phase of a project and the role of management procedures and of tools to avoid/reduce these risks.

Mr. Mc Tigue's paper (39), "F/A-18 Software Development - A Case Study", presented the successful avionics software development for the F/A-18 Hornet Fighter/Attack Weapon System, and emphasized the different phases of the development process and associated test strategies and support facilities.

Mr. Schaaff's paper (40), "A Life Cycle Model for Avionic Systems", presented a phase concept highlighting the early project phases functional/technical design with respect to cost and time for the whole software life cycle.

Mr. Ferens' paper (41), "Avionics Software Support Cost Model was presented by Mr. Shaw. The model is based on historical data from USAF Logistics Centers and is designed to evaluate software life cycle costs during the conceptual phase of a project.

Mr. Dekker's paper (42), "A Software Cost Database for Aerospace Software Development" also described a software cost estimation method based on historical data. The problem of the method is the collection of representative data from completed projects.

Wg Cdr. Barker's paper (43), "The Military User View of Software Support Throughout the In-Service Life of Avionic Systems", was an excellent presentation on the current software life cycle considerations. It emphasized first the need for a better communication of user and producer with respect to operational, technical and budgetary aspects, and second the work on a Software Support Environment like APSE/ADA as a significant step to reduce software life cycle costs.

Mr. Whalley's paper (44) "Design of a Software Maintenance Facility for the RAF", described the tool which is being used by the Royal Air Force for the maintenance of navigation and mission system software. It pointed out that future software maintenance costs can only be reduced by establishing suitable hardware and software standards.

Mr. Stuebing's paper (45), "A Software Engineering Environment for Weapon System Software", discussed the experiences since 1975 with a "Facility for Automated Software Production" (FASP) at the US Naval Air Development Center which supports the coding and testing by an integrated environment.

It emphasized the necessity that a System Engineering Environment should be an integrated environment which supports the weapon system over the entire life cycle from mission requirements to in-service maintenance.

Dr. Klenk's paper (46), "On Aircraft Test Software for First Line Maintenance", described a separation method for avionics test software design and implementation in order to reduce software maintenance costs in the in-service phase.

4. [illegible]

The following summarizes the comments made by the Program Chairman Mr. [illegible] during the closing session of the Symposium.

The major challenge facing the software community today is the fact that the [illegible] will be [illegible] more [illegible] in nature. A tremendous [illegible] which [illegible] be inherent. In modern aircraft we are with [illegible] real-time [illegible] [illegible] Advanced Systems containing a large amount of [illegible] processing interacting in various digital [illegible] schemes. The software [illegible] this will not be [illegible] an [illegible] [illegible]. All this [illegible] [illegible].

We must realistically accept that progress in the software field will be slow, [illegible] slow, so that software will remain [illegible]. It will require [illegible] improve the [illegible] situation in the [illegible] [illegible] terms.

[illegible] the many problems of software design and maintenance, [illegible] tools and methodologies as well as a [illegible] number of different software [illegible] have been presented in the meeting. It became [illegible] that [illegible] [illegible] ways, resulting in [illegible]. [illegible] be efficient [illegible] software production, efforts [illegible] be better coordinated and [illegible] [illegible] be improved.

Furthermore, in addition to a standard High Order Language (HOL), it is necessary [illegible] extent [illegible] of design techniques, [illegible] systems, [illegible] systems and software environment.

The objective of ADA, namely the creation of reliable, maintainable and efficient [illegible] and programming environments motivated the software community. The necessity of ADA as the High Order Language for embedded systems seems to be generally accepted by most of the NATO nations.

Expectations are high, but in view of its early stage of development, there is no [illegible] that our problems can be solved until ADA and its support environment becomes fully available and sufficient experience can be gained.

ADA will therefore remain an issue of continuing concern and is to be monitored carefully.

Software reliability has become another major topic in mission and safety critical avionic systems. New software fault tolerance techniques and methodologies for quantifying software quality have been presented.

Software Life Cycle Costs(LCC) are becoming more and more dominant. Logistic people regard forgotten or underestimated software maintenance as the sleeping giant. Still software maintainability is the key in future programs for keeping software life cycle costs down.

In the future, some of the main areas in software-research will be:

1. Formalization of requirements
2. Quantitive metrics
   - Software performance
   - Software reliability
   - Software Fault Tolerance
3. Fault Tolerant Software Architecture
4. Real Time High Order Language
5. Integrated tool environments
6. Software Reliability Techniques
7. Automation in Software Production

The Symposium was closed with thanks to all who contributed to its performance.

5. CONCLUSIONS

It is comforting for the present and encouraging for the future to see how many managers of software projects are concerned with the quality of their products (design, analysis, management, production, documentation, maintenance ...) and how much effort they envisage to invest in the improvement of this quality.

Practically all the objectives aimed at, however, result in an increase in:

- size of project management teams;
- hardware and software environment facilities;

- volume of code;
- capacity and performance of computers.

This is why the user must become well aware of these problems and their consequences prior to defining his development policy and to making his technical choices. He must learn to adapt his ambitions to his possibilities and his real requirements.

As the complexity of software systems increases, it becomes increasingly difficult to predict how they will behave in any given scenario. Testing is of value in this area, but has its limitations. The highest priority in improving the quality, maintainability and efficiency of software designs must, therefore, be assigned to improvement of the design process itself.

It is certain that nothing will prevent ADA from substantially characterizing the field of design and development of avionics software in the future. However, ADA, complete with APSE, must be carefully developed such that its full potential benefits are realized. There is a great preoccupation with developing integrated sets of software tools. This has been recognized in the ADA development program, with the requirement for APSE (ADA Programming Support Environment) assuming prominence equal to the language development. Whatever language is being used, a need exists for an integrated support environment. More work must to be done in clarifying and examining standardization options in this field. An urgent requirement exists for becoming familiar with the potential and limitations of ADA/APSE.

The software Validation and Verification (V & V) efforts are non-standard, individualized approaches to software testing with each company, agency, country, etc. generating testing techniques for their specialized use without the concern of reusability. The disappointment is the inability for others to use not only the ideas but also the actual code without totally recreating specialized versions. Uniform definitions and agreement on testing methodologies would be beneficial.

Although software experts believe that V & V is an acceptable approach to software development, the initial investment has discouraged many acquisition managers. Still, software experts believe that V & V is a cost effective venture. However, the cost reductions are minimum during the development phase but more extensive during the in-service support phase. Since there is no documented data that accurately assesses the value added (cost and quality) for systems using this technique, and since most of the system cost growth will occur during the in-service support phase, it has been impossible to get a full commitment to the V & V approach to take advantage of the potential such a process holds.

Many papers centered around the existing software development process and sought to define or develop techniques for its implementation. Based on the content of these papers, the foreseeable advancements are evolutionary and will not provide the quantum step that is going to propel software development into a highly productive, low cost, very reliable undertaking; to this end, rather some revolutionary action is required. The level of exploratory resources indicates no such change at this time.

Software technology requires improvements in all areas to cope with future needs. Research and development should be directed towards areas promising improved software quality and reduced software cost. Comprehensive near-term software policy initiatives are required by NATO.

AVIONICS SOFTWARE: WHERE ARE WE?

Willis H. Ware
The Rand Corporation
Santa Monica, California

Abstract

Since the digital computer first flew in an avionics system 25 years ago, the art has progressed from small very slow vacuum tube machines with limited memory to fast chip-based machines that not only do sensor processing but also integrate a variety of data sources into many capabilities--among others, navigation, sophisticated weapons delivery, programmed menu-displays to the air crew. As onboard computer hardware has proliferated, software inescapably has also. From a few hundreds of program words at the beginning, flight software is commonly many tens-of-thousands of words; frequently, a few hundred thousands; and in some cases, even a million. Thus, implementation and management of software resources has become a major problem area for military services. The paper explores dimensions of the issue as it now exists, suggests many positive actions unerway, and proposes a direction in which the future may well move. It concludes that software will continue to be troublesome; progress will come slowly.

Before asking where are we in avionics software, we first ought to ask where have we been; then we can ask where we are going. Among the earliest, if not the first, digital computer to fly, was one in the MA-1 fire control system. It was developed in the early 1950s for the F-102 fighter to control the Falcon missile and folding fin rockets. A vacuum tube machine using subminiature tubes, it operated only at hundreds of operations per second. It could not accommodate the full dynamic behavior of the fire control and launch calculations; primarily it supplied constants to the analog computation of the fire control equations. Its operational program, measured in hundreds of words, was contained in a small magnetic drum on which individual instructions were spaced circumferentially to accommodate the latency time of memory access. Parenthetically one might wonder whether anyone today would still know how to do minimum latency programming. The program was done in a machine language, and life-cycle support of software had not yet emerged as an ongoing operational issue for military services. Interestingly, MA-1 still flies in the F-106; its operational life has exceeded 25 years.

By contrast, the Air Force F-16 fighter is among the most highly automated of deployed operational aircraft. Instead of vacuum tubes, F-16 avionics uses modern microelectronics; its fire control computer has 32,000 words of memory and runs at some half million operations per second. While 128,000 words of computer program fly in every operational F-16, of this only about 30,000 words are written in the high-order language JOVIAL J-3B-2. The rest are in the machine language of whatever computer happens to be involved, e.g. the microprocessors in the stores management subsystem, the signal processor in the radar, the microprocessor in the heads-up display. As a second example, the U.S. Navy F-18 flies 400,000 words of program which executes in some 13 machines; about 2 percent of it is in an HOL. The LAMPS helicopter also has some 400,000 words of program on board; of it about 30 percent is in an HOL. Turning to larger aircraft, the U.S. Navy P-3 land-based airborne early-warning vehicle has 700,000 words on board, 128,000 words of core memory, but also over one million words of diagnostic and maintenance programs. Originally done in assembly language, it is now approximately 70 percent in HOL after a major upgrade. The analogous Air Force AWACS has 512,000 words of memory. Some modern-day military aircraft have a centralized computing function with point-to-point wiring for signal paths, whereas others--notably recent tactical fighters or upgrades--use data bus architectures with some measure at least of distributed processing. Some onboard systems have a capability for degraded modes of operations as equipment fails, but others have virtually no fallback capability.

Indeed, we have come a very long way in terms of the amount of software that flies with modern-day aircraft, a long way in terms of the speed, size, and power attributes of electronics, and a long way in the level of automation. Obviously, though, the penetration of HOL languages is neither as uniform nor as deep as often believed. It varies from a few tens of percent in contemporary fighters to many tens of percent in the larger vehicles, especially those that have been upgraded. The MA-1 system is where we came from; the F-16, F-18, P-3, and AWACS are representative of where we are.

To explore where we are, let us note some dimensions of modern avionics software. First of all, what is "the total software job" of a modern aircraft? The first and most obvious component is that which flies onboard--software in a radar, in the heads-up display, in the stores management system, in the inertial navigation computer, the air data computer, possibly a fuel management subsystem, perhaps a separate display and controls management subsystem, or perhaps an entirely separate system (as in the B-1 bomber) to monitor all else for malfunctions and to capture maintenance data. In the most complex of aircraft, all such systems will be networked together by a bus arrangement often presided over by a central computer complex. Just as we have not progressed all that far in the use of HOLs, neither does the reliability of equipment always exhibit outstanding field reliability. It might be, however, if one were to measure some ratio such as reliability-to-complexity, perhaps progress is better than normally perceived. Be that as it may, onboard equipment does malfunction and so modern aircraft have an extensive array of ground support equipment, which for the most part is software controlled.

Among the ground-based diagnostic and repair facilities is one often called the Avionics Intermediate Shop, generally a highly automated complex of test stands that can examine equipments which either have or are presumed to have failed. Supporting the AIS level of maintenance is a depot or rear echelon capability that generally deals with problems at the electronic card level rather than at overall equipment level. Here one finds a wide variety of automated test equipments, most of which are software driven and controlled.

Just as equipment must be maintained in operational status, so aircrews have to be properly trained. For that purpose crew training devices of many kinds are utilized for modern aircraft. Perhaps the most widely known example of this technology is not an airborne one but rather the crew training simulators used by NASA for manned space missions.

All the software implied by the prior discussion had to have been developed somehow, so one commonly finds one or more program development environments for the working programmers which includes all the diagnostics, testing tools, recordkeeping tools, languages, compilers, etc. used in modern software development and its management. Contrary to the common perception of 25 years ago that operational software would never change, it is now widely understood that it does and will continue to change for a variety of legitimate reasons. Therefore, a modern deployed highly computerized fleet of aircraft must be supported by a facility for the life-cycle support of software. Among other things, the latter includes a software development environment for each of the computers whose software is to be maintained and for each of the languages that are flying and in ground support.

However, it must also include a variety of test and simulation tools to assure that software changes have been properly made and will not lead to new anomalies of behavior. In addition, there usually also must be appropriate flight vehicles, perhaps especially instrumented, to make certain that all is well with the updated software before it is dispatched to the field. Sometimes, especially when the computer involved is a microprocessor, any software involved will be regarded as simply another component of the equipment, which will be tested end-to-end for functional performance without special explicit tests of the software. This view has been taken, for example, in new commercial aircraft; and hence, any life-cycle software support in such instances is seen as an obligation of the vendor supplying the basic equipment.

Thus, when one speaks of "the software job" for a modern aircraft, it proves to be an enormously large undertaking. Moreover, it implies ongoing attention to change just as does physical modification of aircraft. At minimum, the software job for a modern combat aircraft is a few hundred thousand lines of code; but if all of the software development tools must be built as well, then it can be many hundreds of thousands. For large sensor platforms with extensive diagnostics the corresponding number can exceed a million. The many components of the total software job share a unique characteristic however; they must all be kept in lock step. The ground-based test equipment must examine equipment boxes as they are, not as *they were a year or more earlier*. The crew trainers must mimic the aircraft as it is, not as it was. Life-cycle support must match equipment as it is, or in some circumstances as it will be. Finally, we may have to do configuration control of the software by production block numbers, and in some circumstances even by the tail number of an aircraft.

Thus, in addition to all the usual management difficulties associated with large and diverse software undertakings, there is now a time synchrony aspect that can be difficult to accommodate. It is especially so, given that the several components of software are, in the United States Air Force at least, handled by different groups of people at different places, in different organizations--some military and some contractor--and often with different planning and funding arrangements. In one sense it can be observed that the wide exploitation of computer technology in modern aircraft has lead us into a morass of difficult organizational and technical issues; but it is part of the price to pay for military air power capability.

Where do we stand? Computing technology continues to be very dynamic. Software is a very manpower-intensive undertaking that is difficult to manage and frought with danger. Seemingly, the number of successes as measured by budget and schedule are far outweighed by the number of failures. Software generally is entirely too unpredictable at the outset in regard to its eventual cost, the date of its eventual completion, and its ability to realize the user requirements as he really wants them to be, versus what he perceived them to be at the outset of the program. Finally, there is always the question of the resources needed for ongoing life-cycle support.

What is being done about this array of problems? Special languages have been developed and are gradually coming into use to support the initial requirements analysis phase, and to assist in tracking the translation of such requirements into corresponding software capability. Some automated design techniques are slowly appearing, and there is discussion of, but not too much success in, *reusable portable software*. New languages--such as ADA--are being completed and will be introduced into military software programs. Standardization is finally achieving some level of success; at least in the Department of Defense there now exists military standards for data buses, for languages, for instruction set architectures, and for acceptable HOLs.

Some of these techniques have just moved from the research laboratory into the development world, and hence their effect has yet to be felt. Clearly, some installations and some contractors are in the lead; but even in the best places software disasters continue to occur. Beyond that though, there is still a major part of the software community supporting the Department of Defense that has yet to be introduced to the most contemporary tools, techniques, and management approaches, and to be trained in their effective use. Some things thought originally to have been truths are yet unvalidated and may prove to be mythology. It remains to be seen, for example, whether extensive use of an HOL will result in cost savings either in the initial development or in the ongoing life-cycle support. Limited evidence suggests that it will not happen. Similarly, the arguments for standardization suggest significant cost savings in terms of logistic support, replication of software development support tools and environment, training of people, etc; but in this case also, the penetration of standardization is presently minimal and its payoffs are yet to be realized and measured.

Hence, there are advances underway that promise future benefits for accommodating the difficulties that have appeared through wide exploitation of computing technology in modern aircraft. There are other

things coming along, however, whose consequence is harder to judge. Ahead, for example, is a new VHSIC era in semiconductor technology. Might it make possible capturing specific functional capabilities on a chip, thereby eliminating at least some part of a software job? Yet to be felt are the fullest impacts of the microprocessor advances. As they grow smaller and more capable, will we see an increasing array of equipments that are microprocessor based, may contain extensive software, but are treated, tested and maintained as functional equipment without regard to software content? Is there some possibility, for example, that some part of the software job can be passed off to the suppliers of equipment? Clearly, there has been progress in dealing with software; there are new ideas and advances in train. What though might the future look like? What conjectures can be made about it?

Admittedly, it is a judgment call, but in my view the odds are that every problem that we now have with software, will continue to be so and perhaps more so. Why? The automation levels on newer aircraft are bound to increase as the task environment in which the aircrew must function becomes evermore complex. Even today's pilots commonly state that the job in the cockpit is far beyond what can be accommodated by an individual. Many, for example, say that of an aircraft's total capabilities, a given pilot is familiar with and exploits perhaps only a third of them. Probably, however, each pilot specializes himself to a different third.

One can project, I think, that avionics systems will get smarter in the sense that their behavior will be perceived by users as having some level of intelligence. The evolving technology of expert systems and knowledge-engineered systems is bound to find its way into aircraft, and as it does the "IQ", so to speak, of the avionics system will gradually increase. There are some very profound technical consequences of projections such as these not only for implementing better systems but also for maintaining them. If we are successful in building smarter avionics systems, will we be equally successful in assuring that they can be maintained in operational status and repaired in the field by military manpower?

The computer programs to provide higher levels of automation and expert systems will be enormously more complex than even the worst of today. The programming job will be more difficult to do and will use more sophisticated techniques; thus one can imagine that the management task of producing such software, and testing it to assure that it is adequately error-free, will also increase in difficulty. If smart and highly automated systems are to be accepted by their users, then they must be available when needed and they must perform as expected when needed. Thus we have ahead of us the task of handling much better the whole business of malfunctions, anomalous behavior, fault detection, maintenance and repair.

It is commonly acknowledged throughout the industry that there now is a serious shortage of properly trained personnel--both for creating computer programs as well as for managing the process--and the situation is not likely to improve. Computer technology is still diffusing so rapidly through the world at large that the demand for such talent is high everywhere. Hence, the commercial organizations that support military systems will have to compete with a rapidly expanding commercial world for what is already in short supply.

Everything clearly suggests that aircraft will continue to get ever more expensive; therefore looked at as a resource to be maximally exploited for military advantage, a faster sortie turnaround rate will be essential. This point reflects itself partly in the issue of managing faults and error problems, but it also reflects itself in the ground maintenance aspect--notably rapid testability and identification of trouble, efficient means for removing-and-replacing, and fast checkout of an aircraft on its way to the next sortie.

Look now at technology push and technology suction. Whether one country likes it or not, potential opponents will make technological advances that result in new capabilities and opportunities for military action. To counter such ever-increasing threat, at least equal progress will be needed in order to remain superior. Thus, the military establishment will always require the best of technology to accommodate an enlarging threat. The technologist himself, as he perceives the need of the military services, will encourage use of his capability. Thus, military systems will always be on the forefront of technology.

For most of the parameters important to flight performance and the technologies behind them, there simply is not much room to grow. Progress in such things as propulsion, lift-to-drag ratio, or thrust-to-weight ratio will be measured in a few tens of percent at the best. In contrast, growth in raw computing power still promises at least a factor of 10--and perhaps even 100--as we move into new more elegant semiconductor and switching technology. If--and it's a big if--we can build software whose growth in capability approximates that of computing hardware, then we may be able to vigorously exploit a resource not commonly perceived as part of an aircraft, namely data available to it from sensors and information that can be derived from such data. It is conceivable to my mind, that while the usual vehicle performance parameters may grow only slowly, large increases in military capability may nonetheless be achieved through *a highly automated, wholly integrated information infrastructure* to manage the vehicle and support its crew.

As we better understand decisions and actions that a pilot makes, complex information processing systems can be designed to do things that were formerly the prerogative solely of the human mind. With new progress in electronic technology and therefore in computer hardware, we will be able to architect systems whose individual components are not only highly reliable but whose system availability is even higher. This will not minimize the whole issue of health status monitoring, fault diagnosis, and repair; but it probably will affect how maintenance is done, the magnitude of the logistics tail, and the geographical location of repair facilities. Military aircraft will continue to push the forefront of information technology; and because skilled personnel will continue to be in short supply, and because we will be building ever more elegant systems, things in the software world are not likely to get much better in terms of the overall job, in spite of all of the positive efforts now underway.

Life-cycle support is not likely to change either, although its details may somewhat. Historically, what was once called maintenance of software was seen as a nuisance to be minimized. It is now understood that the *changeability of software--as awkward as* that may be or as demanding of resources as it may be--is still the easiest way to accommodate inevitable change in the military threat and the unavoidable changes demanded by operational users. Whatever the problems of modifying software might be, it is still easier

and less expensive than modifying physical equipment. Among other things, replication of software is automated, error-free, and inexpensive. It can be shipped to the field for installation rather than bringing every serial number of some equipment back to a depot for modification.

While computer hardware apparently will be no problem in a real sense because of progress in the semiconductor industry and such specialized efforts as VHSIC, sensors and instrumentation may be a problem; they have been in the past in other areas of automation. We will need to know, for example, what the aircrew is up to, which way they are looking, what they are intending to do, what they are planning to do in the next minutes. How will the crew communicate its intentions to automated systems? I can imagine that the development of appropriate sensors to make measurements not only on the world but also on the crew and the aircraft--and they will obviously be computer-based ones--might be a pacing item in moving up the level of automation.

If this is a valid projection for the future, then one must focus immediately on the software issue of the future. Clearly, airborne computer hardware is progressing extremely rapidly, and one can stipulate that it is unlikely to be the pacing item. On the other hand, we will continue to have all the problem dimensions now associated with and understood about software. We will certainly encounter new ones, some of which have been suggested. We may have to build software intensive systems that function on various aircraft, systems that are cross-service, or even multinational systems. One way or another though, we who are in the computer hardware and software world of military aircraft can look forward to a continuing growth of onboard computery, and a continuing ever-widening exploitation of information. After all, information is the universal commodity that keeps all complex systems functioning, and to make better use of it, we have only digital technology.

The military user will state his operational requirements as best he can perceive them, but software intensive systems are so complex in their eventual behavior that the user often cannot understand what he wants until he has first seen and has used the end product. For relatively simple software-based systems, this issue may be inconsequential or even absent. For the most involved ones, however, we will probably have to accept a fairly intense modification, update, and support activity in the early part of operational life as the user really comes to understand what the system can do or might do. This is quite aside, of course, from whatever design flaws or anomalies of behavior remain in the delivered software for whatever reason. The odds are that life-cycle support of software will never stop throughout the entire life of an air vehicle or its systems. Moreover it has become clear that everything is likely to have a much longer operational lifetime.

The real pacer of an information intensive avionics future is almost certainly to be the frontend intellectual understanding of just what functions the user wants, and how they become software. We will need research emphasis for many years to understand how to develop and manage not only user requirements, but also their translation into software requirements. In this regard, one must note that the precision of dialogue demanded by the software design process is rarely matched by the precision with which user requirements can be stated and transferred to software designers.

While we should not predict the future with gloom and doom, nonetheless I think we must realistically acknowledge that software will continue to be troublesome. Moreover, it will almost certainly continue to be seen as the culprit for a whole variety of ills. I do think that changeability of software, and hence its life-cycle support demand, will increasingly be acknowledged as a positive feature rather than an annoying nuisance that cannot be avoided. Thus, some of the past negative attitudes toward software in this regard will gradually erode. While the software world has devoted itself in the decade of the 1970s to understanding and developing specialized management techniques for implementing it, the problem in software is often an incomplete intellectual understanding at the beginning of what it is really supposed to do.

Of course the computer software scene will improve somewhat; there are a lot of positive actions underway. We will improve our mechanics of organizing and managing software projects. Occasionally some software project will even go smoothly because it will represent a task that is conceptually simple; or a task that is fully understood intellectually; or a task that has already been done before, and therefore a kind of software prototype exists. In particular, the intellectual prototype of the job to be done will be understood. To be sure, HOLs will improve and they will help; but system developers are moving slowly into them. New HOLs such as ADA will undeniably be very useful in providing a structured way in which to describe software requirements. Moreover, such languages will provide an unambiguous way to communicate among people that are involved in the user requirements, in system requirements, and in software requirements phases.

Let us acknowledge all the positive steps now underway to improve the whole software situation in the broad. Let us take credit for all the improvements that will happen in and from HOLs, improvement in management techniques, improvements in descriptive languages, and all else. No matter how good we get, software in my view will continue to be troublesome and progress will come slowly. There will be the ever-increasing demand for military capability to counter new threats; software that must be built will ever-increasingly try to implement complex intellectual information processes; the unavoidable intellectual understanding of the frontend requirements process will continue to be a pacer. It is not a dreaded future though; it is an exciting one as we in the avionics business learn how to create a fully integrated information infrastructure for military aircraft. Unlike the relatively stable future of many technologies involved with the air vehicle itself, we in the avionics business have several decades of dynamic future as we learn how to exploit information as a resource to achieve large advances in military capability.

AVIONIC SOFTWARE DESIGN

Dr. David E. Sundstrom
General Dynamics
Fort Worth Division
Fort Worth, Texas 76101

SUMMARY

Time, theory, and applications experience have lead us to an understanding of a very specific software product, the avionics operational flight program. The knowledge we now have permits us to identify a common software structure and design methodology. The structure reflects the characteristics of the avionics environment, and is applicable to mission processors, stores management processors, integrated cockpit control processors, and possibly others.

In this paper we identify a number of characteristics of the avionics environment and relate these characteristics to design requirements for a common software operating structure. The adoption of this structure supports a design methodology which has many desirable features, beginning with common naming conventions, an emphasis on data flow specification, and clearly identified design responsibilities. A discussion of the structure and the design methodology are provided in the paper.

This structure and methodology has been demonstrated in production F-16 airplanes and is currently the developmental basis for multiple software programs in advanced F-16 versions. It represents a mature and transferable technology.

I. Introduction

Digital processing in the airborne environment has reached a point of maturity in the core area of avionics. Processing tasks in this area, which may be conveniently called mission data processing include navigation, weapons delivery, displays control and annotation, and self-test fault data reporting. These tasks were among the first to be automated in early analog computers and the first to be automated in digital processors.

The shared experiences of the avionics community in achieving numerous implementations of mission knowledge processing has brought this area to a maturity. It is a maturity in the sense of our ability to understand, analyze, and trade-off requirements and in the sense that implementations are now generally of low risk. As a community, we have a broad understanding of the organizational and management requirements and the computing and support resources required. We are even beginning to converge on the appropriate cost factors. Other signs of maturity include the emergence of standards, of applied high-order languages, quality assurance practices, and user groups.

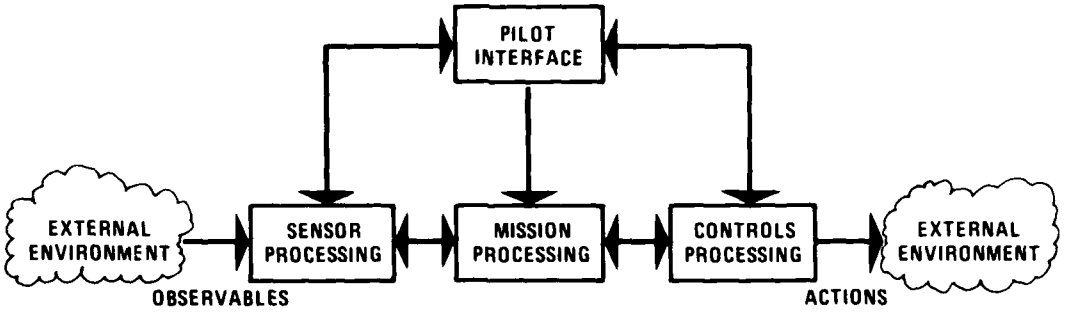
But this is not to say that all avionics processing has reached this stage of maturity. Integrating software, that associated with mission knowledge processing, is best known. The lesser understood areas run in three directions: towards high speed, high data rate signal processing applications, toward the replacement of hardwired logic, and toward the specialized control of sensors and control devices. Our topic in this paper is the design of avionic software, specifically the mature areas of mission knowledge processing software.

We can understand the general characteristics of mission knowledge processing software by understanding the avionics environment. Hence, our paper provides first some thoughts on the characteristics of the avionics environment. Following this, we explore the structure of an avionics software system. Finally, we discuss the design process itself and identify several concepts in design methodology which have proven benefits.

II. Characteristics of the Avionics Environment

The primary function of an avionic system is to integrate the data acquired by sensors into information forms useful to the pilot. Interactive controls, displays, sensors, and targeting devices are included in current systems, and it is clear that advanced systems will involve many forms of automatic control of flight.

There are three significant characteristics (reference Figure 1) related to this functional environment that strongly influence avionic system design and the associated software design. First, we are dealing with a real-time, sampled-data environment which contains numerous feedback control loops. In many cases, we rely on the pilot to close the loop, but in others, such as controlled pointing of a radar antenna or a target



- REAL-TIME, SAMPLED DATA SYSTEM WITH DISTRIBUTED PROCESSING
- ALL PROCESSING AND COMMUNICATION AMONG SYSTEM ELEMENTS IS DETERMINED IN DESIGN PHASE
- EFFECTIVE PILOT INTERFACING REQUIRES A PREDICTABLE, REPEATABLE SYSTEM RESPONSE

Figure 1 The Characteristics of the Core Avionics Environment Are Known

tracking device, the control process has been built into the mission processing function. The second factor influencing design is the simple fact that all systems users, tasks, and communication are known a priori. Finally, the cost of development and operation, and the need to limit pilot workload requires that a deterministic execution of modes and tasks be pursued, in which predictability of response is essential. The recognition of these primary characteristics can lead to significant design implications as outlined below

Real-time, sampled-data control system environment. This environmental factor requires that sampling frequencies be compatible with hardware stability and system accuracy requirements. Further, the inter-equipment data bus which transports the sampled data must enable periodic data transactions having known propagation delay.

These factors must be recognized and dealt with in structuring the avionics interface and associated software designs. The data bus control protocol for example, must provide for the required periodic data transactions that are so critical to the operation of sampled-data control systems. (In this regard, protocol designers must recognize that their systems must meet the stringent requirements of control systems and not simply the more flexible requirements of communications systems.) Within the software bus control algorithm, provisions must exist to carry out the required data transactions at the predetermined rates and phases.

Known tasks design environment. The avionic software environment is benign in that all computing users are known at design time. This is in contrast to the usual university or industrial computing environment where the users and their programs are unknown and some may be hostile. This fact, that all applications are known a priori, leads to significant reductions in operating system complexity.

University courses in operating systems stress generality, protective features, capabilities for tasks to request I/O or scheduling of other tasks, and privileged modes. These general capabilities, when applied directly to the avionics environment, are wasteful. They are wasteful because the powerful features of a general operating system require considerable memory to store and duty cycle to execute.

A better solution is to downscope the generality and arrive at a simple operating system tailored to the avionics environment. Privileged modes are not needed, for example, since the access rules and control can be exercised through the normal design process. It is better not to allow tasks to request I/O or to schedule other tasks: recall that we are dealing with a control systems application where time-predictability must be a major design consideration.

Deterministic execution design. The software to control the operations of avionics needs to be repeatable in operation and predictable in performance. Several factors lead to this conclusion. First, the need for time periodicity in control systems. Second, the need to be able to analyze and predict behavior of algorithms. Third, the need to test the software and to be able to reproduce test conditions and replicate events. Fourth, the essential need of a human operator to rely on the consistent actions time after time.

Deterministic software design can be achieved through a priori scheduling, through exclusive mode definitions, and state transition tables. For example, the application configuration of the avionics flight program should be a function only of cockpit switch positions at each instant. The alternative is to remember previous switch settings so that operational history becomes a part of state logic. This is very undesirable because of the uncertainties induced when testing or troubleshooting, not to mention pilot operations.

III. A General Structure for the Avionics Flight Program

The characteristics outlined above identify the environment of the avionics flight program. The characteristics, when coupled with the design and structural advances in the understanding of software technology, make possible the identification of a general structure for an avionics flight program. The technology advances referred to are those in areas of data flow and control partitioning.

The general structure of the avionics flight program (Figure 2) consists of only five major parts. As may be seen from the figure, the internal partitioning of functions reflects the hierarchy present in the external environment. Recognizing the existence of a functional separation is in itself a significant achievement, for it allows categorization of signals and events and helps to avoid multi-purpose types of input data. Klos (Klos, L. C., 1978) addresses the interface approach to avionic software definition in his NAECON paper.

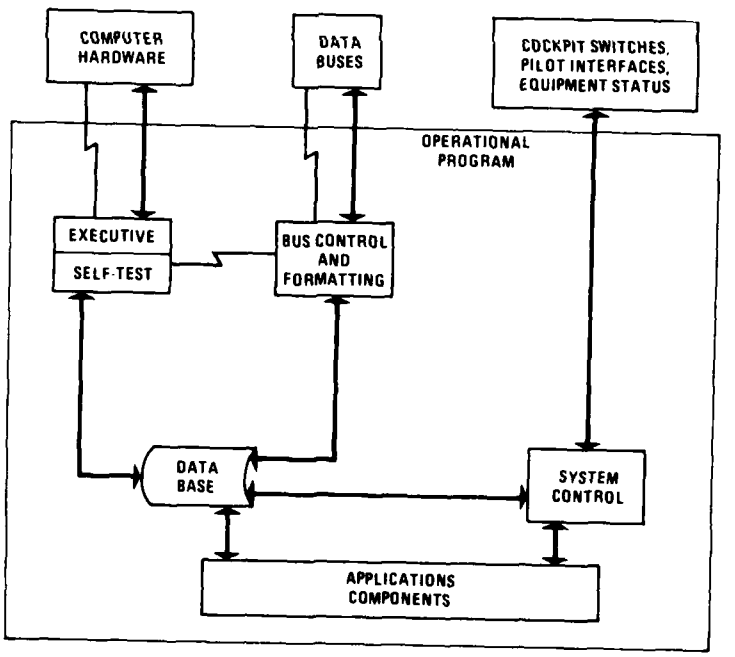


Figure 2 A Generalized Avionics Software Product Structure Has Been Identified

In this general structure, the executive has only the function of time partitioning. The executive enables periodic tasks, background tasks, and interrupts. The executive, however, has no knowledge of the applications or the switch commands or of control modes. These are handled elsewhere. Closely related to the executive is the computer self-test which has the continuing task of validating the operations of the machine.

The bus control and formatting section deals only with getting data on and off the bus, the timing of data transmittals, and the handling of transmission errors. The formatting portion handles the scaling and packing of data as well as validity checking. Hence, the program component has the function of gatekeeper, allowing only valid data to reach the program data base.

The data base itself has two sections. One section reflects the interface to the external environment while the other reflects the data shared by applications. When structured via a high-order language such as JOVIAL, the data base provides an excellent place from which software interface management can proceed; more discussion is provided in Section IV.

The system control component may be thought of as the applications executive. This component provides the single-point focus within the flight program for processing cockpit switch commands, pilot interfaces, and mode status of external equipment. By having a single-point control component, displays content can be insured consistent and modes can be clearly defined for entry and exit. Further, this approach proves favorable to implementation through table structures, which provide a great help to insuring a deterministic, state-oriented system. Edwards, in her AGARDograph article (Edwards, J. A., 1980) discusses this in detail.

Finally, the applications components emerge. These represent the classical functions of navigation, weapons delivery, and other mission-related functions. It is in the applications area that the specialization of the program occurs, where, for example, the difference between fire control and stores management processors is most distinct.

This simple structure has been used in several versions of the F-16 and has been proven effective. It provides a design environment that is manageable on understandable program framework. Moreover, it shows the way to providing executive-to-application interface standards. The following section will elaborate on these considerations.

IV. A Supportive Design Methodology

Simple ideas, such as module naming conventions, can be more useful than perhaps is commonly recognized. Nomenclature, then, is one element of a design methodology. Two other elements to be discussed here are the interface approach to data flow identification and task assignment partitioning for the design team. These elements fall into place once the structure of the flight program is recognized.

Nomenclature conventions support the development by clarifying relationships and ownership of modules. We use two levels of identification (Figures 3 and 4). First, a component is identified. A component is a major task area that usually requires one or more engineers to develop. A component consists of segments. A segment corresponds to a module or procedure.

Once identified, a component is given a name and a two-letter tag that will be used to prefix all segment names. For example, the navigation support component has tag NS and all segments within navigation support are coded with NS as a prefix. Thus, we have NSALT, the altitude adjusting segment, NSWINDS, the wind computing segment, and NSINIT, an initializing segment.

This convention introduces wonderful coherence into the development and maintenance of the program. Since the nomenclature appears in the product specification - where components correspond to chapters - and in the listings, it is easy to read the documentation and to relate the parts. In the laboratory environment, the nomenclature is a great aid in recognizing code.

Suppose, for example, an engineer in the laboratory observes a computer error occur. By inspecting the registers, the faulted instruction is found and the indexed listings then identify the troubled segment. The name of the segment immediately identifies the component and hence, the responsible design engineer.

FUNCTIONS
- TASKS PERFORMED BY AN OFP WHEN VIEWED EXTERNALLY (EXAMPLE – AIR TO GROUND)
- APPEARS IN DEVELOPMENT SPECIFICATION

COMPONENT
- A MAJOR BUILDING BLOCK OF AN OFP (10 TO 20 PER OFP). NORMALLY CONSISTS OF A NUMBER OF SEGMENTS (10 TO 40 EACH), RELATED TO IDENTIFIABLE FUNCTIONS.

SEGMENT
- AN ELEMENT OF A COMPONENT (E.G. PROCEDURE, MODULE, OR SUBROUTINE ARE EQUIVALENT TERMS). A SEGMENT IS A SCHEDULED TASK, EITHER THROUGH THE EXECUTIVE OR SYSTEM CONTROL ACTIONS.

SYSTEM CONTROL
- THE COMPONENT OF AN OFP THAT CONFIGURES THE APPLICATIONS TO SUPPORT THE REQUIRED FUNCTIONS.

Figure 3 Operational Flight Program (OFP) Nomenclature Conventions

- ALL COMPONENTS HAVE A TWO CHARACTER DESCRIPTOR, FOR EXAMPLE

COMPONENT	ABBREVIATION
EXECUTIVE	EX
SYSTEM CONTROL	SC
AIR-TO-GROUND	AG

- ALL SEGMENTS CARRY A COMPONENT PREFIX AS PART OF THE SEGMENT NAME. EXAMPLE

SEGMENT	COMPONENT
AGLADD	AIR TO GROUND
FXFRZ	FIXTAKING

- DOCUMENTATION FURTHER USES CPC NUMBER IN IDENTIFYING INTERMEDIATE FLOWCHARTS, EQUATION SETS, AND SEGMENT NUMBERS EXAMPLE

CPC 6 FIXTAKING	6.1 THRU 6.26 SEGMENT NUMBERS
	6.1 THRU 6.33 EQUATIONS SETS

Figure 4 Software Item Naming Conventions

Of course, the flight program structure has further helped us here because the coherent partitioning of functions has made it possible to isolate problems. Under this approach, we are not troubled by interacting changes. Flags, hidden logic, and assumptions are minimized. Since these are things which result in interacting events, the cases of change-induced problems have been minimized.

The interface approach to data flow identifications recognizes that data interfaces must be complete and well-specified. Two aspects emerge, first, the specification of the software to the external environment, and second, the specification of segment-to-segment data relations. Interface management has long been recognized as a critical discipline in systems design; our extension has been to carry the same recognition down into the detail design of software within a single processor. An example of the types of structures and approaches being used is illustrated in the graph of Figure 5. More detail on the structuring of interfaces with a program is contained in the Klos and Edwards papers previously cited.

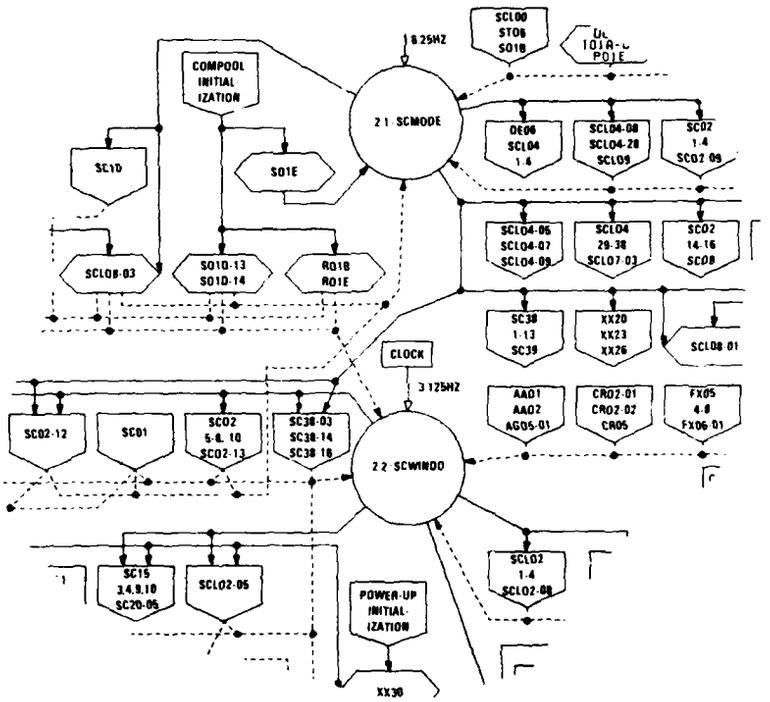


Figure 5 This Graphic Approach to Interface Definition is Backed by a Detailed Specification of Signal Attribute

It turns out often enough that external interfaces may be negotiated by engineers who are not familiar with details of machines and languages. Where this condition exists, interfaces may be structured without adequate consideration made of software impacts. For those implementing MIL-STD-1553 data busses, the Edwards paper (Edwards, J. A., 1979), on formatting will provide excellent guidelines.

The final element of a supportive methodology relates to organizing to accomplish the design task. The flight program design team can be organized around the structure identified in Section II. Such an organization will have the hardware/software integration experts developing the executive and bus control components and the mechanization experts who understand the pilot-cockpit interface and the modes and functions of the weapon system will be assigned to develop the system control component. The application components will be identified through functional partitioning and will be assigned to experts in navigation, altitude filtering, numerical integration, and other such functions. And very importantly, the data base can be managed by an identified person whose responsibility it is to negotiate data flows and structure the organization and distribution of data.

V. Recent Experience

The structure and methodology described above was initiated through funded research (Engelland, J. D., 1977), and brought to bear on the full-scale development of F-16A/B fire control software. Most recently, the concepts expressed in this paper have been applied to the development of multiple software programs for the F-16C/D airplanes.

The F-16C/D represents a considerable avionic extension over that of the F-16A/B, and requires software to be developed for a fire control computer (different from that of F-16A/B), for a stores management processor (greatly different from F-16A/B), for an up-front integrated cockpit controls processor, and for a multi-function displays processor. The latter processors are completely new development items for the F-16C/D.

Our review of appropriate design approaches and methodologies supported the basic concepts outlined in this paper. In several cases, the desire to fully embrace standardization and to introduce common elements into our approach caused us to strengthen our perceptions. The resulting approach (illustrated in Figure 6) introduced hardware features common to all the processors and to support equipment. Software commonality was introduced through the general program structure and design methodology outlined here. A summary of the extent of software-related commonality in this project is provided in Figure 7.

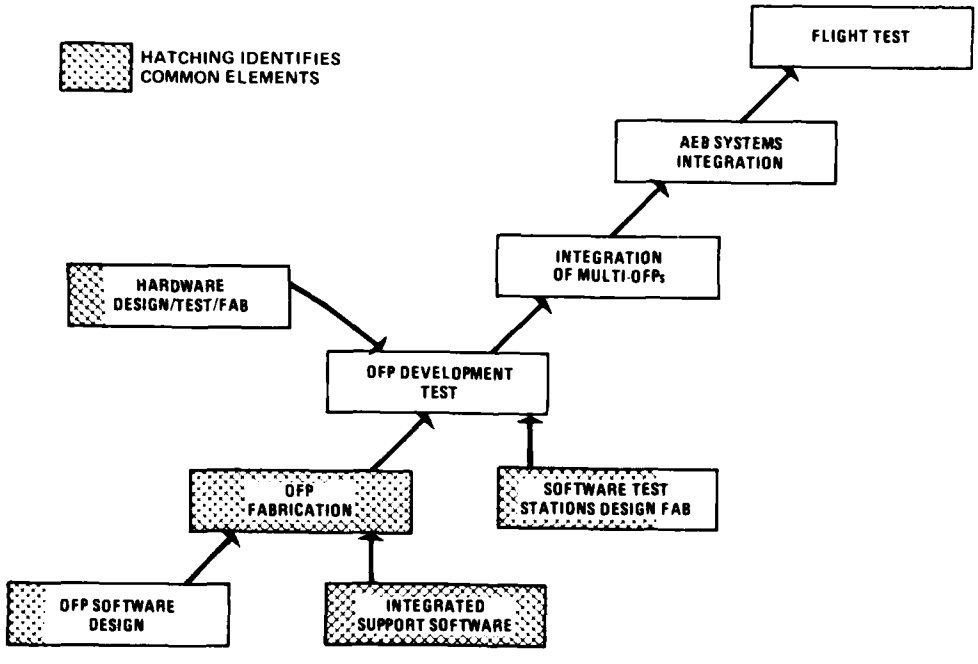


Figure 6 The Design Approach Emphasizes the Use of Common Elements to Minimize Development Effort

OFP SOFTWARE DESIGN
- **COMMON DESIGN METHODOLOGY**
- **COMMON EXECUTIVE, DATA BUS COMPONENTS**
- **COMMON STRUCTURAL DESIGN**

INTEGRATED SUPPORT SOFTWARE & OFP FAB
- **COMMON HIGH ORDER LANGUAGE**
- **COMMON DEVELOPMENT TOOLS**
- **COMMON FAB PROCEDURES**

SOFTWARE TEST STATIONS
- **COMMON PROCESSORS AND OPERATING SYSTEMS**
- **COMMON REAL-TIME MONITOR HARDWARE**
- **COMMON OPERATOR INTERFACE AND CONTROLS**
- **COMMON RECORDING & RETRIEVAL SOFTWARE**

AIRBORNE HARDWARE
- **1750A OR Z8000 PROCESSORS (INTERIM PROCESSOR)**
- **COMMON DESIGN OF INTERRUPTS & FAULT REGISTER**
- **COMMON BUS INTERFACE OPERATIONS**

Figure 7 Common Elements Simplify Development and Anticipate MIL-STD-1750A in Production

V. Summary

The approach described above has been successfully used in the development of F-16A/B fire control software. The same approach is proving valid in the development of F-16C/D software, which includes fire control, stores management, integrated cockpit controls, and multi-function display management. Before adopting this structure and methodology on the major F-16C/D project, we undertook a careful scrutiny of methodology. The resulting approach added new ideas but also sharpened the old.

The F-16C/D software flight programs all use the general structure illustrated in this apper. The executive was developed once and provided as a common element to four design teams. The same is true of the bus control software. Data base management practices have been made common and the nomenclature conventions also adopted.

In short, the maturing of a general structure for avionic software design is at hand.

Acknowledgements

The structure and system reported on in this paper builds heavily on foundation work accomplished by J. D. Engelland, L. C. Klos, and J. A. Edwards. D. H. Daggett has supported this structuring and constructively steered it in the maturing process. S. A. Alford has been instrumental in the implementation of the executive, control algorithm, and hardware provisions.

References

1. Edwards, J. A., 1979, Inside MIL-STD-1553: Interface Format Guidelines, NAECON, pp. 419-425.

2. Edwards, J. A., Jan. 1980, Advanced Design Concepts and Practices in the F-16 Mission Computer Software, NATO AGARDograph on Guidance and Control Software.

3. Engelland, J. D., et. al., August 1977, Operational Software Concept, Final Report, AFAL-TR-77-78.

4. Klos, L. C., 1978, An Interface Management Approach to Software Development, NAECON, pp. 741-748.

Software Development; Design and Reality

Dr. H.von Groote and Dr. F.Schwegler

Messerschmitt-Bölkow-Blohm GmbH
Unternehmensbereich
Flugzeuge

Postfach 801160, D-8000 München

SUMMARY

This paper describes some experiences gained during the development of the Operational Flight Program of the MRCA Tornado. A short outline of the organisational structure and of the Avionic System is given, followed by the description of the different design and test stages of the OFP. Then some general reasons are presented which caused major changes to the software specifications and which are believed to be true for any development of a new avionic system. The paper concludes with a description of the purging process of the assembler program.

1. Introduction

The Panavia MRCA-Tornado has been developed as a joint European venture of the three Nations Great Britain, Federal Republic of Germany and Italy for the operational scenario

* Interdiction	IDS Tornado
* Strike	
* Close Air Support	
* Reconnaissance	
* Air Defence	ADV Tornado

the ADV being a UK-only requirement. The first production aircraft of the IDS was delivered in 1980 to Cottesmore training centre; right now the third batch is in production, and operational service has started.

Fig. 1 shows the overall organisational structure for the design, development and production of the aircraft. The Ministries of Defence of the three Nations have established the Nato MRCA Management Agency (NAMMA) to act as their Executive Agency who in turn contracted Panavia Aircraft GmbH as industrial organisation formed by British Aerospace (BAe), Messerschmitt-Bölkow-Blohm (MBB) and Aeritalia (AIT). The Panavia Partner Companies are the prime subcontractors for design, development and production. With respect to the avionics system they each act as Panavia's agents towards the national equipment suppliers. Each of these partner companies also manage an avionic system test rig for full scale integration of the avionic system. The responsibility for the avionic system's design and development was placed on EASAMS, UK, who in turn subcontracted ESG of Germany, SIA of Italy and itself to staff the international Central Design and Management Team (CDMT), which was responsible for the avionics system & subsystem design, and the International Software Team (IST) which formed part of CDMT and was responsible for the generation, testing, and configuration control of the Tornado Main Computer software. Each company formed "In House Teams" who managed the CDMT controlled development rigs and implemented CDMT design requirements on equipment level.

Since this can only be a rough outline of the tri-national organisational structure we refer to previous AGARD presentations (Sanderson K., 1980 and Harris D.J., 1979) where the different tasks and activities during the development phase are described in more detail.

Today the tasks for CDMT have ended, the remaining activities for productionising and modifications are placed on the a/c companies under the leadership of MBB for the IDS Tornado. The now-named International Programming Team (IPT) still co-resides with ESG who manage their test rigs.

2. The Avionic System of the IDS-Tornado

In order to provide a background for what the software was written for, we give a short outline of the avionic system describing those features which have major implications on the development of the Operational Flight Program residing in the Tornado Main Computer. The most critical requirements for the system are

* high speed low level flight
* first pass aquisition of targets
* accurate weapon delivery
* acceptable workload of crew
* high reliability and maintainability.

These requirements are fulfilled by a semi distributed system which is divided in the following subsystems (see Fig. 2)

* Navigation
* Weapon Delivery
* Terrain Following/Automatic Flight Director System (TF/AFDS)
* Displays and Controls
* Recording
* Computing

Generally speaking each subsystem has its own computing power which allows for a stand alone capability. The full performance, however, is provided by integration of the subsystems performed by the computing subsystem which consists of the central computer, the Tornado Main Computer (MC), and two Interface Units (IFU). The MC is programmed in Assembler and communicates with the other systems via one-way serial data links. For those equipments which do not conform with the serial data standard, the IFUs perform the necessary data conversion.

The system encompasses dual redundant high precision navigation optimised by a Kalman Filter, automatic track and attack steering, A/G and A/A target aquisition and tracking, weapon aiming calculations and generation of release cues, and various modes of navigation and target fixes. Ample use is made of electronic displays to provide comprehensive information and guidance to the crew.

More detailed description of the system and its functions can be found in Bross, P.A., 1981.

3. System Design and Test Stages

The System Design was broken down into subsequent stages which provided more and more detailed specification of the system and its components. (Fig. 3)

The prime document which formed the contractual basis between NAMMA and Panavia for full scale development was the Performance and Design Requirements (PDR) document. In the PDR the operational requirements, the avionic equipment fit, and the overall functional characteristics were laid down.

The avionic system was then divided into subsystems which were defined in subsystem specifications. These reflected the relevant parts of the PDR and describe the structure of the subsystem, the internal and external interfaces, the operationally visible performance, and general aspects like safety, reliability and maintainability. They explicitly or implicitly became the high level specification of the operational software.

On the hardware side we then have the equipment specifications which became the contractual technical description of the equipment. They implicitly also defined the dedicated software specific for that equipment.

The system software, within the Main Computer, is defined in a series of Software Requirements (SWR), which describe the operational requirements by defining the logical and mathematical functions to be carried out by the software. They include description of crew actions, switching functions, priority and iteration rate of tasks, and relations to other documents like other SWR, equipment specifications, the Interface Control Document (ICD). They also contain two annexes, the Logic and Equation Development and the Outline of Test Requirements.

The ICD describes all the interfaces of the avionic systems detailing all signals, signal types, ranges, accuracies, iteration rates, sources and sinks, and so on.

These documents were generated by CDMT. Except for the ICD they were written in plain English and maintained manually since at that time computer aided tools and methods of specifying real-time systems were not yet available. We shall come to that point later.

The SWR formed the basis for the program definition performed by IST. The first level document was to be the program specification describing the program structure, the functions of the scheduler, packages and subroutines, the data base and I/O areas. The internal structure of the packages, their routines, subroutines and interfaces are defined in the Package Specifications. Then follow the flow charts and finally the source code listing which was to be carefully commented in order to put the assembler code in context with the applicable documents and how they were realized by the code. The rules for programming and documentation are laid down in the Programming Guide.

Testing of the software and the system is performed in basically 5 stages. (Fig. 4)

Stage 1, located at ESG Munich, is the workbench for the IST where the programs are generated and tested using a host computer, early development models of the MC and TV/TABs, and an external computer for open loop simulation of the avionic environment with the aid of models derived by CDMT for evaluation of avionic equipments and SWRs.

Stage 2 is the test facility for integration of the software with the prime avionic equipments using test procedures derived from the Outline Test Requirements of the SWRs. Initially there were two rigs, one at ESG and one at EASAMS, the latter has been closed now. They comprise development models of the prime avionic equipments with associated Special to Type Test Equipment (STTE), and computing facilities which allow for simulation of avionic equipments, stimulation of data, and limited closed loop simulation using a simplified aircraft model. The necessary recording, replay and display facilities are also provided. From Stage 2 the software is released to the subsequent Stages 3 and 4 with formal Quality Assurance Certificate.

Stage 3 was the early flight test facility run by CDMT where critical functions of the subsystems Navigation, Terrain Following and Weapon Delivery were tested and evaluated under dynamic airborne conditions in two Buccaneers as test bed aircraft. The facility also provided a test bench harness for integration tests and a ground replay and analysis system. The activities at Stage 3 have stopped several years ago.

Stage 4 are the full scale integration rigs at each a/c company. Essentially they are ground mock ups of the complete avionic system, fitted with the most recent hardware standard. They provide ample test facilities by STTEs, external computers for data acquisition, simulation and stimulation, interfaces to other a/c systems like flight control, power supply, and cooling. They also allow for full scale closed loop simulation, at MBB not only including the avionic system but also the flight control rig.

The primary tasks are

* integration and performance testing of avionic hardware and software within the avionic system and with other a/c systems
* flight clearance of the software for flight test and production aircraft and support to flight line and production line
* definition and control of software configurations which became a progressive task due to the evolution of the software, its adaption to the flight line requirements, and the differing equipment build standards of the flight test and production aircraft.

The tests are laid down in test procedures derived from the Subsystem Specifications and SWRs and are performed under the supervision of the industrial and national quality assurance authorities.

Stage 5 finally are the flight test facilities of the a/c companies. The flight tests are performed with fully instrumented prototype Tornados supported by appropriate ground facilities like recording, replay and analysis systems and for some time a tracking radar.

During the development of such a complex system, of course, a considerable amount of changes to the system's software is to be expected. In order to deal with these changes, formal procedures were established which are divided into three levels:

* Software Query (SWQ)
* Program Change Request (PCR)
* Software Requirement Change Request (SWR CR).

The purpose of a SWQ is to report a software problem encountered at any of the test sites and to call for investigation by CDMT/IST. Depending on the result of the investigation a program change and possibly a SWR change may be required.

A change to the software officially delivered by IST is initiated by a Program Change Request raised by one of the test sites. The PCR is forwarded to all test sites and to CDMT/IST for assessment. If agreed, a software change is generated by IST and delivered to all test sites. A PCR may also result in a SWRCR if there was an error or insufficient definition in the underlying SWR.

A Software Requirement Change Request may be raised by either NAMMA, the a/c companies, CDMT, or IST. All SWRCRs were assessed by CDMT and subsequently by IST and the a/c companies prior to inclusion into the relevant SWR. If a corresponding change to a Subsystem or Equipment Specification would result from a SWRCR, NAMMA approval is also required.

There are many reasons for a SWRCR some of which will be described and illustrated by examples in the following.

4. The law of permanent changes verified.

Certainly the ultimate aim of all SWRs is a complete, unambiguous, physically logical, and consistent description of what the software should do. So the programming team can then go ahead and generate a nice structured program which is well documented and easy to maintain. However, in the development of a complex real-time system the prerequisites of completeness, unambiguity and consistency are very hard to fulfil, and unavoidably many changes to the SWR and of course subsequent changes to the program will occur. Some of the reasons will be described in the following.

4.1 Ambiguity of Software Requirements

Let us look at unambiguity and let us for a moment assume the SWRs were complete and consistent in the sense that they contain all functions to be performed and the definition of all data.

As already mentioned the SWRs are written in plain English. Furthermore they are written by system engineers who know much about the system's and equipments' functions and inter-relationships and usually less about the conditions and restrictions inherent to the computer hardware and programming language. Conversion of these SWRs to program and package specifications then demands quite some system knowledge by the programming team and often resulted in SWRCR for clarifying purpose in order to avoid misinterpretation of the requirements. To gain and maintain this knowledge is not easy especially in view of the fact that the fluctuations of the IST staff was quite high due to the international structure.

The advent of modern tools and methods for unique specification of real-time systems like formal specification languages very much avoids the possibility of misinterpretations of SWR and also reduces the necessary level of system knowledge of the programmers. However, these specification languages are usually aimed at program specification, not so much at formulating software requirements. Anyway, somewhere down the line the verbally expressed PDR have to be converted into an unambiguous formulation, be it at the very last step, i.e. the coding or be it at an earlier stage, if formal specification languages are used. And that is the place where misinterpretations come in.

4.2 Evolution of SWR

The statement that a SWR is complete can only be a relative statement, it reflects the present status of the knowledge, experience and imagination of the people involved in the definition of the operational requirements and the design of the system and its functions.

Let us consider the development of the Navigation Kalman Filter (KF) as an illustrative example (Fig. 5). After selection of data sources (IN, Doppler Radar) and position fixing methods, the equations including the moding and cockpit interface were laid down to form the draft issue of the KF SWR. The filter was then programmed in Fortran and optimised using "real world" simulation models of IN, Doppler, and fixing methods. This resulted in the first issue of the SWR which was complete in the sense that it contained all information available from the simulation. The next step was to generate the assembler program to debug it and to test it against the Fortran KF. Eventually recorded flight test data of the IN and Doppler became available from Stage 3 together with external references provided by a DECCA Navigation System. This allowed for numeric optimisation of the KF and assessment of sensor and system performance. The resulting SWR now also reflected the real equipment hardware, however, with the restrictions of the installation in the test-bed aircraft and the accuracy of the external reference. When the filter finally became airborne in a Tornado prototype at Stage 5 with a high precision navigation tracking radar as external reference it turned out not only that again numeric optimisation was necessary but also that the performance could be considerably enhanced if some structural changes to the filter were introduced. This evaluation was not possible in the previous stage because of the different installation of the equipments and the lower accuracy of the external reference.

With these changes incorporated in the SWR it is now also complete with respect to the real world but not with respect to the imagination of the system designer. For example they were and still are inventing new fixing methods of automatic nature or with crew involvement that further enhance the system performance.

Looking back we realise that the specification and generation of the airborne program started with a SWR which defined the modelled function but was still incomplete. At that time dummy routines were implemented to cater for the identified but then still undefined function. However, during development changes to the structure, new functions, and different iteration rates became necessary that not only affected the KF package itself but also the scheduler and overall program organisation.

This example demonstrates the evolution of a SWR and the inevitable changes to it although the objective was formulated right from the beginning and the realisation had been carefully assessed in computer simulation before the programme specification and generation started.

4.3 Unpredictable errors

There are other sources for changes to a SWR. These are effects which were hard to predict or were thought to be covered by other means so that they need not be considered in the SWR.

Take e.g. unpredictable malfunctions of an equipment, errors that were not generally known as are e.g. gyrodrifts or the time delay of air data sensors. Originally it was thought that malfunctions will be detected by the Built In Test (BIT) and signalled to the system. Maliciously enough, however, they did occur and they were not detected by the BIT because from the point of view of the equipment everything was correct, no component was broken. In most cases these errors occur very seldom and cannot be reproduced on STTE since they are probably caused by external interference. Therefore they tend to reveal themselves very late in the test stages usually during flight test. So what you do is to design a software monitor which checks the output of that equipment and in case of a malfunction generates warnings to the system and/or the crew. For example if the error is of a jumpy nature where physically there cannot be jumps the monitor simply checks for the continuity of the data.
In any case you end up with a major change to a SWR and the program itself.

4.4 Trade-offs between H/W and S/W changes

Inevitably during the development of a complex integrated avionic system one is faced with the decision whether to modify an equipment or to cure the problem by a software change or even to do both: To cure the problem at least partially by an interim software solution until the equipment modification has been performed.

The reasons for hardware modifications are manifold, e.g.

* misinterpretation of specification often resulting in a wrong sign of I/O data which of course can easily be cured by the software.
* erroneous behaviour under unspecified conditions which often can be solved by monitors.
* insufficient performance of equipment under extreme conditions which in some cases could also be cured by a software solution.

For example during flight test it turned out that the attitude data of the SAHR were not accurate enough to provide redundant inputs for terrain following under all flight conditions. The problem could be cured by adding high precision accelerometers to the gyro platform which of course was very costly and would impose a considerable delay to flight testing. There was, however, an alternative solution to the problem using the interface between MC and SAHR: In order to compensate for the earth rotation and the transport rate of the aircraft the MC provides slewing commands to the SAHR which are well defined functions of basic navigations. The solution was ot use other redundant information available to the MC, namely the air data vertical velocity and the vertical velocity derived from Doppler velocities and the SAHR attitude data. Comparison of the two leads to correction terms which are added to the original SAHR compensation terms. This software change was implemented and flight tested within about half a year and improved the accuracy of the SAHR attitude data by a factor of two.

Although software solutions instead of hardware modifications are very attractive: They are faster implemented, often are cheaper and in some cases avoid long negotiations with a supplier and retrofit to the aircraft. But they also have their drawbacks as can be seen from the last example: the rather simple and logically clear-cut interdependence between the data source (MC) and data sink (SAHR) became more complex, mixing two physically different functions and involves new data sources coming through the backdoor. This makes it harder to test and maintain the software and the hardware.

The list of reasons for changes to SWR we have mentioned so far is certainly not complete. There are others like changing operational requirements, replacement of older equipment by a new generation, or expansion of the system by new equipments. But the reasons given already clearly demonstrate the impossibility to fulfil the postulation that a complete, unambiguous and consistent description of what the software should do must be laid down before the program specification may start, unless you do it for a system that already exists.

5. Clean programs, patches, and all that

There is a unanimous wish for clean programs, i.e. a program that does not contain any patches. But we have seen that changes to a program are unavoidable for a variety of reasons and in an assembler program these changes are realised by patches, since rearranging the source program instead takes quite some time and is likely to create more errors in the program and the configuration control than you wanted to correct.

Furthermore, many of the changes are of experimental nature, they have still to be evaluated at the different test stages previously described and you don't want to permanently incorporate them into the program too soon.

Last but not least flight tests need long term planning due to lay ups, reserving ranges or tracking radars, etc. The test program itself then usually must run through in a short time. So errors found before and during flight tests have to be corrected very quickly if the corrections are to be tested within the same flight test programme.

However, from time to time you make a freeze and define a new baseline program which is to contain all program packages implemented so far and all program changes available and tested at that time. Incorporation of the patches and testing of the new baseline take at least half a year up to one year depending on the number of patches and occupy most of the manpower and facilities at Stages 1 and 2.

When the job is done, already a new pile of patches immediately builds up for the following reasons:

* Some program changes could not be incorporated in the baseline since they were not yet fully tested at all Stages. New errors had been found and corrections generated.
* Not all Software Requirement Changes had been implemented at time of freeze and new ones were generated in the meantime.
* The baseline obviously was targeted for the most recent hardware standard of the avionic equipments, e.g. the production standard. For reasons of costs and availability, however, the different test stages were outfitted with quite a mixture of early development, late development, and production equipment which in some cases necessitated hardware related patches specific to the individual test stages and aircraft.
* Eventually the decision was made to double the MC memory, and the full OFP was targeted for that memory size. However, the new hardware was not available until recently. So in order to fit into the smaller memory the OFP was divided into a baseline and major amendments and patches which were configured to the test programs at the different flight test stages. With the new MC being available, of course a new baseline was to be generated.

Especially the last two points are the reasons for the fact that the software tested at Stage 2 is different from the OFP released from Stage 4 to flight test and production aircraft, contrary to the original development planning.

In total, patches cannot be avoided unless development is rigorously stopped, no new requirements are accepted and one year of intense work including flight testing is spent for cleaning the program.

This statement, however, is put into a new perspective with the advent of real-time high order languages which allow for separate compilation of program packages or modules. Here the cleaning is more or less automatically done by the compiler and one is "merely" left with configuration control which in itself is already a hard job.

References:

P.A.Bross, The Computer System of the Tornado, 31st AGARD G&C Panel Symposium, Roros/Norway, 1981

D.J.Harris, Software Development for Tornado - a Case History from the Reliability and Maintainability Aspect, AGARD-CP-261, paper 37, 1979

K.Sanderson, Main Computer Software for the MRCA Tornado, AGARDOGRAPH No. 258, paper 11, 1980

List of Abbreviations

A/A	Air to Air
ADC	Air Data Computer
ADV	Air Defence Variant
AFDS	Automatic Flight Director System
A/G	Air to Ground
BIT	Built In Test
CDMT	Central Design and Management Team
ESRRD	E-Scope and Radar Repeater Display
ICD	Interface Control Document
IDS	Interdiction Strike
IFU	Interface Unit
IN	Inertial Navigator
IPT	International Programming Team
IST	International Software Team
KF	Kalman Filter
MC	Main Computer
MFK	Multi Function Keyboard
MRCA	Multi Role Combat Aircraft
NAMMA	Nato MRCA Management Agency
OFP	Operational Flight Program
PCR	Program Change Request
PDR	Performance and Design Requirements
SAHR	Secondary Attitude and Heading Reference
SMS	Stores Management System
STTE	Special to Type Test Equipment
SWQ	Software Query
SWR	Software Requirement
SWRCR	Software Requirement Change Request
TF	Terrain Following
TV/TAB	TV/Tabular Display

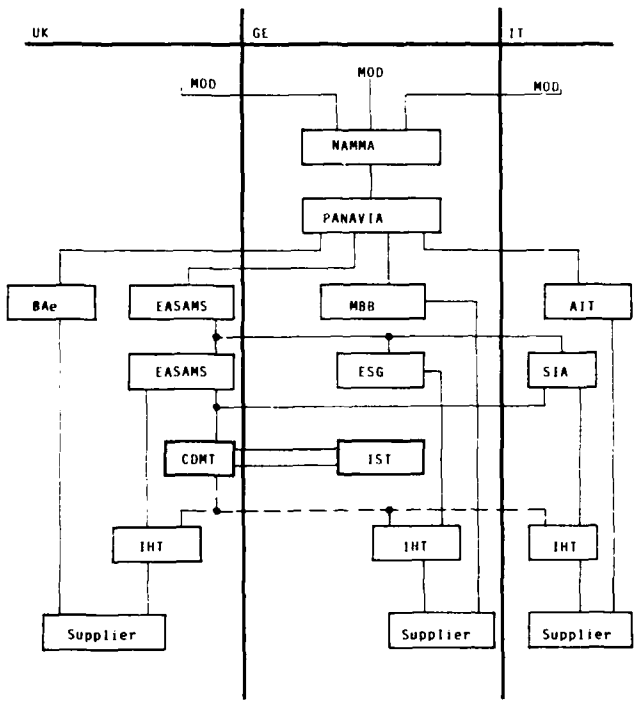


Fig. 1 : Organisational Structure of Development Phase

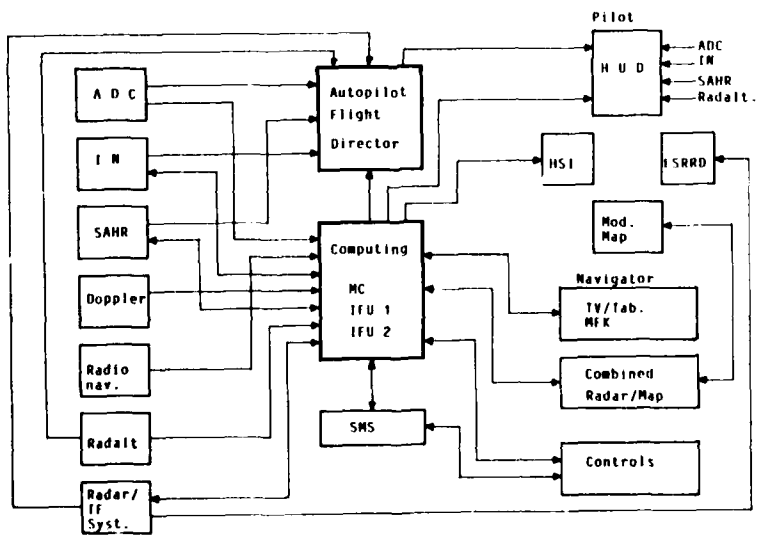


Fig. 2 : Tornado Nav/Attack System

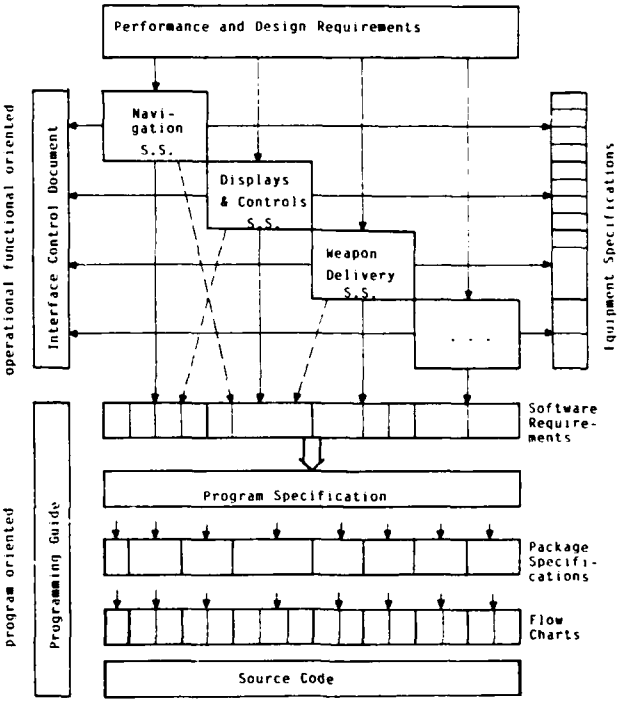


Fig. 3 : Design Stages

location; User	Facilities	Tasks
Stage 1 ESG, IST	Host computer MC, TV/tabs (electr. representative) Simulation by computer models	Program Generation Testing and Integration of modules
Stage 2 ESG, (EASAMS); CDMT	Development models of prime equip-ments and STTE Data simulation, stimulation, re-cording and replay (Integration Software) Limited closed loop simulation	Integration of prime avionic fit Integration and proving of S/W Performance assessment Release to Stage 3 & 4
(Stage 3) BAe Warton; CDMT	2 Test-bed Buccaneer fully instru-mented Tet bench harness Recording & replay facilities	Early flight test of critical functions in Nav., TF & Weapon Aiming
Stage 4 BAe Warton, MBB Munich, AIT Turin; a/c comp.	Ground mock up with full av.syst. fit & STTE Most recent H/W standard Data aquisition, simulation, sti-mulation & replay (DASS, IDAS) Interfaces to other a/c systems Full scale closed loop simulation	Integration & performance testing of H/W & S/W Integration with other a/c syst. Flight clearance & support for flight test & production S/W configuration definition and control
Stage 5 BAe Warton MBB Manching AIT Turin & Sardinia	Prototype Tornado a/c's fully instrumented Recording & Replay Tracking radar	Full flight test of av.syst. H/W and S/W

Fig. 4 : Test Stages

KALMAN FILTER

TEST DATA	DEVELOPMENT STAGES	CONTENT OF SWR
	SELECTION OF DATA SOURCES (IN,DOPPLER) & FIXING METHODS DEFINITION OF EQUATIONS, MODING & COCKPIT INTERFACE FORTRAN KF	THEORY
SIMULATED SENSORS ARTIFICIEL DATA AND FORTAN KF	ZERO ORDER OPTIMISATION, PERFORMANCE PREDICTION CONFIDENCE CHECKS MC - PROGRAM DEBUGGING AND TESTING OF MC PROGRAM	SIMULATED WORLD
STAGE 3 F.T. DATA DECCA REF.	NUMERICAL OPTIMISATION, ASSESSMENT OF PERFORMANCE	REAL EQUIPMENT DECCA REFERENCE TEST-BED A/C
STAGE 5 F.T. DATA TRACKING RADAR	OPTIMISATION OF STRUCTURE & NUMERICS, ASSESSMENT OF FULL SENSOR & SYSTEM PERFORMANCE WITH POSITION AIDING NEW FIXING METHODS	REAL WORLD

Fig. 5 : Evolution of a Software Requirement

MASCOT DEVELOPMENTS TO IMPROVE SOFTWARE STRUCTURE AND INTEGRITY

H R Simpson
British Aerospace Dynamics Group
Six Hills Way
Stevenage
HERTS SG1 2DA
UNITED KINGDOM

SUMMARY

The principal features of the MASCOT approach to design are described and some possible developments to give improved software structure and integrity are proposed. These developments are concerned with three areas of the MASCOT approach:

- Subsystem Structure. The present definition of subsystem structure can be generalised to give a hierarchical framework for functional decomposition.
- Process Synchronisation. The present range of synchronisation primitives can be extended to allow complex requirements to be met in a more direct and efficient manner.
- Data Access Control. The kernel executive can be extended to give run time enforcement of the data access constraints implicit in the network structure which is the product of the design phase.

The proposed developments are entirely consistent with existing MASCOT concepts and, with little difficulty, can be incorporated into the system building, run time executive and dynamic monitoring software. It is a powerful feature of MASCOT that the supporting software which underpins the method (sometimes called the MASCOT machine) is not excessively complex and so can be reasonably easily implemented and readily understood. The proposed developments preserve this feature.

1. INTRODUCTION

A recent 'Guided Tour of Program Design Methodologies' (Bergland 1981) idendifies four program design methodologies which are used or discussed more than most:

- Functional Decomposition
- Data Flow Design
- Data Structure Design
- Programming Calculus

These four headings are reasonably self explanatory; each methodology is described in some detail in Bergland's paper together with a supporting bibliography which need not be repeated here. The important point to note at this stage is that these methodologies are traditionally concerned with program design rather than system design and thus are primarily directed toward single thread sequential program structures. It is hardly surprising therefore that these methods as conventionally described do not cope well with multi processor target environments and take little advantage of the possibilities for using parallel programming constructs to give improved software structure within a single processor.

Recent developments in the programming language field eg ADA and the Communicating Sequential Processes technique of Hoare, 1978 contain parallel programming features designed to overcome the limitations imposed by single thread structures. The approach is generally based on a form of direct process to process inter-communication which results in a highly synchronous relationship between adjacent parallel processes. Furthermore the interaction mechanisms require that processes know the names of the other processes with which they wish to communicate. This erodes modularity and results in functional dependence between conceptually parallel processes.

MASCOT (Modular Approach to Software Construction Operation and Test) is a language independent method which places the expression of software parallelism above single thread structuring by conventional programming techniques. It results in a high degree of asynchronism and functional decoupling which is well suited to multi processor target environments. The method is based on a general approach to information system design and so ensures that software design is carried out in a manner which is consistent with sound system design principles.

Many of the principles of the functional decomposition, data flow design and data structure design methodologies can be applied to system design in which parallelism is exploited. MASCOT provides a medium in which these techniques may be used with the confidence that the resulting software design can be implemented in a reasonably efficient and straightforward manner. It is not easy at this stage to see how the programming calculus approach relates to MASCOT. However in terms of verification of correctness the strong modularity of MASCOT offers distinct advantages by breaking down the verification task into more manageable components.

2. DESIGN DECOMPOSITION

The software design problem can be considered within the wider context of information system design. An information system is concerned with data representation and manipulation; it is considered to be bounded by the data environment which separates it from the external elements (eg electro-mechanical components, chemical processes, human operators) with which it interacts. Although such external elements do not form part of an information system they are of course its "raison d'etre" since the ultimate purpose of any information system is to react to and affect the physical environment in which it exists. The data interface with external elements is of vital importance since it constitutes an information system's perception of the outside world.

At its broadest, but not especially helpful, level of representation we can regard any particular system (S) as being contained within its own particular environment (E) as shown in figure 1.

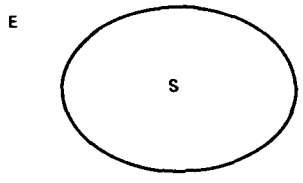


Figure 1 The Problem

This form of expression becomes more useful if we insist that E is described purely in terms of the information interface (static and dynamic properties) as seen by S and forbid the inclusion of any active processing elements (explicit or implicit) in the environment definition. In general this will not be easy but the discipline enforces a clear and complete specification of interfaces and prevents that class of design error which arises from misunderstandings concerning the dynamic properties of separate but related processes.

Clarity of expression demands some method of partitioning the environment into various elements. The processing system can now be defined in terms of its reaction to, and impact on, the individual elements in the environment. This can be represented diagrammatically as shown in figure 2.

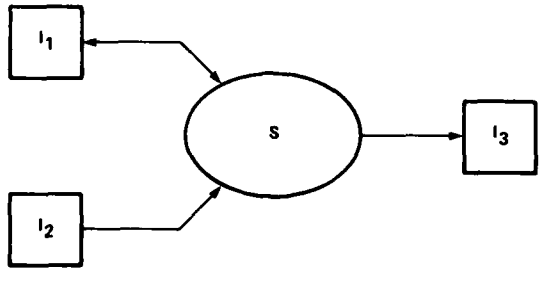


Figure 2 Environment Definition

In such a diagram a rectangular box labelled I represents an information element in the data environment and the closed curve labelled S represents the active processing element. A line joining S to I represents an interaction path. Arrows may be placed on interaction paths to indicate information flow.

Although at this level we have chosen to represent the system as a single processing element surrounded by a number of information elements this does not necessarily imply the absence of active processes outside the system boundary. Such processes will generally exist and often give rise to interactions between elements in the information environment. The characteristics of external processes must be taken into account when carrying out the top level partitioning and element definition. However the method does require that the functional specification of the processing element must be couched in terms of the information with which it interacts.

In describing an information element it will be necessary to consider a number of aspects including structure (formats, syntax), flow rates, semantic content, signals and stimuli, etc. It may also be necessary, as in MASCOT, to define the formal mechanisms (access procedures) by which interaction between active processes and the information environment takes place. This is relatively straightforward compared with the specification and description of processing functions where it often becomes extremely difficult to define precisely what is to be done without making assumptions as to how the processing function will be implemented. A method of functional decomposition is therefore required not only to break up the task into more manageable components for implementation and integration purposes but also to give substance to functional specifications by identifying the underlying functional elements.

Functional decomposition should be carried out in a manner consistent with the principles outlined above and should also be capable of logical extension to any depth necessary to achieve a satisfactory measure of functional modularity and simplicity. Accordingly decomposition of an active processing element is carried out by identifying component processes together with an internal information environment to be used for intercommunication purposes. Thus a first level of decomposition might take the form shown in figure 3.

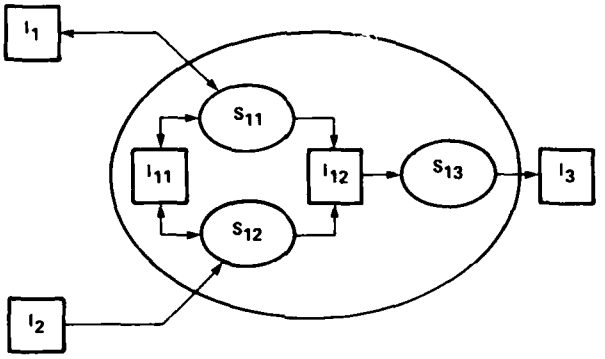


Figure 3 First Level Decomposition

This form of representation is based on functional issues and does not directly depend on the underlying supporting resources (processors, operating systems, communication highways). Of course the nature of the computing and communications environment does affect the functional decomposition but the design approach is one that emphasises functional considerations. Resources will always impose the ultimate constraint on realisable functions but ideally we seek an approach where there is considerable flexibility and ample capacity to meet all foreseeable requirements.

A key feature of this method of decomposition is the emphasis placed on inter process communication with no direct process to process interaction allowed. Communication is regarded as a function in its own right with all the attendant analysis, specification and design implications. It would be wrong to regard the method as producing a set of passive information elements which are operated on by active processing elements. The dynamic properties and functional significance of the information indicate a vital active role. Information propagation aspects are treated explicitly which allows timing considerations to be formally addressed in real time system design.

Design decomposition into processing elements which are always buffered from one another by intermediate information elements induces a powerful form of modularity and maximises functional decoupling. When designing any processing element it is only necessary to consider the characteristics of the surrounding information environment. When designing any information element it is only necessary to consider the characteristics of the processing elements with which it interacts.

3. CURRENT MASCOT

It is clear that the proposed design method conforms with the basic principles of MASCOT. Various aspects of MASCOT have been described in a number of reports and papers eg Simpson and Jackson 1979, Jackson 1977, Simpson 1982 and is the subject of an official Ministry of Defence Handbook (1980). The principal structural features of MASCOT as currently defined can be summarised as follows.

MASCOT is a machine and language independent method for software design and implementation which has at its heart a particular form of software structure based on independent parallel processes (known as ACTIVITIES) whose sole means of communicating with one another is through INTERCOMMUNICATION DATA AREAs known as IDAs. IDAs conventionally fall into two broad classes: CHANNELs which are used to pass message data between producer activities on one side to consumer activities on the other; and POOLs which are used to hold data of a more permanent nature. The processing function of an activity is defined by a ROOT PROCEDURE and activities operate on IDAs by means of ACCESS PROCEDUREs. The overall structure of software can be represented by a network of activities interconnected through IDAs and shown as SYSTEM ELEMEMTs on an ACTIVITY-CHANNEL-POOL (ACP) diagram. System elements may be grouped together to FORM SUBSYSTEMs. All activities and some IDAs are contained in these subsystems which provide a higher level of functional definition; other IDAs lie on the boundaries of subsystems and are known as SUBSYSTEM IDAs (SIDAs).

Activities and IDAs correspond to the processing and information elements discussed in the last section. Since an activity cannot be further decomposed it can be regarded as a basic processing element. Subsystems constitute a single form of higher level grouping to represent more complex processing elements. Thus the current form of MASCOT provides for two levels of formal functional decomposition, first into subsystems and then into activities.

Each activity and IDA is an object in the execution environment and can be regarded as a particular 'instance' of a general 'type'. The concept of type allows a distinction to be made between passive definition (ie the code specifying program logic or data structure) and active use (ie the actual system component representing a processing or information element). The type of an activity is defined by the root procedure which supports it and the type of an IDA is the associated data structure definition (or template). Element types are ENROLLED into the programming support environment and are used to CREATE element instances in the target execution environment.

4. HIERARCHICAL SUBSYSTEMS

Although there is a high degree of correspondence between MASCOT as currently defined and the more general design decomposition method described in section 2, there are several structural and diagrammatic differences. The following relatively minor changes to MASCOT are needed to make it conform fully with the requirements of the method.

a. Subsystem Structure. At present subsystems can only be composed of component activities and internal IDAs. It should be possible to also include component subsystems thus allowing full hierarchical decomposition to be used where required.

b. Single Activities. Currently all activities must be contained within subsystem boundaries and there is no provision for forming an activity which exists in its own right. It should be possible to represent and form individual activities without necessarily making reference to a containing subsystem.

c. Boundary IDAs. Current ACP diagram conventions place external IDAs on the boundaries of subsystems. This is misleading. External IDAs should be placed outside subsystem boundaries to indicate their role as separate design elements.

d. Activity Creation and Connection. At present MASCOT separates out the creation of an activity from its connection to the IDAs with which it interacts. This is unnecessary since an activity which cannot communicate has no purpose.

e. Subsystem and Activity Representation. At present the boundary of a subsystem is shown as a dashed closed curve whereas an activity is shown as a solid circle. The same convention should be used for both subsystems and activities, namely a solid curve enclosing the internal structure if any. It is up to the designer to decide whether he wishes to represent the internal structure of any particular processing function in terms of lower level subsystems or activities.

f. IDA Representation. Conventionally MASCOT allows two broad classes of IDA, channels and pools, with different symbols to represent them. In practice the distinction between these two classes is sometimes difficult to make and it is suggested that a rectangular box is all that is needed.

These changes are small but significant and must be backed up by facilities for enrolment of types and creation of instances. The following are proposed:

a. enrol ida type (data struct(....))

This allows a given data structure (expressed in source text form) to be used for subsequent IDA creation. If necessary the named structure may be parameterised (eg to permit the formation of different sizes of IDA from a single IDA type). IDA parameterisation must not affect the access procedure interface to an IDA in any way. Access procedures should be associated with a given IDA type by conventional procedure parameter typing mechanisms.

b. enrol act type (root proc(ida type, ...))

This allows a given root procedure (expressed in source text form) to be used for subsequent activity creation. The named root is programmed as a procedure which can make use of all conventional program structuring techniques.

c. *enrol ss type (create file(ida type, ...))*

This allows a given parameterised command file defining a subsystem type to be used for subsequent subsystem creation. The named file, whose parameters are the external IDA types, contain a set of create commands (see below).

d. create ida(ida inst, data struct(....))

This creates an ida instance from a previously enrolled IDA type. All formal parameters in the structure specification must be replaced by actuals at the time of creation.

e. create act(act inst, root proc(ida inst, ...))

This creates an activity instance from a previously enrolled activity type. If necessary the facility may make provision for a software priority to be associated with the activity as it is created.

f. create ss(ss inst, create file(ida inst, ...))

This creates a subsystem instance from a previously enrolled subsystem type.

g. define ss(ss inst, (ida inst, ...),
(act inst, ...),
(ss inst, ...))

This defines a new subsystem by placing a subsystem boundary around a collection of already created IDA, activity and subsystem instances. Once an element has been enclosed in a subsystem it is only accessible by means of a compound name.

Enrolment requires that the type being enrolled is already fully defined. This results in a bottom-up approach to type enrolment which prevents formal representation of type relationships prior to the full definition of supporting data structures, root procedures and create files. To achieve top-down representation of type relationships it is necessary to introduce types prior to full definition by means of a further set of facilities:

introduce ida type (data struct (....))

introduce act type (root proc (ida type, ...))

introduce ss type (create file(ida type, ...))

These facilities allow types to be specified and used in other type specifications. However no system elements can be created from a type introduced in this way until the code representing all supporting (explicit and implicit) root procedures, create files and data structures with their associated access procedures have been enrolled.

At any level of design decomposition it is not immediately clear as to whether a given processing element will be implemented in terms of a subsystem or an activity. The above facilities require a decision to be made so that the appropriate act or ss commands may be used. This is unlikely to cause much inconvenience but if desired facilities can be defined for introducing and creating (but not enrolling) processes, where a process may be either a subsystem or an activity.

5. INTERACTIONS

The structural concepts already discussed show that formal decomposition during design can be directly supported by facilities for creating the corresponding software processing and information elements in a form which accurately reflects the static structure of the solution. These facilities must be backed up by run time mechanisms which ensure that the basic processing elements (ie activities) execute when necessary and that information propagation is sustained through the information elements (ie IDAs). It should be noted that those processing elements which are further decomposed (ie subsystems) represent functional configuration definitions which play little direct part in software execution; the execution properties of a subsystem are embodied in the component activities and IDAs.

The MASCOT run time environment contains a compact kernel executive which sustains the operation of activities by the scheduling of processor resources. Propagation of information through IDAs leans heavily on a special type of control variable known as a CONTROL QUEUE. An IDA may have any number of associated control queues which can be operated on by a set of standard primitive operations normally embedded within access procedures.

MASCOT attaches great importance to the tight control of interaction. Various relevant aspects, and the way in which MASCOT covers them, can be briefly summarised as follows:

a. Connectivity. Facilities must be provided to build the network of basic processing and information elements and to establish the connections between them. MASCOT system building features allow IDAs and activities to be created and interconnected in a formal manner. By this means static control of access is achieved.

b. Legality. Within the limits imposed by connectivity constraints it is necessary to control dynamic access by restricting the range of interaction operations available. MASCOT makes use of conventional type checking mechanisms to ensure legality of operations. First, an activity may only be connected to IDAs of the type specified in the relevant root procedure parameters. This is checked at system build time. Second, access procedures used by a root procedure must be selected from the set defining legal operations on the associated IDA type. This is checked at root procedure compile time.

c. Exclusion. The method assumes that all processing elements are potentially asynchronous and it is therefore possible for a given information element to be operated on by several processing elements simultaneously (in this context operations which overlap in time are said to be simultaneous). Coherency of data in an information element must be preserved under these conditions. MASCOT allows simultaneous operations but provides facilities (the JOIN and LEAVE primitives) to enforce mutual exclusion (ie to suspend concurrent operation) where this would adversely affect data coherency.

d. Stimulation. Each processing element potentially has a continuous interest in its relevant information environment but it may not be advisable or possible to arrange for it to monitor this environment continuously. Where a processing element has completed its current processing task it may assume a dormant state whilst awaiting further work. Some means is therefore needed to alert dormant processing elements to changes in the information environment which may be of interest. MASCOT contains facilities (the STIM and WAIT primitives) to arrange for appropriate direct cross stimulation but at present this is limited to a single activity-IDA interaction at any one time. It is also possible to set up polling mechanisms (by means of the DELAY primitive to arrange for periodic interrogation of the information environment. Further facilities are needed to combine direct cross stimulation with stimulation on time (by means of a time out on wait primitive - see section 6) and to allow multiple IDA interactions (by means of a multiple wait primitive and composite IDA constructs - see section 7).

e. Sequencing. In some circumstances it is necessary to ensure that separate interaction operations are carried out in the correct sequence. For example it may be necessary for data transfer interaction operations to be preceded by an 'open access' operation. MASCOT as currently defined does not cope adequately with this requirement and a further facility (the CHECK primitive - see section 8) is needed.

f. Integrity. The aspects discussed above identify a number of software mechanisms which can be used to control data access and hence ensure system integrity. However the ultimate guarantee of integrity rests in the use of hardware facilities to limit access to authorised data areas. The MASCOT kernel, if supported by appropriate privileged mode and address base limit facilities, can readily incorporate an efficient form of memory management to provide a program execution environment of the utmost integrity (see section 9).

6. TIME OUT ON WAIT

Many MASCOT users have experienced the need for a variation to the wait primitive which allows escape (ie rescheduling) after a specified time has elapsed. A form of time out can be implemented with the existing primitives but this is inefficient and requires the introduction of additional 'watchdog' activities.

A time out on wait primitive is essentially a combination of the existing wait and delay primitives. It is proposed, therefore, to call the new primitive WAITDEL, specified as follows:

```
PRIMITIVE PROCEDURE waitdel (q : CONTROLQ, delay : INTEGER)
```

The primitive requires two parameters, a control queue and a time delay. A stim to a control queue which has been the subject of a waitdel operation causes rescheduling in the normal manner. If no stim has been received before the time delay has expired then rescheduling occurs as in the conventional delay operation. A case can be made for an indication of the cause of rescheduling (stim or delay) by returning a boolean value although this is not essential since the associated IDA data and the real time clock are both available for inspection and are more relevant to the current data/time situation.

As an illustration of the use of waitdel consider an access procedure named readword designed to read a word from an IDA of type WORDCHAN (using the open access protocol - see section 8). WORDCHAN and readword take the following form:

```
IDA TYPE WORDCHAN
RECORD
      iq, oq : CONTROLQ
      word : WORD
      empty : BOOL
END
```

```
ACCESS PROCEDURE readword (ida : WORDCHAN, delay : INTEGER,
                           value : WORD, nodata : BOOL)
BEGIN
     time : INTEGER
     WITH ida DO
     time := TIMENOW + delay
     WHILE empty AND TIMENOW < time
     DO waitdel (oq, time - TIMENOW) ENDWHILE
     value := word
     nodata := empty
     empty := TRUE
     stim (iq)
     ENDWITH
END
```

7. COMPOSITE IDAS AND MULTIPLE WAIT OPERATIONS

It is the task of the kernel to sustain information propagation under all conditions. There is one particular situation where the existing MASCOT facilities are not well suited to the stimulation requirement. This arises when an activity becomes dormant but needs to be altered to changes in the data associated with any one of several IDAs. Specific extensions to MASCOT are necessary to cope with two aspects:

a. An activity needs to express a requirement for simultaneous access to more than one IDA.

b. An activity needs to be rescheduled in response to stims received on any one of several control queues.

A simple solution to this problem is to permit access procedures to have more than one IDA parameter and to use a form of multiple wait operation which can take as its parameter a set containing any number of queues. Such a solution has an inbuilt source of inefficiency in that queue sets are formed dynamically and repeatedly when the multiple wait operation is called.

This latter aspect can be avoided by allowing the enrolment of Composite IDA (CIDA) data structures which can define a set or fixed size array of component IDA types, together with a declaration of control queue sets whose elements are drawn from these components. Such a CIDA type is a legitimate parameter of an activity type specification. When an activity instance is created the IDA instances satisfying the CIDA parameter are inserted. Thus a CIDA only exists as a grouping of already created IDAs and its only unique components are the control queue sets declared in its data structure definition (it follows that these control queue sets are formed at the time the activity is created). The CIDA facility should be regarded as providing a collective view of a defined IDA set. Access procedures may be written to operate on a CIDA to gain access to any component IDAs; alternatively each individual component IDA may be operated on by its own access procedures in the normal way.

The control queue sets defined within CIDA data structures are intended to be used as a basis for synchronising mechanisms embedded within CIDA access procedures. The following forms of multiple wait operation are proposed for this purpose:

a. PRIMITIVE PROCEDURE orwait(qset: ( )CONTROLQ)

The using activity is rescheduled when a stim is received on any queue in the set. All stims are cancelled on rescheduling.

b. PRIMITIVE PROCEDURE andwait(qset : ( )CONTROLQ)

The using activity is rescheduled when a stim has been received on all queues in the set. All stims are cancelled on rescheduling.

c. PRIMITIVE PROCEDURE multwait(qset : ( )CONTROLQ, n : INTEGER)

The using activity is rescheduled when a stim is received on any queue in the set. The parameter n (0...) indicates a queue which has received a stim. The value of n is remembered between multwait calls and stims are taken in order counting up from the last value of n in an end around fashion. Only the stim corresponding to queue number n is cancelled on rescheduling.

Time out variants of the above would also be needed and it is likely that other qset primitives (eg joinqs, stimqs, etc) would prove useful.

The above CIDA multiple wait proposal must be seen as a fairly complex extension to MASCOT. It does however plug a gap which is significant if the basic method is to cope with all interaction requirements.

As an illustration of the use of the multiple wait primitive consider an access procedure named readone designed to read a word from any of four IDAs of type WORDCHAN (see section 6) which have been grouped into a CIDA of type FOURWORDS. FOURWORDS and readone take the following form:

```
CIDA TYPE FOURWORDS ( w : (0..3)WORDCHAN)
RECORD
     oqs : (0..3)CONTROLQ
     oqs = (w(0).oq, w(1).oq, w(2).oq, w(3).oq)
END


ACCESS PROCEDURE readone (cida : FOURWORDS, value : WORD)
BEGIN
    n : INTEGER
    WITH cida DO
    REPEAT multwait(oqs, n)
    UNTIL NOT w(n). empty ENDREPEAT
    WITH w(n) DO
    value := word
    empty := TRUE
    stim(iq)
    ENDWITH
    ENDWITH
END
```

8. THE CHECK PRIMITIVE

It is important that all members of the set of access procedures which can be used to operate on a given type of IDA should include safeguards to prevent any misuse which would corrupt the IDA data. The check primitive is proposed to close an existing loop-hole and also to provide a more efficient and straightforward means of achieving certain synchronisation effects.

Access procedures may need to impose mutual exclusion constraints to maintain IDA data integrity. Where control queues are used for this purpose two different forms of access protocol can be distinguished.

a. Closed Access Protocol. This protocol is characterised by the presence of join and leave primitives at the access procedure entry and exit points respectively. Exclusion is confined to the duration of the access procedure call.

b. Open Access Protocol. This protocol separates the securing (using join) and releasing (using leave) of an interface from the data transfer operations, and each of these functions is programmed as an individual access procedure. Mutual exclusion operates between the secure and release access procedure calls.

The closed access protocol is completely safe since the necessary exclusion constraint is imposed by a single call. However the open access protocol relies on a correct sequence of calls and the responsibility for the maintenance of data integrity passes from the access procedure designer to the root procedure designer.

Efficiency and simplicity considerations support the need for an open access protocol:

a. Efficiency. Where an activity is the sole user of an interface it is more efficient to use the open access protocol rather than repeated calls of join and leave for each data transfer.

b. Simplicity. The open access protocol allows a set of related data transfers to be made as a series of access procedure calls. The same effect could be achieved by means of a further synchronising queue to provide mutual exclusion over the related calls but this would be more complex and less efficient.

To make the open access protocol safe from the access procedure designer's point of view requires a new primitive which can be inserted at the start of a data transfer access procedure to ensure that the appropriate IDA interface is currently secured by the activity making the data transfer. The check primitive is introduced to meet this need and has the following specification:

```
PRIMITIVE PROCEDURE check (q : CONTROLQ)
```

This primitive has the effect of raising a fault if the queue is not currently joined. Of course the check on the queue ownership itself constitutes an overhead which would seem to amount to a continuing waste of resourses at run time; albeit this waste will be substantially less than that arising from a join primitive at the same position (plus the associated leave on exit). As with other potential fault conditions this run time overhead can be avoided by using off line program analysis. In this case the analysis must ensure that check always occurs between appropriate join, leave brackets.

9. HIGH INTEGRITY AND MEMORY MANAGEMENT

High integrity features in computer software are concerned with the prevention of access (unintentional or deliberate) to unauthorised data areas. Achievement of high integrity will generally involve direct manipulation of hardware base and limit registers together with strict control of the construction processes which are used to build the executable software.

Many machines which are appropriate for use in conjunction with the MASCOT software design and implementation methodology have memory management facilities. Generally speaking these facilities are provided to extend the basic address range of the machine rather than for tight access control purposes. Consequently the facilities are usually coarse in their effect and only allow memory to be managed in relitively large chunks. Nevertheless they do constitute a basis for the provision of high integrity features.

The MASCOT ACP diagram is an explicit expression of authorised data access since the lines joining IDAs to activities show permitted interactions. Not shown explicitly on an ACP diagram are the access procedures and private data areas (activity stacks) which are involved in any information transfer between the IDA and an activity. The kernel executive is an additional implicit element of the run time software. High integrity and memory management considerations indicate that the run time software should be broken down into six broad categories:

a. Primitive procedure code
b. Access procedure code
c. Root procedure code
d. Kernel data base
e. IDA data
f. Activity stacks

The relationship between these six categories is shown in figure 4 which also partitions the software into privileged and application areas. This partitioning expresses the MASCOT principle that communication between processes takes place within the application area and is not the special preserve of the operating system.

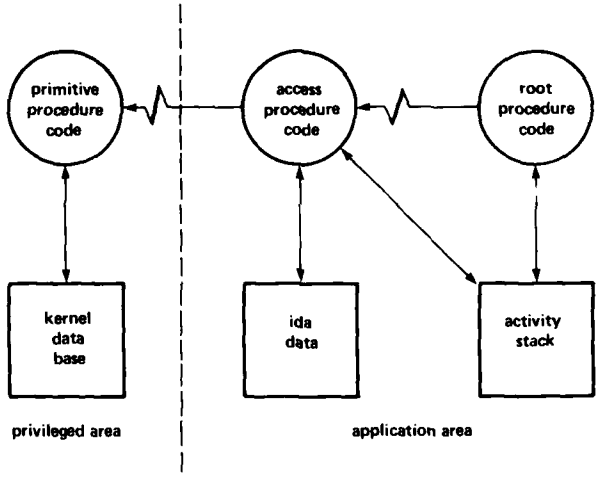


Figure 4 MASCOT Software Categories

At this point we should make a clear distinction between high integrity in particular and memory management in general. High integrity requires only base and limit facilities to constrain data access and code execution to defined ranges. Memory management requires address extension facilities to enlarge the overall direct address range of a machine by mapping virtual addresses onto a larger range of physical addresses. Thus from the high integrity point of view we do not need to concern ourselves with the various mechanisms which extend addresses but can concentrate on access and execution protection.

A further assumption is necessary. The base and limit information for the IDA data and activity stack areas will form part of the kernel data base and should be inaccessible to access procedure and root procedure code. This implies two different modes (privileged and application), the mode being automatically set on transition across the privileged/application boundary, together with a restriction that instructions which change hardware base and limit registers can only be executed in privileged mode. If the hardware architecture contains base and limit facilities but makes no distinction between privileged and application modes then a system which prevents unintentional unauthorised access can be created but deliberate unauthorised access will still be possible.

Conventionally the primary function of the MASCOT kernel has been to schedule activities. The introduction of high integrity features requires that the kernel is also concerned with the dynamic control of IDA access and the kernel data base must contain IDA control structures together with the relevant linkage information. Figure 5 shows the relationship between elements of an appropriate kernel data base structure. Each activity control structure in the kernel contains a reference to a vector (idas) indicating the IDAs to which the

activity has access. These are the only IDA-activity connections and the kernel has complete control over the data path interactions implicit in the ACP diagram.

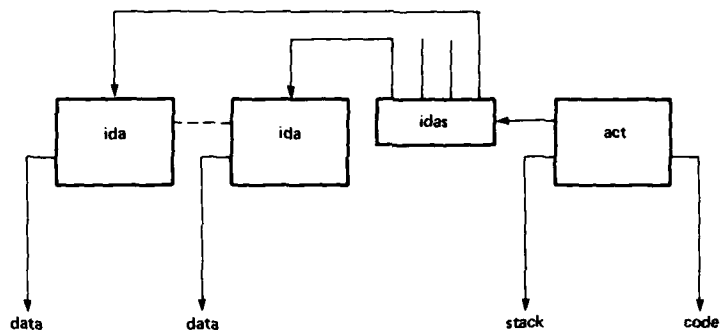


Figure 5 Kernel Data Base Structure

Figures 4 and 5 taken together imply that the virtual address space associated with the application area is divided into a minimum of three regions: code (root and access procedures), activity stack, and IDA data. (Note that if the access procedures are placed in a separate fourth region, or if they are placed in the same region as the IDA data, then each IDA control structure must locate the relevant access procedure area and entry to these procedures must be by special kernel call; the discussion here assumes that this is not the case and that access procedures are effectively directly linked into the root code). Primitive calls result in a transition into privileged mode which it is assumed allow manipulation of the base-limit register sets for the three regions. These three register sets form part of the context of an activity and must be set up by the kernel each time an activity is scheduled. In addition the registers associated with the IDA data must be set at the start of every access procedure call. This can be accomplished by an additional primitive:

```
PRIMITIVE PROCEDURE setida(ida : IDA)
```

This procedure takes as its parameter an IDA of any type and has the straightforward effect of loading the hardware base and limit registers for the subject IDA.

As an illustration of the use of the setida and check primitives consider an access procedure named read designed to fetch a word from an IDA of type WORDCHAN (see section 6):

```
ACCESS PROCEDURE read(ida : WORDCHAN, value : WORD)
BEGIN
    WITH ida DO
    check(oq)
    setida(ida)
    WHILE empty
    DO wait(oq) ENDWHILE
    value := word
    empty := TRUE
    stim(iq)
    ENDWITH
END
```

The check primitive ensures that the using activity has previously secured the IDA's output interface. This avoids a possible latent fault in that read might be called many times without invoking the wait primitive, and it prevents this access procedure from operating on the interface when it has in fact been secured by another activity. The setida primitive merely adjusts base and limit fields to allow access to the data fields of the IDA (note that the control queues are regarded as being in the kernel data base and are always visible to a primitive).

10. APPLICATION OF THE METHOD

MASCOT has been set in the framework of a general design method based on hierarchical functional decomposition in terms of information and processing elements. A clear distinction has been made between types and instances of such elements and this has been used as a basis for the formulation of a coordinated set of facilities for generating MASCOT software.It is useful to summarise the part played by these concepts in the overall approach to software design and implementation.

Figure 6 shows the route from problem environment through to target software. The first stage of the design is to identify the information elements which surround the outermost processing element. Since these are the points at which the system will interact with the outside world this generally involves designing software representations of information elements (often hardware components) which already exist. The IDA instances

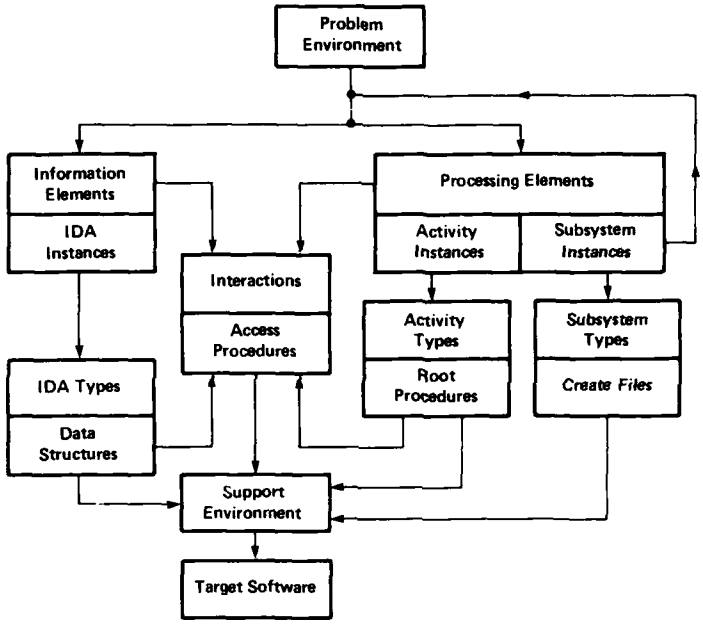


Figure 6 Design and Implementation Approach

which are created at this outer level can be regarded as software extensions to the information elements on the boundary of the system (see section 11 for an example). It is convenient to include all elements at this level within a single subsystem boundary which has no explicit connections to the outside world. A subsystem type can then be introduced to represent this configuration and can be used to generate the target software when the design is complete.

The method then proceeds with decomposition of the outermost processing element to yield information and processing elements at the next level down, and this is followed by further decomposition as necessary. After the first level of information environment definition the design process produces additional internal information elements which supplement the external elements identified at the previous level.

As each element instance is identified the corresponding type should generally be enrolled (as a data structure, root procedure or create file) or at least introduced into the support environment. This step of course will be unnecessary where use can be made of a type which already exists. The names of types have global significance for any given system support environment but the instance names are local to the enclosing subsystem.

It is important at this point to consider the nature of the decomposition process in more detail. The method formally expresses the decomposition of a subsystem type in terms of its internal component instances, and the external interface definition for a subsystem type is expressed in terms of IDA types. An ACP diagram should show the internal structure as instances, but it is a debatable point as to whether the outermost subsystem and external IDAs should be identified by type or by instance. Complete specification of an IDA instance must include the semantic and dynamic properties of the associated information. Thus the formal specification of subsystem connections in terms of external IDA types alone is inadequate and it is recommended that the ACP diagram shows both the type and instance names for such IDAs. Similarly the outermost subsystem should also be shown by type and instance. The role of the define ss facility during decomposition is worth noting. This allows more than one level of nested decomposition to take place without enrolling intermediate subsystem types.

Access procedures service the interactions between processing and information elements. They have an 'operates on' relationship with IDA types and a 'used by' relationship with activity types. Access procedures to meet all interaction requirements must be inserted in the support environment either at the time the associated IDA type is enrolled or separately by facilities specially provided for this purpose.

When all supporting software (explicit and implicit) for a given processing element type (activity or subsystem) has been enrolled a corresponding instance may be created either in a special test environment or in the final target system. Similarly an instance of each IDA type may be specially created to test for correct operation of the assosiated access procedures. It should be noted that the create files effectively provide a recursive facility for building the system bottom-up from a top-down design specification.

11. AN EXAMPLE

The overall decomposition approach is aimed at system design in the large and it is impossible in a paper of this length to give an example which covers all stages of the design process. However the first and most important steps of a fairly straightforward design task can be described.

Consider a hypothetical control problem where input data is derived from two sensors of identical type and, after processing by a controller, is used to drive an actuator. As far as the function to be performed is concerned there are three peripheral items for which some form of software representation is required. This can be achieved by enrolling two IDA types from which conceptual instances can be created:

```
enrol ida type(SENSOR(LOCATION))
enrol ida type(ACTUATOR(LOCATION))
```

The SENSOR and ACTUATOR types have been given a parameter on the assumption that each individual instance will have an associated unique address. Such instances can be regarded as software extensions to hardware information elements which physically exist. Thus the total software system boundary (shown in figure 7)passes through the sensor and actuator information elements, containing on its inside the components which are part of the software design.

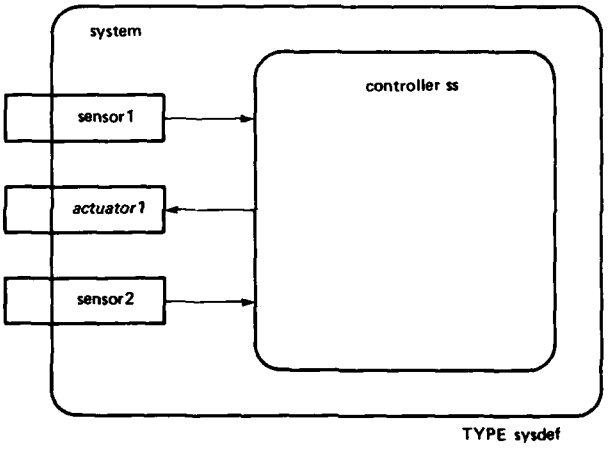


Figure 7 A Control Problem

At the topmost level the software design can be regarded as a single subsystem instance (named system) with no external IDAs. The type of this subsystem (which we will designate sysdef) expresses the internal structure of the system. As shown in figure 7 sysdef contains the subsystem instance named controller ss and in order to define sysdef in terms of a create file we must first introduce a subsystem type (which we will name controller) for controller ss:

```
introduce ss type(controller(SENSOR, SENSOR, ACTUATOR))
```

The create file defining sysdef can now be expressed as follows:

```
CREATE FILE sysdef
BEGIN
    create ida (sensor1, SENSOR(1000))
    create ida (sensor2, SENSOR(2000))
    create ida (actuator1, ACTUATOR(5000))
    create ss (controller ss, controller(sensor1, sensor2, actuator1))
END
```

The numbers in the above create ida statements have been arbitrarily chosen. They can be regarded as representing the address space associations between a real physical element and its corresponding software component. Subsystem type sysdef can now be enrolled:

```
enrol ss type (sysdef)
```

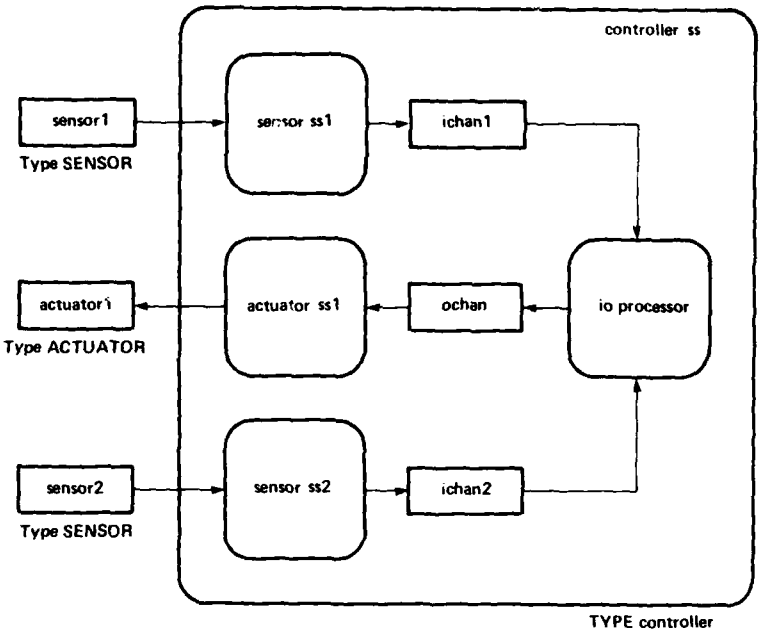


Figure 8 Controller Subsystem Decomposition

The next stage of design decomposition is shown in figure 8. Three handler subsystems interact with a set of internal data buffers, and an io processor subsystem controls the flow of data between input and output. All buffers are of type DATACHAN which, presuming that we have written the necessary source text describing the data structure, may be enrolled as follows:

```
enrol ida type (DATACHAN)
```

Subsystem types must be introduced for the handler subsystem (of type sense and actuate) and the processing subsystem (of type compute):

```
introduce ss type (sense (SENSOR, DATACHAN))
introduce ss type (actuate(ACTUATOR, DATACHAN))
introduce ss type (compute(DATACHAN, DATACHAN, DATACHAN))
```

The definition of the controller subsystem type introduced above can now be completed:

```
CREATE FILE controller (s1, s2 : SENSOR, a1 : ACTUATOR)
BEGIN
    create ida (ichan1, DATACHAN)
    create ida (ichan2, DATACHAN)
    create ida (ochan, DATACHAN)
    create ss (sensor ss1, sense (s1, ichan1))
    create ss (sensor ss2, sense (s2, ichan2))
    create ss (actuator ss1, actuate(a1, ochan))
    create ss (io processor, compute (ichan1, ichan2, ochan))
END
```

Further decomposition proceeds until all processing elements which are to be implemented as activities have been identified. Although the solution method is apparently rather long winded it is entirely logical and systematic and ensures that a sound,highly visible software structure is achieved.

It is interesting to consider the case where sensor, actuator and io processing functions are resident in separate processors in a distributed system. The approach adopted is to duplicate the internal IDAs (eg ichan1 is split into ichan1a and ichan1b) and to install a communications subsystem to drive data between associated IDA pairs. Note that the communications hardware is conceptually embedded within the communications subsystem and that the communications function is kept entirely separate from the rest of the design which is hardly altered at all provided that a satisfactory intercommunication performance can be achieved.

12 CONCLUSION

The design approach outlined in this paper is based on functional decomposition in terms of data driven, independent and potentially asynchronous processes which interact through well defined information flow paths. The approach is formal yet flexible enough to allow progressive tightening of functional and design specifications.

Although applicable to both hardware and software the approach is particularly suitable for the software elements of a design where it can be used to achieve visibility of the functional partitioning which of necessity must be applied to any large capacity computing or communications resource.

MASCOT as currently defined enforces a useful degree of functional decoupling and interface definition. The proposed changes, which are small but significant, bring MASCOT into line with a more general design decomposition approach. In addition a number of possibilities for improving the control over activity-IDA interactions are described.

The problem of real time performance analysis remains to be resolved. However the design framework calls for explicit expression of information propagation aspects. This is considered to be an essential prerequisite for any worthwhile consideration of real time performance characteristics and opens the door to the development of further formal analysis methods.

REFERENCES

BERGLAND G D, 1981. A Guided Tour of Program Design Methodologies. Computer Vol 14 No 10 pp 13-37

HOARE C A R, 1978. Communicating Sequential Processes. CACM Vol 21 No 8 pp 666-677

JACKSON K, 1977. Language Design for Modular Software Construction. IFIP 1977 Congress Proceedings pp 577-581

MASCOT SUPPLIERS ASSOCIATION, 1980. The Official Handbook of Mascot. RSRE Publication

SIMPSON H R and JACKSON K, 1979. Process Synchronisation in Mascot. The Computer Journal Vol 22 No 4 pp 332-345

SIMPSON H R, 1982. Act Parallel : Use Mascot. Computer Bulletin 11/31 pp 6-9

VERS UN VERITABLE ATELIER DE LOGICIEL AVIONIQUE

G. BRACON

Electronique Serge Dassault

92214 SAINT CLOUD (FRANCE)

RESUME

L'expérience acquise à l'ESD en matière de développement de logiciels avioniques (Mirage F1, Mirage 2000, *équipements) a conduit à* la définition d'un atelier logiciel.
L'atelier AIGLE a pour vocation la prise en compte des méthodologies et l'assistance à l'ensemble des activités de développement, de maintenance et de suivi de projet. Il comporte un ensemble d'outils fonctionnellement complémentaires, qui utilisent une base de données centrale, et peuvent donc partager des informations. L'intégration de services bureautiques et le confort du dialogue homme-machine permettront l'amélioration de la productivité.
Enfin, la caractéristique essentielle d'AIGLE est la saisie automatique d'informations de contrôle-qualité et de gestion de projet. Ceci permettra de valider le processus de production, élément indispensable à la certification des logiciels.

I - INTRODUCTION

L'Electronique Serge Dassault est spécialisée dans l'étude, le développement et la fabrication d'*équipements* électroniques de pointe, tant dans le domaine militaire que dans le domaine civil.

L'effectif de l'ESD est de plus de 3 000 personnes, dont 1 700 ingénieurs et cadres.

L'informatique aérospatiale (calculateurs, bus numériques, systèmes digitaux, logiciels de base et d'application) constitue une des activités principales de l'ESD : 20 à 25 % du chiffre d'affaires est réalisé dans ce domaine.

Une communication effectuée à la conférence AGARD d'OTTAWA, en mai 1979, présentait le contexte de développement des logiciels avioniques et la méthodologie MINERVE mise en oeuvre à l'ESD.

L'adoption d'un cycle de vie du logiciel conduit à constater rapidement le faible taux de couverture des outils utilisés au cours d'un projet. Ces outils couvrent traditionnellement surtout l'étape de codage, qui ne représente qu'environ 20 % de l'effort total.

L'ESD s'est, depuis 1977, dotées de moyens de tests puissants : les B.V.L (Baie de Validation du Logiciel, décrites dans une communication AGARD en septembre 1979), particulièrement adaptées aux tests de logiciels avioniques destinés aux calculateurs ESD. Cependant, il est apparu nécessaire de compléter la panoplie d'outils, afin de prendre en compte la spécification des besoins, la conception, et la gestion de projet. D'autre part, l'expérience acquise dans le domaine du test nous a paru mériter une généralisation qui permette de s'affranchir de la machine et du langage-cible. Enfin, les problèmes de l'assurance et du contrôle-qualité, et particulièrement de la mesure de la qualité, ont été pris en compte dans notre réflexion globale sur le problème du génie logiciel.

C'est pourquoi, depuis 1979, nous avons entrepris une série d'actions concertées dans ce domaine avec la SNIAS/DSBS. En effet, une telle réflexion nécessite la mise en commun d'expériences diverses afin que puissent être imaginées des solutions à vocation générale permettant des investissements à la hauteur des besoins.

Parmi ces actions, la définition d'un atelier de génie logiciel (AIGLE) constitue le cadre général des orientations adoptées, et est donc présentée ci-dessous.

II - LES PRINCIPALES ACTIONS GENIE LOGICIEL

Les travaux présentés ici ont été effectués ou sont en cours à l'ESD ; certains sont menés conjointement avec la SNIAS/DSBS, tous étant réalisés de façon concertée. La terminologie utilisée pour situer les travaux dans un cycle de développement s'appuie sur celle adoptée dans les plans-qualité établis par l'IEEE et l'AFCIQ, à savoir :

- spécification des besoins (ultérieurement abrégée en spécification) ;
- conception ;
- codage et tests unitaires ;
- tests d'intégration ;
- tests de validation.

2.1. Etudes en matière de méthodes

Une assistance technique a été fournie au SCTI-CELAR pour la standardisation d'un schéma de développement de systèmes militaires intégrant du logiciel, ainsi que l'établissement d'une terminologie et d'une liste de plans-type des documents à produire.

Une étude est en cours à la SEFT pour mettre en place des moyens de développement de projets, en particulier pour la définition et la recette des systèmes et la gestion de projet.

2.2. Etudes en matière de définition et de conception de logiciel

Un système de définition de logiciel assistée par ordinateur (D.L.A.O.) a été spécifié dans le cadre d'un contrat DRET. Il s'appuie sur un langage principalement conçu en fonction des applications avioniques. La conception de ce système est en cours. Une communication concernant cette étude sera faite au cours du présent symposium (#10).

Un système de définition et de conception de logiciel a été défini pour l'Agence Spatiale Européenne en s'appuyant sur les résultats de l'étude D.L.A.O. et de l'étude S.S.P. (Système Support de Programmation, développé par la SNIAS/DSBS), et en généralisant les domaines d'application.

2.3. Travaux dans le domaine des tests

Les Baies de Validation de Logiciel telles qu'elles sont utilisées actuellement permettent la mise en oeuvre, l'observation et la simulation de l'environnement des calculateurs opérationnels. Elles possèdent aussi une interface opérateur, spécifique du type d'application, qui permet à l'ingénieur conduisant le test d'un logiciel d'agir soit sur l'environnement, en simulant des actions du pilote ou des événements liés aux équipements, par exemple, soit sur le logiciel par l'intermédiaire de commandes de mise au point. Deux directions de développement ont été suivies pour le perfectionnement des BVL :

- L'automatisation des moyens de test actuels, sous contrat STTE, permettant l'enregistrement des échanges au cours d'une séance de validation entre machine de test et machine testée, afin de pouvoir effectuer des revalidations automatiques qui réduiront le coût et les délais des tests de non-régression.

- La généralisation du système B.V.L. , afin de permettre sa mise en oeuvre sur diverses machines-cibles et pour divers langages de programmation. Ceci a fait l'objet d'une étude, intitulée IDA, dans le cadre d'une convention Agence de l'Informatique, qui a abouti à la définition d'un langage de test du logiciel, d'une interface standard entre machine de test et machine cible, et d'une bibliothèque d'outils spécialisés. Une communication concernant cette étude sera faite au cours du présent symposium (#30)

2.4. Etude de méthodes pour la mesure de la qualité et de la fiabilité des logiciels

Ces études, effectuées sous contrat DTEn, sont actuelllement en cours et ont pour but :

- la détermination de facteurs, critères et métriques de qualité du logiciel ;
- la définition de procédures d'évaluation de la qualité ;
- l'établissement de classes de fiabilité du logiciel ;
- la définition de procédures de mesure de la fiabilité ;
- l'expérimentation en vue d'une évaluation des études.

III - L'ATELIER DE GENIE LOGICIEL : AIGLE

Les diverses réflexions menées dans les domaines de la méthodologie et des outils logiciels ont conduit l'ESD à s'associer à la SNIAS/DSBS et à la STERIA pour définir un atelier intégré de génie logiciel qui puisse constituer une infrastructure dans laquelle les outils développés devront s'intégrer, et qui permette d'articuler les outils entre eux en tenant compte du cadre méthodologique [RID 80]. Cette définition, effectuée avec l'aide d'une convention Agence de l'Informatique, a abouti à une spécification fonctionnelle, ainsi qu'à des choix globaux d'implémentation.

3.1. Contexte d'utilisation

L'atelier a pour domaines d'application privilégiés les logiciels intégrés à des systèmes, en particulier dans le domaine aérospatial, et, par ailleurs, les applications de gestion. Ce second domaine n'est pas développé ici, la dualité ayant été introduite pour favoriser l'amortissement économique de l'investissement (outre que nombre de fonctionnalités de base s'avèrent être communes).

Compte tenu des moyens mis en oeuvre, l'Atelier est plus particulièrement destiné aux organisations ayant à développer des projets de moyenne et grande envergure, ou nombre de petits projets.
L'Atelier s'adresse à toutes les personnes intervenant dans le cycle de développement du logiciel.

L'Atelier est indépendant :

- des machines-cibles ;
- des langages de programmation ;
- de l'organisation de l'équipe, qu'il permettra cependant d'exprimer et dont le fonctionnement sera facilité grâce aux outils de gestion de projet et d'assurance-qualité.

Selon les implémentations, diverses méthodes pourront être supportées en fonction des contextes d'utilisation ; ceci impliquera la mise en place de plusieurs "gammes" d'outils.

3.2. Activités prises en compte

L'Atelier fournit des aides automatisées pour toutes les activités (et tous les acteurs) concourant à la vie d'un logiciel) :

- Activités de développement :
  L'Atelier adopte le cycle de vie préconisé par l'IEEE et l'AFCIQ.

- Activités de maintenance :
  Des moyens seront fournis pour la saisie, la consultation et le rapprochement des rapports d'anomalie ou demandes de modification, des fiches descriptives de modifications, la gestion des versions et configurations.

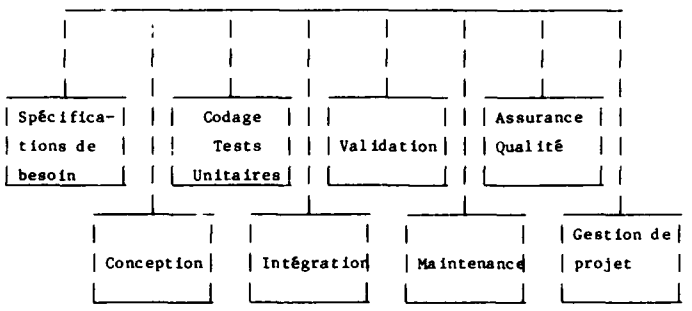
- Activités "horizontales" :
  Sous cette appellation sont regroupées les activités de gestion de projet, assurance-qualité et production de documents.

Le support d'une activité pourra être selon le cas :

- un ensemble spécifique d'outils : système d'aide à la spécification ou à la conception, moyens de production de programmes, moyens de test, outils de planification,...
- l'utilisation dans le contexte d'une activité d'un service à vocation générale : édition de texte, gestion de documents, gestion de configurations,...
- le contrôle par l'atelier de procédures, en particulier d'assurance-qualité, matérialisées par le respect de plans-type, de règles de précédence entre activités, le maintien de la cohérence des configurations, etc...

Les activités prises en compte déterminent le schéma conceptuel d'ensemble d'un projet développé sur AIGLE (cf. Fig.1). Ce schéma global se détaille selon les outils mis en oeuvre ou constituants plus précis.

Fig.1 : Schéma conceptuel d'ensemble



3.3. Principes de conception

3.3.1. Innover et non pas inventer

Compte tenu de l'étendue de la tâche de réalisation de l'Atelier, le principe fondamental retenu afin d'aboutir à un premier produit utilisable à l'horizon 1985, est de privilégier l'approche innovative, c'est-à-dire la mise en oeuvre de concepts établis. AIGLE se fixe donc pour but principal d'intégrer des outils existants en un même système, c'est-à-dire de les doter d'interfaces homogènes tant vis à vis de l'utilisateur qu'en termes d'échange d'informations entre eux.

3.3.2. L'approche système

Plutôt que de rechercher une portabilité généralisée, toujours délicate et coûteuse, voire inefficace, il nous a paru plus prometteur de privilégier un certain type de configuration matérielle. En conséquence, l'architecture physique visée est constituée d'un réseau local reliant des postes de travail autonomes et puissants, qui partagent des services centralisés, variables selon les installations. Le service central essentiel est le stockage des informations dans une base de données, dont la gestion assure la sécurité, le partage et la mise en relation des informations constituant les produits des projets.

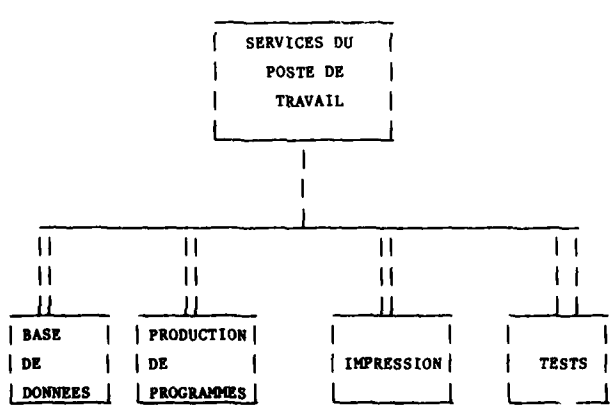
3.4. Architecture

3.4.1. Architecture fonctionnelle

L'Atelier est perçu par l'utilisateur comme étant structuré en deux niveaux de service : des services locaux assurés de façon individuelle par le poste de travail, des services généraux partagés avec les autres utilisateurs. (cf. Fig.2). Il n'y a pas de répartition a priori entre les deux catégories de services. Cependant, d'une manière générale, tous les services seront locaux exceptés :

- le service base de données : normalement utilisé au travers des outils, il pourra servir à l'interrogation directe par l'utilisateur avec les restrictions d'accès qui lui sont attachées ;
- le service production de programmes : ce type de service (compilateur, assembleur...) étant généralement disponible dans les environnements actuels, et représentant un investissement très lourd, l'atelier se contentera d'en assurer l'utilisation par une mise en oeuvre de la machine hôte ;
- le service impression de masse : dans le cadre du poste autonome de travail, une impression locale parait nécessaire en général ; cependant, un tel dispositif n'a pas pour vocation d'aller au-delà du "hard-copy" ; il est donc nécessaire de disposer d'un site central pour les impressions en quantité (listings, gros documents), bien que la nature du poste de travail ait pour effet de diminuer notablement les manipulations de papier.
- le service test ; il permet l'accès à la machine-cible, afin d'exécuter des tests "réels", c'est-à-dire ne perturbant pas le programme sous test ; AIGLE comprendra une machine de test, outil de mise en oeuvre et d'observation du calculateur-cible, telle que l'a définie l'étude IDA ;
- le service réseau : au cas où l'accès à un réseau généralisé, tel que TRANSPAC ou la communication avec une autre installation AIGLE, serait nécessaire, une passerelle permettra de disposer des services existants sur d'autres sites.

Fig. 2 - ARCHITECTURE FONCTIONNELLE D'AIGLE



3.4.2. Architecture physique

L'architecture matérielle d'AIGLE est une architecture répartie comprenant :

- d'un réseau local de type ETHERNET ;
- des postes de travail autonome et puissants, c'est-à-dire dotés d'un processeur puissant (68000, 8086...) d'une mémoire centrale de plusieurs centaines de K octets, de disques Winchester, d'un écran bit-map, d'un dispositif de manipulation de type souris. Le poste de travail offre dans ce cas un confort important d'utilisation et de par la localité de la majorité des services, garantit la disponibilité et la stabilité des temps de réponse. Son coût, du fait de la généralisation prévisible de ce type de matériel, doit s'avérer rapidement compétitif en raison de la décharge obtenue sur les machines "serveurs".
- des serveurs assurant les services centraux partagés. Ces services peuvent être assurés sur un serveur unique, ou confiés à des machines dédiées, selon les installations. Les services considérés sont essentiellement : le SGBD, les moyens de production de programme (compilation/édition), les moyens d'impression de masse et l'accès à la machine-cible. En ce qui concerne le SGBD, la solution du serveur spécialisé (machine de base de données du type Copernique) apparaît être à terme garante de la qualité du serveur.

3.5. Organisation logicielle

Une caractéristique importante d'AIGLE étant l'évolutivité à la fois vis à vis des outils mis en oeuvre et du matériel support, il est nécessaire d'organiser le logiciel en niveaux de services successifs.

3.5.1. Structure d'accueil

La base du système est constituée par des éléments préexistants : architecture matérielle, système UNIX/SOL [BEL78] [CAM82], SGBD.
Ces éléments doivent être encapsulés de façon à présenter une interface de service invariante aux couches supérieures : ce niveau est baptisé structure d'accueil, puisqu'il assure la pérennité des outils vis à vis de l'architecture matérielle.
La structure d'accueil assure trois classes de services :

- la communication dans l'atelier, c'est-à-dire la gestion de la localisation des services utilisables par les niveaux supérieurs, et des protocoles de communication entre constituants de base du système ;
- la gestion des informations, appelées "objets" dans AIGLE ; les activités prises en compte par l'atelier permettent de dégager un schéma conceptuel, qui identifie des classes d'objets (de spécification, de code, de gestion de projet...) et des classes de relations entre ces objets. A chaque ensemble d'outils constituant une version de l'atelier correspond ainsi une instantiation du schéma conceptuel, prenant en compte à des niveaux plus fins les types d'objets et de relations dépendants des outils. La structure d'accueil a pour rôle de gérer, de manière générique pour tous les objets, leur manipulation et les évolutions de versions ; cette gestion est assurée par la distinction, pour chaque objet, entre son descripteur (identifiant, type, version, état...), entité propre à la structure d'accueil, et son corps manipulable par les outils compatibles avec l'objet considéré
- la gestion de l'interface utilisateur, qui assure l'homogénéité des protocoles de dialogue entre l'utilisateur et le système, fournissant un standard d'interfaces aux outils interactifs.

3.5.2. Services généraux

Au delà de la structure d'accueil, des services généraux peuvent être mis à disposition de tous les outils spécifiques. Ils regroupent principalement :

. les moyens bureautiques : édition de texte, édition graphique, messagerie, gestion des documents ;
. la gestion des configurations : dans le contexte d'AIGLE, cette gestion couvre l'ensemble des produits issus du développement (des spécifications aux tests et aux informations de maintenance), ainsi que la gestion des sites d'implantation et des moyens de production et d'exploitation, indispensable aux contrôles de cohérence.

3.5.3. Gestion de projet et assurance-qualité

Les outils de gestion de projet et d'assurance-qualité sont invoqués automatiquement lors de l'utilisation d'outils de développement, et effectuent en temps réel les mises à jour et les mesures concernant le développement. Le suivi permanent du planning de projet et du plan-qualité peut ainsi être consulté par les outils d'interrogations spécifiques à ces activités, et les dépassements de prévisions ou non-respect de procédures signalés aux divers intéressés.

3.5.4. Développement et maintenance

Les outils de développement (et les outils complémentaires de maintenance : saisie des incidents, demandes de modifications et modifications effectives) constituent des ensembles couvrant le cycle de vie adopté, dont l'utilisation entraîne automatiquement des questions définies au moyen de l'outil de gestion de projet et des contrôles définis au moyen des outils d'assurance-qualité. C'est en particulier à partir des exigences du plan-qualité que devront être établis des liens entre produits d'activités diverses le long du cycle de vie.

3.6. Le dialogue utilisateur-système

Il repose en particulier sur l'intégration de deux concepts : le langage de commande SHELL d'UNIX, et les interfaces utilisateurs développés dans les projets SMALLTALK [SMA 81] et KAYAK [NAF 81].

Les caractéristiques principales en sont :

- le déroulement parallèle de plusieurs tâches pour un même usager et la possibilité d'enchaîner des commandes par le jeu des filtres ;
- la présence simultanée d'informations de provenances diverses, apparaissant dans des fenêtres distinctes, ce qui facilite la consultation de documents ;
- le fonctionnement à l'intérieur d'un mode, défini par l'activité choisie par l'utilisateur parmi celles offertes par AIGLE, qui autorise la manipulation des informations liées à cette activité, et ne permet que la consultation des autres informations ; par exemple, il est possible d'utiliser un éditeur syntaxique de langage de programmation dans un texte de spécification, mais le produit n'a vis à vis de l'activité de spécification qu'une valeur de texte informel (inversement, un texte de spécification formalisée inséré dans un programme-source n'a plus que la valeur de commentaire) ;

- la présence à l'écran de menus contextuels guidant l'utilisateur en permanence ; de plus, l'appel contextuel au guide auto-éducatif est toujours possible (cette fonctionnalité est indispensable, du fait du grand nombre de services susceptibles d'être fournis par AIGLE) ;
- l'interprétation contextuelle des commandes, allégeant et uniformisant la syntaxe ;
- l'interrogation par un "query", sous-ensemble des primitives base de données de la structure d'accueil, des informations accessibles à un utilisateur, en particulier afin d'en consulter les relations ; ce service, utilisable au travers d'outils de formattage, permettra l'édition de listes d'objets, de références croisées, d'extraits spécialisés (un minimum de procédures pré-établies étant évidemment fourni).

3.7. Les services généraux

Ils se répartissent en deux catégories : la bureautique et la gestion des configurations.

3.7.1. Moyens bureautiques

La bureautique d'AIGLE offre trois types de services :

- l'édition textuelle et graphique, interface standard entre l'utilisateur du poste de travail et les outils finalisés ;
- la messagerie, qui comporte :
  - le bloc-notes d'un utilisateur qui lui permet de gérer ses notes personnelles (en particulier de stocker les messages reçus) ;
  - la gestion des boîtes à lettres, qui permet aux différents utilisateurs de s'adresser des messages, et aux outils de contrôle (gestion de projet, assurance-qualité) de prévenir par message les divers intéressés quant aux anomalies rencontrées ;
  - la gestion des documents : l'utilisateur dialogue avec le système au moyen de textes et graphiques, que nous appelons "composants documentaires" ; ainsi peuvent être constitués divers documents, à partir de plans-type prédéfinis et de composants documentaires élaborés à l'aide des outils (élément de spécifications, de conception, code source de programme, jeu de test,...).

Le documentaliste assure ainsi la gestion des plans-type, la saisie d'un document selon un plan-type, et la mise en forme pour édition.

3.7.2. Gestion des configurations

Cette fonction, essentielle dans AIGLE, assure la cohérence des produits issus d'un projet, d'un bout à l'autre du cycle de vie. Elle s'appuie sur la gestion des objets de la structure d'accueil, qui contrôle la validité des relations établies entre objets, et les changements de version d'un même objet. En revanche, la gestion des configurations doit s'assurer de la cohérence d'évolution des versions entre des objets en relation. Par exemple, la modification de l'interface d'un module doit entraîner la signalisation à l'utilisateur des modules liés ; une fiche de modification portant sur les spécifications détermine la trace vers les éléments de conception, codage et tests concernés.

Tout projet est représenté, pour une version donnée, par une nomenclature des produits qui en sont issus. De plus, pour chaque type de produit, il sera possible d'y rattacher les références des moyens de production et d'exploitation associés, et les sites d'implantation. En particulier, il peut être utile de conserver la désignation de la version et/ou des options de compilation ou d'édition de liens utilisées pour produire un programme ; de même la version de moniteur d'exploitation auquel est destinée une version d'application est une information qu'il faut avoir.

Ainsi, la gestion des configurations est le régulateur du système d'informations, capable de déterminer si une application est complète ou non, et si l'ensemble de ses constituants sont homogènes et peuvent donc être intégrés pour exploitation. De ce fait, elle constitue un élément important du système d'assurance-qualité.

3.8. L'assurance-qualité

L'assurance-qualité est une fonction répartie dans le système AIGLE, mise en oeuvre à l'occasion de toutes les opérations de développement et de maintenance. Elle s'appuie pour sa réalisation sur une grande quantité d'outils (existants et à développer).

3.8.1. Le plan-qualité

La définition du plan-qualité d'un projet s'effectue à l'aide d'un outil spécialisé qui permet de saisir :

- les produits exigés pour chaque étape du cycle de vie ;
- la désignation pour chaque produit du standard à lui appliquer (plan-type, langage, outil de contrôle et paramètres correspondants, ...) ;
- les règles spécifiques du projet : validation d'un produit avant passage à l'étape suivante, droits d'accès des divers participants, cycle d'autorisation de modifications ;
- les mesures demandées sur le développement.

3.8.2. Le suivi de la qualité

Toute utilisation d'outil en développement ou maintenance entraîne un contrôle par rapport au plan-qualité :

- de respect du cycle de vie ;
- de conformité aux standards ;
- de respect des règles spécifiques de passage d'une activité à une autre, de modifications, de validation de produits ;
- de métriques (à certains états des produits)

3.8.3. Le suivi de l'assurance-qualité manuelle

Bien que l'atelier ait pour objectif d'automatiser l'assurance-qualité autant que possible, certaines opérations demeurent soumises à l'appréciation humaine. Afin d'intégrer leurs résultats, AIGLE permettra la saisie :

- des résultats de revues de projet et "walkthroughs", qui portent une appréciation sur un produit (en particulier, le valident et peuvent demander des modifications) ;
- des cycles auteur-lecteurs de produits, c'est-à-dire des interlocuteurs, produits concernés et remarques.

Les informations seront datées de la même manière que toute opération à effet rémanent réalisée par l'atelier.

3.8.4. La métrologie

AIGLE permettra l'insertion d'outils de mesures de toutes sortes [LAW 81].

En particulier, des mesures seront faites ;
- sur la durée des activités (temps calendaire, temps d'utilisation de l'atelier) ;
- sur les quantités produites (nombre de lignes, de pages, d'objets, ...) ;
- sur les caractéristiques des produits (complexité textuelle ou structurelle) ;
- sur les temps d'exécution des tests dynamiques ;
- sur les taux de couverture des tests ;
- sur la fréquence et l'incidence des modifications.

3.9. Méthodes et outils

3.9.1. La démarche [BOE]

L'atelier a pour but d'offrir un ensemble complet d'outils assistant aux tâches de développement, suivi, contrôle et gestion de projets logiciels. Nous avons dit plus haut qu'il était indispensable d'utiliser des outils établissant un cadre méthodologique afin d'assurer la formalisation des produits, la mise en relation à un niveau de détail suffisant des éléments de décomposition, et la possibilité d'utiliser des outils de métrologie.

L'optique d'une boîte à outils indépendants étant rejetée, deux voies restent possibles :

a) Définir un ensemble d'outils adaptés à une méthodologie particulière et complète (couvrant toutes les activités). Le couplage de ces outils et leur intégration seront alors aisés, et le système résultant d'une totale homogénéité. Cette solution comporte pourtant des inconvénients notables :
   - elle implique la conception de l'ensemble des outils, voire la définition complémentaire de la méthodologie, un ensemble homogène et complet d'outils n'étant pas disponible aujourd'hui ;
   - elle lie AIGLE aux paris techniques impliqués par cette définition totale.

   Cette voie a donc été rejetée.

b) S'attacher à rassembler dans une même structure matérielle et d'interfaces des outils existants ou dont la définition existe, outils qui sont retenus du fait de leur adaptation aux besoins identifiés, et dont les concepts sont complémentaires.

Cette approche permet :

. l'adoption d'outils existants, dont la spécification est donc acquise ;
. plus de souplesse à l'égard de l'adaptation de nouvelles méthodes ou de nouveaux outils.

C'est cette solution qui a été retenue ; des choix de première implémentation ont été effectués.

3.9.2. Les outils

- La phase de spécification est assistée par le système D.L.A.O., dont le modèle conceptuel et la syntaxe semi-formelle sont bien adaptés à l'expression de problèmes avioniques temps réel, et à la description des interfaces du logiciel avec le matériel environnant.

- La phase de conception est conduite selon la méthode des machines abstraites, supporté par le système SSP. Ce système débouche sur un service de programmation assistée vers les divers langages pris en compte : LTR, PASCAL, FORTRAN...

- Le code produit fait l'objet d'analyses statiques ; il sera éventuellement possible d'intégrer des analyseurs dynamiques sur le poste de travail, mais ceci n'est pas prévu dans un premier temps.

- Les tests d'intégration en temps réel sont effectués depuis le poste de travail sur la machine-cible, sous contrôle de la machine de test, avec le langage de test défini par l'étude IDA.

- La gestion de projet sera assurée par le produit SGPL (Système de Gestion de Projets Logiciels), caractérisé par son orientation vers les travaux logiciels, la prise en compte des études de Putnam, l'utilisation des historiques de projet et la saisie automatique des données à l'intérieur d'un atelier.

3.9.3. Intégration des outils

Les divers outils évoqués ci-dessus, ainsi que les fonctions générales et les utilitaires fournis en particulier grâce à UNIX, sont intégrés selon divers points de vue :

- par leur accès à travers le même interface homme-machine ;
- par l'utilisation d'une même base de données ;
- par la possibilité, déterminée dans le plan-qualité d'un projet, de mettre en rapport dans la base de données des éléments de produits issus d'activités distinctes (par exemple : un programme et la partie de conception qui le décrit, ou les parties de spécification qu'il implémente). Ceci assure la possibilité de parcourir les liens entre les informations et vers les modifications, et donc d'assurer la cohérence et la complétude des produits.

IV - CONCLUSION

Le volume et la complexité, et par voie de conséquence l'importance et le coût du logiciel ne cessant de croître [BOE 82], le risque technique encouru à chaque projet, en termes de qualité, coûts et délais nécessite de nouvelles solutions pour être maîtrisé. Ces solutions ne peuvent résider uniquement dans les progrès, par ailleurs souhaitables, des langages de programmation.

En effet, les progrès en matière de langages mettent au contraire toujours plus en évidence les aspects mal connus, et donc incontrôlés, des développements de logiciel :

- l'insuffisance des spécifications, trop floues, incomplètes, voire contradictoires, la difficulté d'appréhender de gros documents, ... ;

- l'incapacité dans laquelle on se trouve aujourd'hui de valider totalement de gros logiciels complexes en dehors de leur environnement opérationnel, ce qui est coûteux, voire dangereux. De façon plus générale, il est impossible à l'heure actuelle d'estimer la qualité d'un logiciel avant une utilisation prolongée qui permette de constater les coûts d'exploitation et de maintenance ;

- la difficulté qu'il y a d'estimer et de contrôler les coûts et délais, et d'évaluer les impacts des modifications.

Tous ces problèmes démontrent le besoin de disposer d'outils qui prennent en compte les divers points de vue sur un projet, standardisent les produits (documents, programmes), fournissent des éléments chiffrés objectivement sur le processus de développement et de maintenance, donc assurent la véritable visibilité du logiciel.

En fait, la conclusion qui s'impose maintenant est qu'il faut pouvoir valider le processus de développement et d'entretien du logiciel au moins autant que le produit, comme cela se fait dans d'autres techniques [HOW 82]. Ceci n'est évidemment possible qu'avec un degré important d'automatisation [SOR 79].

AIGLE nous paraît être une façon de s'orienter résolument vers une telle automatisation. Après une phase expérimentale, un tel atelier permettra de capitaliser une réelle connaissance sur des logiciels développés, tant en termes de qualité et de gestion de projet qu'en termes de "mémoire technique". En effet, il est envisageable d'élargir le système à l'interrogation sur des sujets déjà traités, ce qui permettra de fournir des éléments réutilisables par de nouveaux projets [WAS 82].

BIBLIOGRAPHIE

[BEL 78] BELL System Technical Journal - 1978

[BOE 79] BOEHM, Barry - 1979
Software Engineering - As it is - Software Engineering Conf.

[BEL 82] BOEHM, Barry - 1982
Les facteurs de coût du logiciel (Software cost factors) - TSI n° 1

[CAM 82] CAMPBELL et al - 1982
Le système d'exploitation SOL - Note INRIA

[HOW 82] HOWDEN, William - 1982
Life-cycle software validation - IEEE Transactions on S.E.

[LAW 81] LAWLER, R.W. - 1981
System Perspective on Software Quality - COMPSAC'81

[LED 81] LEDGARD et al - 1981
Directions in human factors for interactive systems -
Springer Verlag

[NAF 81] NAFFAH, N. - 1981
Le projet-pilote bureautique KAYAK
Bulletin de liaison de l'INRIA n° 69.

[RID 80] RIDDLE, William - 1980
PANEL ; Software development environments
IEEE Transactions on S.E.

[SMA 81] SMALLTALK - August 1981
A language for the 1980's - BYTE

[SOR 79] SORKOWITZ Alfred - 1979
Certification Testing : A procedure to impose the quality of software testing -
IEEE Transactions on S.E.

[WAS 82] WASSERMAN Anthony and GUTZ Steven - March 1982
The future of programming - CACM.

DISCUSSION FROM AVIONICS PANEL FALL 1982 MEETING ON
SOFTWARE FOR AVIONICS

Session 1 : SOFTWARE (S/W) TECHNOLOGY (TUTORIAL) - Chmn Dr. A. A. Callaway (UK)

Paper Nr. 1 - AVIONICS SOFTWARE: WHERE ARE WE?
Presented by - Dr. W. Ware
Speaker - D. M. Weiss
Comment - I notice that you pointed to the development of expert systems as a way of improving the information management handling problem in the cockpit. I wonder if you see anything in the software engineering technology that will help with the problem of developing software that is easy to change?

Response - There are things around that might contribute to that. However, I don't think one can develop a recipe that says we if we do the following things we will be on top of that particular issue. Dr. Sundberg's paper identifies an obviously important step. Standardizations are also an important step. There are a lot of bits and pieces which are also important steps. In the whole they will all contribute to that end goal that you have identified. But there are not any magic bullets on that issue or any other issue in the software business.

Paper Nr. 2 - AVIONICS SOFTWARE DESIGN
Presented by - Dr. D. E. Sundstrom

No Questions

Paper Nr. 3 - Cancelled

Paper Nr. 4 - S/W DEVELOPMENT: DESIGN AND REALITY
Presented by - Dr. H. Von Groote

No Questions

Paper Nr. 5 - MASCOT DEVELOPMENTS TO IMPROVE SOFTWARE STRUCTURE AND INTEGRITY
Presented by - Dr. H. R. Simpson
Speaker - Dr. D. J. Martin
Comment - MASCOT, I believe, was originally aimed at large, asynchronously operating systems. I am not sure of the usefulness in the highly synchronous, sequential processing of the flight controls task. Could you please comment?

Response - MASCOT was created some 10 years ago to deal with the problems of large distributed processing systems for ground applications. Modern avionic systems are approaching the complexity of these earlier ground based systems and the method can be used to advantage. The MASCOT software structure can also give benefits in essentially sequential systems where the intercommunication data areas can be used to compose a very strong form of partitioning between different elements of the processing task thus enhancing testability and flexibility (see Computer Bulletin, March 1982, for example). In smaller applications of MASCOT it will be necessary to tailor the kernel to avoid unnecessary overheads.

Paper Nr. 5 - MASCOT DEVELOPMENTS TO IMPROVE SOFTWARE STRUCTURE AND INTEGRITY
Presented by - Dr. H. R. Simpson
Speaker - J. Whalley
Comment - What is the official MOD(UK) view on the proposed extensions and what are the implications for MASCOT with the advent of ADA?

Response - The proposed extensions will shortly be put before the MASCOT technical management committee. If adopted by the Committee (after review and an amendment as necessary) they will be incorporated in what will probably be known as MASCOT III.
MASCOT is a language independent method with a formal means for expressing the design and supported by certain software construction and test tools. In principle the ADA language can be used with MASCOT although it must be said that the tasking features of ADA are in some conflict with the MASCOT approach to parallelism (as explained in the introduction). However this difference is not sufficient to rule out the possibility of an effective combination of the MASCOT design method and the ADA programming languages.

Paper Nr. 6 - VERS UN VERITABLE ATELIER DE LOGICIEL AVIONIQUE Presented by - G. Bracon
Speaker - Unknown
Comment - Est-ce que vous pourriez nous dire ou en est votre developpement actuellement?

Response - Comme je viens de l'ecrire, ce sont les résultats d'une étude qui a conduit à une spécification globale que nous considérons comme une action un peu federatrice sur tout ce que nous faisons dans le domaine du génie logiciel, mais nous n'avons, pour l'instant, pas depassse le stage de cette specification. Par contre nous espérons bien qu'en fonction des actions qui seront lancées en particulier par les pouvoirs publics français dans ce domaine, nous aurons l'opportunité d'entamer les developpements de ce type.

Paper Nr. 6 - VERS UN VERITABLE ATELIER DE LOGICIEL AVIONIQUE Presented by - G. Bracon
Speaker - N. P. Haigh
Comment - The "AIGLE" system contains a large data base. Would you care to comment on the security aspect with regard to unauthorized access?

Response - Oui, nous avons envisagé d'avoir des sécurités a ce niveau là, c'est à dire qu'il y aura des mots de passe, il y aura un certain nombre de procédures de reconnaissance de l'utilisateur. Chaque utilisateur devra être declaré. Je pense que, à cet egard, nous aurons une politique relativement classique sur ce qui se passe sur les systèmes de temps partagé, on n'a pas de spécifications tres précises de ce point là.

REQUIREMENTS DECOMPOSITION AND OTHER MYTHS

Dr. T. G. Swann
Marconi Avionics Ltd.,
Elstree Way,
Borehamwood,
Herts.
WD6 1RX

SUMMARY

A myth is a traditional fiction that reveals a greater truth. This paper is concerned with the procurement of large, innovative, systems - why do we never get what we want? The problems are often blamed on certain types of faults in the requirement specification and the design process.

Requirements specifications are discussed and it is seen that they are far more complex than we are asked to believe. It is argued that the description of design as Requirements Decomposition is more than a simplification: it is positively misleading. Similarly the virtues of good specifications, such as completeness and formality, are not just pre-requisites for success: they are unattainable ideals. It is concluded that many traditional maxims, though valuable, should not be taken too literally. They are, perhaps myths.

1. INTRODUCTION

A myth is a traditional fiction that reveals a greater truth.

Once, long ago, a powerful king wished to buy a toy. He described his requirement like this:-

> "I want a toy plastic car, to give to my son on his fourth birthday - one of those brightly coloured plastic ones. It must be safe as he might chew it. Keep it secret, don't let him know".

All the king's ministers were appalled at the careless informal style of this document and resolved to tidy it up before passing it to the executive office of the civil service. Each had a suggestion for improvement.

i) The word "toy" is vague and hard to define, surely the essence here is that it should be small and cheap.

ii) We do not need to say who the car is for, or that it is a birthday present. These are irrelevant to the specification.

iii) There is no need to specify "brightly-coloured", all such toys are brightly-coloured. This is putting in too much detail.

iv) The sentence would then read: "One of those plastic ones". This is a redundant, as plastic has already been specified.

v) There is no need to mention the possibility of the toy being chewed. There are regulations for toy safety that include this.

vi) If the toy is kept secret, then the son would not know of it. The injunction "Don't let him know" is therefore superfluous.

The revised specification read:-

> "A small, cheap, plastic car is required. It must be safe. Keep it secret.

A sufficient, formal statement expressing all the necessary aspects of the requirement with no irrelevant detail. Naturally it was several months before the king discovered the existence of a secret project to develop a small, cheap, plastic automobile that set new standards in passenger safety.

What went wrong? At first sight this seems a trivial story. But, if we examine each of the six amendments in turn, we see that none are outrageous. They are all perfectly valid and could be supported by convincing arguments. If, instead of three lines, the specification were 200 pages, the amendments might pass unquestioned.

I believe that things go wrong for more reasons than are generally acknowledged. And the most important reasons lie in an area scarcely touched by existing methodologies, namely the very nature of language and formal notation.

This paper does not attempt a deep study of the many philosophical questions that could be raised (Marcuse H., 1964; Weinberg G, 1971; Woods W. A. 1981). It merely uses some elementary arguments to shed light on a vexing, practical, problem.

2. PURPOSE OF SPECIFICATION DOCUMENTS

If we were building a small system, we might do all the design in a single session, write it down in a single document and build the whole system ourselves. But life is rarely like that. In practice we must write a variety of documents to suit the needs of a host of people. They will vary widely in content, style, vocabulary and so on; from wiring schedules, that refer to some specific technique or machine tool, to explanations written in simple language for the benefit of the customer's management.

Among the documents, we identify some that embody the design of the system. These form a hierarchy or pyramid which, read from the top downwards, describes the design in more and more detail. And we expect them to be written in that order, starting with an overview of the design and ending with many detailed descriptions of local features.

But this does not mean that the actual design work is performed topdown. On a major project the design path will be paved with feasibility studies and working papers. Many high level documents will be written with quite specific design details already in mind.

The documents in the pyramid can be regarded as both specifications of what has been designed and requirements to be met by further design. They therefore tend to be called wither Specifications (specs.) or Requirement Specifications. Their authors emphasise the "Specification" part, to show how much design work has been done. Their readers, the design teams for the next level down, emphasise "Requirement" to show how much is still to be done. And rightly so, for there is no guarantee that this"Requirement" can be met at all.

The specs., then, have several distinct purposes. They capture or describe the design for the benefit of the authors (designers). They convince the customer (the level above) that the designers have understood his requirement: or at least they describe what he is going to get. They communicate the design to other design teams, who may know nothing about it. These others must be able to distinguish what has been designed, what remains to be designed and the various constraints on design.

If a problem occurs and the design must be changed, it must be possible to see the origin of each part of the design and hence the impact of any change. In this context, the design can be seen as a hierarchy of decisions: this many modules, that sort of interface, and so on. The decisions may stem from the customers requirements, working papers or designers' hunches. Often they will be purely arbitrary. But whatever the origin, the specs. are more than just a description, they are a living record of these design decisions and their consequences.

As design proceeds, more and more people become involved, and not just in design. Test engineers need to know which microprocessor will be used, how much BITE there will be, and so on. Production need to know how many circuit boards there will be. Training officers need to know how the system will appear to the users. Above all Company Management need to keep track of the project timescale and budget. It has been said that, at the early stages of a project, the prime purpose of the documents is to allow Management to estimate the resources that will be needed.

3. INFORMATION

When all the purposes are spelt out, it seems remarkable that we even attempt to compress so much into a single series of documents. An even more remarkable notion is that the whole hierarchy is generated by the decomposition of a single document: the Specification.

The conventional story is that we take this document and expand it into a number of lower-level documents, each containing more detail. The process is repeated until we arrive at the lowest possible level, binary code, say, or circuit diagrams. But this cannot be the whole story. The Requirement Specification for a project may be only 100 pages while the lowest level of documents is several thousand. They contain more information.

Of course "number of pages" is not a very good measure of quantity of information, but it serves to illustrate the point. Even allowing for repetition, the same idea occuring in many documents, there is still a great disparity.

Where does all this information come from? It is not as if we can examine documents under a microscope to reveal greater detail. No, the information is added by the design process. Decomposition is a myth. Design means making decisions, adding information, creating order out of disorder.

So perhaps we should be concerned not with specifications as such, but with the information flow they imply. (Parnas D.L. 1971). One of the prime functions of a specification must surely be to direct this information gathering activity. If we can understand this activity we will learn a great deal about writing specifications.

3.1. Requirements and Design

The requirement for a new product or system rarely springs to life as a precise, well thought out idea. Initially there will be only some human "need" which is perceived, and expressed, in terms of existing familiar things. Deisgn, in its widest sense, is the job of taking this "need" and producing a product to satisfy it.

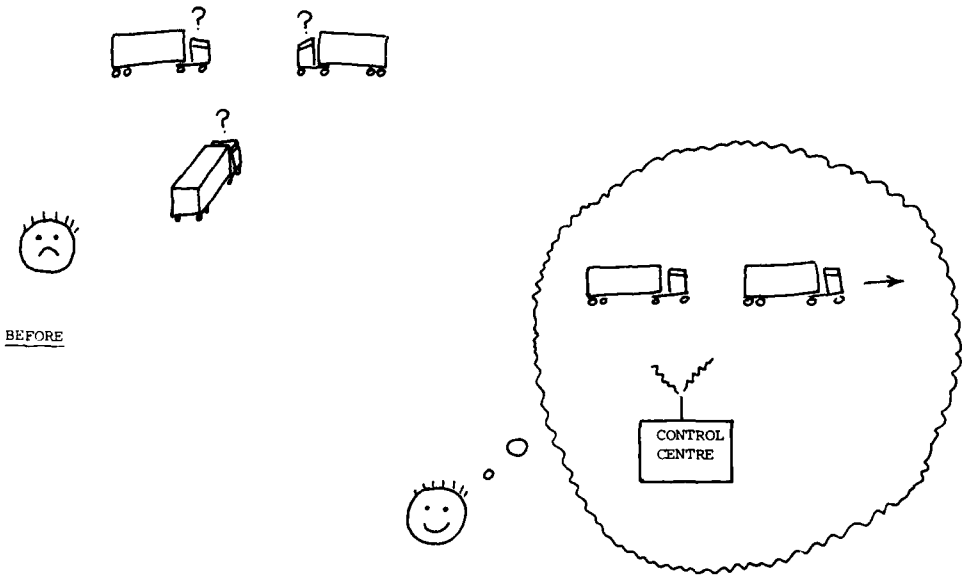
At first sight we might imagine that this was a well defined, if difficult job. The person with the need (the customer) writes down what is wanted in a Requirement Specification. Then someone else (the contractor, engineer) builds it.

But things are not that easy. Suppose a customer expresses the need for a control c to control the movements of a fleet of delivery trucks. He has said what is needed "a control centre , and what it is to do: "control the movements etc". But this brief description cannot be "decomposed" to produce a design, there is simply not enough information. No-one really knows what is wanted, someone has to find out. Or rather someone has to specify the need, and this means making decisions.

It is clear that even writing down the requirements is going to involve some decisions. There has already been an implicit decision to have a control "centre" rather than a network, say. We would expect further decisions such as: "it shall control all vehicles in the London area" and "it shall perform the following functions". But can we, as designers, ask for more detail of these functions? And more detail? Can we ask for them to be expressed as computer programs? I believe that we cannot ask for any more, simply because no-one knows any more about what is wanted. If they did they would not need us to design it.

The customer can see problems and he can describe these problems. But this may not even suggest a solution. Indeed there may be no solution, it all depends on how the problems are expressed. Even if we believe we can see a solution, we have no guarantee that it can be realised with available technology.

The design process is esssentially iterative. We must float a few ideas, evaluate them, modify them, and slowly turn them into concrete design. We are changing our, and the customer's, world view no less. Before, we saw a world with problems. After the Requirement Specification is written we see a world with the new product and fewer problems:-



BEFORE

AFTER

This is a leap of imagination. But unless we are also gifted with amazing insight we still cannot say, in detail, what is in the "Box": that is a matter of design decisions. We can only define it in terms of the new behaviour of the world around it and its interaction with this world. We have taken some decisions, as can be seen in the picture, but we have not yet designed the box. The functions to be performed only exist as high-level, human ideas. And these ideas cannot, somehow, be analysed to discover the low level detail. That would be magic

Historically, software design has always started with a System Analysis phase. This is because many business systems are intended, in some way, to replace existing, manual clerical systems which must first be analysed to find out what they are and how they work. The analysis is then a major component of the Requirement Specification. But it is only one component, and it must certainly not be confused with the subsequent, design decisions. We can only analyse things that already exist, i.e. the world "before" the new system. We cannot analyse things which are only ideas in the mind of the customer or designer.

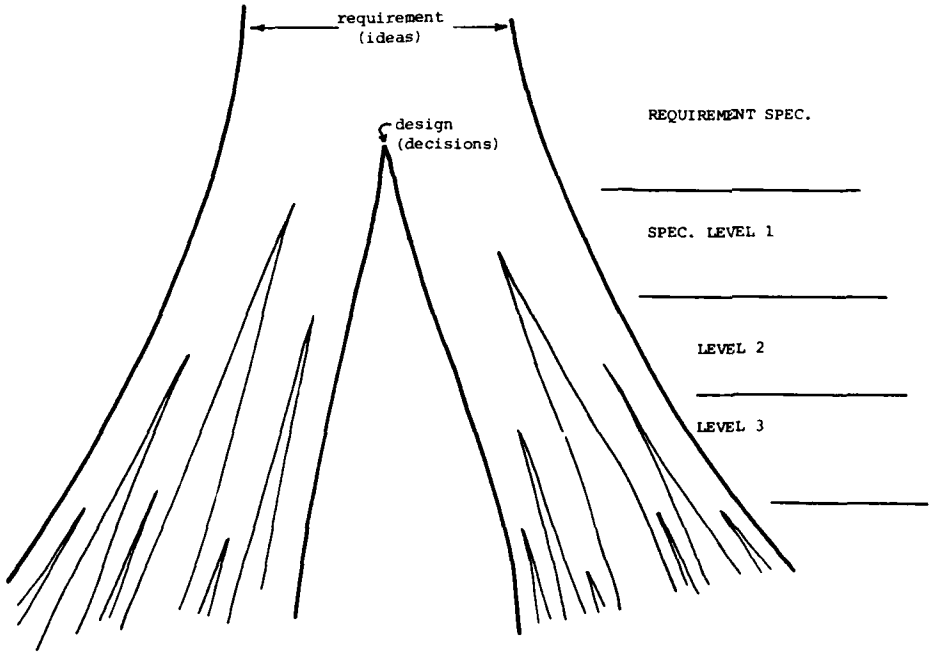
But we do need more information, and it can only come from the customer. If he wants us to do the design, he must describe his need: not as design detail inside the "box", but in terms of the purpose for which it was conceived i.e. going further out into the customer's world. Thus the system shall "save money", "reduce delivery times" and so on. And these purposes, in turn, can only be explained in terms of still higher level purposes: the company shall "make a profit", "increase local employment" and so on.

So the Requirement Specification is not a description of a product it is a statement of aims and intents with the proposed system at the lowest level and no limit to the highest.

Somehow we must construct a system to satisfy the expressed needs. Certainly this will entail careful reading of the Requirement Specification and consideration of its various parts, but if we have to choose just one word to describe our activity then we should choose not "analysis" but "synthesis". The designers job is not to analyse the requirements but to synthesise a design.

The designers must make decisions. And although these arise in response to the customers requirements, they are not contained in it. Someone has to synthesise the design, that is make the whole host of decisions that bridge the gap between the customer's purpose and the final nuts and bolts. They must invent the details of the control centre functions, they must judge what is suitable hardware to implement these functions and so on.

All these decisions are written down to form the hierarchy of specification documents. At the top are the major decisions about the gross nature of the system, these lead to smaller-scale decisions and on down to the final details. We can picture these decisions as a tree, where the "trunk" is the customer requirement and the branches (or rather the branching points) reflect the hierarchy of decisions over successively smaller areas of concern:-



Note... Like all software or family trees it is drawn upside down, with the branches underneath.

The decisions in the tree transform human ideas at the top into engineering detail at the bottom. And this shows the meaning of "requirement decomposition", in that the decisions form a structured list of parts. Structured decomposition is a methodology in the sense that is shows us how to organise and document our design decisions. Just as "genus" and "species" show us how to classify animals - but not how to design one.

3.2. Information Balance

If we group the decisions into appropriate documents the result will be as shown above. And we can see immediately why the higher level documents are thinner than the lower: they contain fewer decisions. We can also see why the Requirement Specification itself should be very thick: it contains not just decisions but an entire world view. And it is this that drives the decisions.

If we were engaged in pure, original, design then all the decisions (and hence the detail) would stem from the customer's stated requirement. The decision that "the control centre shall contain computers, radios and telephones" relates to the customer's description of the size of the vehicle fleet, etc. (For a smaller fleet, a clerk with a bicycle would have sufficed.) The requirement "to increase local employment" must also surely affect the design and will need elaboration in terms of budget and higher level aims. Even the most trivial decision must come from somewhere. The tree has a constant cross-sectional area. If the stated requirement does not cover a decision then we are, in effect, saying: "don't care".

If all goes well, we can trace each statement in the requirement to a set of decisions and each decision to a set of statements in the requirement. And if this is true there will be, in a crude sense, an information balance. A complex requirement, say, will have a correspondingly large number of decisions in the specification.

But this is never strictly true. No design is wholly original. A manufacturer will always try to sell a product that has already been designed. And rightly so, for by reducing cost and risk the existing product may well meet the customer's higher level needs. The decisions in the stated requirement are only the customer's attempt to meet the designers half way.

In reality the requirement does not attempt to encompass all the design detail. Instead it refers to other documents such as working papers and lists of stock items. For example, if we buy a car, the salesman assesses our requirement in terms of the size of our family, the thickness of our wallet, our social aspirations etc. These are translated into a simple formula such as "Model A, 4 doors, 1600cc". And this is the Requirement Specification received by the factory.

Clearly it does not pretend to contain the amount of detail necessary to build a car. That occupies several rooms full of paper. Instead it merely calls on the stock items provided by the factory. It could probably be encoded in just 6 digits.

The real Requirement Specification to which the cars are built is a far larger document (or pile of documents) It is this hidden document, with its attendant studies and reports, that is used to design the car. Once designed, minor variations can be called up by just the "Customer Specification".

So we can formulate a more general law of information balance: a Requirement Specification must contain as much information as is novel or non-standard in the final system. Any less is an invitation to guesswork.

4. THE PERFECT SPECIFICATION

We are often told that good specifications are complete, formal, non-redundant and unambiguous. We are also told that all requirements should be testable. Let us see, then, if we can test this requirement for good specifications.

4.1. Completeness

Consider the information tree above. At the top is the Requirement Specification: an explanation of what the customer wants in terms of his own aims and purposes. We have already noted that there is, in principle, no upper limit to this explanation.

However much we are told, questions or problems will always arise that can only be answered by referring back to the customer. And they can occur surprisingly late in the day. If we discover that our truck control-centre customer wishes to transport valuable cargo, it might only mean encryption on the telephone lines and passwords in the software: it might mean relocating the whole centre to a more secure building and redesigning the software. The excuse would be that it did not seem necessary for the designers to know the nature of the cargo. At this point a good system designer would wonder what else he had not been told.

So completeness at the highest levels, though desirable, is impossible. We do not have the patience or the paper to write the history of the universe.

But what about the design specifications? Surely these can be complete if we describe the control-centre, the telephones, the computers, the programs, all down to the very last detail? In practice we do not even try. We specify a certain type of computer, but leave the internal details to the manufacturer. We specify a certain quality of paper, but again leave out the details. The specifications are packed with details, but only describe the novel, non-standard features. The rest calls for stock items in stock situations. The bulk of the system is specified just like the saloon car: "Computer Model A, 1 Mbyte, 4 discs".

Of course the real system contains more detail than even the manufacturers of the stock items can define. The complexity of any real object is infinite. A microcomputer and a piece of impure silicon, say, are both infinitely complex - but the microcomputer has the impurities in useful places. In either case the exact shape, size, position etc. all require an infinite sequence to define them. If we only specify a finite amount of information we are, again, specifying "don't care" for the infinite remainder.

The list of things that could be specified is endless. How much we choose to say is determined not by "completeness" but by the law of diminishing returns. Beyond a certain point it is not worthwhile specifying any more detail. Where this point lies depends on many factors, not least the time and money at our disposal and the risk of boring the recipient.

So even the detailed specifications are not literally complete. And as we saw above, all these documents have many more purposes than merely describing design. The information tree is an oversimplification. Somewhere in the hierarchy between customer and designers will be the documents that we really need to build the syster including at least one that can form the basis of a contract. These much certainly be complete, but in the sense that they describe all the things we need to know, sufficiently for us to proceed with design.

This is a different, less formal, meaning of the word complete. Literal completeness is a myth. The true measure of a specification is not completeness but whether or not it is cost-effective.

4.2. Formal Languages and Meaning

We have seen above that, to make documents manageable, we do not try to put all the detail into the specification. Instead we leave it in supporting documents such as other manufacturer's codes of practice. Instead of calling for a car of a certain "style", "power" and "size", all open to interpretation, we say "Model A, 4 doors, 1600cc". This is a formal specification language, and it overcomes the traditional language barrier by means of formal syntax and semantics. Each term is defined in the supporting documents which are open to the customer and manufacturer alike.

It is even possible nowadays to perform computer analysis of such languages. A car manufacturer can analyse the customer specification to check for consistency (and completeness) and may even pass the result directly to an automated production line. Yet, for some reason, this is not hailed as a major breakthrough in the automation of system specification and design. Why not?

The answer is simple, but instructive. When we buy a car we know that other cars, very similar to the one we "specify", have already been designed, built, tested and (we hope) sold to satisfied customers. We are only specifying a minor change to the assembly of stock items. The formal terms are merely pointers to other documents where the real specification details are held. It is these documents that give the formal terms their meaning. The syntax of the language is controlled by the design engineers who restrict it to defining cars which they have already designed and tested.

If we want a system with novel, untested features we cannot procure it in this way. We cannot just specify a "control-centre" as these words are not backed up by supporting documents. Only the customer knows what they mean, the system has not yet been designed.

As design proceeds we can certainly make many formal statements, calling up stock items such as computers and telephones. But the novel features, that make this system different from all others, cannot be described in this way. Whatever terms we employ, the words can only be defined, or understood, by reference to the Requirement Specification. And this in turn, is only understood by looking further out into the customer's world. The meaning of our words rests ultimately in natural language and human ideas.

Suppose, then, we are asked for "Car Model A, 4 Doors, converted for disabled driver". The first two parameters are formal pointers to stock items, no problem, only the third is informal and must be referred to the customer. But we cannot simply cut the spec. in two parts, feeding "Car Model A, 4 Doors" to the production line and designing the rest ourselves. The presence of this one novel feature throws the whole design into doubt. We have to find out exactly what "Model A" means to see if it can be converted. We can no longer merely convert the formal parameters into a string of digits for computer processing, we have to study their definitions. And these too rest ultimately in natural language and the infinite complexity of real objects.

Certainly the term "4 Doors" can be understood, but only in the sense that it gives us a mental picture of the car. The words contain the germs of ideas, "semantic hooks" which grasp at a variety of informal images in the human mind. And if the words are chosen carefully they can be of great value. They allow us to think in high level ideas by providing "handles" to manipulate the real, underlying definitions. But we must not confuse the informal image with the definition of the real object. The final car, after conversion, may have three doors or five - but it will still be labelled "4 Door" on the production line. By allowing these "semantic hooks" on a formal word, we are deliberately being informal in the interests of cost-effective communication.

The use of these "hooks" is already a familiar technique in computer programming, where identifiers are given "meaningful" names. In this case, the names are equivalent to comments in the program and we can demonstrate this by editing all the names to random text strings before compilation. The program is rendered meaningless to a human reader but there is no effect on run-time behaviour. This demonstrates again the differences between the formal notation of the programming language and the intended real effect of the computer system as implied by the variable names.

Of course once the design is complete, then all the words can be given formal definitions merely by pointing to the finished system. But by that time the problem has been solved and the greatest need for communication and rigour is gone. It is no breakthrough to formally specify a system that has already been designed.

It might be argued that there are other means of conveying meaning, mathematics for example, that are strictly formal. But on reflection we see that mathematics only describes formal relationships between abstract objects. For example V1 - V2 describes an abstract operation on two algebraic quantitities. If we wish to compare the velocities of two real objects must first define the objects, define their velocities, decide on Newton's or relativistic laws, and so on. The translation from a real-world problem to a mathematical model requires a leap of imagination and a lot of text to describe it. The translation back, from a computed result to its significance in the real world is even harder. We do not live in a world of certainty and billiard-ball dynamics, we live in a world of uncertainty, approximations, and systems that we do not understand. We can, and should, use the notations of mathematics to help us describe relationships. But we must remember that the formality is at best an analogy and at worst a cosmetic.

4.3. Redundancy and Ambiguity

We have concluded that the meaning of any specification rests ultimately on natural language words. But what do these words mean? Words by themselves have no meaning. Meaning is introduced by putting

together groups of words to elicit appropriate responses in human brains. To reach a wider audience, or achieve a more precise meaning, we need more words without limit. Many quite simple ideas need whole books to explain them, merely because they conflict with our stock set of mental images.

This puts redundancy and ambiguity in a new light. In the literal (trivial) sense, redundancy means having things which are not necessary, ambiguity means having more than one possible meaning. We can reduce both by being more concise and formal. But if we do this crudely, by throwing out the explanations, then we throw out the meaning too, the baby goes out with the bathwater. In the end we have just mathematical equations with no ambiguity and no meaning either.

We do want to reduce ambiguity but not at the expense of the overall meaning. If we want to be understood we must explain ourselves, that is say the same thing in different ways. We do not just write as much as is necessary for one explanation, we write as much as is necessary to be understood. The spec. that explains something three ways is like a triplexed "redundant" computer system. If is proof against a whole class of errors. (Patterson D. A. 1981.)

Once again, we must write the amount that is most cost-effective. The unambiguous spec. is a myth.

5. STYLE

The perfect specification is complete, formal, non-redundant and unambiguous. But we have seen above that these are not themselves formal and precise terms. We cannot interpret them in their literal, mathematical, sense. It is more constructive to think of them as indicators of good literary style:-

- "Complete" means that it includes all relevant information, ideas, and explanation.
- "Formal" means a formal prose style with no colloquial phrases or jargon.
- "Non-redundant" means it does not bore the reader with repetition.
- "Unambiguous" means that everything is adequately explained.

This does not imply that we should be deliberately imprecise. On the contrary. Even the simplest idea can be very hard to explain. We need all the help we can get if we are to communicate anything at all. And this means careful definition of terms, formalised diagrams and notations, painstaking attention to detail, and so on. This is all good style (Henninger K.L, 1980).

Indeed, wherever possible we should be strictly formal, in the mathematical sense. If we can capture the essence of some mechanism in a formal analogy, i.e. a mathematical model, then we have gone a long way towards describing it - provided we flesh out this backbone with sufficient words of explanation.

What we must avoid is the compulsive brevity that gives the illusion of perfection, merely by throwing away all the bits that do not fit in with this idealistic aim.

We may then avoid the "defensive author syndrome":- A beautiful document in elegant, precise English is presented for design review. What happens? The whole day is spent explaining the meaning of the document, word by word. At the end of the day everyone understands it and agrees that it is a precise description after all. No one suggests it is deficient and the author departs without changing a word.

This is an elementary law of human behaviour. If people believe that what they are writing is formal, unambiguous and complete, they do not surround it with explanation. Once we are freed from this belief, we can see why it is that we are always misunderstood. We can then take practical steps to remedy the matter.

The most important of these is to tell people everthing we can about their task, and this involves more than is at first obvious. Designers need to know who the customer is, why he wants the product, and so on and so forth. In particular, where there are working papers etc., these should be treated as part of the Requirement Specification. If designers do not have the information from these papers, they will make it up themselves by guesswork.

We must remember too that documents contain far more information than they show at face value. Good designers are also good at "reading between the lines" to assess the thinking behind each statement. If explanations are inadequate or missing then they will invent these too by guesswork. Even good designers are only human.

And since we are communicating with human beings, we must not forget the practical observation that, in an appropriate context, one informal phrase may give more enlightenment than a page of careful prose (or pictures). Often, understanding only comes when we are given a verbal explanation, face to face, with full vocal inflection and waving of arms. How we can assimilate these into a specification system is a problem yet to be solved.

6. CONCLUSION

We can at last explain the moral tale told at the start of this paper. What the king wrote, then, was not a formal requirement specification, it was a cost-effective document designed to direct the procurement of an item in the real world. And in this extreme case cost-effectiveness dictated an informal style throughout.

Unfortunately, misled by the myths of their time, his ministers made three gross mistakes. First they tried to make it into a "functional requirement specification" by throwing away the "irrelevant", non-functional explanations. Second they tried to make it formal by throwing away the informal "semantic hooks". Third they tried to make it concise by throwing away the redundancy and omitting any further explanation. Taken literally the myths are recipes for disaster.

REFERENCES

Henninger K. L., 1980. "Specifying Software Requirements for Complex Systems New Techniques and their Application", IEE Trans. Soft. Eng.

Marcuse H, 1964. "One Dimensional Man", Routledge and Kegan Paul.

Parnas D. L. 1971. "Information Distribution Aspects of Design Methodology", Proc. Int. Fed. Inform. Processing Conf.

Patterson D. A., 1981. "An Experiment in High Level Language Microprogramming and Verification". Comm. ACM.

Weinberg G. M. 1971. "The Psychology of Computer Programming". Van Nostrand.

Woods W. A. 1981. "Procedural Semantics as a Theory of Meaning", Bolt Beranek and Newman Inc.

PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES

BY

C. P. PRICE
D. Y. FORSYTH

British Aerospace Public Limited Company
Aircraft Group
Warton Division
Warton Aerodrome
PRESTON
PR4 1AX
United Kingdom

SUMMARY

It is likely that the coming decade will witness a wider use of requirements analysis techniques in the development of avionic systems. They may be employed in the production of software requirements in particular or the development of higher level system requirements.

In general such approaches may be said to consist of a methodology to be used in the production process, software tools to assist in analysis and the existence of a specific target software design interface such as language and architecture.

The predicted quality and productivity improvements will only be attained if the selection of tools and techniques is tempered by practical considerations. This paper discusses the main issues any organisation contemplating the use of requirements analysis techniques will have to consider. They include the scope of application, system or software, the special needs of users, attributes of the methodology, the level of automation and the means by which they can be introduced to a project. The latter includes training, development of appropriate examples, project route map and support.

As an illustration the approach developed and used by the Warton Division of British Aerospace, Semi Automated Functional Requirements Analysis (SAFRA) is briefly described. In SAFRA, Controlled Requirements Expression (CORE) is the method of production embracing data collection, system analysis and notation. Storage and validation of the description is achieved using the Problem Statement Language and Problem Statement Analyser (PSL/PSA) of the Unviversity of Michigan's ISDOS project including a system description language, database management system and a suite of appropriate reports.

Software design utilises the Modular Approach to Software Construction Operation and Test (MASCOT) methodology with its rationale for module inter-communication, control, scheduling etc. Coding is arrived at through formal procedures which link the CORE diagrams with the U.K. standard CORAL 66 via PSL.

Two case histories of applying SAFRA are presented, one for a small task and preliminary results on its use in a larger project.

1. SAFRA OVERVIEW

SAFRA suggests a specific approach to requirements and software design and encompasses a number of methods and tools to aid the engineer in this task. A phased life cycle approach to system development is advocated as shown in Fig. (1), with techniques and tools applicable to each phase.

The method used by the engineer to develop and express his requirement is Controlled Requirements Expression (CORE), a new technique developed jointly by BAe Warton Division, and Systems Designers Limited. CORE is a method for the assembly and analysis of information relevant to a requirement with an easily understood diagrammatic notation.

Validation and storage of system and software requirements and design documentation is achieved by using the University of Michigan's Problem Statement Language and Problem Statement Analyser (PSL/PSA). The continued use of the CORE notation is made during the software design phase with storage using PSL/PSA but aimed at the use of a rationalised executive and High Order Language. The former is the Modular Approach to Software Construction Operation and Test (MASCOT) (1) and the latter is the U.K. MOD Standard CORAL 66. A further assumption is the use of a commercially available MASCOT based software development system.

To support the use of SAFRA in a project environment, quidelines are available suggesting basic Configuration Control procedures for the management of the data base and associated documentation.

2. CONTROLLED REQUIREMENTS EXPRESSION (CORE)

2.1 General

CORE is a method of analysing and expressing system or software requirement in a controlled and precise manner. It enables a subject requirement to be expressed as either a number of lower level requirements or as a component part of some higher level, (Fig. 2). A lower level requirement derived using CORE may in turn be subjected to the method to produce a hierarchy of further

2.1 General continued

levels. The lowest is that at which the full method need no longer be applied, and basic software design may proceed. Further decomposition through detailed software design continues to make use of the notation.

2.2 Diagrammatic Notation

CORE diagrams utilise boxes to represent processes and arrows to represent data. The diagrams are time ordered from left to right and thus the box order specifies the sequence in which the processes must occur.

Symbol free boxes shown in parallel represent indeterminate order and overlapping boxes indicate a number of identical processes occurring concurrently. All data entering a CORE diagram is referenced to a source and all output data to a destination.

Data arrows may also be used to describe repetition, selection and condition. One emerging from the top of a process box indicates that this process is functionally equivalent to that referenced by the arrow. Those appearing at the bottom of a process box indicate the mechanism that performs the process. Iteration is shown by an asterisk in the top right-hand corner of a process box and mutual exclusion by a small circle in the top left-hand side. Fig. 3 summarises the main features of the notation.

2.3 The Method

The method comprises eleven logical steps which must be applied in total for each level of requirement. There are three stages for each level of decomposition summarised as:

- Gather Information
- Propose Relationships
- Prove Relationships

Information is gathered with respect to a number of sub-divisions of the problem, referred to as Viewpoints, in terms of input and output data and gross functions. This information is refined by a data decomposition step which specifies in more detail the data already tabulated.

Relationships are proposed between inputs and outputs from each viewpoint in turn as well as for data flowing across the viewpoint, and these are termed 'Single Threads'. The relationships are assessed in two ways:

- The inter-relationship between viewpoints are examined and where specific links exist new diagrams in the form of 'Combined Threads' are constructed.
- Threads represent particular paths through system operation and do not depict parallelism or the operational time ordering of processes. This is achieved by the construction of a 'Combined Operational' diagram. Both of these will lead to iteration through the previous steps prompting a more detailed examination of the single threads for correct combination and in order to establish operational relationships.

2.4 Node Notes

This is not a step in the method as such but simply a means by which a small amount of english text may accompany any CORE diagram. Node notes provide simple descriptions of data and process names and may include design related information such as constraints, assumptions or decisions.

3. SMALL PROJECT CASE HISTORY

3.1 Introduction

By early 1980 the initial CORE development phase was complete and it had been used within the development group on anumber of tasks to exercise and evaluate the method. The tasks had been undertaken in a sympathetic environment and it was not known how the method would prosper when subjected to the constraints of a real project. We were particularly interested in how difficult it would be to transfer the method to a project team and to this end a small task sponsored by the Royal Aircraft Establishment was undertaken which required engineers, unfamiliar with software or CORE, to generate the software requirements for a Fuel Management System (FMS).

3.2 CORE Transfer

In order to transfer the CORE method to this and other projects a one week CORE course had been developed with an appropriate blend of

- formal lecture
- tutorial exercise
- workshop practice

The formal lectures provided a complete overview of the method and detailed definitions of each step.

The tutorial exercises were designed to bridge the gap between the formal definition and practical application.

The workshop practice gives the student 'hands on' experience and requires that he generate the requirements for a simple Vehicle Performance Monitoring System through two levels of decomposition.

3.2 CORE Transfer continued

The initial course was given to several engineering staff, two of whom were scheduled for the FMS task, by three members of the CORE development team.

3.3 The FMS Task

The FMS had two principle objectives:

- To develop the software requirements for the system
- To explore, at a technical level, the feasibility of using CORE outside the development environment

A team of three engineers was formed to undertake the task consisting of two fuel system experts and a CORE specialist. The latters' role was not only as consultant on the method but also as a requirement producer. It is worth noting that an engineer, trained in the use of CORE, is able to produce detailed requirements for systems outside his own specialisation by assuming the role of an analyst and using people or documentation as a source of specialist information.

3.3.1 Start-Up

Prior to embarking upon any requirement exercise it is necessary to establish the exact nature of the problem and the strategy by which it will be solved. With CORE this involves establishing a customer requirement and preparing a route map through that requirement and its sub-parts to the point where software design may begin.

The FMS customer requirement consisted of a simple hardware description detailing fuel tanks, fuel pumps, valves etc., along with a statement of the major system functions such as refuel, defuel and leak detection. In common with most customer requirements this document was not generated in a structured manner and addressed many levels of detail.

A route map through the problem was prepared (Fig. 4) which indicated that three levels of decomposition would be necessary to provide a functional software description. Level I would restructure the customer requirement, remove any ambiguities and ascertain whether the requirement constituted a viable system. Level II would confirm the validity of level I and provide a stepping stone to the detailed software requirements generated at level III.

3.3.2 Level I Application

Level I was undertaken to define how the customer's view of the fuel system operated and communicated within its immediate environment. This was achieved by taking the requirement, establishing its direct interfaces and then defining precisely how all these interfaces and the requirement operated together.

The project team undertaking the task encountered a number of problems and these are discussed under the following headings:

- Iteration
- Functional thinking
- Information limiting

- CORE is an iterative method and will not progress unless descriptions are valid whereas conventional approaches allow progress to be made regardless of validity. Conventional experience leads engineers to assume that once a requirement is expressed it is correct and requires little or no modification. When a problem was encountered in CORE (i.e. the method enforced iteration), the initial reaction was to assume that the method was inadequate. This syndrome can only be overcome by providing specialist support through the first level of decomposition and it is interesting to note that the it usually disappears at subsequent levels.

- Functional thinking proved to be a difficult concept to adopt, particularly by hardware engineers, who were inclined to provide hardware solutions before the problem had been defined. This tendancy is understandable as hardware system architectures and computing capacity are often defined before the functional or software requirements are generated.

- CORE demands that the information contained within one level of decomposition be limited to an amount that can easily be controlled by the persons producing that level. This is of particular importance in the early stages of a project as too much information will cause an unmanageable information explosion in the later stages. The most obvious way of limiting information is to consider only that information which is of immediate use. Engineers have a natural affinity for detail and will, if unrestrained, enter a problem in a maze of detail most of which is superfluous to the immediate task.

3.3.3 Levels II and III, Application

Level I had restructured the customer requirement and proved it to be viable by establishing its interfaces with the outside world. Level II took this restructured requirement and decomposed it into its subparts to provide a detailed description of the FMS in isolation. This description proved that the interfaces defined at level I could be maintained by the system and provided the basis for the level III decomposition process. Level III transcends basic software design and did not require the full CORE method to achieve a hierarchic decomposition of the level II subparts.

3.3.3 Levels II and III, Application continued

Apart from some assistance being required in the early stages of level II both levels required little support demonstrating that a complete transfer had taken place, successfully, within a small project environment.

3.4 Management of FMS Task

In the interests of solving transfer problems the management of the FMS task was undertaken at local rather than project level and limited to the establishment and maintenance of milestones, allocation of manpower and review/approval.

3.4.1 Milestones

Establishing the milestones at the onset of the task highlighted some hitherto unexposed problems. The CORE method iterates and no step in the method can be said to be complete until all steps have been undertaken. This makes it meaningless to allocate completion milestones to individual CORE products within a level of decomposition and implies that the smallest work unit to which milestones can be applied is the level itself. Unfortunately, this is not easily defined as it can vary in depth and scope, the exact details of which are dependent upon system complexity and available manpower.

Previous experience, albeit in the development environment, had indicated that a level of decomposition took approximately three months to complete and so this figure was tentatively allocated to each level. In the event the total task took ten months, four months for level I and three months each for levels II and III.

This and previous development tasks indicate that the three months period is a constant for any one level of decomposition. This has been substantiated in subsequent applications and it is interesting to note that whilst system complexity and manpower allocations effect the depth and scope of any one level they do not appear to affect the time period.

As stated previously CORE iterates and therefore milestones which mark the completion of a step cannot be set, this does not, however, prohibit the setting of milestones which indicate when a task is to begin. Each step of the method can be allocated a start date within a period and although none of these steps will be completed until the period has elapsed it provides management with a very good indication of progress.

3.4.2 Allocation of Manpower

Initially three engineers were allocated but part way through level I it became apparent that the task was unnecessarily overmanned and one engineer was removed. The new manning level proved to be consistent with that required of a conventional approach. Subsequent applications have confirmed that the number of persons required to undertake a CORE exercise is, in general, the same as that required for a conventional approach.

3.4.3 Review/Approval

Because the task was under local management control the review and approval procedures were somewhat limited. They did, however, demonstrate that persons unfamiliar with CORE could, with guidance, quickly come to terms with the notation and provide valuable comment on the CORE products.

3.5 Configuration Control

The FMS task was not subject to the rigorous configuration control procedures and standards normally imposed at project level and it was left to local management to maintain basic standards. Whilst this approach ensured that the FMS technical objectives were not jeopordised by unwieldy standards procedures it did little for the configuration control aspects of the task. At this time experience with PSL/PSA was limited although a procedure for encoding CORE diagrams into PSL had been evolved and the usefulness of some of the many PSA reports had been identified. It was clear that this area required further work and an informal one man PSL/PSA configuration control function was established to operate in parallel with the FMS task.

PSL/PSA was introduced into the FMS task once the initial level I data/process relationships had been established. All CORE diagrams produced up to this point were manually encoded into PSL and then entered into the Divisional IBM 3032 to form the initial FMS data base. Consistency checks were undertaken using PSA and the results fed back to the FMS engineers. Unfortunately, this process took a considerable time and the FMS engineers, eager to progress, took the opportunity to perform their own manual consistency checks and continue with the method. The iterative nature of CORE resulted in significant changes being made to the initial data/process relationships and consequently the FMS data base bore little resemblence to the level I descriptions at the time of issue. It was obvious that both the point of entry was incorrect and that the process itself was too slow. Whilst nothing could be done at this time as regards speed it was clear that the point of entry could be adjusted. The frequency of change occurring during the production of level I made intermediate application of PSL/PSA useless. The requirement remained for a data base providing a useful and accurate master record of the design and it was considered that this could only be usefully achieved if the PSL/PSA tasks were deferred until a complete level had been produced. This philosophy was applied successfully to the remainder of level I and subsequent levels allowing consistency checks to be undertaken in retrospect for each level of decomposition as well as providing a complete record of the FMS design.

3.6 Concensus

The exercise demonstrated that SAFRA could be installed in a small project environment successfully and that the procedures evolved for its transfer were practicable.

The applicaton of SAFRA although sucessful, had revealed two areas which required further development:

PSL/PSA procedures
CORE documentation

- The FMS exercise showed that the existing PSL/PSA procedures were best deferred to the end of a level. Whilst this approach is satisfactory for small tasks it was considered that large complex tasks, involving significant amounts of data, would require consistency checks to be undertaken in the early stages of a level in as near real time as possible.

- Approximately 40% of the total effort on the FMS exercise was expended on the manual production and maintenance of CORE diagrams and notes. It was clear that significant benefit would result if this task were reduced.

Both of these areas were procedurally correct but unsatisfactory due to the manual nature of their implementation and in early '81 a development programme was undertaken to provide an automated aid which would:

- Generate and edit CORE diagrams
- Generate notes and reports
- Translate CORE diagrams into PSL
- Execute consistency checks.

4. SAFRA DEVELOPMENT STATUS

Experience on the FMS and other small tasks undertaken using SAFRA highlighted areas within the SAFRA project that required further development. Before discussing the large project case history a brief account of these developments and their status in relationship to the point of entry will now be given.

The three major areas of development were:

- Software Design
- PSL Interface Package
- CORE Work Station Requirement

4.1 Software Design

A set of procedures have been developed which enable CORE diagrams, at the detailed software design level, to be translated into the programming language CORAL 66. The procedures also capitalise on the use of MASCOT, which advocates a structured approach to software design. Many of the concepts of system partitioning, particularly relating to MASCOT Activities and their associated data relationships, were found to be closely allied to those used in CORE thus providing some measure of correspondence between CORE and MASCOT diagrams. These procedures have been developed and evaluated over the last two years and are expected to be transferrable into a project in the near future.

4.2 PSL Interface Package

To gain maximum benefit from using PSL/PSA, particularly on a large project, the mechanism by which information is entered on to the data base for storage and analysis must involve a high degree of automation. This was evident from experience on the FMS task where CORE diagrams had to be manually encoded into PSL input files. As a consequence, a PSL interface package was developed. This allows the engineer to transfer the information contained within CORE diagrams into a local file, using a terminal, under the direction of software generated cues. A useful degree of syntax checking is provided as well as automatic production of all the PSL language elements, but the package is not directly linked with the data base for integrity reasons. This package became available in early 1982.

4.3 CORE Workstation Requirement

Experience on the FMS task highlighted four areas where automation was desirable if not essential, namely:

- The production and editing of CORE diagrams
- Consistency checking
- Automatic translation of CORE diagrams into PSL input file
- Report generation

A detailed requirement, expressed in CORE, was developed for a workstation with the objective of automating as many of the CORE tasks as practicable. The requirement is being implemented in three phases:

- CORE diagram editor
- Automatic translation of CORE diagrams into PSL/PSA
- Automatic production of CORAL 66 from CORE diagrams

5. LARGE PROJECT CASE HISTORY

5.1 Introduction

SAFRA has more recently been introduced to the production of mission Avionic and Utility system and software requirements for a new military aircraft.

The basic design philosophy adopted encouraged the development of a fully integrated Avionic and Utilities system using distributed processing. The Utilities system comprises Power plant control, Secondary power, Environmental control, Hydraulic and Fuel management which communicate via a dedicated Utilities serial data highway. The Avionic system encompasses a sensor suite providing sufficient information to allow basic navigation, detection of hostiles, and a comprehensive weapon delivery capability. In order to reduce development timescales and costs and provide some degree of commonality with current production aircraft existing sensors had been considered, where appropriate. The Avionic and Utility system software is required to execute the following basic functions:

- Elaboration of raw sensor data
- System and Cockpit executive moding and control
- System testing.

This project is still at a very early stage of development but some observations can be made about the introduction of SAFRA to a task of this size.

5.2 The Problem

The Aircraft Operational requirement, expressed in english details the operational and performance capability required by the customer for the aircraft he wishes to procure. The response to this document the proposed hardware system architecture (see para. 5.1) is a document similarly expressed in english. The information contained within these documents crosses many levels of detail, has little structure and can be found to be both ambiguous and inconsistent.

Real world constraints dictate that this type of documentation exists at the start of the system and software definition phase and the problems facing the requirement specification team as a result can be summarized as follows:

- The documentation does not attempt to express, or allow a natural entry point to the development of, the software requirements
- Conformance between the two documents cannot be demonstrated.
- Neither document can be shown to represent a complete or consistent system requirement/description.

SAFRA was introduced specifically to overcome these problems and we will now consider the approach adopted.

5.3 Approach to Problem Entry

Any method of decomposition relies on the highest level of description, from which all subsequent levels are derived, being consistent and complete. If the detailed system description is based on an expansion of the operational requirement and hardware system architecture, as described, latent requirement errors would remain undetected and multiply at subsequent levels. In order to overcome such difficulties CORE was introduced to the project and the first level of decomposition was set the following specific objectives:

- Add structure to (and thus help to validate) the proposed hardware system architecture
- Identify the major software elements of the system thus providing a platform for the development of the detailed software requirements.

A route map was prepared (Fig. 5) and the level I viewpoint selection shown was aimed at satisfying these objectives. Viewpoints VOI and VO4 isolate those areas of the system that were either selected hardware or aspects of the system outside the influence of the requirement specification team. VO2 and VO3 includes all system operational test and moding software as yet undefined, these being the subject viewpoints to be further decomposed. The route map also shows that on completion of level 2 all software requirements are combined to produce an overall software operational diagram thus providing confidence that all software specified for the aircraft is compatible and consistent. It will be noted that the route map shows an input to the overall software operational daigram from the aeromechanical systems in VO4 thus allowing integration of software requirements produced in CORE by engineers working on the Utilities system.

This overall strategy also allows the proposed hardware system architecture to be described in a consistent and unambiguous manner and provides a document which can be realistically assessed against the customers operational/performance requirements.

5.4 Project Staff Training

Project staff can be placed in one of three catagories:

- Requirement producers
- Requirement interpreters at a technical level
- Requirement interpreters at a management/customer level.

5.4 Project Staff Training continued

At the technical level requirements expressed diagrammatically in CORE and supported by node notes, have been accepted as the means of recording and communicating information. However, circumstances dictate that requirements expresssion at the project management and customer levels should follow the more conventional approach and be expressed in english.

Ambiguity can be resolved by reference to the CORE documentation, and these english descriptions do not compromise the quality of the authoritative CORE version stored on the PSL/PSA data base. This situation was not totally unexpected as it would be unrealistic to assume that CORE diagrams would be acceptable as an unsupported statement of requirement at all technical and managerial levels within the project. Electrical circuit diagrams for example are a medium for express design but they are only acceptable at a specific technical working level.

In order to satisfy the needs of producers and interpreters of requirements expressed in CORE two courses were prepared:

- A modified one week CORE course
- A one day CORE appreciation course

The one week course was a refinement of the one first developed for small projects with the same basic outline and objectives. Attendees of this course were prospective requirement producers or those requiring a detailed knowledge of the concepts of CORE. The latter included engineering representatives from equipment suppliers who were required to supply a product in response to requirements expressed in CORE.

A one day CORE appreciation course was also introduced aimed specifically at project management. This course had three objectives:

- Provide an appreciation of the method and an understanding of the CORE notation
- Suggest areas that must be considered in planning a strategy for the management of CORE products.
- Ensure that the project staff fully understand the role SAFRA was expected to fulfill.

5.4.1 Personnel Requirements

A system requirement specification team was formed to develop level 1, consisting of six engineers, two of whom were CORE specialists. The remaining four were systems engineers experienced on current production aircraft but unfamiliar with the method. Here it must be stressed that CORE replaces none of the engineering skills necessary to develop good systems and hence the quality of personnel required in the systems design team is consistent with that needed for a conventional approach.

5.5 Level 1 Application

Level I was required to express the proposed hardware system architecture in a structured manner and gathering this information was achieved by applying the first four steps of the method. This demanded that all project information sources be identified such as air vehicle specifications, existing hardware documentation and hardware/software specialists. Inconsistencies and omissions in the proposed hardware system description, highlighted during the production of the level, were resolved by referring these queries to the customer and system specialists. The completed level I represented a validated, unambiguous and consistent system description which could be provisionally approved at the technical management level. The hardware system description was subsequently updated to reflect level I and issued in draft for review across all project levels.

The validated level I provided the project with a stepping stone to demonstrating conformance between the customer operational requirements and the proposed hardware system architecture expressed in CORE, essential to the validity of software requirements generated at subsequent levels. Assessing level I as a response to the operational requirement was not an easy task. In common with many documents of its type the information it contained crossed many levels of detail ranging from broad statements such as a required mission success rate to a specific hardware equipment selection. The approach adopted was to assess each combined thread and operational diagram, in turn, against the operational requirement, queries were referred back to the customer and, where appropriate, diagrams changed and performance criteria added to the node notes accompanying each diagram. Level I, now reflecting a validated hardware/software system description conforming to the customer operational requirement, was exposed to formal technical and standards reviews and subsequently PSL encoded for storage and analysis on the PSL/PSA data base.

5.6 Project Milestones

Level I took 6 months to complete which was predictably longer than the earlier assertion that a level of decomposition takes approximately 3 months irrespective of task complexity. Involvement by the requirement specification team in peripheral tasks such as the production of sub-system english descriptions, project review meetings and technical/standards reviews accounted for the additional 3 months required to issue an approved level I.

5.7 Technical Review

The review of a level of decomposition was approached in 3 stages:

- CORE Standards review
- Internal review
- Independent (customer) System audit

CORE standards review requires that each CORE diagram is assessed for the approved use of notation, corect annotations and that each step in the method has been correctly applied. Each diagram carries the standards reviewers signature who is a member of the project standards management group and independent of the CORE producers.

The internal review serves to ensure that the level under consideration is an accurate interpretation of how management and specialists within the project envisage the system. To this end the level was partitioned into areas of specialisation such as Navigation, Weapon Aiming and Power Generation and issued, in draft, for comment. The reviewer was provided with the operational diagram and those combined thread diagrams impacting on his specialist area. In addition the diagram originators were available to support the review activity if they were required. All the diagrams relating to a particular subsystem were identified and reviewed by the specialists collectively and comments minuted and issued to the requirement specification team. This process iterates until an agreed representation is reached and approved.

5.8 Data Base

The PSL/PSA data base is used for the storage and analysis of approved CORE products and standards to control all aspects of the handling were introduced accordingly. The complete and technically approved level 1 was encoded in PSL, using the PSL interface package (ref. para. 5.2), comprising Tabular entries, Data decompositions, Combined Threads and associated node notes. The task of encoding the data set into PSL was undertaken by non-technical personnel and took approximately four weeks to complete. The main problem encountered is highlighted by examination of the table below which presents a brief analysis of the data base size and error count.

DATA BASE SIZE	
NAME TYPE	TOTAL
PROCESS NAMES	177
DATA NAMES	547
TOTAL NAMES	724
ERROR SOURCE	
LEVEL I INCONSISTENCIES/ OMISSIONS	15
CORE TO PSL	70
TOTAL ERRORS	85

- 10,000 PSL STATEMENTS
- 14,500 LINES

Genuine data set errors such as naming inconsistencies and omissions accounted for only 17% of the total errors detected by PSA, the remaining errors being directly attributable to the manual transfer from diagrammatic information to PSL. These errors were either misinterpretations of CORE diagrams or simply misread names. Notational idiosyncrasies from one diagram producer to another can cause confusion when read by a third party, and are difficult to control by the introduction of standards done.

The workstation will enforce such standards automatically but the errors represent only about 0.6% when expressed on a percentage of the total number of lines of PSL.

6. CONCLUSIONS

The main conclusion that can be reached from the experience of applying SAFRA, particularly to the larger tasks, it is that unless adequate preparation is made prior to starting a requirement specification task the gains to be made by adopting methods such as CORE will be lost. In summary these considerations should include:

- Staff training
- Project needs for requirements documentation
- Project standards
- Route map from operational to software requirements
- Interfacing computer based tools

7. ACKNOWLEDGEMENTS

The authors wish to acknowledge the support of the Flight Systems Department of the Royal Aircraft Establishment in the development of SAFRA and British Aerospace for the opportunity to report this work.

8. REFERENCES

1) MASCOT a Structured Software Design Mehodology for Real Time Systems.
The official handbook of MASCOT Published by MASCCT Suppliers Association Dec. 1980.

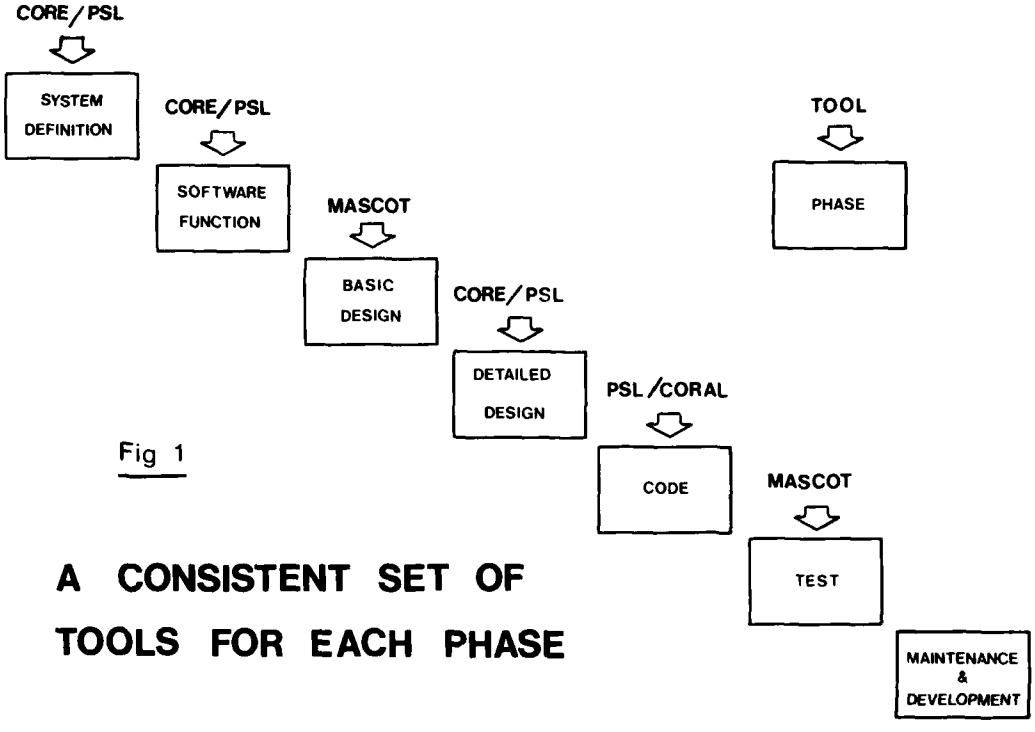


Fig 1

A CONSISTENT SET OF TOOLS FOR EACH PHASE

CONTROLLED REQUIREMENTS EXPRESSION

A SUBJECT REQUIREMENT CAN BE EXPRESSED

- AS A COMPONENT PART OF A HIGHER LEVEL REQUIREMENT

OR

- AS A NUMBER OF LOWER LEVEL REQUIREMENTS

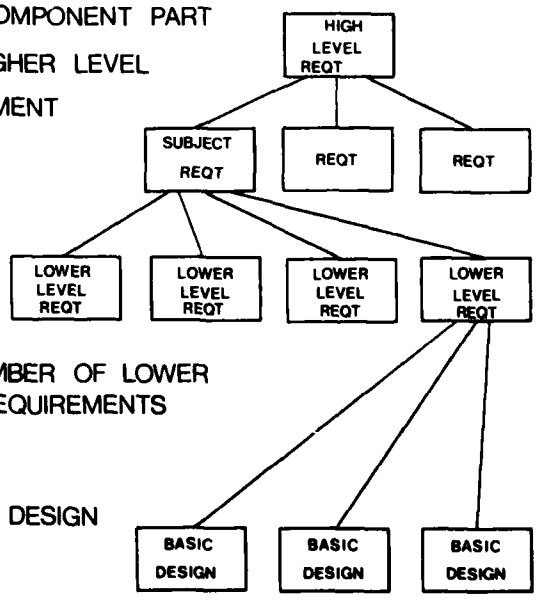


Fig 2

- TO PRODUCE A HIERARCHY OF REQUIREMENT DOWN TO BASIC DESIGN

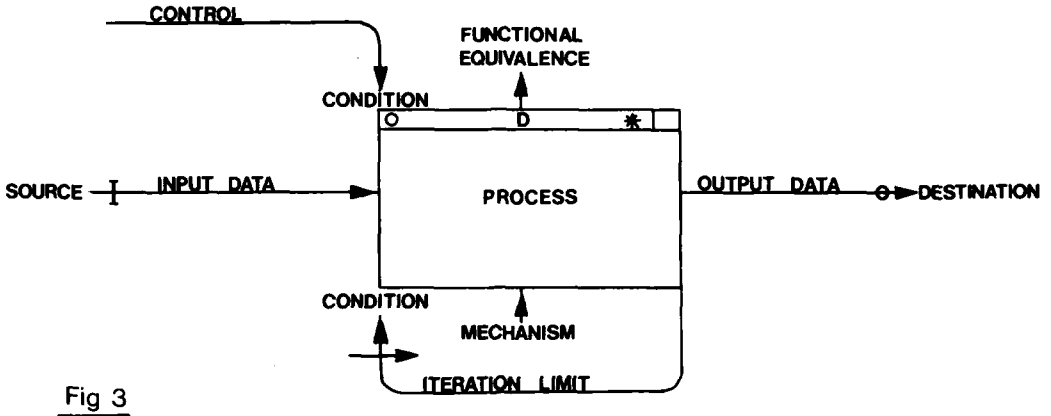


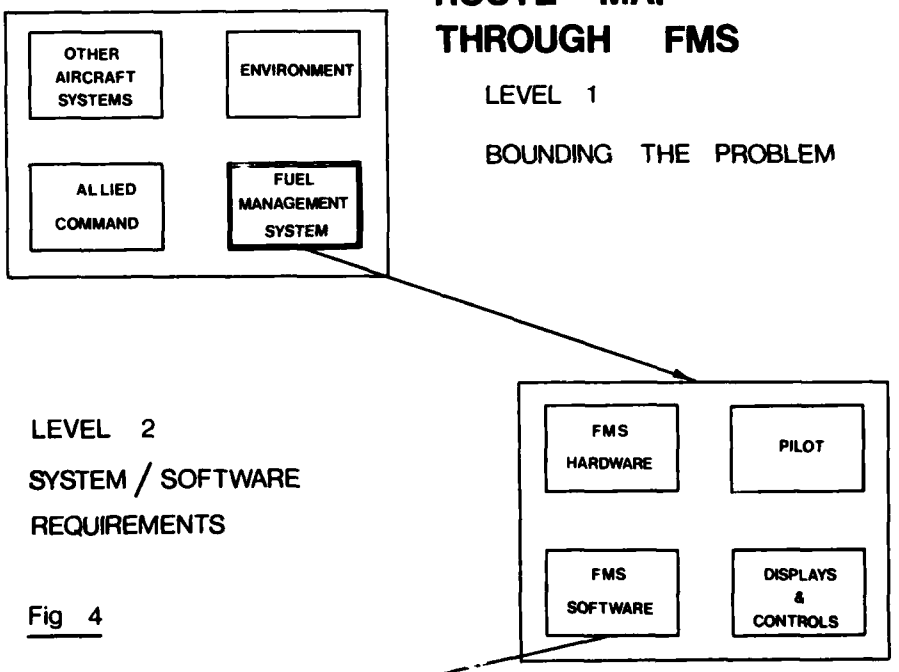
Fig 3

THE CORE NOTATION

ROUTE MAP THROUGH FMS

LEVEL 1

BOUNDING THE PROBLEM



LEVEL 2

SYSTEM / SOFTWARE REQUIREMENTS

Fig 4

LEVEL 3

DETAILED SOFTWARE REQUIREMENTS

ROUTE MAP THROUGH LARGE PROJECT

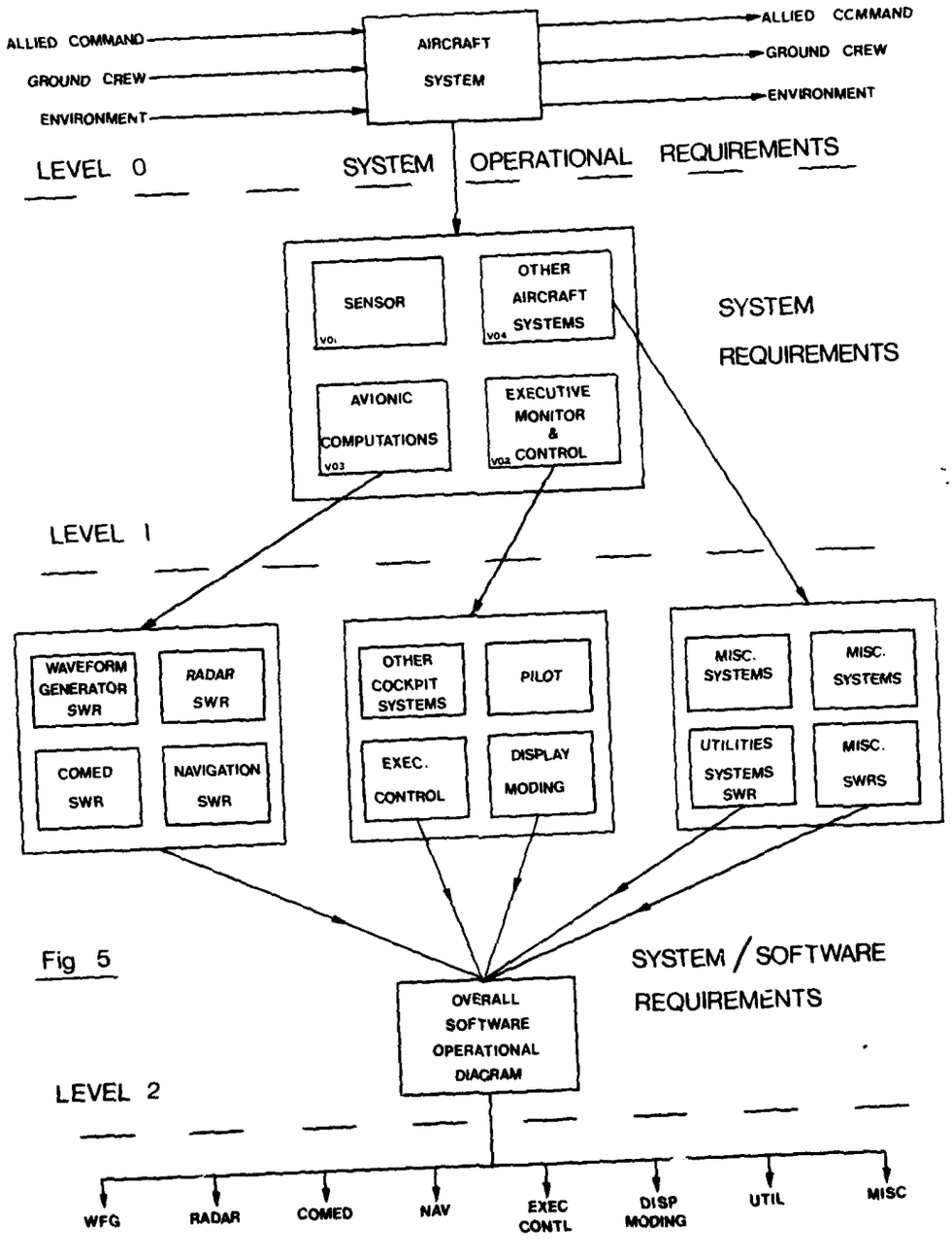


Fig 5

THE A-7E SOFTWARE REQUIREMENTS DOCUMENT:

THREE YEARS OF CHANGE DATA

Louis J. Chmura and David M. Weiss
Computer Science and Systems Branch (Code 7590)
Naval Research Laboratory
Washington, DC 20375
United States

SUMMARY

A major product of the Naval Research Laboratory's Software Cost Reduction project is the software requirements document for the A-7E operational flight program. The document, which was first published in November 1978, is intended to serve as a model for specifying complex software systems. We have carefully tracked changes to the document. The data have consistently suggested that the specification has several desirable qualities, for example, it is easily maintained and is remarkably free of inappropriate implementation detail.

1. INTRODUCTION

1.1. General

Basili and Weiss (1981) as well as Fryer and Weiss (1981) have reported earlier on change data being collected to evaluate the software requirements document (SRD) for the A-7E aircraft (Heninger et al. 1978). The SRD is a product of the Naval Research Laboratory's Software Cost Reduction (SCR) project. Here, we present a more recent picture of the data and, where possible, compare with other published data. In preparation for this paper, we have reexamined all submitted change report forms for consistency and completeness of information. This has resulted in the reclassification of some earlier data with variable effects on the earlier published observations.

The remaining three sections of this introduction review the SCR project, the SRD, and goal-directed data collection for those who are unfamiliar with these topics.

1.2. Software Cost Reduction (SCR) Project

Since January 1978, personnel at the Naval Research Laboratory (NRL) and the Naval Weapons Center (NWC) have been redeveloping version NWC-2 of the operational flight program (OFP) for the A-7E aircraft using such software engineering techniques as information hiding (Parnas 1972), abstract interfaces (Parnas 1977), cooperating sequential processes (Dijkstra 1968), and resource monitors (Hoare 1974). The A-7E OFP is part of the Navigation/Weapon Delivery System on the A-7E aircraft. It receives input data from sensors, cockpit switches, and a panel with which a pilot keys in data. It controls several displays in the cockpit and positions several sensors. In all, twenty-two devices are connected to the TC-2; examples include an inertial measurement set providing velocity data and a head-up display. The head-up display projects symbols into a pilot's field of view, so that he sees them overlaying the world ahead of the aircraft. The OFP calculates navigation information such as present position, speed, and heading; it also controls weapon delivery, giving a pilot steering cues and calculating when to release weapons.

The A-7E OFP is an operational Navy program with severe memory and execution-time constraints. The code consists of about 12,000 assembler language instructions for the IBM System 4 PI model TC-2 computer. The TC-2 has 16K bytes of memory.

The goals of the SCR project are (1) to demonstrate the feasibility of using the selected software engineering techniques in developing complex, real-time software and (2) to provide the Navy with a model for the design of avionics software. One of the reasons for choosing to redevelop the A-7E OFP is the challenge of showing that any memory or execution-time overhead incurred by using the software engineering techniques is not prohibitive for such real-time programs. A second reason is that maintenance personnel at NWC feel that the OFP is difficult to change; the claim for the selected software engineering techniques is that they facilitate the development of software that is easy to change.

The A-7E SRD (Heninger et al. 1978) is the first major product of the SCR project. More recent products of ongoing software design include a guide to SCR software modules (Britton and Parnas 1981), interface specifications for the device interface module (Parker et al. 1980), specifications for the function driver module (Clements 1981), specifications for the extended computer module (Britton et al. 1982), and specifications for the shared services module (Clements 1982).

The projected completion date for the SCR project is September 1985. As of the end of 1981, approximately 10 man-years of technical engineering effort have been expended on the redevelopment.

1.3. A-7E Software Requirements Document (SRD)

The SRD is an attempt to provide a complete and concise description of the A-7E's OFP requirements. No other such requirements description exists despite the existence of a working OFP. The SRD is also an attempt to improve upon present methods of specifying software requirements. Four principles motivate the document: (1) state questions before trying to answer them, (2) separate concerns, (3) be as formal as possible, and (4) organize according to OFP outputs (Heninger 1980). The resultant organization is shown in figure 1.

Personnel at NWC who are maintaining the current A-7E OFP have been active consultants and reviewers of the document, both prior to initial publication and subsequently. The document has been under configuration control since it was first published in November 1978.

As of the end of 1981, approximately 2.5 man-years of technical engineering effort has been expended on the document: approximately 1.5 man-years on producing the original document and the remaining 1 man-year on changing and issuing three updates.

1.4. Goal-Directed Data Collection

The opportunity to apply recent software engineering technology liberally to the development of a complex system is rare. Though true evaluation of the result must wait until delivery of the software, we believe it would be unfortunate to wait until then. From the start of the redevelopment, we have collected data to permit evaluation of the project and the products in the interim.

The data collection methodology being used is goal directed (Weiss 1981). Briefly, it consists of the following five elements.

1. Identify Goals. The data collection effort can be geared to determine how well the goals for a product or process are met. Twelve goals drive data collection for the SRD. Six are original goals for the document (Heninger 1980).

    1. Specify external behavior only.
    2. Specify constraints on the implementation.
    3. Be easy to change.
    4. Serve as a reference tool.
    5. Record forethought about the system life cycle.
    6. Characterize acceptable responses to undesired events or errors.

    Three other goals apply to requirements documents in general.

    7. Be correct.
    8. Promote detection and correction of errors.
    9. Be useful.

    Another three concern the software development process.

    10. Discover effective ways of finding errors.
    11. Characterize changes.
    12. Characterize errors.

2. Determine Questions Of Interest From The Goals. The answer to each question can then help measure how well a stated goal has been obtained. For example, given the goal that the SRD be easy to change, a reasonable question would be:

    Are changes confined to a single section of the document?

    Such a question suggests that data be collected on how many sections were modified for each requirements change.

    Figure 2 is a complete list of questions for the SRD.

3. Develop A Data Collection Form. This is an iterative process. The form is best tailored to the product being studied and to the questions of interest. Figure 3 is the change report form (CRF) currently used to request and record resolution of changes to the SRD.

4. Develop Data Collection Procedures. Procedures for collecting change data are best part of normal configuration control procedures.

5. Validate And Analyze The Data. Reviews and analyses of the accumulating data are concurrent with software development. We have found that validation of the data should occur weekly. The validation should include examining the forms for completeness and consistency. Sometimes it is necessary to interview the originator or the person making the change to obtain omitted data or to resolve problems. Validation of A-7E requirements change forms has several times detected incomplete changes.

2. A-7E SOFTWARE REQUIREMENTS CHANGE DATA

2.1. Background

There are 343 change report forms (CRFs) that were generated against the SRD from November 1978, when it was first published, through December 1981. Only 284 of the requested changes have been resolved (i.e., carefully analyzed and formally accepted or rejected) by the end of 1981. Specifically, 276 have been accepted; 8 rejected (only 2 of which were rejected because the change was deemed not worth the effort.) We have validated the 276 accepted CRFs. We feel most confident in using the data to answer the following six questions from figure 2.

1. Is the requirements document easy to change?
2. Is it clear where a change has to be made?
3. Are changes confined to a single section?
4. What use of the document reveals the most errors?
5. How many errors are found in the document?
6. What kinds of errors are contained in the document?

It is important to understand what we believe properly constitutes a "change." For the SRD, we define a change to be an alteration to a baselined version. We consider two changes to be the same if they have the same cause and are written against the same version of the SRD. As an example, connecting a new device to the A-7E computer might require numerous updates to the SRD, but all the updates would make up a single change. A change that is a completion or correction of a prior change is a separate change.

Occasionally, a CRF is submitted containing two changes. When this happens, we generate a second CRF and separate the two changes onto the two forms. For example, if the following correction was submitted on a CRF, we would generate a second CRF. The first would address just the ambiguity, the second would address the misspelling.

> "The last sentence of the description is ambiguous. Replace it with .... Note also that the word descripter is misspelled."

If two CRFs are submitted that are different parts of the same change, we merge the two CRFs.

We consider that there are two classes of change data: error corrections and non-error corrections. An error correction is either an original error correction (i.e., the first correction of the error) or a completion or correction of a previous change. In terms of figure 3, the CRF for an error correction would have one of the two first two boxes in item 4 marked and the second page filled in. Hereafter, we use the term error to refer to error corrections and the term modification to refer to non-error corrections. The following table shows the number of errors and modifications reported and accepted each year.

	YEAR CHANGE ACCEPTED				
Change Class	1978	1979	1980	1981	1978-1981
Error	4	72	97	74	247
Modification	0	5	13	11	29
Total:	4	77	110	85	276

The small number of modifications can be explained by the fact the the SCR project is a faithful redevelopment of version NWC-2 of the A-7E OFP.

There are eight classes of requirements errors: clerical, ambiguity, omission, inconsistency, incorrect fact, information put into wrong section, implementation fact included, and other. The classification scheme is generally in terms of cause; that is, we classify errors according to their causes, not according to their symptoms. Accordingly, we use the following definitions for the different error classes.

Clerical Error: An error resulting from a mechanical transcription process from one medium to another (e.g., keypunch error when copying from handwritten to computer-processable form).

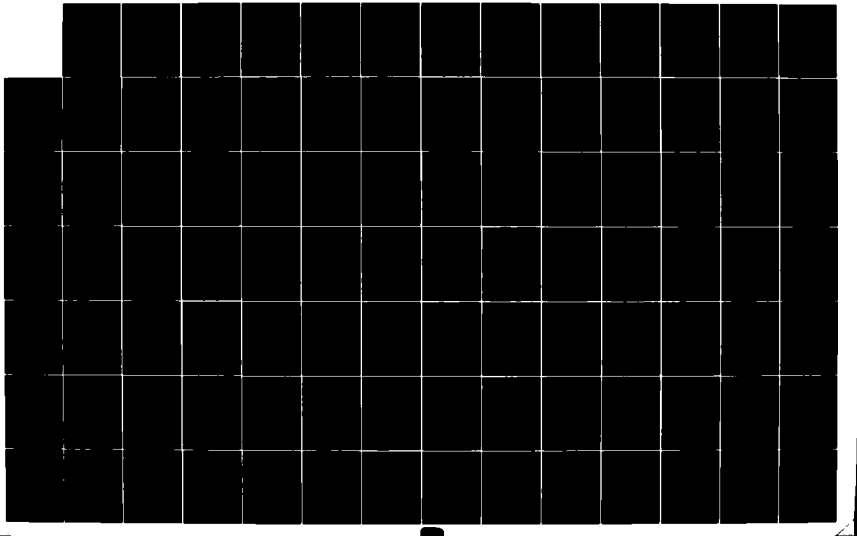
Ambiguity: An error resulting from an author's inability to distinguish clearly among several alternatives. Note that an ambiguity might result from unavailability of information, carelessness, or other reasons.

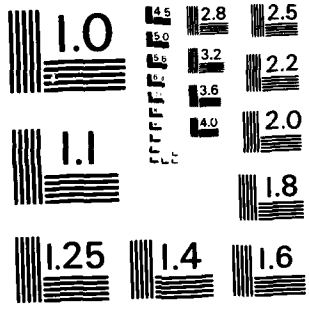
Omission: An error resulting from an author knowing necessary information but not including it in the document.

Inconsistency: An error resulting from authors of two or more different sections of the document believing contradictory statements. The result is that two different parts of the document contradict each other.

AD-A127 131 SOFTWARE FOR AVIONICS(U) ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT NEUILLY-SUR-SEINE (FRANCE) JAN 83 AGARD-CP-330 2/5

UNCLASSIFIED F/G 9/2 NL





Incorrect Fact: An error resulting from an author having the wrong information. The result is that the document contradicts other sources of information.

Information Put Into Wrong Section: An error resulting from improper separation of concerns, such as including a description of the data available from a radar in the same section as a description of the computer interrupt structure.

Implementation Fact Included: An error resulting from an author including information about how to implement a requirement.

Other: An error resulting from some cause other than those specified in the foregoing.

Note that determining the cause, and hence the class, of an error sometimes requires the help of the authors of the document. Furthermore, there are some errors for which the proper class is not completely clear. To reduce the number of such situations, we train all data analysts to use the same classification criteria and use more than one analyst to review questionable cases. We estimate that fewer than 5% of the changes may be improperly classified.

The statistics we present on the effort required to design changes follow from the times supplied in item 5 of the requirements CRF. The times are those that technical persons expend on understanding and specifying (i.e., designing) changes in sufficient detail so that an editor or typist can update the document maintained on a word processor. Thus, editorial and secretarial time are not included.

We classify the effort expended in designing changes as follows:

Class	Design Effort (e)
Trivial (T)	0 < e ≤ 1 work hour (wh)
Easy (E)	1 wh < e ≤ 1 work day (wd)
Moderate (M)	1 wd < e ≤ 1 work week (ww)
Difficult (D)	1 ww < e ≤ 1 work month (wm)
Formidable (F)	1 wm < e

It is interesting that, of the 276 accepted changes, none have been Difficult (D) changes and only 2 have been Formidable (F) changes.

2.2. Is the document easy to change?

Figure 4 is the distribution of effort for designing most of the changes to the SRD. Excluded are changes to correct clerical errors because such changes typically involve simple fixes such as correcting misspellings or punctuation. Excluded also are change completions, though the efforts for change completions have been added to the efforts reported for the corresponding original changes. The fact that 94% of the changes required a day or less to design, together with the fact that only two requested changes were rejected because the efforts to make them seemed unjustified, suggest that the SRD is indeed easy to change. What's more, the contrasts between figure 5 and figure 6 suggest that recent changes tend to be even easier than earlier changes.

Figure 7 is an alternative for figure 4. Changes to correct clerical errors are again excluded, but change completions are included. Because the majority of the change completions continued the work initiated by two formidable changes, the distribution of figure 7 is close to that of figure 4.

Actually, the general form of figures 4 through 7 is to be expected; it seems to be characteristic of accepted changes in general. It is quite similar to figure 8, a distribution of effort to design software changes for a NASA software development project (Weiss 1981). It is also similar to the distribution of working time to correct errors during test and integration as presented by Shooman and Bolsky (1975). The simple fact of the matter may be that a software project simply cannot survive if personnel tie themselves up with making too many time-consuming changes; potentially difficult or formidable changes will tend to be rejected unless there is no other way out. What is remarkable about figures 4 through 7 is that only two requested changes have been rejected because they were deemed not worth the effort.

As a note of caution, recall that the SCR project is different from many software-production projects in that it is a faithful redevelopment of an existing OFP. The variety of changes for the SRD is less than for many such documents. For example, the OFP interfaces have not changed because of the introduction of new interfacing equipment.

2.3. Is it clear where a change has to be made?

Of the 276 requirements changes, 70 correct clerical errors. Of the remaining 206 changes, only 23 (11%) are change completions. This small percentage by itself suggests it is clear where requirements changes have to be made. Another measure suggests the same: Of the 206 changes, only 35 (17%) required examination of more sections than were changed.

Figure 9 shows the yearly percentage of changes that required examination of more sections than the number actually changed. The trend is not encouraging. Recent changes, which tend to be easier as noted in section 2.2, are nevertheless requiring the examination of more material. One explanation for the trend may be the fact that the original authors of the requirements document no longer design most of the changes; persons less expert in the document have taken their places. Another explanation may be simply that recent changes are more subtle than earlier changes. We will have to wait to see which explanation applies. If the trend slows or stops in the future, then the first explanation would seem to hold. If the trend continues, then the second would seem more valid.

2.4. Are changes confined to a single section?

There are 183 changes excluding change completions and clerical errors. If we combine change completion data with the corresponding original changes, then 40 (22%) of the 183 changes involve more than one section. Figure 10 plots this statistic over the years. Together these numbers indicate that, early on, changes tended only slightly to be confined to one section. More recently, confinement to one section tends to be the rule.

2.5. What use of the document reveals the most errors?

Of the 74 errors corrected in 1981, 18 are clerical. The distribution of ways in which the remaining 56 nonclerical errors have been detected is given in figure 11. The distribution clearly shows that, in 1981, the SCR project was heavily into design. Figure 12 shows that design activity has been the primary way in which errors have been detected. Although not surprising statistics, they are nevertheless encouraging because they indicate that the SRD is being heavily used by designers. Apparently, the document is meeting two of the goals stated for it: (1) that it be useful and (2) that it serve as a reference tool.

The cumulative distribution of figure 12 should be of special interest toward the end of the SCR project. If the requirements document is of high quality and if the approach being taken to software development is successful, then relatively few nonclerical requirements errors should be reported as a result of testing. In other words, the distribution of figure 12 should retain its same general shape.

2.6. How many errors are found in the document?

There are 247 requirements errors that were corrected from 1978 through 1981, 70 (28%) of which are clerical. This seems to be a small number of nonclerical errors considering that the SRD contains approximately 600 pages. The error-per-page ratio is only 0.30.

Bell and Thayer (1976) report on 972 problems with a B-5 level software specification that comprised approximately 2500 pages. About 50 (5%) are problems of new or changed requirements, which cannot occur in the A-7E OFP redevelopment. The remaining 922 problems yield a problem-per-page ratio of 0.37, which is not very different from the above ratio for the SRD. But the problems reported by Bell and Thayer are the result largely of two formal requirements reviews conducted during system development. They comprise some fraction of the total number of problems found. Thus, the actual problem-per-page ratio is likely much greater than 0.37.

2.7. What kinds of errors are contained in the document?

The distribution of error classes in 1981 is shown in figure 13; in 1979, figure 14; cumulatively, figure 15. Except for some variation in the percentage of inconsistencies, there is little difference in the distributions. The data suggest that the requirements document is much more precise (has relatively few ambiguities) and somewhat more consistent than it is complete and correct. Perhaps most remarkable is that there are no errors of the type "inclusion of implementation facts." The authors seem to have succeeded extremely well in their goal of specifying external behavior only.

It is also interesting to note that the percentage of clerical errors found has remained constant with time. Of the 70 clerical errors so far found, only two were introduced by earlier changes.

3. CONCLUSION

We have two objectives in monitoring changes to the SRD. The first is to test the feasibility of goal-directed data collection. The second is to measure the success of the document's authors in meeting their objectives. This second objective is important because one of the main goals of the SCR project is to produce a model for engineering OFP software. The SRD is an important part of that model.

Our work has shown that goal-directed data collection can be feasibly integrated with traditional configuration control activities. The major difficulty is that there must be constant careful attention to ensure accurate information; that is, validation must be concurrent with data collection.

Our analysis of the three years of change data suggests that the authors of the SRD have

successfully met some major objectives. In particular:

1. The document seems to be easily maintained, even though a growing percentage of recent changes requires the examination of more sections than those that must be changed.

2. The document seems to be well structured. Changes tend to be confined to single sections.

3. Based on comparison with published data, relatively few errors have been found to date.

4. The document is remarkably free of ambiguities and inappropriate implementation details.

5. The document is a living document. It has been heavily used during design.

These are of course interim results, and data collection will continue throughout the SCR project. Nevertheless, the change data from the start has consistently supported this positive evaluation of the SRD (see Basili and Weiss 1981). There do not appear to be any significant trends in the data that suggest the conclusions will change.

4. ACKNOWLEDGEMENTS

Special thanks go to Paul Clements who patiently answered our seemingly endless questions about submitted CRFs. Miss Tamara Lewis is responsible for the high-quality histograms, which she produced despite problems with our home-grown plot package. Of course, we owe much to all those who have patiently filled out CRFs and continue to do so.

5. REFERENCES

Basili, Victor R., and Weiss, David M. 1981.
Evaluation of a Software Requirements Document By Analysis of Change Data.
Proceedings, Fifth International Conference on Software Engineering, pp. 314-323.
Long Beach CA: IEEE Computer Society.

Bell, Thomas E.; and Thayer, Thomas A. 1976.
Software Requirements: Are They Really A Problem?
Proceedings, 2nd International Conference On Software Engineering, pp 61-68. Long Beach CA: IEEE Computer Society.

Britton, Kathryn H., and Parnas, David L. 1981.
A-7E Software Module Guide.
NRL Memorandum Report 4702. Washington, DC: Naval Research Laboratory.

Britton, Kathryn H.; Parnas, David L.; and Weiss, David M. 1982.
Interface Specifications For The SCR (A-7E) Extended Computer Module.
NRL Memorandum Report. Forthcoming. Washington DC: Naval Research Laboratory.

Clements, Paul C. 1981.
Function Specifications for the A-7E Function Driver Module.
NRL Memorandum Report 4658. Washington DC: Naval Research Laboratory.

------. 1982.
Interface Specifications for the A-7E Shared Services Module
NRL Memorandum Report. Forthcoming. Washington DC: Naval Research Laboratory.

Dijkstra, Edsger W. 1968.
Cooperating Sequential Processes.
Programming Languages, ed. F. Genuys, pp. 43-112. New York: Academic Press.

Fryer, Sandra R., and Weiss, David M. 1981.
Evaluation of the A-7E Software Requirements Document By Analysis of Change Data: Two Years of Change Data.
Paper presented at the 15th Annual Asilomar Conference On Circuits, Systems, and Computers, November 1981.

Heninger, Kathryn L. 1980.
Specifying Software Requirements for Complex Systems: New Techniques and Their Application.
IEEE Transactions on Software Engineering, vol SE-6, no 1, January 1980.

Heninger, Kathryn L.; Kallander, John; Parnas, David L.; and Shore, John E. 1978.
Software Requirements for the A-7E Aircraft.
NRL Memorandum Report 3876. Washington, DC: Naval Research Laboratory.

Hoare, C. A. R. 1974.
Monitors: An Operating System Structuring Concept.
Communications of the ACM, vol. 17, no. 10 (October 1974), pp. 549-557.

Parker, Robert A.; Heninger, Kathryn L.; Parnas, David L.; and Shore, John E. 1980.
Abstract Interface Specification for the A-7E Device Interface Module.
NRL Memorandum Report 4385. Washington DC: Naval Research Laboratory.

Parnas, David L. 1972.
On the Criteria To Be Used in Decomposing Systems into Modules.
Communications of the ACM, vol. 15, no. 12 (December 1972) pp. 1053-1058.

------. 1977.
Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems.
NRL Report 8047. Washington, DC: Naval Research Laboratory.

Shooman, M. L., and Bolsky, M. I. 1975.
Types, Distribution, and Test and Correction Times For Programming Errors.
Proceedings -- 1975 International Conference on Reliable Software. SIGPLAN Notices, vol 10, no 6 (6 June 1975) pp 347-357.

Weiss, David M. 1981.
Evaluating Software Development By Analysis Of Change Data.
Ph.D. Dissertation. Technical Report TR-1120. College Park: University of Maryland.

0. Introduction: A description of the document organization, an abstract for each section, and a guide to the notation used.

1. Distinguishing Characteristics of the TC-2 Computer

2. Input and Output Data Items: Description of the information received and transmitted by the computer, organized according to device, one subsection per device connected to the computer.

3. Modes of Operation: States of the program corresponding to aircraft operating conditions.

4. Time-independent Description of A-7 Software Functions: Each function description characterizes one or more output date items and specifies the conditions under which they are updated.

5. Timing Requirements: Timing requirements for all functions described in section 4.

6. Accuracy Constraints on Software Functions

7. Undesired Event (UE) Responses: Desired behavior of the system when undesired events occur.

8. Required Subsets: Useful subsets of the system obtainable by omitting parts of the code.

9. Possible Changes: Possible future modifications to the OFP.

10. Glossary: Glossary of acronyms and technical terms used by the A-7 community.

11. References

Indices: Alphabetical indices to data item descriptions, mode overviews, and functions.

Dictionary: Definitions of standard terms used in the mode (section 3) and function (section 4) descriptions.

FIGURE 1. Sections of the A-7E Software Requirements Document

1. Is externally-visible behavior only specified without implying a particular implementation?
2. Are the appropriate external interfaces specified?
3. Are the external interfaces specified correctly?
4. Is the document easy to change?
5. Is it clear where a change has to be made?
6. Are the changes likely to occur predicted correctly?
7. Are changes confined to a single section?
8. Is the proper set of undesired events described?
9. Is the notation used unambiguous?
10. Which sections have the most errors?
11. Where do the most changes have to be made?
12. Which type of tables has the most errors?
13. Does the document contain unnecessary information?
14. What use of the document reveals the most errors?
15. Are sections 3 (Modes) and 4 (Functions) consistent with each other?
16. Is the dictionary complete, correct, and consistent with the rest of the document, and will it remain so?
17. Which subsections of sections 2 (Data Items), 3 (Modes), and 4 (Functions) are most error-prone?
18. How is the document being used?
19. Why are changes being made?
20. How many errors are found in the document?
21. What kinds of errors are contained in the document?

FIGURE 2. Questions Underlying the Change Report Form (CRF) for the A-7E Software Requirements Document

A-7 Requirements Document Change Report Form

Number _____

Current Date..... Month Day Year
Change Started On...

Change Discovery

1. How was the document being used when the need for the change was discovered?
   - ☐ Validation review by the authors
   - ☐ Validation review by non-authors
   - ☐ As a maintenance reference
   - ☐ As a software design reference
   - ☐ As a coding reference
   - ☐ As a test reference
   - ☐ Other: _____

Identification

2. Description of change: _____

3.

Section(s) Changed		Section(s) Examined But Not Changed	
Name	Sectio	Subsection(s)	Subsection(s)
Intro.	0.		
TC 2	1.		
Data Items	2.		
Modes	3.		
Functions	4.		
Timing	5.		
Accuracy	6.		
UE's	7.		
Subsets	8.		
Changes	9.		
Glossary	10.		
Sources	11.		
Data Item Index			
Mode Index			
Function Index			
Dictionary			

Type Of Change

4. Why is the change being made (check one)?
   - ☐ To correct an original error
   - ☐ To complete or correct a previous change (Previous CRF # _____)
   - ☐ To comply with unexpected requirement change (violates sect. 9 assumption)
   - ☐ To comply with expected requirement change (assumed in sect. 9 subsect. __)
   - ☐ To remove unnecessary information
   - ☐ To reorganize within one section } Readability
   - ☐ To reorganize among several sections } Improvement
   - ☐ Other: _____

5. What was the effort in person-time required to understand and make the change?

   0 | 1 man-hr | 1 man-day | 1 man-week | 1 man-month

6. Estimate the percentage of the effort just to understand the change.

   0 | 20 | 40 | 60 | 80 | 100

FOR ERROR CORRECTIONS ONLY

Type Of Error

7. How is the error best characterized (check one)?
   - ☐ Clerical
   - ☐ Ambiguity
   - ☐ Omission
   - ☐ Inconsistency
   - ☐ Incorrect Fact
   - ☐ Information put in wrong section } Inappropriate Fact
   - ☐ Implementation fact included
   - ☐ Other: _____

For Section 2 (Data Item) Errors

8. Which part(s) of the subsection were incorrect (check all that apply)?
   - ☐ Hardware
   - ☐ Description
   - ☐ Value Characteristics
   - ☐ Instruction Sequence
   - ☐ Data Representation
   - ☐ Comment

For Section 3 (Modes) and Section 4 (Functions) Errors

How is the error best characterized (check all that apply)?

9. Section 3
   - ☐ Mode Transition Error
   - ☐ Mode Condition Error
   - ☐ Mode Overview Error

10. Section 4
    - ☐ Applicable Mode List Error
    - ☐ Output Data Item Error
    - ☐ Output Description Error
    - ☐ Event Table Error
    - ☐ Selector Table Error
    - ☐ Condition Table Error
    - ☐ Inconsistent or incomplete table

11. ☐ Corresponding errors were made in both sections 3 and 4.

For Errors Involving Dictionary Items

12. How is the error best characterized (check all that apply)?
    - ☐ Incorrect item
    - ☐ Incomplete item
    - ☐ Dictionary inconsistent with usage elsewhere

For Notational Errors

13. Which notation type was in error?
    - ☐ /input/
    - ☐ //output//
    - ☐ $value$
    - ☐ !item!
    - ☐ *mode*
    - ☐ operator
    - ☐ event
    - ☐ simple condition
    - ☐ compound condition

14. How is the error best characterized?
    - ☐ incorrect grouping
    - ☐ incorrect symbol
    - ☐ incorrect value

Comments

15. Please give any information that may help understand the change and its cause.

Name _____ Date _____

FIGURE 3. A-7 Requirements Document Change Report Form

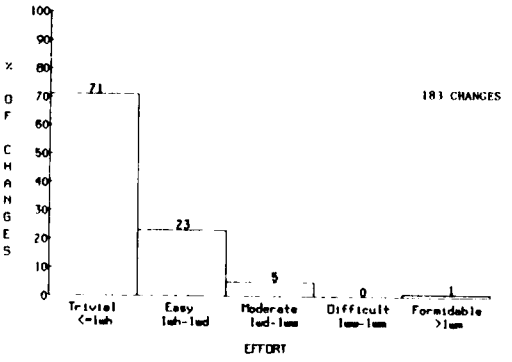


FIGURE 4. Effort to Design Changes Excluding Clerical Errors and Change Completions (1978-1981)

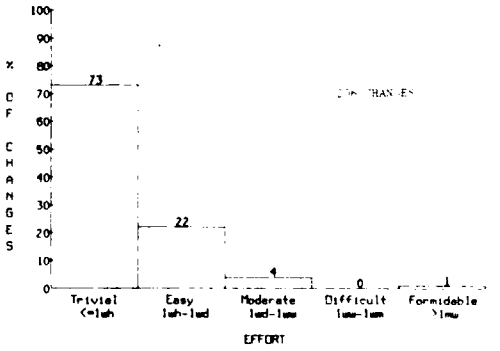


FIGURE 7. Effort to Design Changes Excluding Clerical Errors (1978-1981)

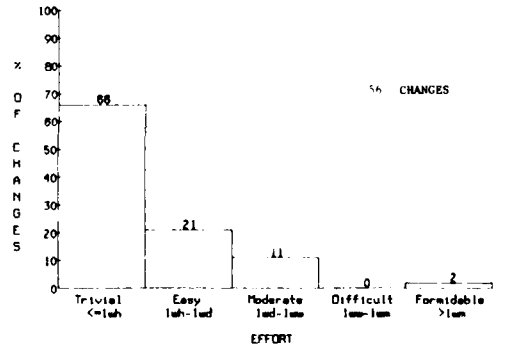


FIGURE 5. Effort to Design Changes Excluding Clerical Errors and Change Completions (1978-1979)

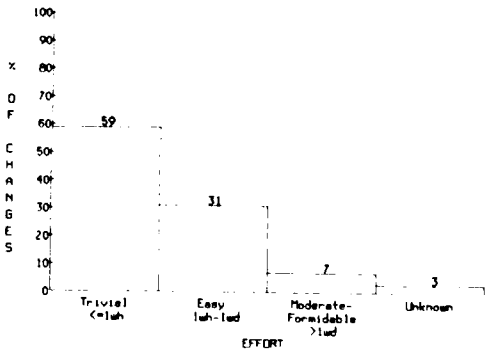


FIGURE 8. SEL1 Effort to Design Changes (Figure 5.8 from Weiss 1981)

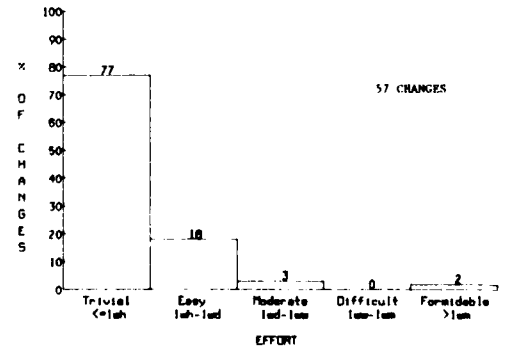


FIGURE 6. Effort to Design Changes Excluding Clerical Errors and Change Completions (1981)

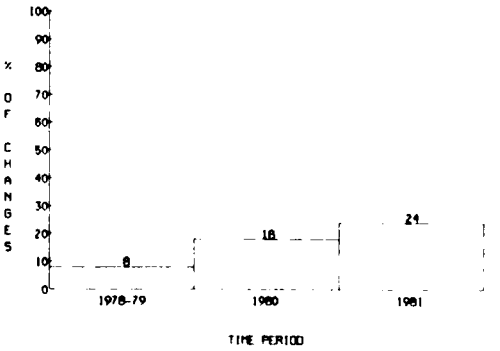


FIGURE 9. Percentage of Changes Requiring Examination of More Sections Than Those Changed (1978-1981)

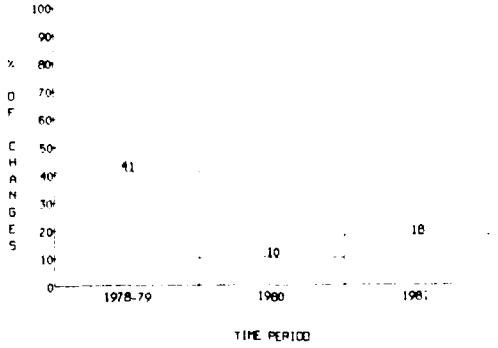


FIGURE 10. Percentage of Changes Involving More Than One Section (1978-1981)

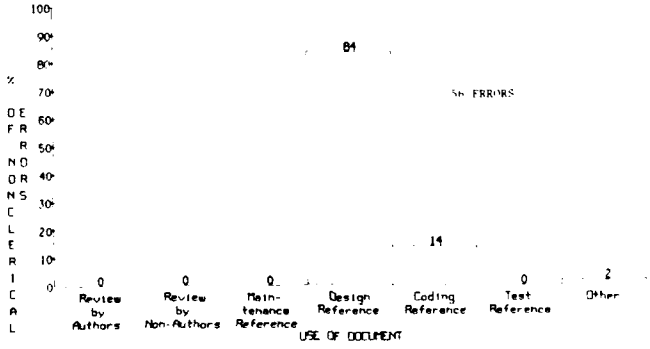


FIGURE 11. Detection of Nonclerical Errors (1981)

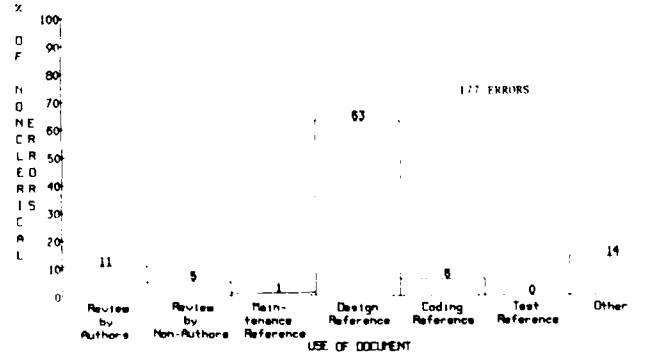


FIGURE 12. Detection of Nonclerical Errors (1978-1981)

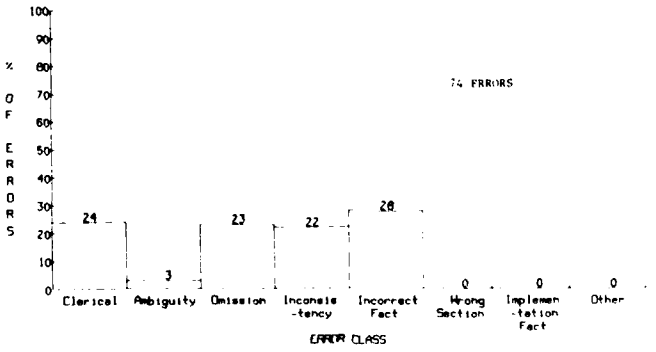


FIGURE 13. Error Classes (1981)

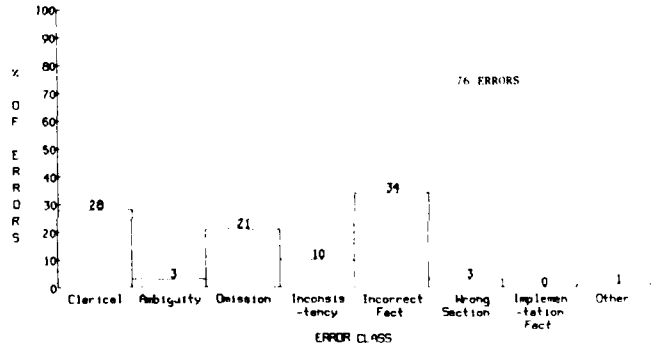


FIGURE 14. Error Classes (1978-1979)

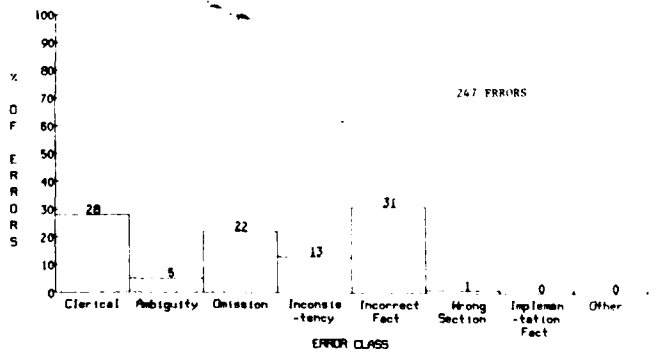


FIGURE 15. Error Classes (1978-1981)

D. L. A. O. : UN SYSTEME D'AIDE

A LA DEFINITION DE LOGICIELS AVIONIQUES

Sylvie CHENUT-MARTIN et François DOLADILLE

Electronique Serge Dassault
55, Quai Carnot
92214 SAINT CLOUD
Tél. : 602.70.17. / 602.50.00

RESUME

Cet article décrit les grandes lignes du système D.L.A.O. (Définition de Logiciel Assistée par Ordinateur), dont l'objectif est d'aider à l'élaboration de spécifications de logiciel temps réel, et en particulier de logiciel avionique. Il présente tout d'abord les buts de l'étude et la démarche qui a été suivie pour les atteindre : analyse du processus de définition appliqué par l'ESD pour le logiciel des calculateurs embarqués, analyse des besoins des utilisateurs et des systèmes existants.

Les solutions retenues sont ensuite développées en soulignant plus particulièrement les points suivants :

- simplicité de mise en oeuvre : S.O.S., saisie des informations assistée par l'outil, ...
- facilité de construction de documents.

A terme, le système doit s'intégrer dans un ensemble cohérent d'outils couvrant les différentes étapes du cycle de vie du logiciel ; les problèmes posés par l'intégration avec un système d'aide à la conception sont évoqués.

I - INTRODUCTION

La taille et la complexité, sans cesse croissantes, des logiciels développés dans la dernière décennie ont mis en évidence l'importance primordiale de la phase de définition dans le cycle de vie d'un logiciel. Certaines mesures effectuées sur de gros projets [BOE 78] [BOE 82] permettent ainsi d'estimer que le coût des seules erreurs de définition représente en moyenne le tiers du coût total du logiciel.

A l'origine de ces erreurs, que l'on peut répertorier en différentes catégories : ambiguité, omission, incohérence, surspécification, incompréhensibilité, ... , se trouvent évidemment la complexité et le volume des spécifications, mais leur cause immédiate tient souvent à l'absence d'outils spécifiques à cette étape de définition et en particulier l'écriture des spécifications en langage naturel [MEY 79].

Cette absence de moyens automatiques est particulièrement ressentie par tous ceux qui ont à développer des logiciels complexes : la définition du calculateur principal d'avions militaires peut représenter plus de 2000 pages de textes et de schémas. Les modifications apportées aux spécifications initiales, pendant la phase même de réalisation du logiciel, y sont très nombreuses (de l'ordre de 1000). Ces mises à jour sont délicates du fait d'une structure de documents mal adaptée à l'évaluation de l'impact de ces évolutions sur l'ensemble du logiciel.
C'est pourquoi l'E.S.D. développe actuellement dans le cadre de deux contrats, l'un avec la Direction des Recherches, Etudes et Techniques (DRET), l'autre avec l'Agence Spatiale Européenne (ESA) un système dont l'objectif principal est d'offrir un support simple et facile à mettre en oeuvre pour la phase de définition de logiciels. Ces travaux sont le résultat d'une collaboration avec la SNIAS-DSBS et visent à satisfaire les besoins de différents industriels dans des contextes variés (il est à noter que le système, développé aujourd'hui en français, peut être, par des modifications mineures, adapté à la langue anglaise).

II - OBJECTIFS

Les objectifs que doit atteindre un système d'aide à la définition sont multiples :

- Accroître la qualité de la spécification , c'est-à-dire réduire de façon importante les erreurs citées ci-dessus. Le système doit offrir à l'utilisateur un "langage" simple, autorisant de nombreux contrôles mais lui permettant également de mieux analyser ses besoins.

- Faciliter les mises à jour, c'est-à-dire prendre en compte en temps réel toutes les modifications, les contrôler, détecter leurs répercussions et éviter qu'elles ne rendent incorrect le reste de la spécification.

- Fournir une documentation fiable et adaptée aux besoins des différents utilisateurs. Le système doit offrir des documents à la demande qui satisfont l'ensemble des points de vue ; pour le document de spécifications, qui est la plupart du temps la base contractuelle d'engagement entre le réalisateur et le demandeur de logiciel, il est indispensable d'assurer une très grande lisibilité qui permette à un non - informaticien de s'engager.

D'autre part, il est évident que l'utilité et donc l'efficacité d'un tel système sont liées très étroitement aux avantages ergonomiques de ses interfaces ; il est donc indispensable que le système fournisse à son utilisateur une structure souple mais efficace donc des fonctions simples mais faciles à mettre en oeuvre.

Pour satisfaire ces objectifs, la démarche suivie lors de la définition du système D.L.A.O. a consisté :

1 - A choisir comme thème d'expérimentation un document de spécifications existant dont le contenu soit représentatif des principaux problèmes rencontrés lors de la définition de logiciels complexes. L'étude de ce document a permis, d'une part d'en analyser les défauts et leurs causes, d'autre part d'identifier la forme d'un texte de spécification et la nature des concepts qui y étaient traités.

2 - A interroger des rédacteurs de spécifications, utilisateurs potentiels du système, pour connaître leurs besoins.

3 - A étudier les systèmes existants (ISDOS [TEI 77], SREM [ALF 80], SADT [SOF 76], ZAIDE [CHE 80],...) pour identifier les écueils à éviter et pour voir dans quelle mesure ces systèmes pouvaient satisfaire les besoins mis à jour lors de l'étape précédente.

Cette démarche, qui a permis de dresser un bilan détaillé des besoins réels d'un industriel en matière de définition de logiciels, a mis en évidence les soucis fondamentaux des utilisateurs :

. facilité d'apprentissage et d'utilisation du système, notamment au niveau du langage,
. bonne lisibilité des documents produits, possibilité d'éditions graphiques du type SADT ou R-nets [ALF 76],
. adaptabilité du langage aux problèmes traités.

C'est en fonction de ces soucis que nous avons défini D.L.A.O. dont certains aspects sont développés ci-dessous.

III - LE SYSTEME D.L.A.O.

3.1. Le langage

Il a été élaboré à partir d'une analyse des documents de spécification de logiciels temps réel développés à l'ESD ; chacun des concepts qui était manipulé dans de tels documents a été identifié et est exprimé dans le langage. L'objectif était de trouver un compromis satisfaisant entre formalisme, lisibilité et facilité d'emploi. Il a été trouvé en réduisant à 5 les types d'objets à définir. La spécification se présente comme une suite de définitions d'objets de l'un des 5 types suivants :

. Les informations : Ce sont les données opérationnelles manipulées dans le logiciel.
Exemple : "Mode de fonctionnement du radar" est une donnée opérationnelle en entrée du logiciel dont on exploitera les différentes valeurs : télémétrie Air-Sol, visualisation du sol,... pour la spécification.

Les interfaces : Ce sont les supports physiques des informations.
Exemple : L'information "Mode de fonctionnement du radar" est implémentée par l'interface "Mode radar", qui est un champ de 10 bits.

. Les événements : Ce sont les faits aléatoires ou non, éventuellement accompagnés d'informations, auxquels le logiciel doit réagir.

Exemple : Le "passage du radar du mode télémétrie Air-Sol au mode Visualisation du sol" est un événement, le logiciel doit en effet dans ce cas effectuer certaines initialisations et des traitements spécifiques au nouveau mode.

. Les états : Ce sont des ensembles de propriétés vérifiées par le logiciel à un instant donné.

Exemple : Au temps t, le logiciel est dans l'un des états suivants : mode Navigation, mode Air-Air, mode Air-Sol.

. Les traitements : Ce sont les fonctions accomplies par le logiciel.

Les exemples qui suivent (définitions successives d'une information et d'une interface) sont prélevés dans un logiciel avionique. Ils ne tiennent pas compte des facilités de saisie offertes au spécificateur (cf. 3.2.), ni des possibilités d'éditions fournies par le système (cf. 3.5.)

Définition d'une information

```
Latitude_additionnelle : INFORMATION
  SYNONYME            ecart_de_latitude
  ENTREE
  REEL
  UNITE    degre
  DOMAINE DE VARIATION [-180, 180]
  VALIDE SI coordonnees_but = valide     /
```

Définition d'une interface de type message

```
centrale_inertielle_6 : INTERFACE
MESSAGE
EMETTEUR        centrale_inertielle
RECEPTEUR       calculateur1,
                calculateur2,
                pilote_automatique,
                radar,
                viseur
6,25 HZ
CARACTERISATION * commande label 88, commande complementaire 13 *
LONGUEUR 80 BITS
        COMPOSITION
        MOT          validite_centrale_inertielle ;
                     etat_centrale_inertielle ;
        DOUBLE-MOT   mode_centrale_inertielle ; /
```

3.2. <u>Facilités de saisie</u>

Lors de l'exécution de chacune des fonctions offertes par le système, l'utilisateur dispose d'un certain nombre de facilités de niveau général ; en particulier, il peut avoir recours, à tout moment, à un 'S.O.S.', pour lui expliquer le fonctionnement du système quand il le désire ; de même, il peut définir et utiliser des macros, aussi bien au niveau du langage de commande (interprêteur SHELL d'UNIX [BOU 78]) qu'au niveau du langage de spécification, ce qui lui évite ainsi certaines tâches répétitives (procédures de commandes ou définitions d'objets souvent utilisées).

D'autre part, pour la saisie interactive des spécifications, il existe, outre un mode <u>"libre"</u> dans lequel la seule assistance apportée par le système est le S.O.S., un mode <u>"assisté"</u>. Dans ce mode, l'utilisateur est guidé pas à pas dans la rédaction de sa spécification grâce à un jeu de questions/réponses.
L'enchaînement de la saisie est assuré par le système en fonction des options choisies par l'utilisateur. La syntaxe du langage lui est alors parfaitement transparente.

Le passage d'un mode à l'autre peut se faire à tout moment. Dans le cas d'un passage mode libre - mode assisté, le système se branche directement au questionnaire nécesaire à la poursuite de la spécification.

L'exemple ci-dessous présente un court scénario d'utilisation du mode assisté pour un objet de type information (fin de définition).

Chaque figure symbolise un écran :
- la partie supérieure de l'écran est réservée au système pour présenter la spécification indentée, rappeller la fonction en cours ;
- la partie inférieure permet à l'utilisateur de fournir le texte de sa spécification ou de répondre aux questions posées par le système et de sélecter la fonction en cours ; les notations utilisées sont les suivantes : majuscules pour les affichages du système et minuscules pour celles du spécificateur, le (T) signifie transmission.

```
but_de_preparation : INFORMATION
   SYNONYME         but_prep
   ENTREE
   ENTIER
   DOMAINE DE VARIATION [1,63]
   * les buts 93,94 sont pour le moment en reserve *

                         FONCTION : DEFINITION MODE ASSISTE
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

L'INFORMATION EST-ELLE PARAMETREE PAR LE TEMPS ? 1 (T)

1 - OUI
2 - NON


COMMANDE :
```

```
but_de_preparation : INFORMATION
   SYNONYME          but_prep
   ENTREE
   ENTIER
   DOMAINE DE VARIATION [1,63]
   * les buts 93,94 sont pour le moment en reserve *
   PARAMETRE PAR LE TEMPS
                              FONCTION : DEFINITION MODE ASSISTE
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

DES CONDITIONS DE VALIDITE SONT-ELLES ATTACHEES A
L'INFORMATION ? 2 (T)

1 - OUI
2 - NON


COMMANDE :
```

```
but_de_preparation : INFORMATION
   SYNONYME          but_prep
   ENTREE
   ENTIER
   DOMAINE DE VARIATION [1,63]
   * les buts 93,94 sont pour le moment en reserve *
   PARAMETRE PAR LE TEMPS     /
                               FONCTION :
-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-.-

MISE EN GARDE : L'OBJET CHANGEMENT BUT EST
                UTILISE ET NON DECLARE
                VEUILLEZ LE DEFINIR




COMMANDE :
```

<u>FIN DE DEFINITION AVEC DIAGNOSTIC</u>

3.3. <u>Mise à jour</u>

L'un des buts du système est de faciliter les mises à jour et, en particulier, d'éviter que des modifications anarchiques aux répercussions mal définies provoquent une dégradation du texte écrit. Pour cela, à chaque mise à jour, le système contrôle le texte modifié (les contrôles étant de même nature qu'en saisie : cohérence, complétude) mais aussi visualise l'impact de la modification, c'est-à-dire l'ensemble des objets en relation avec l'objet modifié, ce qui permet à l'utilisateur de cerner complètement les répercussions de sa mise à jour, qu'il peut valider ou non.

3.4. Bibliothèque de spécifications

Il arrive souvent que les spécifications de deux projets différents aient des parties communes, notamment dans le cas de logiciels avioniques où les missions à remplir sont souvent de même nature. Il est donc intéressant de pouvoir récupérer tout ou partie d'une spécification déjà existante plutôt que d'avoir à la réécrire. A partir des spécifications archivées dans les bases de données projet, on va donc créer des bibliothèques de spécifications équivalentes aux bibliothèques de modules en programmation.

3.5. Editions

Il existe deux types de documents fournis par le système :

- Les éditions ponctuelles, qui peuvent être demandées à tout moment par l'utilisateur pour obtenir le ou les renseignements spécifiques nécessaires à la poursuite de son activité. L'emploi d'une base de données comme structure d'archivage permet l'accès sélectif à toutes les informations relatives à un problème particulier et facilitent grandement le travail par rapport à la documentation sur papier utilisée aujourd'hui où il est difficile de rassembler tous ces éléments.
  Ces éditions, qui travaillent directement à partir des objets contenus dans la spécification, permettent d'obtenir différents types de sortie :

  - Edition de la spécification d'un objet : trois formats peuvent être demandés : squelette syntaxique seul, avec commentaire et "en pseudo-français", le système insérant des bribes de phrases aux endroits voulus.
  - Edition de références croisées : répondant aux requêtes du type "Quels sont les objets vérifiant telle ou telle propriété ?"
  - Arbre de décomposition des événements ou des informations, qui montrent la construction de ces objets à partir d'objets de plus bas niveau.
  - Diagramme d'enchaînement des traitements (de type R-nets)
  - Flux de données, qui montre comment circulent et où sont élaborées les données.
  - Diagramme d'état, qui symbolise graphiquement les transitions entre états
  - ...

- L'édition du document de référence, qui est la plupart du temps la base contractuelle d'engagement entre le demandeur et le réalisateur de logiciel et qui doit donc être aussi lisible et aussi proche que possible des documents actuels de spécification. Pour cela, un outil documentaire, simple et facile à mettre en oeuvre, a été défini. Il permet à l'utilisateur de construire son document comme il le désire à partir des éléments documentaires constitués par les éditions ponctuelles. Grâce à cet outil, la gestion de la "structure" documentaire est totalement indépendante du contenu du document.

La construction du document peut se faire sous le contrôle d'un plan-type qui est lui-même défini par un utilisateur privilégié pour chaque projet, suivant les standards méthodologiques employés. Cet outil, vu sa généralité, peut s'appliquer à chacune des phases du cycle de vie du logiciel.

Le texte qui suit est un exemple de documentation fournie par le système : il est constitué de 2 chapitres ; le premier est une édition du type diagramme d'état (les états et les événements qui interviennent dans leurs transitions sont précisés avant le diagramme proprement dit), le deuxième est constitué par la spécification d'un objet de type traitement.

Exemple de document édité

CHAPITRE I

RECALAGE PAR PASSAGE A LA VERTICALE

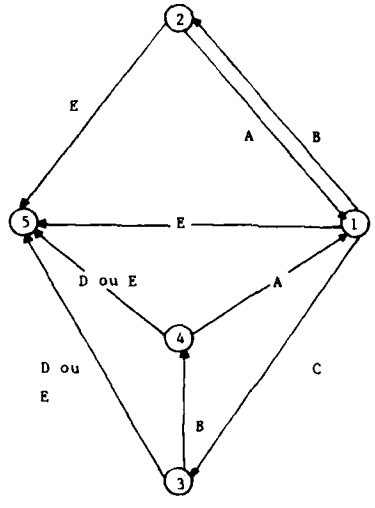
1.1. Etats possibles

Durant l'exécution de cette fonction, le système est dans l'un des cinq états définis ci-après :

1	Recalage par passage a la verticale avant designation du but avec des coordonnees but valides
2	Recalage par passage a la verticale avant designation du but avec des coordonnees but invalides
3	Recalage par passage a la verticale apres designation du but avec des coordonnees but valides
4	Recalage par passage a la verticale apres designation du but avec des coordonnees invalides
5	Recalage non selectionne

Les événements qui interviennent dans les transitions entre états sont les suivants :

A	Changement but avec coordonnees but valides
B	Changement but avec coordonnees but invalides
C	Premiere designation
D	Appui sur touche ANNULATION ou VALIDATION recalage
E	Appui sur touche VERTICAL



TRANSITIONS ENTRE ETATS RECALAGE VERTICAL

1.2. Nature des traitements

TRAITEMENT recalage par passage a la verticale.

Ses principales caractéristiques sont :

- synonyme : recalage vertical
- description : l'equipage selectionne..... erreurs de recalage
- informations en entrée : but de preparation, position avion, coordonnees du but, touche but additionnel
- informations en sortie : type de recalage, ordre de sortie de recalage, erreurs, voyant S sous la touche VERTICAL

DYNAMIQUE

Si le système est dans l'état recalage par passage a la verticale avant designation du but avec des coordonnees but valides

Alors le logiciel effectue les actions suivantes :

1. type de recalage : = vertical ;
2. voyant S sous la touche VERTICAL : = allume ;
3. quand apparaît l'evenement premiere designation
   alors le logiciel elabore l'information erreurs a partir de position avion coordonnees du but ;
4. quand apparaît l'événement appui sur touche ANNULATION ou VALIDATION recalage
   alors le système passe dans l'état recalage non selectionne.

...

IV - LIAISON AVEC LA CONCEPTION

L'E.S.D. développe également un système d'aide à la conception des logiciels. Ce système est le support d'une méthode dérivée de celle des machines abstraites [GAL 78] en prenant en compte un environnement temps-réel. Il est donc indispensable de lier les deux systèmes, cela est fait actuellement grâce à des relations définies manuellement par l'utilisateur entre les objets manipulés dans la spécification et ceux manipulés dans la conception. Grâce à ces liaisons, on obtient une traçabilité minimum entre les deux étapes, qui peut être intéressante notamment en cas de modification d'une partie des spécifications. En effet, dans ce cas, le système fournit l'impact de la mise à jour, et permet à l'utilisateur de connaître les éléments de conception qui sont susceptibles d'évoluer.
Une étude est en cours pour analyser la faisabilité d'une automatisation de la mise en place de ces relations, qui permettrait ainsi une intégration totale entre les deux systèmes.

V - CONCLUSION

Le système que nous venons de décrire apportera une aide significative à ses utilisateurs pour l'analyse des problèmes de définition du logiciel. Il permettra un accroissement de la qualité des documents produits ainsi qu'une facilité et une sûreté de mise à jour. Il rendra disponibles et fiables les informations relatives à la spécification et facilitera leur communication.

La définition du système est maintenant terminée, la phase de réalisation est en cours. Le système sera intégré à l'atelier de génie logiciel AIGLE en cours d'étude (projet commun aux sociétés ESD, SNIAS/DSBS, STERIA [BRA 82]).

BIBLIOGRAPHIE

[ALF 76] M.W. ALFORD, I.F. BURNS
"R-nets : A graph model for real-time software requirements"
Symposium on Computer Software Engineering.
Polytechnic - Institute of New-York
1976

[ALF 80] M.W. ALFORD
"Software requirements engineering methodology (SREM) at the age of four"
Proceedings COMPSAC 80
1980

[BOE 78] B.W. BOEHM
Software Engineering
Ecole d'été informatique IRIA, EDF, CEA. PARIS
Juillet 1978

[BOE 82] B.W. BOEHM
"Les facteurs du coût du logiciel"
Technique et science Informatique n°1
Janvier 1982

[BOU 78] S.R. BOURNE
"The Unix Shell"
Bell System Technical Journal Vol. 57 N° 6 part. 2
Juillet-Août 1978

[BRA 82] G. BRACON
"Présentation de l'atelier AIGLE"
Actes des J.I.I.A. - Paris (à paraître)
Juin 1982

[CHE 80] S. CHENUT, J.M. NERSON
"Le système ZAIDE : conception générale"
Note EDF HI/3468-01
Juin 1980

[GAL 78] M. GALINIER , A. MATHIS
"La machine abstraite comme unité de conception, implantation et modification de programmes"
Ecole d'Eté Informatique CEA, IRIA, EDF
Juillet 1978

[MEY 79] B. MEYER
"Sur le formalisme dans les spécifications"
Atelier logiciel n° 23 - Note EDF HI/3206-01
Août 1979

[SOF 76] SOFTECH Inc.
"An introduction to SADT : Structured Analysis and Design Technique"
Document SOFTECH n° 9022 - 78 R
Novembre 1976

[TEI 77] D. TEICHROEW, E.A. HERSHEY III
"Computer - aided technique for structured documentation and analysis of information processing systems"
IEEE Transactions on Software Engineering - Vol SE 3 N° 1
Janvier 1977

THE MENTOR APPROACH TO REQUIREMENTS SPECIFICATION

D. Jordan, B. Hauxwell,
Marconi Avionics Ltd.,
Elstree Way,
Borehamwood,
Herts.
England.

SUMMARY

Requirements Specification methodologies and documentation systems which are currently available adopt a range of differing viewpoints on the system development problem. Each is oriented to enabling the user to specify efficiently some particular aspect of the target system, and each places a particular emphasis on certain features of the system.

At Marconi Avionics we believe that the role of an automated documentation system should be to accommodate all information relevant to a system design, to check rigorously that information for consistency, and to make it visible in a range of reports which individually provide various emphasis on the information.

To this end we are developing an integrated system, MENTOR, for the development of engineering designs and documentation. MENTOR is intended to assist development teams in the gathering of information, definition of terms, and development of designs for avionics systems. The system has the following major features:-

a) A new specification method which guides the user in the decomposition of his problem by partitioning the behaviour of the target system among a number of operational capabilities.

b) A specification language NATTER (A NATural TERminology) which is intended for use by any personnel involved in the system development and is sufficiently flexible to accommodate all functional documentation for the target system.

c) An advanced algebraic analysis technique which enables the MENTOR system to perform a powerful dynamic consistency analysis of the evolving specification, and provide valuable feedback to the development teams.

1. INTRODUCTION

In recent years a wide variety of techniques have been proposed in an attempt to inject a degree of methodological rigour into the systems engineering process. These proposals have ranged from the rather esoteric, formal specification and program transformation techniques (Jon 1980, Ham 1976), to the pragmatic, documentation oriented approach of a number of semi-formal documentation methodologies(Tho 1976, Jac 1975) and Engineering Design Documentation systems (Tei 1976, Dav 1977). Each of these schools can offer some valuable assistance in the generation of effective system designs, however the more formal techniques in general have little direct relevance to the day to day engineering process, and will, we believe, prove difficult or impossible to integrate into the procedures involved in the development of large systems in a commercial environment.

At Marconi Avionics we have been investigating the relationship between formal and semi-formal techniques in some detail, and we have concluded that a broadly based approach is feasible. In outline, this approach would allow for development and evolution of system concepts in a structured documentation-oriented fashion, but would nevertheless provide a degree of formal verification sufficient to give confidence in the correctness of the designs which are achieved. We are currently engaged in the development of an Engineering Design Documentation System, MENTOR which embodies this approach to the specification and verification of systems.

At its core MENTOR is a database system for the maintenance of engineering documentation in much the same spirit as ISDOS (Tei 1976) and SDS (Dav 1977). MENTOR, however, supplements this information management capability with:-

a) A method for the top-down evolution and structuring of system specifications, with emphasis on the separation of concerns in the analysis of required behaviour and the specification of operational design.

b) A semantically processed specification language, NATTER, which captures the specification, in detail, for representation on the MENTOR database, and

c) A new and powerful technique for consistency analysis, based on a formal interpretation of the NATTER language primitives, and implementing a concept of logical flow, developed from the recently described theory of flow algebra (Mil 1978).

In addition MENTOR will provide a comprehensive report generation package for the retrieval of information from the database, and will provide facilities for security management, and completeness monitoring.

The completeness monitoring facility will be linked to the consistency checking mechanism, and whenever insufficient information is available to complete the structures anticipated by the consistency checking process, so that a consistency check cannot be completed, the completeness monitoring facility will prompt the user to provide the missing information.

It is however, the central language analysis and verification capabilities of the MENTOR system, together with the MENTOR specification technique which form the substance of this paper. The next section comprises a discussion of the capabilities required in the area of language analysis, and an introduction to the technique of language analysis which we have adopted in the MENTOR system. Subsequent sections treat the MENTOR specification technique, and the verification of systems by flow logical analysis similarly.

2. THE NATTER SPECIFICATION LANGUAGE

In order to discuss meaningfully the capabilities of a specification language, it is necessary to adopt some position on the nature and role of the specifications which are to be prepared using the language. One view which is widely held, is that a full functional requirements specification, stating in detail the behaviour to be implemented by the system, should be provided. Furthermore this view holds that the requirements specification should be completely neutral, to the extent that no feasible design should be excluded.

While we have a certain amount of sympathy with this position,we feel that it is overstated, with regard to the practicalities of the systems engineering discipline. To put it baldly, we believe that there is a wide gulf between the products of engineering design on the one hand, and those algorithmic problems which can be precisely specified in a formal, design free manner, on the other.

Looking at this from another angle, we would say that the elucidation of detailed system requirements is a central part of the engineering process, and that the generation and evaluation of designs is an essential technique in this process. We therefore take the view that the specification language is essentially a design documentation tool. We do however recognise that the documentation of design characteristics must be correlated to the requirements of the application, so that the language must be sufficiently powerful to support a detailed external view of the behaviour offered by the system.

Any terminology which is designed to provide this comprehensive capability will have a great deal in common with natural languages as they have been used for engineering documentation. If, however, as in the MENTOR system, it is required to provide automatic tools to facilitate the engineering process, we must confront the fact that the specification language must be accommodated to the system by some measure of formalisation. There is however a well known risk in the formalisation of natural languages, in that it can be difficult to confine usage of familiar natural language terms within the limits of a rigid formal syntax.

Nevertheless, in view of our overall approach, we believe that it is imperative to retain much of the vocabulary and some of the style of natural English in the specification language for MENTOR. Therefore we have specified the use of semantic techniques of language analysis in order to retain also some of the flexibility which characterises the natural language. Thus the MENTOR specification language NATTER (a NATural TERminology) does not have a rigid defined syntax, and any statement is regarded as syntactically valid, provided that it can be reduced to an appropriate set of semantic relationships.

Broadly speaking this is possible only because NATTER is restricted in its expressive power to be a terminology for describing the behaviour and operation of systems of processing capabilities. Only a limited number of types of object need to be recognised, and only a limited range of interactions and interconnections between documentation objects require to be specified. Furthermore, strict naming conventions for documentation objects can be adopted in order to facilitate the processing of noun phrases.

Nevertheless the use of semantic techniques lends to NATTER the flexibility to be used in a natural way for the preparation of engineering documentation. While retaining the capability to generate a fully detailed model of the meaning of the documentation to be maintained on the system database.

The output of the NATTER language analyser is called a second order entity/relationship model of the meaning of the input statements. Noun phrases in the input statements are processed to generate unique names for a set of named entity nodes, and these nodes are interconnected by a network of relationships which represent the verbs and other connective phrases in the input statements.

Both the entities and the relationships which form the primitive components of the entity relationship model have associated type attributes, and the semantics of the NATTER language are defined as constraints on the association of entities with relationships depending on their type attributes. Relationships are only permitted to be specified between objects of appropriate types, and the appropriate types for each relationship are defined by means of a schema which is associated with the relationship type. The semantic processing of NATTER therefore consists of the use of the relationship schemata associated with the connective phrases in the input statement, to derive a consistent allocation of the noun phrases to appropriate object types.

In principle this scheme can accommodate the processing of arbitrary input statements to generate an equivalent entity/relationship model, however its use with NATTER is restricted to provide only sufficient capability for an engineering design terminology. Nevertheless, in order to accommodate both the specification of designs and the explanation of those designs in terms of the behaviour which they implement, this capability extends to the ability to process statements of causation, inhibition and consequence which, in many cases, must be modelled as relationships acting upon other relationships.

It is this feature, that some relationships act upon others, which is referred to as the second order characteristic of the entity/relationship model. We have however found that in all such cases it is appropriate to regard one or other of the entities participating in the acted upon relationship as mediating the action. This means that the second order characteristic can be conveniently handled in MENTOR by a technique of parameterisation. Consider, for example the relationships

A derives B; and, C causes A to derive B.

In this case the action implied by A derives B is under the control of the process entity A, therefore we model these relationships in MENTOR as

A(N) derives B; and, C causes A(N)

Where N uniquely identifies the relationship A derives B among all the relationships in which A participates.

3. THE MENTOR SPECIFICATION METHOD

We have supplemented the powerful NATTER language analysis capability of the MENTOR system with detailed guidelines for the structuring of system specifications. These guidelines comprise the MENTOR specification method, and amount to a top-down, abstract machine approach to specification generating very strongly structured engineering designs. The central idea in this specification technique is to separate the *concerns of how* the system is required to behave, and how it will operate in order to achieve that behaviour, in terms of the independent behaviours of a number of contributory processes and the control flows amongst those processes. In order to achieve this separation of concerns, without prejudicing the further detailed analysis of the requirements of the application, we recommend that a certain attitude be adopted towards the role of a process in engineering design. A process is to be regarded as adding to the system the capability to take account of some natural subdivision of the interactions in the system's environment. It has as a central concern the responsibility of maintaining some set of entities in an appropriate relationship to be a meaningful model of some aspect of the activities which the system is required to control.

Adopting this approach means that, typically, each function provided by the system requires the collaboration of several processes. Conversely each process may be required to perform a number of different actions in support of various functions. This means that it is necessary to specify precisely the circumstances in which each action is performed, so that each process has its own behavioural specification in terms of conditions recognised and actions performed in response to those conditions. This amounts to an explanation of the role of the process in the management of the systems operating environment.

Operational processing sequences among processes can be specified by indicating the consequence of particular actions as conditions becoming true, which then act as the stimulus for subsequent actions. Additionally triggering actions may be specified between processes, wherever it is necessary to activate an idle process to perform some action.

We believe that this comprises a fully general approach to the specification of engineering designs, and that by employing this method in conjunction with the NATTER specification language, documentation of a high quality can be generated which parallels and quite closely resembles the best specifications in natural English.

The main purpose of the NATTER language, however is to make engineering specifications machine processable, and one of the results of the specification method outlined is the generation of specifications of sufficient detail to support an automatic consistency analysis.

4. VERIFICATION OF SPECIFICATIONS BY FLOW LOGIC ANALYSIS

The MENTOR system will not, in any sense, guarantee that a design is appropriate for a given application. To look for such guarantee would in fact be misguided, since there can be no rigorous way of determining what would be an appropriate response by the system in any circumstances which might in practice arise. The determination that the behaviour of a system fulfils the requirements of the application is a process that we call validation, and however the behaviour of the system is defined, validation is ultimately a matter for human judgement. There is, however, another aspect of the verification of systems which is amenable to automation. That is, establishing that the operations of the system do in fact conform to and achieve some specified behaviour. In other contexts this activity has been called program proof, but in the context of the MENTOR system it is more appropriate to regard it as a dynamic consistency check of the specification.

To appreciate why this is so, it is necessary to understand that the documentary form of specification supported by MENTOR is one that is designed to facilitate the validation process. There is at no point a full formal specification of the behaviour of the system, but instead a succession of abstractions of that behaviour which encompass and progressively delimit the application which the system will fulfil. In effect this refinement is achieved by incorporating specific assumptions about the intended application in the specifications of system operation. But because of the hierarchical nature of systems it is possible to make contradictory assumptions at different points, leading to inconsistencies between the operation of separate parts of the system. This is the sense in which the automatic verification technique in the MENTOR system is called *consistency checking*.

The technique which we have developed to perform this analysis is based on the concept that the instantaneous state of a system can be represented by the current truth assignments of a system of predicates (truth functions) which specify the states of individual entities in the system. Flow logic is a technique which enables us to model the evolution of truth assignments in such a system, dependent on the correlations between predicates implied by the relationships specified in the system description.

Meaning representations (i.e. representations of meaning) for relationships in flow logic specify these correlations between predicates, and are subject to algebraic manipulation conforming to a system of axioms known as the Laws of Flow (Mil 1978). These axioms have the effect that each relationship is perceived as having an immutable intrinsic meaning, while permitting the manipulation of meaning representations in such a way as to derive representations of the effect of relationships acting in concert.

In practice each meaning representation is equivalent to a number of interconnected states which correspond to partial states of the system considered. Each state of the meaning representation comprises a number of statements which in general describe the changes of state of the participating entities associated with changes of state of the system. In fact each statement has the general form h:A T where h, if it is present, denotes the hypothesis that the state of some participating entity changes, initiating a change of the partial state of the system. A, if it is present, denotes the argument that certain other state changes occur amongst the participating entities in consequence of the hypothesis, and T, which is always present, denotes the conclusion that the system evolves to a new partial state corresponding to a given transformation state of the meaning representation.

If the hypothesis is absent from a statement then the transformation is such that it may occur spontaneously, while the absence of an argument has no particular significance.

In some statements, the transformation T, instead of identifying a state of the meaning representation, may be set to the null value, Ø. Statements of this form effectively assert that the state change, h, cannot occur while the system is in the corresponding partial state, and that an attempt to cause h to occur in that state comprises a pitfall which is to be avoided. For this reason we pronounce the value Ø as "pit", and use statements of this type in such a way that if,and only if, a system of relationships is inconsistent then at some point a transformation to Ø must be realised.

The flow operations which we use to manipulate the meaning representations of flow logic are specifically designed to treat the transformation value Ø in such a way as to facilitate the identification of circumstances in which an illegal transformation to Ø cannot be avoided. In fact Ø is analogous in its treatment to an algebraic zero so that the meaning representation for an inconsistent family of relationships reduces to Ø. However the main benefit of this approach is to identify an intermediate level of latent inconsistency.

A latent inconsistency occurs whenever some partial state of the system is found to lead inexorably to a transformation to Ø, and must therefore be avoided. The significance of these states is to impart a direction to the process of consistency analysis, and their effect is to minimise the amount of processing required to establish that a proposed update is consistent with a known consistent database. It is therefore feasible with a suitably structured database to verify interactively the consistency of each statement input to the MENTOR system, and to maintain the system database at all times in a consistent state.

When an inconsistent update is proposed the MENTOR system will not update its database, but will print an appropriate message together with a trace of the flow analysis which detected the inconsistency. By expressing this trace in terms of the relationships considered, rather than the details of the flow logic, it will amount to an explanation of the specification error and will therefore facilitate the rectification of the problem

5. CONCLUDING REMARKS

The MENTOR approach to systems development is essentially a specification technique which relies heavily on the use of a computer tool to provide a range of valuable clerical and logical tasks. In developing the concepts for the MENTOR system we have drawn together a number of threads of contemporary research and, we believe, have indicated the direction in which systems engineering practices must evolve.

It is to be anticipated that as time progresses more and more sophisticated techniques will be added to the central core of methods outlined here. For example we can foresee the day when systems like MENTOR will have the capability to enter into dialogues, and to explain the purpose and design of systems in a more natural way than is currently possible. Indeed the catalogue of possible developments seems almost limitless.

Only time will tell which tools and techniques will prove cost effective in the long run. However tools and techniques they are, and they will do nothing to alter the fact that systems engineering is a specialist human activity. The human engineer is the most valuable resource in any systems development activity, his inventiveness is and will remain indispensable. Our objective is to provide a tool with sufficient specialised knowledge of the engineering task, to act as a guide, and mentor to systems development teams.

It will take a system of an entirely different order to achieve more than that.

REFERENCES

Jon 1980 — C.B. Jones "Software Developments: A Rigorous Approach"
Prentice Hall International, Eaglewood Cliffs N. J. 1980.

Ham 1976 — M. Hamilton, S. Zeldin. "Higher order Software - A Methodology for Defining Software" IEEE Trans. Software Eng. pp 9 - 32, March 1976

Tho 1976 — M. Thomas. "Functional Decomposition: SADT".
Infotech conf. on structured design. Amsterdam October 1976.

Jac 1975 — M. A. Jackson. "Principles of Program Design"
Academic press New York, 1975

Tei 1976 — D. Teichrow, E. Hershey. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems".
2nd Int. Conf. Software Eng. October 1976.

Dav 1977 — C.G. Davis + C.R. Vick. "The Software Development System".
IEEE Trans. Software Eng. pp 69 - 84. January 1977.

Mil 1978 — R. Milner "Flowgraphs and Flow Algebras"
Report CSR-5-77. Computer Science Dept. Edinburgh Univ. Revised December 1978.

The Computer Aided Specification System Easy

Lutz Hirschmann / Niels Christensen

mbp Mathematischer Beratungs-
und Programmierungsdienst GmbH
Se ierteichstr. 47
D-4600 Dortmund,
Federal Republic of Germany

Abstract

Easy is a formalism which support the description of the specification in a simple semiformal manner and emphasises the use of data types. It is our view, that a specification should not only be precise, unambigious and complete, but is also a document which serves for the communication between people (software engineers). We use the term specification to describe the result of the system design phase i.e. for the description of the decomposition of the system into modules and the interaction between these moduls.

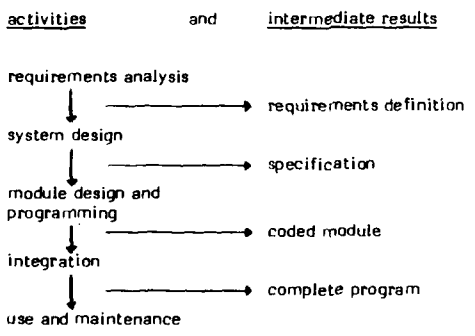
A specification written in Easy consists of packages which provide the encapsulation of logically related information, data and functions. Packages communicate by interfaces which consist of procedures, types and constants. Objects of the export-interface are resources implemented in the exporting package and are made available to other packages. Objects used from other packages are listed in the import-interface. Restrictions in the use of exported procedures must be mentioned in the description of the procedu.. or (preferably) in the paragraphs "sequence" and "error".

A software tool - Easy Tool - has been developed to support the use of Easy. It checks specifications for syntactical and semantical correctness, consistency and completeness, supports the editing, stores and maintains specifications in a data base and generates several crossreference lists. Future versions of the tool will support the transition from specification to implementation to make sure the program matches the specification.

Easy has been used in several industrial projects. Some of the experience is reported in the final chapter.

1. Requirements for a Specification System in an Industrial Environment

Since there is no general agreement about the term "specification" it seems advisable to start a paper about a specification system with our definition of this term. A possible, though very rough software development life cycle would contain the following:



Specification in this sense denotes the result of the activity "system design", i.e. the decomposition of the whole system into modules and the definition of the interactions between them. The specification is the starting point for the programming phase and the basis for integration.

A specification language is a formalism to describe the specification.

In the field of programming languages a certain consensus has been reached. Current research work in language design is directed towards the development of formalisms in which the other results (not programs) of the software development life cycle can be expressed. At least for system and real time software development we consider this approach as being superior to the attempt to formalize the activities of the software life cycle e.g. programming, as is described in /Jack 75/.

Specification languages are a topic of current research in computer science. However, improvements in software quality, development time and man-power effort as a result of their use in practical applications may already be expected.

A specification language should support the following concepts

- information hiding
- completeness of interfaces
- implementation independency

However, the crucial criterion for acceptance and usefulness of a specification language is its practicability in an industrial environment.

In detail, practicability imposes the following requirements on a specification language.

- It must offer significant support for real world problems. The degree of formality of a really useful specification language has to be determined carefully and must not prevent its applicability to complex problems, on the other hand completely informal specification techniques have been proved not to be useful. A perfect specification of a stack does not imply the applicability of the formalism to a project with 100,000s lines of code.

- It must be usable by dp industry professionals with average qualifications. Not all of them have recently completed their PhD in axiomatic semantics, thus the specification language must be easy to learn and use. Furthermore, it should be easy to read since communication among the project team is based to a large extent on the specification.

- It should improve the quality of the software but there must be a reasonable trade-off between the costs of writing the specification and the gains in later development phases and maintenance.

Experience has shown that recent research results, e.g. from Guttay /Gut 77/ (algebraic specification) or Parnas /Par 78/, are not applicable in practice, although we do consider them to be very important.

The current challenge to industrial software engineering is the application of new research results in a less formal but more practical manner. This approach is represented in our specification system **Easy**. Like most of the recent specification techniques, e.g. /Koch 79/, it is based on the principles of abstract machines and abstract data types.

2. Concepts of the Specification Language Easy

A specification in Easy consists of packages which provide the syntactic frame for the decomposition of the system. A package is a static construct to describe the modules obtained by decomposition of the whole system and encapsulates logically related information, data and functions. Easy packages are not nested.

The interaction between packages is defined by interfaces which consist of procedures, types and constants. Objects of the export interface are resources implemented within the exporting package and are made available to other packages. Objects used from other packages are listed in the import interface.

Procedures are defined by their name and their parameters. Each parameter has a name, a type and a binding mode (in, out, in-out) and controls the data flow between packages. Access to a package's data from outside can only be performed by using exported procedures, thus hiding the representation of local data as has become standard for abstract data types.

While the syntax of procedures is precisely formalised, no definite formalism is required for the semantics. The insistence on a formal operational or even on an axiomatic semantical description would not correspond to the required practicability of Easy. Therefore the effects of an exported procedure are normally described in "careful natural language" although Easy also permits other notations.

Since Easy supports the concept of abstract data types, the package interface also contains types. A package which imports procedures that access a data structure must also import the types of the accessed elements. Some types such as integer or character may be declared as default and then need not be passed through the interfaces. Types are specified by their name and an informal description.

The types of parameters of imported procedures and the types of imported constants must be imported by the package which uses them, so that it is clear from the specification which data structures are used within each package.

The import of a data type into a package allows the declaration and use of objects of this type within the implementation of this package. This also applies, of course, to exported types. As for procedures and constants, two types are equivalent if they have the same name.

Constants are defined in the interface description by their name and type.

Once the export-import relations have been specified, possible restrictions in the use of exported procedures have to be given. The Easy specification therefore contains two paragraphs called "sequence" and "error". The "sequence" paragraph describes the order in which exported procedures may be legally called, whereas the "error" paragraph lists general limitations, restrictions and the error handling routines of that package. As is the case for the description of the effect of exported procedures, no particular formalism is required for these two paragraphs, e.g. for the definition of sequence restrictions a graphical notation for path expressions could be used.

The following example is intended to give an impression of Easy. It is an artificial example, it is not complete, real world problems usually lead to much bigger package specifications and it is impossible to demonstrate all features of Easy with one example.

```
# Example for a package specification in Easy.

# Declared implicit types:
#   integer
#   string  =  character string of variable length

PACKAGE   Radiocomm
  VERSION 0
  FROM    01 June 82
  BY      C. Hort

TERMINOLOGIE
  msg:        logically connected string of information from one sender to one destination.                    ))

  msg-block:  physically connected string of information of fixed length, can be part of msg or may contain several
              msg's.                                                                                          ))

  disabled:   disabled communication indicates either hardware problems or restrictions due to jamming, radio silence
              or specific order.                                                                              ))

DESCRIPTION
    This package provides the service procedures offered by a simple radio communication utility. It deals with a
    single-layer network without priority scheme and fixed participants addressed by a destination code. There is an
    acknowledge/validate shakehand to ensure the arrival of messages.

    Queue-handling, channel-control, en/decoding and network command are supposed to be contained in separate
    packages.                                                                                                 ))

EXPORT
PROC  Xmit      ( IN    port-no:      home
                                      destination,
                  IN    string:       message,
                  OUT   result-Xmit:  result,
                  OUT   integer:      msg-id)
        DESCR:    Put a message in the sending queue; get result indicator and associated message-identification.
                                                                                                          ))

TYPE  result-Xmit
        DESCR:    possible states:
                  -   o.k.
                  -   communication disabled
                  -   destination port disabled
                  -   not ready
                  -   invalid home port
                  -   invalid destination port                                                            ))

PROC  validate  ( IN    port-no:      home,
                  IN    integer:      msg-id,
                  OUT   result-validate: answer)
        DESCR:    validate whether message has arrived and was acknowledged by addresses.                 ))

TYPE  result-validate
        DESCR:    possible states:
                  -   msg arrived & acknowledged yet
                  -   not arrived yet
                  -   not acknowledged yet
                  -   invalid home port
                  -   msg-id unknown or invalid
                  -   msg lost                                                                            ))

PROC  receive   ( IN    port-no:      home,
                  OUT   port-no:      from,
                  OUT   string:       message,
                  OUT   result-receive: result,
                  OUT   result-receive: msg-id)
        DESCR:    Receive first message from arrival queue or result = "no msg" if there is none. If the msg received
                  previously had not been acknowledged yet, it will be delivered again.                   ))

TYPE  result-receive
        DESCR:    possible states:
                  -   o.k.
                  -   no msg in queue
                  -   acknowledge missing - same msg again
                  -   disabled communication
                  -   invalid home port                                                                   ))

PROC  acknowledge   ( IN    port-no:        home,
                      IN    integer:        msg-id,
                      OUT   result-acknowl: result)
        DESCR:    Acknowledge (receipt) msg in order to remove it from arrival queue and allow validation for
                  sender.                                                                                 ))
```

```
TYPE  result-acknowl
      DESCR:  possible states:
              -   o.k.
              -   unknown msg-id
              -   invalid home port                                        ))

TYPE  msg-block
      DESCR:  Formatted and encoded information block ready to be xmitted across the channel.
              Length = msg-block-length;
              More layout description of msg-block ...                     ))

CONST integer:  msg-block-length,
                max-msg-id

IMPORT # # #  use abbreviated form of proc imports  # # #

FROM queuing:
  PROC    insert
  PROC    remove
  PROC    createqu
  PROC    deletequ
  PROC    qstatusof   (IN queue) RETURN qstatus
  TYPE    queue
  TYPE    message
  TYPE    receipt
  TYPE    qstatus
  TYPE    result-q

FROM channelctl:
  PROC    primary-ch          RETURN channel
  PROC    alternate-ch        RETURN channel
  PROC    activate-ch         ( IN channel, IN msg-block, ?)
  PROC    ch-status           ( OUT ch-quality, OUT jamming-level, ?)
  TYPE    channel
  TYPE    ch-quality
  TYPE    jamming-level

FROM cypher:
  PROC    encode
  PROC    decode
  PROC    setcode
  TYPE    code

FROM netcommand:
  PROC    current-net         RETURN net
  PROC    current-code        RETURN code
  PROC    silence             RETURN deg-of-silence
  TYPE    net
  TYPE    port-no
  CONST   net: basic-net
  CONST   integer: max-port-no
  TYPE    deg-of-silence
  CONST   jamming-level: max-jamming-level

SEQUENCE
      for sending:
          1.   Xmit          (.. out integer: msg-id ...)
          2.   (optional)
               validate      (.. in integer: msg-id,
                             (out result-validate ...)
               2. can be repeated until result-validate = "msg arrived yet"   ))

      for receiving:
          1.   receive       (.. out result-receive ...)
               repeat until result-receive = "o.k."
               then obligatory:
          2.   acknowledge                                                    ))

ERROR
      Use errors (= sequence/id/data errors) and/or transfer errors (= comm.disabled, channel malfunctions) are
      reported in the various result parameters.                             ))

ENDPACK radiocomm
```

For the specification of the interaction between packages the import-export relationship is adequate. However, this is not the case at the periphery of the system, i.e. between the system and its environment, for example dialogues and interrupts. This kind of communication is specified in Easy in a special paragraph called "external" which contains parameter specifications denoting input or output entities.

Easy provides adequate language constructs to express the use-relation (by the export-import interface) and the data flow between packages (by the binding mode of procedure parameters), but no language contructs are offered to define the control flow because control flow is considered an implementation rather than a specification matter.

Conforming to this philosophy, Easy does not (yet) contain any constructs for the specification of processes. It is thought that realtime applications are best specified by the encapsulation of data structures as recommended by the package concept. Generally, a wide choice of possible process definitions exists in such an application, and it seems inadvisable to restrict this choice prior to implementation. The strict observance of the abstract type principle automatically takes care of many common real time problems (e.g. shared data can only be manipulated by a dedicated handler). It is not even necessary to use visible semaphors.

Easy does not differentiate between different types of packages (e.g. function packages or data packages) and does not insist on a specific allocation of packages (e.g. strong hierarchy). Although it is recommended that the import-export relationships should form a hierarchy of abstraction levels, this is not a restriction. A wide applicability of Easy is thus ensured.

3. The Tool for Easy

To support the use of the specification language a tool has been developed which performs the following functions:

Aid for the Writing of the Specification

A dedicated editor serves to create and maintain syntactically correct specification texts.

To reduce the work of writing the specification, the types of parameters of imported procedures and the types of imported constants can be imported automatically. To import a procedure it is sufficient to mention the procedure name (parameters can be omitted).

In order to use the advantages of the specification system during the design process (in a phase where the specification is necessarily incomplete) certain parts of the specification can be left undefined.

For imported objects the name of the source package may be omitted and be inserted later by the tool. On the other hand all exported resources of a package can be imported simply by giving the package name and asking the tool for an automatic import. However, this means that precision and distinctness of the interfaces suffer considerably and it is advisable to use this facility with care.

Consistency Check

After a package has been specified and checked for syntactic and some semantic correctness it will be entered into an Easy data base. In the data base this package can be checked for semantic correctness against the packages, which have already been entered. This includes checks for ambiguities in names of exported objects, whether all imported objects are properly exported, and that no re-export (export of imported objects) has taken place.

Documentation Aid

Cross reference lists are generated for each package as well as for the whole system. With these lists questions like "which packages are affected by the alteration of a certain exported object?" can be answered easily. This is considered to be very important during development and maintenance of software since it is usually difficult to find out the effects of changes on the whole system.
Formatted source listings of the specification and aggregated lists are available. Later versions will generate a graphical overview of the whole system which will include the relationships between all packages.

Ensuring Integrity

A complete and correct specification still does not guarantee a correct implementation. The "classical" way to ensure the correctness of the implementation is its formal verification against the specification. But with real world systems of normal size, this is not feasible.

In any case, because of the informal semantic description in Easy, formal verification can not work. However, the Easy tool will offer a significant aid in achieving a correct implementation. The idea is to generate from the specification a "program frame" in the desired programming language, which ensures that the specified interfaces are obeyed in the implementation. Thus integrity of implementation and specification is preserved without losing the advantages of having different languages for both purposes. It is advisable to distinguish between specification and programming languages in particular for the following reasons:

- programming languages with their rich set of control structures define how something is done whereas specification languages provide only language constructs to express what has to be done and thus ensure the implementation independency of the specification.
- the package concept and abstract data types are very useful for the specification even if the chosen programming language does not contain these features.
- one standardised formalism can be used for the specification, no matter what programming language is used.

The mapping of the specification onto a program frame is relatively simple for modern programming languages like Ada. For each Easy package an Ada package head is generated. Further information like imports are given as an Ada comment. The mapping is more difficult in the case of older languages like Fortran or Assembler, where no user-

definable data types exist and where, with the exception of a few assemblers, there are no constructs to define the overall structure of the program.

Even when older, but still wide spread, languages are used a specification with recent techniques is of particular value. In the case of Fortran, Easy packages can be mapped onto subroutines and Easy procedures can be mapped onto entries of these subroutines. Passing of parameters between entries can be performed either by entry parameters or by common blocks. When common blocks are used, additional programming standards are necessary to avoid a loss of security caused by possible but illegal access to the common area.

Before implementation begins, the representation of each exported type has to be defined and will be passed by the tool to the importing packages. Then the generated program frame contains all the information needed by the programmer to implement a subroutine corresponding to an Easy package.

The example given previously in this paper would be supplemented as follows:

```
    .....
    .....

ERROR..
    .....
IMPLEMENTATION
    TYPE result-Xmit   =  INTEGER
                       1  =  o.k.
                       2  =  communication disabled
                       3  =  destination port disabled
                       4  =  not ready
                       5  =  invalid home port
                       6  =  invalid destination port
    TYPE msg-block     =  INTEGER (80)
    .....
    .....

ENDPACK radiocomm.
```

After the implementation parts of all other packages have been defined, the representations of all imported types of the package "radiocomm" are known and the following Fortran program frame can be generated by the tool:

```
      SUBROUTINE RADIOC
C
C     PACKAGE Radiocomm
C     VERSION  0
C     FROM     01 June 82
C     BY       C. Hort
C
C     DESCRIPTION
C       This package provides the service procedures offered by a simple radio communication utility. It deals with a
C       single-layer network without priority scheme and fixed participants addressed by a destination code. There is
C       an acknowledge/validate shakehand to ensure the arrival of messages.
C
C       Queue-handling, channel-control, en/decoding and network command are supposed to be contained in separate
C       packages.
C
C       The following types are used implicitly
C           integer    =  INTEGER
C           string     =  INTEGER(1)
C
C
        ENTRY XMIT (HOME, DESTIN, MESSAG, RESULT, MSGID)
C
C     DESCR:   Put a message in the sending queue; get result indicator and associated message-identification.
C
C       HOME     type  =  port-no      =  ...
C       DESTIN   type  =  port-no      =  ...
C       MESSAG   type  =  string       =  INTEGER(1)
C       RESULT   type  =  result-Xmit  =  INTEGER
C       MSGID    type  =  integert     =  INTEGER
C
        INTEGER MESSAG(1), RESULT, MSGID
        ...
```

With this information, a programmer should be able to start with the implementation of that subroutine and it is ensured that the interfaces of the implementation conform with the specification. Moreover the code is annotated by a lot of useful comments about identifiers and their data types which have been carried over from the specification and which are very often not present in ordinary Fortran programs.

Besides this semi-automatically programming aid the rather simple mechanism of merging the specification text into the program source has turned out to be useful. It enables quick "visual" validation of the integrity of code versus specification.

Easy strongly recommends that access to data structures should be specified using procedures which will be mapped onto procedures or entries of the implementation language. But for performance reasons it is not desirable to make a

procedure call for every data access. Future versions of the tool will therefore offer more flexibility by mapping procedures of the specification not only onto their implementation counterpart but also onto inline code or macros.

Development of the Tool

The specification of the tool itself has been carried out in Easy. Pascal was chosen as implementation language and the data base is mapped onto single key isam files, so the tool is quite portable.

4. Practical Experience with Easy

Easy was used for the first time in a multi-man-year project for the development of a railway control system. Although this is a somewhat different application area than avionics, the same software problems are believed to be encountered in this field. Thus our experence should be valid for avionic software as well.

Fortran had to be used as implementation language and a further requirement was that the system could easily be adapted to new track structures and new operation conditions. This could only be achieved with a clean specification.

Positive Experience

The acceptance of Easy in the project team was good. The reasons were partly the enthusiasm of the team members for new techniques, but mainly the understanding, that something has to be done for the specification phase and that this could very well be Easy.

Every team member had a good overview of the current state of the design since this was well documented in the specification. It was considered valuable (not only by the project management) to have the result of the system design phase in a checkable form on which design reviews could be based. Thus the long "blind fliging" between problem analysis and programming could be avoided. The requirement of clean and complete interface description imposed by Easy revealed some design errors at an early stage when they could still be easily corrected. This should be considered as a significant contribution to improve the quality of the final product and conforms to the demands of quality assurance which require each intermediate product of software development to be a subject of review.

The standardisation of the specification achieved by the formal definition of the specification language improved its readability for team members other than the author.

The precision of the semantic description of packages, procedures and types was much improved after a glossary had been defined. An optional paragraph for each package, called "terminology", includes the glossary in the specification. This glossary was not part of the first version of Easy.

It is difficult to say what or how much has been saved in software development costs by applying Easy since of course one could not carry out the same project in parallel using "traditional methods". Compared to our initial estimates, the specification caused more effort but the programming proved to be less expensive.

Big savings are expected during maintenance. If maintenance costs can be reduced by 50%, which seems to be realistic, then we will have gained within 2 years after delivery of the system as much as we paid for system design and specification in that project. Current experience shows that these optimistic expectations have been met. This is mainly a result of the data encapsulation imposed by the Easy package concept.

Problems

When applying the specification language for the first time, a change of thinking habits was often necessary since Easy requires a very systematic approach. Sometimes it was difficult to distinguish between the logical concept of package and the physical concept of a load module. Therefore on introductory training course on the principles of Easy was required. This problem can be compared with the problem when introducing new programming languages, e.g. Ada. "It is not the goal to write COBOL-programs in Ada syntax but to write Ada programs."

The work of writing down all the details was sometimes considered as quite arduous in particular when the interfaces were "trivial". On the other hand, the users often attempted to express more features (in particular the control flow) in Easy than the specification language has been designed for. To apply Easy effectivly, the programmers have to be told what should be expressed in Easy and what not. Easy is not intended to be the universal notation for all aspects of system design and specification.

In some cases, the transition from the complete specification to the program was still difficult, in particular when specification and programming were carried out by different people. An Easy package provides sufficient information to make use of its resources but not always to implement it. Since the person who carried out the specification of a package has usually some ideals on how to implement it, she/he should include these ideas in the description part.

Final Remarks

A specification language is not a design method. A specification language can support heuristic recommendations like those from Parnas /Par 72/ or Myers /My75/, but cannot be expected to replace a designer's creativity. Therefore Easy does not contain a construct for refinements of packages (although a design by refinement of packages might be a good method) and gives no guidelines in determining the optimal size of a package. If packages are too big, it often happens that errors in the design are overlooked since too much is hidden in one package. If packages are too small, the interfaces can become so bulky and complex that everybody loses the overview of the system.

Further development of Easy will be concerned with the semantic description of procedures and types and the specification of parallel processes. However it is difficult to find a formalism for these purposes which is precise as well as simple to use in practice.

Another possibility is the extension of Easy to a full software engineering environment. Like the transition from the specification to the program, Easy could support the transition from the requirements definition to the specification by checking whether the requirements, for which an adequate notation has to be developed, are met by the specification.

At present, we live in the age of "software engineering environment enthusiasm" (se^3), because these environments are expected to improve software productivity. But attention should be paid to the implications for the people involved in the production of software.

Bibliography

Gut 77: Guttag, J.V.:
Abstract Data Types and the Development of Data Structures
CACM 20/6, S. 396-404, 1977.

Jack 75: Jackson, M. A.:
Principles of Program Design,
Academic Press, London, 1975

Koch 79: Koch, W.:
SPEZI - eine Sprache zur Formulierung von Spezifikationen
TU Berlin, FB Informatik, Bericht Nr. 79-22, 1979.

My 75: Myers, G.J.:
Reliable Software through Composite Design
Petrocelli/Charter, New York 1975.

Par 72: Parnas, D.L.:
On the Criteria to be used in Decomposing Systems
into Modules, CACM 15/12, S. 1053-1058, 1972.

Par 78: Bartussek, W., Parnas, D.L.:
Using Assertions about Traces to write abstract
Specifications for Software Modules,
Proceedings on Information Systems Methodology,
Springer-Verlag, 1978.

Easy Language Definition

Version 3.2, dated 14.04.1982

Contents

0. Introduction

This is a description of the SYNTAX and SEMANTICS of the specification language **Easy**. Easy is basically a formalism for describing the results of the system design phase of the software production process.

Important features of Easy are

- constructs for modularisation,
- formulation of abstract data types,
- no constructs for expressing algorithms and data representation.

The SYNTAX of Easy is presented in a context-free van Wijngaarden notation:

- each sentence is terminated by period '.'
- alternatives are seperated by semi-colon ';'
- elements of a sequence are separated by comma ','.

The SEMANTICS is described in prose.

The following hyper-rules are used:

ELEMENT OPTION: ELEMENT;.

ELEMENT SEQUENCE:
ELEMENT;
ELEMENT, ELEMENT SEQUENCE.

ELEMENT ENUMERATION:
ELEMENT;
ELEMENT, ⊥, ELEMENT ENUMERATION.

Terminal symbols and keywords are underlined.

1. Global Structure

SPECIFICATION-PROGRAM:
SPECIFICATION-PROG-HEADER,
PACKAGE-SPECIFICATION SEQUENCE,
SPECIFICATION-PROG-END.

All the packages of a specification program are termed a "system of fellow packages". The textual order of the package specifications is of no consequence.

SPECIFICATION-PROG-HEADER:
SYSTEM-DESIGNATION,
IMPLICIT-TYPES,
GLOSSARY OPTION,
DESCRIPTION OPTION.

SYSTEM-DESIGNATION: SYSTEM, NAME.

IMPLICIT-TYPES:
DEFAULT-TYPES,
(TYPE-DESIGNATION SEQUENCE;
NONE).

TYPE-DESIGNATION: NAME, DESCRIPTION OPTION.

Types specified with NAME are regarded in all packages as implicitly imported (e.g. INTEGER, BOOLEAN).

Specifying NONE excludes the usage of implicit types.

SPECIFICATION-PROG-END: ENDSYSTEM, NAME.

Name must be the same as the name given in SPECIFICATION-PROG-HEADER.

PACKAGE-SPECIFICATION: PACKAGE-HEADER,
(EXPORT-SECTION, IMPORT-SECTION;
IMPORT-SECTION, EXPORT-SECTION),
EXTERNAL-SECTION OPTION,
SEQUENCE-SECTION OPTION,
ERROR-SECTION OPTION,
PACKAGE-END;

PACKAGE-NAME,
DESCRIPTION OPTION.

Using PACKAGE-NAME means that the package is described elsewhere but is part of the current system too ("external package definition").

2. Package Header

PACKAGE HEADER: PACKAGE-NAME,
VERSION SEQUENCE,
GLOSSARY OPTION,
DESCRIPTION OPTION.

PACKAGE-NAME: PACKAGE, NAME.

NAME must be unique within the system of fellow packages.

VERSION: VERSION, SHORT-TEXT,
FROM, SHORT-TEXT,
BY, SHORT-TEXT.

After FROM the date of the current version should be stipulated, likewise after BY the name of the person responsible.

GLOSSARY: TERMINOLOGY, DICTIONARY.

DICTIONARY: ELUCIDATION SEQUENCE.

ELUCIDATION: DEFINIENDUM, :, DEFINIENS.

DEFINIENDUM: NAME.

DEFINIENS: TEXT.

Important concepts and expressions (names, abbreviations etc.) that occur in the description of the package and/or system may be enumerated in the DICTIONARY.

The terms mentioned in the dictionary of a package may not coincide with terms taken from the system dictionary.

DESCRIPTION: DESCRIPTION, TEXT;
DESCR, TEXT.

3. Package End

PACKAGE-END: ENDPACK, NAME.

Name must be the same as the name given in the PACKAGE-HEADER.

4. Export Section

EXPORT-SECTION:

EXPORT,
PACKAGE-NAME OPTION,
(EXPORT-SPECIFICATION SEQUENCE;
NONE).

In the EXPORT-SECTION those objects are declared that are to be made available to other packages.

All objects exported from a system of fellow packages are termed visible objects. The names of all visible objects must be unique within their category (e.g. as type, constant etc.). Re-exportation, i.e. exportation of objects that have been imported by the current package, is not allowed.

"NONE" means that the current package does not make any objects available for external use.

PACKAGE-NAME:

NAME.

In order to increase the legibility of the specification the name of the package may be repeated here.

EXPORT-SPECIFICATION:

EXP-PROCEDURE;
EXP-TYPE;
EXP-CONSTANT.

EXP-PROCEDURE:

PROCEDURE-DEFINITION,
DESCRIPTION OPTION.

PROCEDURE-DEFINITION:

PROC, NAME,
PARAMETER-LIST OPTION,
FUNCTION-VALUE OPTION,
CONTROL-CLAUSE OPTION.

PARAMETER-LIST:

(, PARAMETER-SPEC,).

PARAMETER-SPEC:

PARAMETER ENUMERATION;
?;
PARAMETER ENUMERATION, , , ?.

The parameters of exported procedures serve to define the flow of data between packages. The mechanism for passing the parameters is not included in the language definition.

A "?" indicates that (all) parameters have not yet been defined.

PARAMETER:

BINDING-MODE,
PARAMETER-TYPE,
(: , NAME SEQUENCE ;).

All names occuring in a parameter list must be unique within the procedure.

Unnamed parameters may be used, providing

a) only one parameter of this type occurs or

b) all parameters are unnamed. In this case assignment depends on the order in which the parameters are specified.

BINDING-MODE:

IN;
OUT;
IN-OUT.

PARAMETER-TYPE:

NAME.

PARAMETER-TYPE must be a visible type, i.e. either exported by the current package or exported by another package and imported by the current package.

FUNCTION-VALUE:

RETURN, FUNCTION-TYPE.

FUNCTION-TYPE:

NAME.

FUNCTION-TYPE must be a visible type. If the function-value has been specified, the procedure, having been called, returns a value of the specified type.

CONTROL-CLAUSE: CONTROL,
(MAIN; EXTERNAL; HARDWARE).

CONTROL specifies whether the procedure is to be activated as a main program, from outside the system or by a hardware mechanism. If the CONTROL clause is not specified, activation from within the system is assumed.

EXP-TYPE: TYPE-DEFINITION,
DESCRIPTION.

TYPE-DEFINITION: TYPE, NAME.

Exportation of a type means that data structure defined in the current package and designated with NAME will be made available to other packages. Provided suitable access procedures are also exported access to these data structures are possible. Direct access to the data structure is not allowed (data abstraction).

EXP-CONSTANT: CONST,
CONST-TYPE, : ,
NAME ENUMERATION,
DESCRIPTION OPTION.

CONST-TYPE: NAME.

The type of a constant must be visible. The value should be given in the description.

5. Import Section

IMPORT-SECTION: IMPORT, PACKAGE-NAME OPTION,
(IMPORT-SPECIFICATION SEQUENCE;
NONE).

Imported objects are resources that may be used by the current package. Each imported object is exported by only one package within a system of fellow package.

"Name equivalence" is valid for all objects pertaining to the same interface, i.e. two objects are equivalent if they have the same name.

IMPORT-SPECIFICATION: ORIGINATING-PACKAGE,
(IMPORT-OBJECT SEQUENCE; ?).

"?" specifies that the imported objects are still unknown and will be defined later.

ORIGINATING-PACKAGE: FROM, (NAME, : ; ?, :).

NAME must refer to a package within a system of fellow packages. "?" specifies that the originating package is still unknown and will be defined later.

IMPORT-OBJECT: IMP-PROCEDURE;
IMP-TYPE;
IMP-CONSTANT.

IMP-PROCEDURE: PROCEDURE-DEFINITION,
DESCRIPTION OPTION.

The PARAMETER-LIST and the FUNCTION-VALUE may be omitted for imported procedures, no CONTROL-CLAUSE must be specified here.

IMP-TYPE: TYPE-DEFINITON,
DESCRIPTION OPTION.

IMP-CONSTANT: CONST, NAME ENUMERATION,
DESCRIPTION OPTION.

The following objects are implicitly imported:

- those types that are specified as implicit in the SPECIFICATION-PROG-HEADER;
- the types of imported constants, provided these types have not been explicitly imported;
- the types of parameters of imported procedures, provided these types have not been explicitly imported.

6. External Section

EXTERNAL-SECTION: EXTERNAL, PACKAGE-NAME OPTION, PARAMETER-SPEC SEQUENCE.

The declarations in the EXTERNAL section describe the flow of information at the "periphery" of the system, i.e. between the packages and the external environment. This serves as a description of I/O and interrupts.

7. Sequence Section

SEQUENCE-SECTION: SEQUENCE, DESCRIPTION.

The SEQUENCE-SECTION contains sequencing conditions relating to the use of exported procedures.

8. Error Section

ERROR-SECTION: ERROR, DESCRIPTION.

The ERROR-SECTION contains information relating to restrictions, limitations and error handling for the package.

9. Terminal Symbols I

NAME: Character string containing alphanumeric symbols (except for blank) and starting with an alphabetic symbol. Hyphens are not significant.

SHORT-TEXT: A sequence of any symbols terminated by a line feed.

TEXT: A sequence of any symbols, including line feed, terminated by double closing brackets: ....)).

The description should characterize the relevant object precisely without however unnecessarily restricting its implementation.

The description may take one of the following forms:
- formulation based on colloquial speech,
- a notation using axiomatic or operational semantics,
- examples of implementation,
- graphics.

COMMENTS: Text between "#" and the end of the line is regarded as comment. "#" within a description does not open a comment.

10. Terminal Symbols II

A colon ":" preceeded by one of the following symbols, either directly or separated by one or more blanks, is ignored, i.e. skipped over.

BY
CONST
CONTROL
DEFAULT-TYPES
DESCRIPTION
DESCR
ENDPACK
ERROR
EXPORT
EXTERNAL
FROM
HARDWARE
IMPORT
IN
IN-OUT
MAIN
NONE
OUT
PACKAGE
PROC
RETURN
SEQUENCE
TERMINOLOGY
TYPE
VERSION

The following special symbols are used:

(
)
))
?
,
;
#

DISCUSSION FROM AVIONICS PANEL FALL 1982 MEETING ON
SOFTWARE FOR AVIONICS

Session 2 : SOFTWARE AND SYSTEM REQUIREMENT ANALYSIS - Chmn Dr. Ing. L. Crovella (IT)

Paper Nr. 7 - REQUIREMENTS DECOMPOSITION AND OTHER MYTHS
Presented by - Dr. T. G. Swann
Speaker - Dr. L. Crovella
Comment - You seem to be saying that formal methods are of very little value. Do you really mean this?

Response - No! Formal methods are valuable, and with today's large systems they are becoming more and more essential. But, they are only one part of the design process - the tip of the iceberg.
We so rarely know what we really want, that by the time we are able to write this down formally, the design work has nearly all been done.

Paper Nr. 7 - REQUIREMENTS DECOMPOSITION AND OTHER MYTHS
Presented by - Dr. T. G. Swann
Speaker - D. Weiss
Comment - What would a mathematician think of your desirable specifications?

Response - We think of mathematics as producing formal equations, proofs and so on. But a vital part of mathematics is to describe what is going on, what assumptions have been made, what conclusions can be drawn. So if a mathematician proves that $E=Mc^2$, say, we would expect a whole book of text to describe what is meant by the equation. Without the book we cannot make use of the mathematics.

Paper Nr. 8 - PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES
Presented by - C. P. Price
Speaker - Dr. T. G. Swann
Comment - You use PSL/PSA to hold and analyse requirements. Do you find the language and toolset restrictive?

Response - The mapping of PSL to CORE has presented few problems and the comprehensive PSA reports have fulfilled the majority of our needs so far. This is not to say that PSL/PSA could not be improved. The ISDOS project is responsive to sponsors' needs and adopts a policy of continual update.

Paper Nr. 8 - PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES
Presented by - C. P. Price
Speaker - K. Pulford
Comment - There are often formal models or paradigms which form the basis of many methods. Is there one in SAFRA and what sort of model do you pass on to your engineers?

Response - As SAFRA embraces the complete life cycle there are several such models, but for requirements in particular there are two.
The problem statement language (PSL), employs the Entity-Relationship-Attribute model for system description although to some extent this is not visible to the engineer as he works with the core diagrams.
The core model defines processes as being connected via tightly or loosely coupled data relationships (Threads and associated Threads). The engineer using this model, examines the behavior of the requirement in sequential terms (Thread Diagrams) and also for concurrency (Operational Diagram).
There are other aspects to the model but time precludes them being discussed here.

Paper Nr. 8 - PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES
Presented by - C. P. Price
Speaker - J. B. Clary
Comment - Could you please describe the "CORE" work station more fully?

Response - The CORE work station is aimed at providing a single workplace for all phases of software development, including system and software requirements and up to system integration. This is being achieved by integrating a new facility with an existing proprietary software development system.

The new facility will have the following features;

1. Construction, storage and editing of all CORE diagrams
2. Automatic generation of PSL from the diagrams
3. Code generation from basic diagrammatic constructs

The first feature already exists and the code generation while initially aimed at CORAL is likely to include PASCAL.

Paper Nr. 8 - PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES
Presented by - C. P. Price
Speaker - Dr. D. J. Martin
Comment - How compatible is the CORE diagram notation with the currently well understood control law diagram notation (i.e. blocks containing LaPlace transform filter descriptions, diagrams of non-linearities and summation points)?

Response - Control law diagram notation is applicable to a specific area and level of systems design. CORE adopts a notation applicable to all levels of system and software specification and is independent of the type of system being described. More importantly the adopted CORE notation contains features that are compatible with aspects of MASCOT diagram notation thus allowing CORE requirement diagrams to be mapped directly into MASCOT design diagrams. In essence the notations are complimentary rather than being compatible.

Paper Nr. 8 - PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES
Presented by - C. P. Price
Speaker - Dr. L. Crovella
Comment - On how many projects has SAFRA been used at B.Ae.-Warton?

Response - Over the past three years SAFRA has been applied to six projects and is currently being used on a further three, one of which is a major aerospace project.

Paper Nr. 8 - PRACTICAL CONSIDERATIONS IN THE INTRODUCTION OF REQUIREMENTS ANALYSIS TECHNIQUES
Presented by - C. P. Price
Speaker - Dr. A. A. Callaway
Comment - A lot of the emphasis you have given today is concerned with production of software, software specifications, and a couple of times you even mentioned the automatic generation of actual software. Of course today there is a lot more attention being paid in everything we read to items such as higher integration, the US VHSIC program, the Japanese 5th generation project, and similar programs in Europe, where there is going to be greater and greater degrees of integration. We are going to be approaching putting whole systems onto single silicon chips. It seems to me that such a technique as this might also have great potential in acting as a top level design tool to get us down to a standard hardware description language level, rather than a standard software description level. This could then be used as an interface to some computer aided design suite for the silicon. Would you care to comment on that?

Response - I think what we are trying to achieve in CORE, is as you say. We have to be able to demonstrate conformance between the very top level and the lowest level, even down to basic design. The notation allows us to express hardware, system description, software. We can describe any type of system. I think that's where the flexibility of having the method certainly helps us. At the lower levels, this conformance can be demonstrated.

Paper Nr. 9 - THE A-7E SOFTWARE REQUIREMENTS DOCUMENT: THREE YEARS OF CHANGE DATA
Presented by - D. M. Weiss
Speaker - Dr. L. Crovella

Comment - Are there any other sources of data from different environments you can use to compare with your data?

Response - The only other published data on requirements documents that I know are the result of a critical design review. These data were published by Lipson et al and show similar distribution to ours for errors. In addition, they found in a 2 or 3 day critical design review as many errors as we have found in 3 years.

Paper Nr. 9 - THE A-7E SOFTWARE REQUIREMENTS DOCUMENT: THREE YEARS OF CHANGE DATA
Presented by - D. M. Weiss
Speaker - R. E. Westbrook
Comment - The SRD is being updated to reflect the changes that have occurred in the A7E OFP since 1978. Has any data on these changes been collected? If so, what is the distribution of the data?

Response - No such data has been collected yet.

Paper Nr. 9 - THE A-7E SOFTWARE REQUIREMENTS DOCUMENT: THREE YEARS OF CHANGE DATA
Presented by - D. M. Weiss
Speaker - G. Sundberg
Comment - The error categories did not include design or requirements errors. These errors sometimes do not show up until the operational phase. Do you intend to include these errors in your study and track them once the system becomes operational?

Response - All errors shown were errors in the requirements document. Those errors in the category "incorrect facts" were indeed errors where the requirements were wrong.
Errors in design specification and code are being monitored as a separate part of the project. However, it is not clear exactly what a "design" error is, or how to distinguish such errors from implementation errors. To do this one needs a clear definition of design, which is currently a rather ambiguous term.

Paper Nr. 9 - THE A-7E SOFTWARE REQUIREMENTS DOCUMENT: THREE YEARS OF CHANGE DATA
Presented by - D. M. Weiss
Speaker - Dr. T. G. Swann
Comment - You state that there were very few problems caused by ambiguity. But I understand that the specification was for a system that had already been designed and coded, so I would expect few if any ambiguities to be left in it. Would you care to comment?

Response - The ambiguities referred to are ambiguities in the requirements document. This is a new document, written for this project. A single requirements document had not heretofore existed. Consequently, the existence of the code, while somewhat helpful in resolving answers to questions about the system, had no major effect on the way the authors wrote the requirement.

Paper Nr. 9 - THE A-?E SOFTWARE REQUIREMENTS DOCUMENT: THREE YEARS OF CHANGE DATA
Presented by - Dr. D. M. Weiss
Speaker - Wg. Cdr. S. Barker
Comment - An analysis of eight years worth of program change data for the UK LINESTEAN Air Defence System has shown three main things. Even after eight years, errors came to light attributable to faults in the design phase; that during the initial (four) years simpler changes having minimum system disturbance predominate, that later in the system life, far-reaching changes are attempted. This last fact may explain the variation of the effects of changes in your Figure 10.

Response - Because the system on which data is being collected is not yet in the maintenance phase, it is not possible to say whether we will see the same pattern as you. It will be interesting to make comparisons when we are at the stage where we have comparable data.

Paper Nr. 10 - D.L.A.O.: UN SYSTEME D'AIDE A LA DEFINITION DE LOGICIELS AVIONIQUES
Presenter - Ing. F. Doladille
Speaker - Dr. L. Crovella
Comment - La validation des spécifications par une maquette, n'est-elle pas un des objectifs interessants d'un système d'aide à la spécification?

Response - Cet aspect n'était pas un objectif initial du project D.L.A.O. cependant, nous étudions actuellement la possibilité d'adjoindre au langage des constructions permettant la génération de prototypes.

Paper Nr. 11 - THE MENTOR APPROACH TO REQUIREMENT SPECIFICATION presenter - D. Jordan
Speaker - L. Skorczewski
Comment - The mentor methodology appears to consider only the case where definition/specification starts with a clean sheet of paper and proceeds downwards in levels in an orderly fashion - each element being expanded and analyzed for consistency/conformance as the process progresses.
How does the method operate within larger real-time systems where large elements of hardware (or even software within hardware) are imposed as requirements by the customer from the start. e.g. you will use the manufacturers inertial platform and his navigation software as part of your system specification and design?

Response - It is true that the MENTOR approach is based on a hierarchical and predominantly top down specification technique. However, this does not preclude its use in the situation which you describe.
It is certainly the case that any formal verification technique requires a complete description of *all components within some validated system boundary*. Where pre-existing hardware and/or software is required to be incorporated this implies that a detailed interface specification, at minimum must be provided.
When MENTOR is used in such circumstances it will be necessary for a designer to describe in detail how the existing components will interact with new software to achieve the overall system objectives. Such descriptions may be introduced at any appropriate level of detail. Subsequently, subject to validation the specified component can be treated as an internal interface to the new system.

Paper Nr. 11 - THE MENTOR APPROACH TO REQUIREMENT SPECIFICATION presenter - D. Jordan
Speaker - Dr. W. J. Cullyer
Comment - Can you handle Robin Milner s predicate logic automatically within your tools?

Response - Yes, The tools handle this automatically.

Paper Nr. 11 - THE MENTOR APPROACH TO REQUIREMENT SPECIFICATION presenter - D. Jordan
Speaker - Dr. L. Crovella
Comment - What is in reality the status of this program, is it available?

Response - All that we have is a number of standard and experimental programs. We have just recently embarked on a prototype integrated MENTOR system.

Paper Nr. 12. - THE COMPUTER AIDED SPECIFICATION SYSTEM EASY
Presenter - Dr. N. Christensen
Speaker - Unknown
Comment - Your procedures and packages and other aspects sound very much like the ADA language with its procedures and tasks packages and pr· ·te and public etc. Would you care to comment on that?

Response - Yes, there are some similarities with ADA or perhaps we should say that there are some similarities with the state-of-the-art in making programming languages. But, I think there are quite a lot of differences between EASY and ADA. There is the very important difference that description in EASY is informal so that you have to do more handwork to get an operable program than in ADA. The ADA demands more formal packaging of the descriptions and procedures. There are other differences too. For example our intent is not to produce redundant information or specifications. The program is encouraged to specify or to describe a procedure when he uses it or when he defines it. Another thing is the aspect of human communications is different in ADA and EASY, because we have quite a possibility of generating sorted lists and summaries. I am not sure that ADA sorts at the moment contain the possibility to extract these lists. Of course you can supply this information in an ADA program, too. But, only as comments.

Paper Nr. 12. - THE COMPUTER AIDED SPECIFICATION SYSTEM EASY
Presenter - Dr. N. Christensen
Speaker - Unknown
Comment - Are there any mechanisms in EASY allowing for real-time and synchronization problems?

Response - No, there are not. This is intentional, I think one of the previous speakers, Mr. Sundstrom dealt with general architecture of air-flight control systems. We think, as Mr. Sundstrom said that everything that has to do with scheduling and tasking could be put in one package and not be scattered over the entire system. Our attempt is not to specify scheduling and such things in the packages, but to specify one package which will make tasking and scheduling.

Paper Nr. 12. - THE COMPUTER AIDED SPECIFICATION SYSTEM EASY
Presenter - Dr. N. Christensen
Speaker - D. M. Weiss
Comment - What kinds of automated consistency checking have you built in?

Response - We can prove everything which is specified formally in the specification. This is mainly that the parameters and the import and export definitions of the procedures fit together. Of course, we can also prove there are no ambiguities and uniqueness of names and that all export procedures are used some where in the system, and that all import procedures are defined somewhere in the system. For example if you have no constant for data types, where do you get the data for these types.

Paper Nr. 12. - THE COMPUTER AIDED SPECIFICATION SYSTEM EASY
Presenter - Mr. N. Christensen
Speaker - K. Pulford
Comment - One remark. I think it is becoming an occupational hazard these days in giving papers, to be compared with the ADA programming language or the ADA environment. I have an analogy, which prompts the question.
One problem you have in ADA is the recompilation problem, when you actually compile several units and you have to recompile, one unit invalidates several others. You have this problem of deciding which units are invalid. I think you have the same problem in EASY. Have you looked at this problem?

Response - We are aware of this problem. But, because of the fact that we are not able to automatically generate code, we are not able to tackle this problem automatically. What is very important is that the affected specifications are documented. If you have a change you can then see which packages or procedures are affected through the change. The other attempt we are making on this problem is that you can implement procedures like interpoint in the language, and define these in the implementation part of the specification. When you make a reference in your sort then the actual specification can be taken off the data base. This, of course only applies to formal changes.

THE IMPACT OF STANDARDIZATION ON AVIONIC SOFTWARE

J. D. ENGELLAND
GENERAL DYNAMICS/FORT WORTH DIVISION
FORT WORTH, TEXAS, U.S.A.

1. INTRODUCTION

To get a realistic view of the impact of standardization on avionic software, it is important to review the history of avionics in the last twenty years. In the early sixties, avionics was just beginning to edge into the digital world. At that time, systems were still predominately analog subsystems clustered around a central digital mini-computer. (Reference Figure 1)

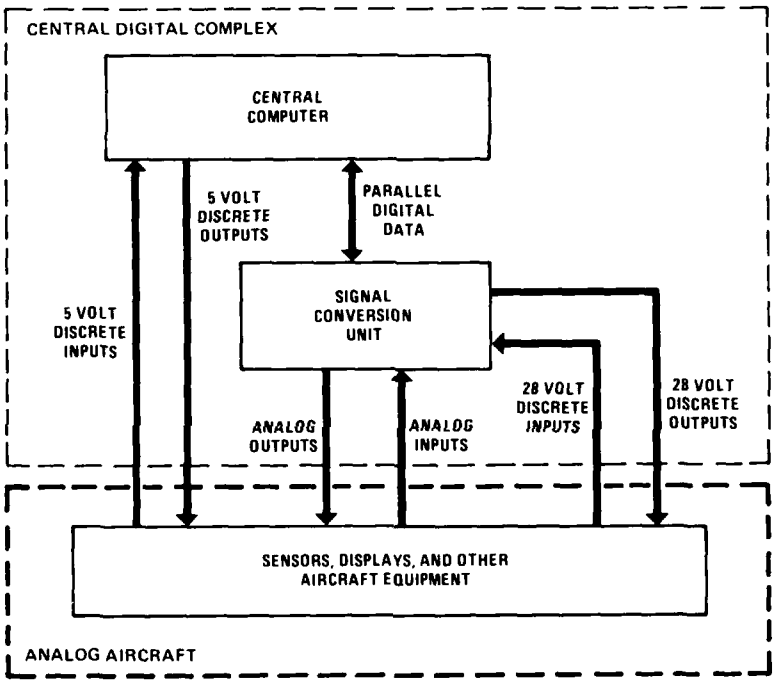


FIGURE 1
TYPICAL CENTRALIZED AVIONIC COMPUTER SYSTEM

This computer still did what the analog computer it replaced did, but better, and it could be easily changed! The software for these computers was painstakingly prepared in assembler (sometimes even in machine) language, typically with little documentation and even less concern about maintenance. Usually, because the computer memory and throughput had been drastically undersized for the real requirements, the programs became marvels of compact, efficient, tricky coding which only the author was able to understand how they worked.

This era of avionic software gave rise to a number of serious problems. Several of the software standards being promulgated today are legacies of the problems encountered with these early digital systems. However, in spite of all problems, the digital systems were clearly demonstrated to be better and more capable and the movement to digital systems accelerated.

With the advent of the micro-processor, it became practical to distribute a processor into virtually every subsystem. As a result, it has become possible to implement nearly any logic or mathematical concept in the avionic system software (e.g., Kalman filters, digital maps, coupled fire/flight control). Avionic systems have, in fact, reached the point where software is no longer simply a part of the system, software is the system.

With software becoming such a dominant factor in avionic systems, it was inevitable that standards would be applied. These standards (e.g., in the U.S. - MIL STD 1553, MIL STD 1589, MIL STD 1750, and MIL STD 1760) are now having major impacts on avionic systems and avionic software. These influences are the subject of this paper. However, one must be careful not to attribute to standardization, changes caused by other events.

1.1 NEW SOFTWARE CONCEPTS

Several key software features found in modern avionic systems are not results of standards at all, but of design methodology advances. The two most significant of these advances are adoption of structured programming concepts and introduction of top-down design methodologies. In both cases, they were quickly recognized as sound engineering approaches to software design and eagerly adopted.

Structured programming, in particular, has had a profound influence on software design (Dahl, Digkstra, and Hoare, 1972). With its emphasis on use of sound programming constructs and disciplined design of program flow paths, structured programming has become an integral part of every avionic system being designed today.

Top-down design methodology, as described in work done by Baker at IBM, has also been widely recognized as a good software design and management concept (IBM System Journal, 1972). Such concepts as structured walk-thrus, program librarians and, of course, top-down coding are being used in some form and combination in nearly all avionic software being developed in the U.S.

1.2 HIGHER ORDER LANGUAGES

Another significant influence on avionic software has been the movement to Higher Order Languages (HOL). Acceptance of HOL for avionic software did not really occur until hardware with sufficient excess capability to permit the 10-20% coding inefficiencies associated with HOL was available. A number of languages have been used in embeeded avionic applications including FORTRAN, JOVIAL/J3B, and AED. While none of these languages is listed as a standard to be used for avionics, they have permanently changed how avionic software is developed. In addition to supporting the top-down structuring and structured programming concepts, they allowed movement of software away from fixed-point data to floating point, caused emphasis to be placed on symbolic debugging tools, and highlighted the need for better compilers and software development environments. The impact of standardization with a single HOL (e.g., ADA) is yet to be seen, although inferences from the use of current standards such as MIL STD 1589B (JOVIAL J73) can be made.

First, use of HOL has increased programmer productivity. This increased productivity occurs in reduced coding effort. Some secondary benefits occur in the design phase in that the HOL is a problem - oriented design medium. This allows the designer to focus more on the function to be implemented and less on the details of the processor.

Second, use of HOL has increased initial code correctness. With a mature compiler, it is not uncommon to achieve execution of a flight program immediately upon compilation as opposed to the typical three or four revisions necessary with assembler programs to reach that same level.

Third, use of HOL has provided significant improvements in the quality of documentation. Since most HOLs are designed to be self-documenting, it is relatively easy to generate good product specifications with accurate descriptions, equations, and data base definitions. As an additional bonus, detailed accurate flow charts can be automatically generated if contractually required.

A fourth impact of HOL, not as widely used in avionics, is that of HOL-level software debugging. HOL debug tools allowing much of the software debugging to be done at the HOL source level have been used extremely in commercial system and personal computers (e.g., TRACE command). Their development and use is just now accelerating in the avionics arena as better compilers, linkers, and loaders which support debug options become available.

A counterbalancing impact of HOL upon avionics is the associated growth in the amount and complexity of support software tools required. If standardization to a single HOL really does lead to a single reusable set of support software that will be the foremost contribution of standardization. Continued development and redevelopment of support software can be avoided and original costs will be amortized over many systems.

1.3 SYSTEMS ATCHITECTURE

A final factor not related to standards which has had a major influence on avionic software is the architecture of the systems. As systems have evolved toward distributed processor networks, an ever-increasing design consideration of system and software design is proper functional partitioning between subsystems, loose coupling of system elements, and graceful reconfiguration/degradation of the integrated system in the presence of failures. The present trend in avionics is toward hierarchical systems architectures. Several systems already display a hierarchical structure (Reference Figure 2). However, a number of initiatives are under way in which new

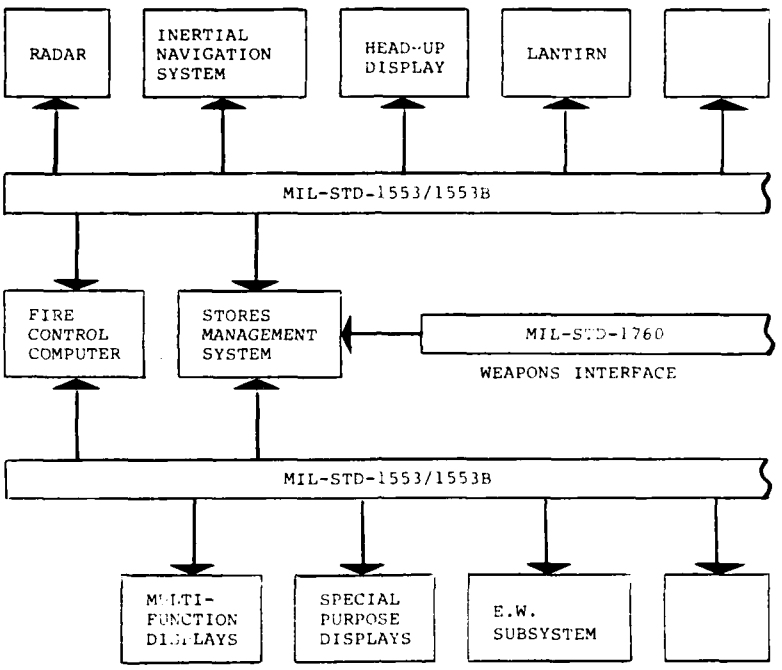


FIGURE 2
TYPICAL AVIONIC SYSTEM ARCHITECTURE

very widely distributed system concepts and architectures will be explored. These initiatives focus on both physical interconnection techniques and communication protocols. One of the significant initiatives currently in the works is the USAF PAVE PILLAR Study. This effort is intended to define the shape of next generation avionic systems architecture, hardware, and partitioning.

2.0 IMPACTS OF STANDARDIZATION

If standards did not cause adoption of structured programming or utilization of top-down design, and did not foster evolution of system architecture, what then are their impacts on avionic software? We will look at five standardization areas in the remainder of this paper, analyze their impacts, and then look to the future.

2.1 DOCUMENTATION AND CONFIGURATION MANAGEMENT STANDARDS CAME FIRST

Although the thrust toward avionic software standardization is generally perceived as a rather recent event, in the United States at least, several key standards have influenced software design and software documentation since the late 1960s. These standards (MIL STDs 483, 490, and 1521) required rigorous configuration management, disciplined control of development programs, with formal reviews and audits, and extensive documentation of software. These early standards provided a significant push toward putting discipline and visibility into software design and producing software that can be maintained by persons other than the original developer. However, they have not significantly affected the structure of embedded software or its reusability across different avionic systems. The four key new avionic standards defined by the U.S. Air Force, (Reference Figure 3), are having an impact on the structure of software.

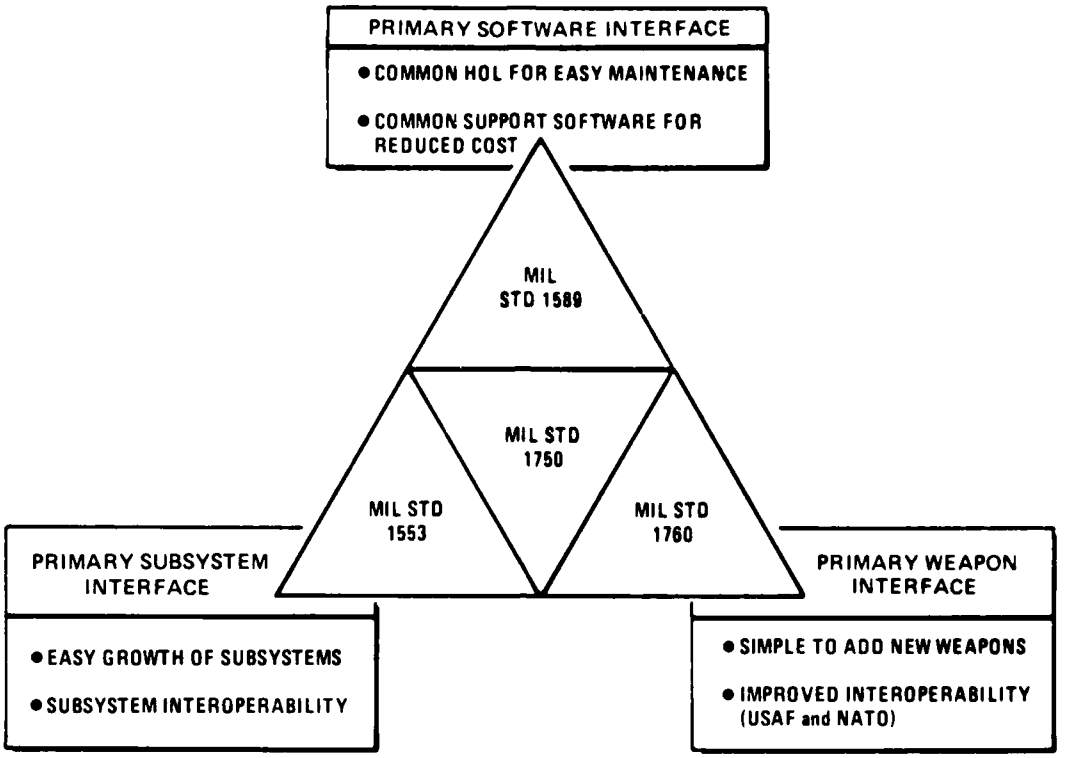


FIGURE 3
CURRENT STANDARDS ARE AT KEY INTERFACES

2.2 STANDARD MULTIPLEX DATA BUS

MIL STD 1553 is a bus interface protocol and signal characteristic standard that has no direct requirements on software. However, it has profoundly impacted software structure in several ways. First, the standard provides a well-known stable communication interface for the embedded software. As a result, alternate software approaches for control and use of the MIL STD 1553 interface have been tried and results reported in public forms (Sundstrom, Edwards, 1981). Because these reports do identify concepts which work well, there has been significant convergence of software design for controller/terminals toward table-driven redundant channel configurations with programmable error handling.

Although the command/response nature of the MIL STD 1553 multiplex data bus is probably its most widely recognized feature, the feature which has the most profound influence on software structure is the asynchronous nature of its data transmission. To achieve data time consistency between subsystems on the bus hardware/software concepts such as time-stamping (Brumback, 1982) have been adopted. In the example of time-stamping, the system must contain a global time reference and the originating processor must append to its data the time validity of that data. The impact on software is obvious:

(1) The originator of data must encode and append the stamp to data,

(2) The data base must be designed with allocations for time stamps,

(3) The user of data must decode and use the stamp for extrapolation/interpolation as manipulation of the data in his algorithms.

An impact perhaps not so obvious is that, because time-stamping is a total system concept, all new equipment or weapons added to the system must either embrace the same concept also or at least be compatible with it. This is an important consideration and impact for the designer of new add-on avionics indeed!

2.3 STANDARD HIGHER ORDER LANGUAGE (HOL)

Selection of a specific HOL as the standard, such as MIL STD 1589, has a rather subtle impact on avionic software structure. The most visible impact will be on data structures since each language defines its data elements differently. (Ada with some of its distinct data concepts will have a significant impact on data structures in the software.) A more subtle impact is one of identifying those language constructs are most code efficient and using them in preference to other constructs. This impact will become significant but hard to measure if use of a standard HOL leads to widespread use of a common compiler. Impact on the embedded computer operating system design, such as foreseen with Ada, has not occurred with MIL STD 1589 since that language does not contain real-time constructs. Until Ada is in full use, impact of HOL standardization will continue to be improved code readibility and development with potential use of a common compiler and support software tools across avionic systems.

2.4 STANDARD COMPUTER INSTRUCTION SET ARCHITECTURE (ISA)

The MIL STD 1750 ISA defines a standard processor instruction set which has applicability to a broad range of embedded avionic processors. This standard can quite correctly be viewed as an interface standard between the embedded computer hardware and the avionic software. As an interface standard, it directly impacts software only in the operating system kernels (interrupt handlers, timer and clock routines, processor-controlled I/O, etc.). However, a synergistic result can occur when MIL STD 1750 is used in conjunction with MIL STD 1553. This result is definition of a common software structure and command protocol for operation of a MIL STD 1553 data channel by a MIL STD 1750 processor (Alford, 1982).

Results in this area have already proven to be beneficial for software development. Even more promising, several manufacturers have expressed interest in implementing a VLSI 1553 data channel device tailored to interface with this common MIL STD 1750 interface. An exciting thought -- software actually leading hardware design.

2.5 STANDARD AIRCRAFT/STORES INTERFACE

MIL STD 1760, newest of the current standards, defines the total signal and power interface between the aircraft and all stores carried on the aircraft from simple ordinance to podded avionics. Since MIL STD 1760 uses MIL STD 1153 as its data interface, software impacts of MIL STD 1760 will largely be the same as for MIL STD 1553. A word of caution is due, however. The time-stamping considerations will also be true. Consequently, smart weapons also must be designed to have time-stamp compatibility.

3.0 A LOOK TO THE FUTURE

As the digital avionic systems continue to grow in capability, complexity and mission and flight criticality, software reliability and maintainability will become even more important than it is now. Accordingly, new standards will be established in a number of areas. Several which appear highly probable are:

- o Standard Mass Storage Data Interface
- o Standard Operating System Environment (Interface)
- o Standard 1750/1553 Channel Command Protocol
- o Standard Reference Frames (i.e., Inertial, Earth Fixed, Body Coordinates)
- o Standard Data Time-Stamp Reference Provisions

Standardization of application modules and algorithms is less probable in the near term, since these modules and algorithms directly affect system performance. In many cases, the aircraft capabilities and mission are dominant factors in selection of algorithms and application modules which meet specified performance requirements. Imposition of specific standard algorithmic implementations could create significant contractual difficulties in assessing performance responsibility if the resulting system then failed to meet all requirements.

4.0 SUMMARY

Profound changes in avionic software have occurred over the past twenty years. Many of these changes were not the result of standardization initiatives but rather of hardware evolution and maturation of software engineering from an art to a discipline, and growth of avionic system architectures from centralized systems to distributed networks.

Recent avionic standards, notably MIL STDs 1553, 1589, 1750, and 1760, although they are interface standards, are having subtle but significant impact on avionic software structure. One of the most significant impacts is that of system-wide data time correlation. With the current trend to independent development of sensors and subsystems for multiple aircraft types, great care must be taken to ensure compatibility with the overall aircraft system is being designed into the new unit. Finally, more new standards will be defined, however, these standards will turn more toward the pure software relm rather than the hardware/software interface, with attempts to standardize key software interfaces.

BIBLIOGRAPHY

Alford, Steven, 1982, "A Common 1553B I/O Channel for the F-16", Internal General Dynamics/Fort Worth Report.

Brumback, B. D., 1982, "Time-Referencing of Data in an Asynchronous Environment", Naecon Proceedings, Dayton, Ohio.

Dahl, O. J.; Digkstra E. W.; and Hoare, C. A. R., 1972, "Notes on Structured Programming", E. W. Digstra in Structured Programming, Academic Press.

Sundstrom, Dr. D. E.; Edwards, Dr. J. A., 1981, "Inside MIL-STD 1553 - Efficient Embedded Protocols", Naecon Proceedings, Dayton, Ohio.

REFERENCES

"Chief Programmer Team Management of Production Programming", 1972, IBM System Journal, Vol. II, No. 1.

Ada® STATUS AND OUTLOOK

by

Lt Cdr John F.Kramer, Jr
ADA Joint Program Office*
Arlington, Virginia
USA

The Ada Joint Program Office (AJPO) is attached to the Office of the Deputy Under Secretary of Defense for Research and Engineering (Research and Advanced Technology). The AJPO is a small office comprised of a Director, Lt Col. Larry E.Druffel, USAF; a Technical Director, Dr Robert F.Mathis and three Service Deputy Directors, one from each of the Military Departments.

1. INTRODUCTION

Ada is a modern high order computer programming language which the US DoD intends to adopt as the standard language for embedded computer applications.

The US DoD language standardization effort has been principally directed at the Embedded Computer system area, and not as a replacement for COBOL in the traditional areas of Financial Management, Inventory or Payroll; or as a replacement for the large scientific computations normally done in FORTRAN. Although there appears to be a growing recognition that Ada, as a modern programming language embodying good software engineering principles and modern language features, is very suitable for those areas as well as the embedded computer applicators on which it was designed, the AJPO is still principally concerned with the application of Ada to the embedded applications and their support systems.

An Embedded Computer system is an integrated hardware and software system that forms part of a larger system. Examples of Embedded Computer systems are communication systems, command and control systems, on-board aircraft navigation and weapon control systems and other real-time control systems, such as the computer in a car or the control processor in a microwave oven. These systems may range in size from a small microcomputer system, such as an aircraft auto pilot, to a network of large computers such as found in large ground based command and control systems.

Embedded Computer systems have difficult life-cycle software problems to contend with. They are often very large (sometimes in the millions of lines of code), they are usually long lived (lasting more than 20 years), and they usually require continuous changes (typically several times a year) in order to keep up with new weapon systems or change in threats.

In addition to these important life-cycle problems, embedded computer applications often have characteristics which require certain features in a language which are not readily available in programming languages. Embedded computer application processes usually occur in parallel rather than sequentially, or often require man-machine interactions involving real time control. These parallel tasks must have some form of inter-task communication to be able to exchange information, a capability that is missing from traditional languages and has resulted in a large number of assembly language inserts into otherwise high order language programs. Embedded computer applications also need some sort of automatic error recovery since they cannot stop when some exceptional situation occurs, such as an arithmetic overflow. The language must be able to specify a recovery action at the application level in such situations. Finally, embedded computer systems must be able to use non-standard Input/Outputs. Inputs are often from sensors and outputs are often in the form of control information to a mechanical device not to and from the usual disk, tape, typewriter, or printers.

2. Ada JOINT PROGRAM OFFICE

The Ada® Joint Program Office (AJPO) was established to implement, introduce, and provide life-cycle support for Ada and its support systems. The AJPO has representation from each of the services and is an excellent example of tri-service cooperation under OSD management.

---

* Attached to the Office of the Deputy Under-Secretary of Defense for Research and Engineering, The Pentagon, Washington DC 20301.

The AJPO is responsible for developing and managing the DoD Ada program, for coordinating all DoD Ada developments, and for DoD cooperation with standardization activities such as ANSI, ISO, and NATO. It provides the principal DoD interface with the computing community both in the US and abroad, and encourages development and application of advanced software techniques using Ada as the vehicle for technology transfer. The AJPO's responsibilities are being expanded to develop and coordinate a plan for research in software and systems.

3. Ada PROGRAM

The Ada Program extends beyond the normal language standardization to include controlling and cost and improving the quality of the software by facilitating the application of modern software engineering practices to embedded computer system developments.

In 1975, the High Order Language Working Group (HOLWG) was established by the US DoD with representation from Army, Navy, Air Force, DCA, NSA and DARPA to investigate the feasibility of adopting a common high order computer programming language for use in embedded computer systems. A comprehensive set of requirements was developed and published in the June 1978 Steelman document. Over 23 existing computer languages were formally evaluated against these requirements, and when no existing language was found sufficiently powerful to serve as the common language, the HOLWG undertook a competitive international procurement to develop a new language. The language design was completed by Cii-Honeywell Bull in July 1980, and is currently in the process of being adopted as a US American National Standards Institute (ANSI) standard. The Ada Program has derived substantial benefit from the very broad world wide participation and cooperation of the government, industry and academic computing community.

4. Ada PROGRAM OBJECTIVES

There are four major Ada® program objectives. First, the AJPO must ensure the implementation and maintenance of Ada as a consistent, unambiguous standard recognized by the DoD and also by the widest possible community. Recognition of Ada as a standard is a necessary step in the realization of software and people portability. Second, the AJPO must ensure the smooth introduction and acceptance of Ada in the DoD as early as possible consistent with the needs of individual components. There are a number of projects which could benefit from an early introduction of the language. However, the advantages offered by the use of Ada will not be realized unless a programming support environment is also available. Therefore, this objective must balance the need for an early introduction of the language against the risk of a premature introduction. The third major objective is to provide Ada education and training to the software community. The success of the Ada program depends upon the development of a broad Ada knowledgeable resource pool of software personnel. The final major objective is to provide for the life-cycle software support for Embedded Computer systems. This will include the design, development and distribution of transportable Ada tools which will provide software support for the application software in embedded computer systems.

5. ECS LANGUAGE STANDARDIZATION

In 1975 almost all embedded software was being written in many different assembly languages and a diversity of High Order Languages (HOL). The HOLs being used did not support modern programming methodologies and were ill-suited for the ECS application. It was also becoming apparent that it was not sufficient to simply have a standard language and computer. What was needed was an integrated set of software tools which supported the whole ECS software life-cycle. In addition each military service was developing its own language which tended to make the situation even worse.

As a result of the US DoD's recognition of these problems, the High Order Language Working (HOLWG) was formed to identify DoD's requirements for computer programming languages, to evaluate the existing languages, and to recommend the implementations and control of a "minimal set".

In 1976, DoDD 5000.29 required DoD approved languages be used on all new projects and a control agent for each standard language, to ensure the use of the language and the compilers for a language complied with the Directive. The Directive required the establishment of a Management Steering Committee for Embedded Computer Resources (MSC-ECR) to correct management problems of computer based systems. The HOLWG became a committee of the MSC-ECR. DoDD 5000.31 implemented DoDD 5000.29 and established seven approved HOLs: FORTRAN, COBOL, CMS-2, SPL-1, JOVIAL-J3, JOVIAL-J73, and TACPOL. Since FORTRAN and COBOL were already ANSI standards, ANSI was designated the control agent for them. The Navy was designated control agent for the two languages CMS-2 and SPL-1, the Air Force was designated control agent of the two JOVIAL dialects, and the Army was designated control agent for TACPOL.

The 5000.31 languages were not viewed as the long term solution to DoD's language, however, they provided a stable starting point for the DoD language use policies. The MSC-ECR requested two independent cost benefit analysis studies to determine the benefits of further reducing the number of languages. The studies showed that on a yearly basis, hundreds of million of dollars could be saved by adopting a suitable language with a reasonable measure of acceptance

5000.31 is currently being revised to include Ada®. Although it is DoD's intention to phase the individual service languages out, they will remain in use since embedded systems tend to be long lived. Eventually 5000.31 may contain only Ada, FORTRAN and COBOL. There is no design to replace the large amount of software written in FORTRAN and COBOL. While individual program managers may use Ada for new systems in traditional FORTRAN and COBOL areas, it would require a substantial investment in library software. It will be several decades before this could be considered for systems already in existence.

6. Ada STANDARDIZATION

The AJPO is in the late stages of the American National Standards Institute (ANSI) canvass process to establish the programming language Ada® as a US ANSI standard.

The proposed standard Ada Language Reference Manual was finalized in July 1980 and has received an enormous amount of review and comment, all of which was included in the ANSI canvass process. On December 10, 1980, the same manual was published by the US DoD as a Military Standard (MIL-STD 1815) so that early DoD projects would be able to use Ada.

The canvass procedure began in April 1981 with ANSI approval of the canvassee list, which was balanced among potential implementors, potential users and general interest categories. ANSI approved 96 canvassees to vote on the proposed Ada standard. The proposed standard Ada Language Reference Manual, MIL-STD 1815, was mailed to the canvassees who were allowed six months to review it and vote, with comment or reservation, if they so chose. There were 380 comments received with the ballots, which closed the first state of the canvass on October 15, 1981. The tally was 66 for, 23 against, and 7 not voting. Concurrent with the formal canvass, a public review was conducted. Although required to consider only those comments made during a specified two-month period by ANSI directives, the DoD initiated the public review in December 1980 and kept it open for a year (until December 1981). The public review was also expanded to include the international community in order to provide more visibility to those interested in considering Ada as an International Standards Organization (ISO) recommendation. There were 758 comments received from the public review which were logged into a file and made publicly available via the ARPANET.

Based on this extensive commentary, the language design team made appropriate language changes and produced a set of chapter reviews documenting the changes to the Language Reference Manual and the Ada Language. These chapter reviews received extensive analysis from the international group of expert computer scientists serving as distinguished reviewers for the later phases of the Ada Program. The canvassees were advised of unresolved issues and given a 30-day period during which to change their vote. Of the 96 canvassees, only four have indicated dissatisfaction with the resolution of the issues raised during the canvass. Although there were some changes to the language, there has been no change to its structure: a user would not generally perceive the changes and certainly would not observe a change in functionality. The user may observe simplifications in the underlying model and greater consistency in the design. Of 400 examples used in a course taught a number of times by the language designer, only four will have to be changed and only one will be completely eliminated.

The publication date of an ANSI version of the Ada reference manual will be determined by ANSI in accordance with its management and administrative procedures. In order to have a US DoD Military standard between the time it is decided that ANSI will approve the language and the time the ANSI manual is published, the US DoD will publish MIL-STD 1815A around September 1982. It will be the July 1982 version with appropriate editorial changes as determined by the ANSI supplemental canvass.

Upon acceptance of the Ada language as a US ANSI Standard, ANSI/ASC-X3 will encourage the adoption of Ada as an international standard through the International Standards Organization (ISO). ISO/TC97/SC5 (programming language subcommittee) recommended establishment of an experts group in October 1981. One organized meeting of experts was held although the group has not yet been formally constituted.

The final ANSI tally represents convincing support for the language as revised. However, the number of changes made to the Language Reference Manual suggests that an opportunity for the canvassees to evaluate the revised reference manual would be appropriate. Therefore, the DoD has obtained ANSI approval of a supplemental canvass, for editorial review only, to commence in July 1982. Additional language changes would not be entertained. A two-month supplemental canvas will give the canvassees an opportunity to verify that the changes made to the Language Reference Manual are consistent with their understanding of the revised language. Any language changes which are suggested will be logged and made publicly available. They will be forwarded as part of the further consideration of Ada as an international standard.

7. Ada VALIDATION ORGANIZATION

A major goal of the Ada® program is to ensure that software is portable across implementations, and that it produces the same results independent of the computer on which it is being run or compiler front-end or back-end that generated the code. While validation has usually been an afterthough in language design, this was not the case for Ada.

Early in the Ada effort it was recognized by the US DoD that a compiler validation capability would be required as part of the enforcement mechanism. In 1979 a competitive procurement was undertaken for a set of test programs, validation test tools to assist in the preparation and analysis of results, and an implementer's guide.

The US DoD will require the validation of all compilers prior to their use in any DoD program. Since the validation suite itself will not ensure that Ada compilers implement the same common language, the Ada Joint Program Office is in the process of establishing an Ada Validation Organization (AVO). The AVO will have to encourage, measure and enforce conformance to the Ada Standard and other related standards which may be defined throughout the program. It must provide technical assistance in the validation process and also must provide a focal point for efforts to advance the state-of-the-art with respect to Ada validation. The duties of the AVO will include but not be limited to: maintenance of a state-of-the-art Ada validation capability; providing public access to the AVO; maintaining the implementer's guide; reporting the results of validations; certifying satellite facilities; and managing the issuance of certificates of validation, retesting and other administrative matters in support of the AVO and the validation process.

The current validation capability is made up of approximately 1400 test programs which were designed to check the presence of features and the absence of non-standard features. The validation capability is made of both compile time and run time tests designed to check for the presence or absence of legal programs; link load rejection of illegal programs, and self testing programs. The AVO will also check for capacity requirements, correct warnings and the operation of standard packages.

The US DoD feels that because of the growing worldwide interest in the use and development of Ada, Satellite Ada Validation Facilities (SAVF) will have to be established. Such a SAVF would operate in a manner consistent with the AVO such that certificates of Ada validation issued by a SAVF would be equivalent to those of the AVO in rights and effect.

8. Ada SUPPORT ENVIRONMENT

The intended use of a machine independent HOL in the embedded computer application area, lead the HOLWG to include a support environment as a critical part of the US DoD's HOL standardization effort. Although Ada© did not require a special environment, the life-cycle maintenance of ECS systems did. It was felt that an integrated software environment containing a set of good tools would encourage acceptance of the language, thereby magnifying the benefits of the language standardization effort. The set of tools that needed to be included in a program support environment were: a compiler, editor, documentation aids, program development aids, and other life-cycle support tools. Failure to develop such a facility would mean that software development would continue to be treated as an art, with little basis for predicting software costs and completion times resulting in late, erroneous and costly software.

The initial US DoD sponsored workshop was held at Irvine, California in June 1978, to discuss alternatives. The result of the workshop was a draft requirements document "PEBBLEMAN", describing all aspects of the problem, including many policy issues. In November 1979, a revision was published treating only the technical issues and a workshop was held at San Diego, California to review it and solicit new ideas. The final "STONEMAN" document was prepared and distributed in February 1980.

9. Ada LANGUAGE ENVIRONMENT (See Figure 1)

By design, Ada© incorporates many of the features needed to support modern programming practices, but just like any tool it can be misused. The capabilities of Ada will only be fully realized when a sophisticated Ada Programming Support Environment, complete with advanced development and management tools, is made available and widely used. The Stoneman model of a support environment calls for the integration of conventional software tools into a framework that is sufficiently open ended to accommodate a wide variety of programming methodologies and automated software tools.

The purpose of the APSE is to support the development and maintenance of application software throughout its life cycle, with particular emphasis on software for embedded computer applications. One of the more important concepts in an APSE is the data base, which acts as the central repository for information associated with each project throughout the life cycle. The data base supports the organizational infrastructure as well as maintining the data critical to the development, testing and life cycle support of software. The data base also serves as the interface through which the set of modular tools can communicate. The data base will contain such management information as version control, library support and project management as well as the code, test data, and documentation required as part of any development.

A second important aspect of the APSE is its host-target relationship. It recognizes that many Embedded Computer systems are not capable of supporting development themselves. The host-target relationship permits the development, testing, and maintenance of software for less capable machines to be done in a symbiotic way from a much more capable machine which can support the sophisticated and often resource intensive tools that need to be applied to the embedded software life cycle problem.

10. APSE STATUS

The US DoD has two minimal Ada Programming Support development efforts underway, the Army Ada Language System (ALS) and the Air Force Integrated Environment (AIE). The Army will begin testing the ALS along with a few selected Navy and Air Force sites in the Fall of 1982. The ALS will begin Beta testing in the Spring of 1983 and be available for wider use in the Fall of 1983. The AIE will be delivered in 1984 with a rehosted version available in 1985. The US Navy will be issuing an RFP for their MAPSE in the Winter of 1982 for delivery in 1985. The Navy MAPSE will be based on the ALS.

In addition to the US DoD APSE work, there are at least two other efforts underway. The commission of the European Community (CEC) awarded contracts for a root compiler and MAPSE based on 50% funding by capital. The German SPERBER project, supported by the German MOD will become a complete MAPSE with the inclusion of the data base in 1985 and the British MOD intends to partially fund an environment based on the extensive British MAPSE design studies completed in 1981 and 1982. In addition to these relatively complete MAPSE developments, there are numerous commercial Ada tool set developments and potentially several MAPSES.

The UK DOI Development Methodology Study assessed Ada as an acceptable HOL for implementing software developed under a wide variety of methods. Since no complete life-cycle method exists today, let alone one which includes Ada as an implementation language, the US DoD has an effort underway to define the requirements for a life cycle methodology. After sufficient international comment, the US DoD intends to build at least one such methodology and a complete set of tools to support it.

Since the Ada Joint Program Office (AJPO) anticipates industry, academia and other governments to support and build APSEs besides the two US DoD supported environments, the Navy has been tasked to lead a joint service review team to identify and recommend conventions for the tool to KAPSE interfaces. The degree of success of this effort at identifying reasonable conventions and standards and the degree to which all KAPSE developers adhere to them will significantly influence the cost of porting a tool from one KAPSE to another and therefore how well we can amortize the cost of sophisticated tool development across a large number of installations.

11. AVAILABILITY AND MATURITY

The availability and maturity of a new language like Ada® are very critical in the decision making process for program managers considering the use of it for a production project. They must weigh the anticipated long term life cycle benefits to be derived from the use of Ada against the current status of the language, support environment, training, general public involvement and their own project time line. With the anticipated adoption of the Ada language as an American National Standard (ANSI) language in the fall of 1982, the language will become stable and will remain so until the International Standards Organization (ISO) changes it during their adoption process around 1985. This will give at least 3 years of experience with the 1982 ANSI version before deciding what changes, if any, should be made as part of the ISO process.

At least four MAPSEs are being developed today, and will begin to appear at potential user sites during the next year. These environments are fairly ambitious projects and will probably require at least a year before they can be considered of production quality for large projects. Since the early Ada environments will really only be MAPSEs, they will not provide any significant improvement over any existing environment, except in their potential to be populated with new and fairly sophisticated life cycle Ada tools as they are developed over the next decade by the US DoD and the many other Ada users.

At present the amount of Ada training being conducted is relatively small, but it can be expected to increase significantly over the next two years as the language becomes a standard, and as the US DoD begins to mandate its use in various projects. Ada Program Design Languages are already being mandated in numerous contracts, which is beginning to increase the demand for people who have some knowledge of the language. The US DoD is developing a training plan which will address the kinds of training required, the materials and facilities required and the ways of measuring the effectiveness of the training. Once the requirements are understood, the US DoD intends to make sure that all required capabilities are available either from the public sector or by them being developed at DoD expense.

The international involvement in the Ada program is astonishing. Not only has this involvement been during the design of the language, but it has been at the commercial and government level by making economic commitments towards some Ada effort. The number of text books that are appearing is increasing daily, as are the courses being offered. As compilers and Ada environments become readily available to the academic community the amount of research revolving around the Ada language will increase greatly.

12. CONCLUSION

The US Dod is totally committed to using the Ada language for its embedded computer applications. Each of the US Military Departments has an introduction strategy. The Army is the most aggressive because they have the least

invested in current standards. They will begin using Ada in 1983 for new low risk programs. The US Navy and Air Force intend to begin to use Ada at a low risk and mandate it for all new programs starting in 1986 when appropriate support environments are in place.

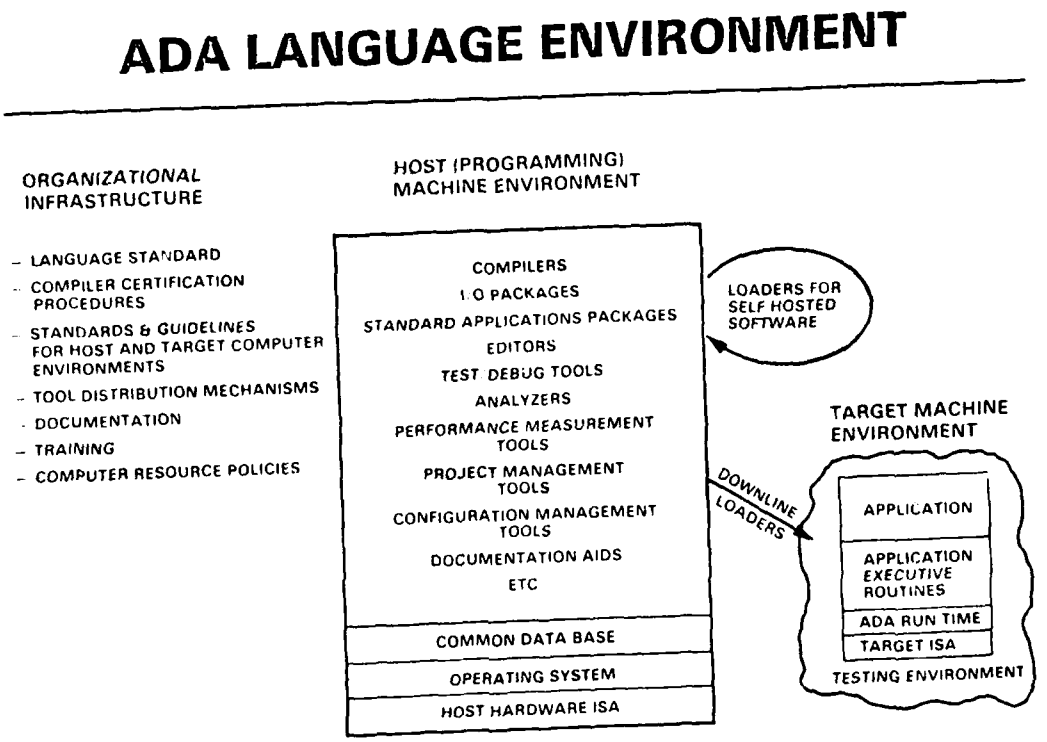


Figure 1

STANDARDISATION DU LTR POUR CALCULATEURS EMBARQUES

LE PRESENT ET LE FUTUR

ICA de Montcheuil

Ministère de la Défense
26 Boulevard Victor
75996 Paris Armées-France

RESUME

Le langage LTR a été standardisé en France pour les applications militaires opérationnelles en 1974. Depuis cette date de nombreux développements ont été réalisés et le langage est effectivement utilisé dans plusieurs systèmes embarqués opérationnels et en développement.

Une courte présentation du langage aujourd'hui appelé LTR V2 montre qu'il répond aux principaux critères permettant une utilisation effective dans des systèmes embarqués : efficacité du code produit, couverture des besoins spécifiques, facilité d'écriture ou de modification de programmes, disponibilité de chaîne de production adéquate.

Cependant en raison des progrès des techniques de programmation, il faut définir une nouvelle stratégie pour l'avenir. Il ne semble pas possible d'attendre en espérant que le développement d'ADA apportera une solution, car l'examen de sa définition fait craindre qu'il ne soit pas effectivement utilisable pour les applications embarquées.

C'est pourquoi une stratégie offrant les meilleures garanties a été définie, celle de rénover le langage LTR afin de profiter des améliorations technologiques tout en préservant les acquis et la possibilité d'utilisation effective. Ce nouveau langage appelé LTR V3 est rapidement présenté avec son atelier de programmation.

1. INTRODUCTION

Après un bref rappel de la politique de standardisation en informatique, il sera présenté un bilan de la standardisation et de l'utilisation du LTR actuellement.

Une analyse des critères d'utilisation d'un langage pour calculateurs embarqués amènera à définir une stratégie pour l'avenir basée sur une rénovation du langage LTR.

2. POLITIQUE DE STANDARDISATION EN INFORMATIQUE

Devant les problèmes soulevés par l'introduction de l'informatique dans les systèmes d'arme, le Ministère de la Défense français a décidé dès 1965 de mener une politique active de standardisation en informatique et créé un organisme pour animer cette politique : le Service Central des Télécommunications et de l'Informatique.

Sur le plan matériel ont été développés des calculateurs militaires interarmées (gamme IRIS M puis gamme 15 M) qui ont été adoptés pour des applications des trois armées (air, mer, terre).

Sur le plan logiciel, les efforts ont porté sur les langages, les bases de données et la méthodologie de développement.

Comme langage, le LTR a été standardisé en 1974 et doit être utilisé dans tous les systèmes opérationnels, embarqués ou non. Depuis cette date des investissements importants ont été faits pour permettre son utilisation effective, et aujourd'hui la plupart des nouveaux systèmes militaires utilisent ce langage.

3. HISTORIQUE DU DEVELOPPEMENT DU LANGAGE LTR

Le langage LTR a été défini en 1968 au Centre de Programmation de la Marine, en le dérivant de l'ALGOL W. Un compilateur maquette a été réalisé de 1969 à 1971. A la suite de cette réalisation a été établie en 1972 la définition du LTR dit "de base" et commencée la réalisation d'un compilateur opérationnel pour les calculateurs de la gamme IRIS militaire.

Le premier compilateur opérationnel a été disponible en 1974, et alors a été prise la décision de standardiser le langage pour toutes les applications opérationnelles.

De nouveaux compilateurs ont été lancés pour les calculateurs militaires de la gamme 15 M et pour les calculateurs IBM série 370, et ont abouti en 1977.

Devant le développement des applications, il a paru nécessaire en 1978 de définir de façon normative le langage, qui a été nommé LTR V2, et donc de réaliser un projet de norme.

Trois compilateurs pour calculateurs aéroportés ont été réalisés en 78-80 (calculateurs EMD série 84, SFENA UMP 7800, THOMSON-CSF MPB 77 et 80).

De nouveaux compilateurs ont été lancés pour aboutir en 1983 pour le microprocesseur Motorola 68000 et les calculateurs SEL 32.

4. UTILISATION ACTUELLE

Quelques systèmes embarqués maintenant en phase opérationnelle sont programmés en LTR. On peut citer :

- SENIT 4, système tactique embarqué sur corvette anti-sous-marine,
- ATTILA, système d'artillerie embarqué en cadres mobiles.

De nombreux systèmes en développement sont en cours de programmation en LTR.

Dans le domaine aéronautique on peut citer :

- l'avion de combat MIRAGE 2000 dans lequel plusieurs calculateurs sont programmés en LTR (Calculateur principal, calculateur de pilotage...),
- l'avion de reconnaissance "Atlantic Nouvelle Génération" dont le système tactique et le système de visualisation sont en cours de programmation en LTR.

Parmi les systèmes embarqués non aéronautiques, on peut citer le nouveau système de missiles balistiques maritime dont tous les sous-systèmes sont programmés en LTR (mise en oeuvre et contrôle des missiles, exploitation tactique, navigation, stabilisation...).

Le nombre de cartes LTR écrites pour des applications opérationnelles dépasse aujourd'hui 800 000.

5. PRESENTATION DES CONCEPTS DU LTR V2

5.1. Concepts de base

Dérivé de l'ALGOL W, le langage a été enrichi pour permettre une description globale d'une application temps réel, et introduire des mécanismes de communications entre tâches d'une application multitâche.

Les principaux points à noter sont les suivants.

5.2 Description de l'application

Une application peut être composée :

- d'un ensemble de données systèmes (SYSTEM DATA),
- d'un ensemble de données de communications (GLOBAL DATA),
- d'un ensemble de sous-programmes communs (GLOBAL PROCEDURE),
- de programmes (PROCESS) activés par appel ou par interruption,
- d'un programme de départ (START PROCESS).

Chaque ensemble ou programme est décomposé en articles : articles de données et sous-programmes (PROCEDURE, FUNCTION, PROCESS).

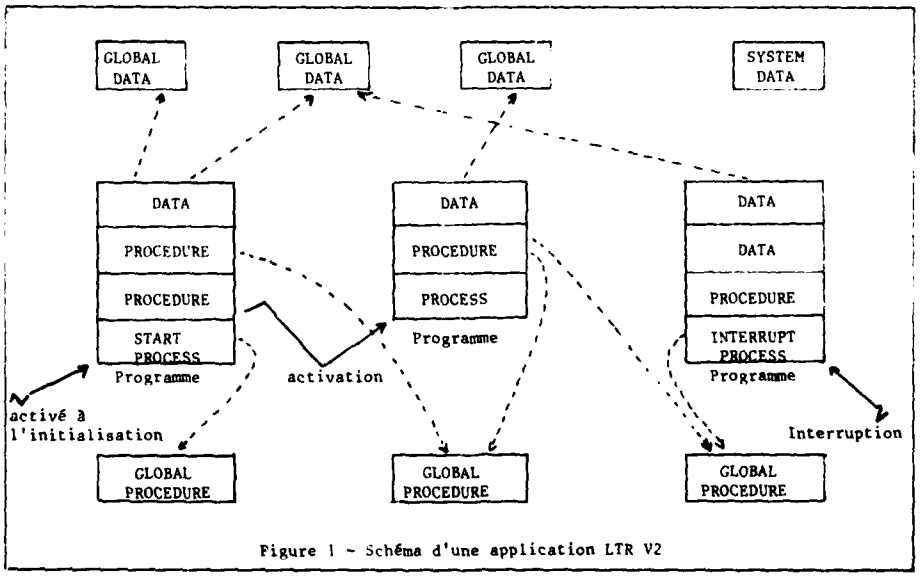


Figure 1 - Schéma d'une application LTR V2

5.3. Mécanismes temps réel

Les mécanismes temps réel sont des phrases du langage permettant de :

- créer des tâches sur des programmes,
- activer ou désactiver des événements et se mettre en attente d'une expression d'événements et délais,
- réserver (conditionnellement ou inconditionnellement) ou libérer un point d'entrée de sémaphore,
- accéder à une horloge temps réel,
- contrôler les interruptions (activation, connexion de programmes, activation de tâche sur interruption),
- effectuer des entrées-sorties soit au niveau physique, soit à un niveau intermédiaire (par bloc de contrôle), soit au niveau fichier formaté.

5.4. Structuration des programmes

Un programme est décomposé en sous-programmes (PROCEDURE, FUNCTION, PROCESS) , non imbriqués.

Un sous-programme comprend des déclarations et un ou plusieurs blocs (BEGIN...END) de phrases.

Les phrases peuvent être des phrases simples ou des structures :

- alternatives IF THEN... ELSE,
- itératives WHILE... DO

  FOR... WHILE... DO

  FOR... UNTIL DO avec incrémentation d'index,
- de choix multiples CASE... OF.

5.5. Choix d'objets

Tous les objets utilisés doivent être déclarés.

On peut déclarer des objets simples :

- entiers 1, 2 ou 4 octets,
- fixes 2 ou 4 octets,
- réels 4 ou 8 octets,
- logiques 1, 2 ou 4 octets,
- booléens,
- qualité (attributs symboliques),
- chaînes de caractère,
- référence à un objet.

On peut également déclarer des objets structurés :

- tableaux à plusieurs dimensions d'objets simples,
- structures d'objets simples,
- tableaux de structures.

5.6. Opérateurs

Un large choix d'opérateurs permet de composer des expressions :

- opérateurs arithmétiques : +, -, *, /, DIV, MOD, EXP,
- opérateurs logiques : décalages droit et gauche arithmétique, logique et circulaire, et, ou, ou exclusif, complément,
- opérateurs de comparaison : égalité, inégalité, plus grand, plus petit, plus grand ou égal, plus petit ou égal.
- opérateurs booléens : non, et, ou.

5.7. Données dynamiques

Des déclarations et des phrases permettent de travailler sur des données dynamiques, soit sous forme d'objets structurés avec pointeur, soit sous forme de listes chaînées ("ensembles").

5.8. Compilations séparées

Des déclarations spécifiques permettent de découper une application en unités de compilation pouvant être compilées séparément.

6. QUALITES REQUISES POUR UN LANGAGE POUR CALCULATEURS EMBARQUES

6.1. Position du problème

Avant d'analyser les raisons d'utilisation effective du LTR, nous allons essayer de définir quelles qualités sont nécessaires pour qu'un langage et son système de production puissent être utilisés dans une large gamme d'applications embarquées.

6.2. Critère d'efficacité du code produit

Les calculateurs militaires embarqués sont relativement coûteux, et souvent, spécialement en aéronautique, on aura un grand nombre de systèmes ayant le même logiciel.

Il faut donc que le langage choisi et son compilateur ne soient pas une cause d'augmentation "sensible" du coût des calculateurs, et donc que le taux d'expansion soit raisonnable par rapport à une écriture en langage de bas niveau, aussi bien en volume mémoire qu'en débit d'instructions.

On notera que cette expansion ne doit pas seulement se mesurer au niveau du code produit par la traduction du programme utilisateur, mais aussi de tout l'environnement d'exécution ajouté pour pouvoir exécuter ce code.

De plus, certains systèmes n'ont besoin que de mécanismes moniteurs très rudimentaires (limités à quelques tâches d'interruptions ou quelques tâches cycliques). Il faut alors pouvoir avoir un environnement d'exécution très réduit.

6.3. Critère de couverture des besoins spécifiques

Pour couvrir de larges besoins et éviter autant que possible de recourir à une programmation bas niveau (ou à un langage spécialisé), le langage doit être assez riche pour couvrir des besoins très divers :

- traitements d'interruptions,
- entrées-sorties au niveau matériel,
- calculs en flottant ordinaire ou très précis,
- traitements de bits ou de caractères,
- synchronisations entre tâches.

.....

6.4 Critère de facilité d'écriture ou de modification des programmes

Un des objectifs essentiels d'un langage de haut niveau est de diminuer le coût de production et d'entretien des logiciels complexes. Un des critères de coût est la facilité d'écriture et de modification des programmes.

Les facteurs intervenant sont :

- la simplicité de la structure de l'application,
- le découpage de l'application en unités ayant des interférences limitées et bien séparées,
- la simplicité du langage et sa facilité d'apprentissage,
- la facilité de compréhension des programmes venant d'une bonne structuration et de la lisibilité du code,
- les sécurités introduites par le langage et le compilateur.

6.5. Critère de disponibilité d'une chaîne de production adéquate

Pour que le langage puisse être effectivement utilisé dans un projet il faut pouvoir disposer d'une chaîne de production suffisamment complète et dans les conditions suivantes :

- la chaîne doit pouvoir être disponible dans des délais compatibles avec le planning du projet,

- la chaîne doit fonctionner sur un calculateur facilement accessible par les réalisateurs du projet, et dans de nombreux cas sur un miniordinateur dédié au projet,

- la chaîne doit produire le code de la machine cible du projet,

- l'investissement pour obtenir cette chaîne doit être acceptable par le projet.

7. RAISONS D'UTILISATION EFFECTIVE DU LTR V2

7.1. Efficacité du code

La définition du langage a permis de réaliser des implémentations ayant un taux d'expansion réduit. Diverses mesures ont données des résultats de 1,2 à 1,3.

De plus, l'environnement d'exécution peut être réduit et performant : en effet pour des petits systèmes il peut ... ier de 10 à20 kilooctets (moniteur opérationnel multitâche et bibliothèque exécutive).

7.2. Couverture des besoins spécifiques

Le langage tel qu'il est défini s'est montré suffisamment riche pour couvrir la totalité des besoins des applications embarquées.

On notera cependant que certaines implémentations n'avaient pas implémenté la totalité du langage et que certaines applications ont écrit des bibliothèques de sous-programmes en assembleur pour pallier ces manques.

7.3. Facilité d'écriture ou de modification

La description des applications et les mécanismes temps réel introduits par le langage se sont révélés bien adaptés aux applications et simples à appréhender.

Le langage permet de découper l'application en unités ayant des interférences relativement limitées.

La syntaxe et les phrases du langage peuvent donner un code structuré et lisible si on suit des règles de programmation.

Enfin le langage s'est révélé facile à apprendre par des programmeurs ayant un peu d'expérience d'autres langages.

7.4. Disponibilité d'une chaîne de production adéquate

C'est le point qui a été le plus critique pour l'utilisation du langage. Cependant grâce à la politique de standardisation des matériels informatiques menée par le Service Central des Télécommunications et de l'Informatique, et aux investissements réalisés par ce service et les industriels, on dispose maintenant de chaînes de production opérationnelles fonctionnant soit sur un miniordinateur français (Mitra 125), soit un calculateur IBM pour les calculateurs cibles standards.

Il n'y a donc maintenant plus de problème d'accessibilité de chaînes de production pour la plupart des projets utilisant des calculateurs embarqués.

8. POLITIQUE POUR L'AVENIR

Le langage LTR V2 se révèle aujourd'hui un très bon outil pour la programmation des systèmes militaires opérationnels.

Cependant depuis sa conception, des progrès importants ont été faits sur les langages et les méthodes de programmation.

En particulier le langage Pascal conçu en 1970 a apporté des concepts très interessants, et récemment le langage ADA a regroupé beaucoup de nouveaux concepts.

Il n'est donc pas possible de se figer sur le LTR actuel, et il est nécessaire pour l'avenir de préparer un langage plus moderne et permettant une meilleure qualité de programmation.

Une solution serait bien sûr de retenir le langage ADA et d'attendre que sa définition soit suffisamment figée et son développement suffisamment avancé aux Etats-Unis pour pouvoir passer à une phase opérationnelle. Malheureusement l'examen de sa définition actuelle nous fait craindre qu'il ne puisse être effectivement utilisé pour des raisons qui seront détaillées par la suite.

Dans ces conditions une nouvelle stratégie est à trouver, et nous avons défini un nouveau langage, baptisé aujourd'hui LTR V3 qui, en assurant la continuité de LTR V2, intègre les progrès technologiques, et devrait répondre aux critères permettant son utilisation effective. Ce nouveau langage va être présenté, ainsi que ses critères d'utilisation.

9. POSSIBILITE D'UTILISATION D'ADA

Le langage ADA apparaît comme un ensemble très homogène et très complet qui devrait pouvoir avoir un vaste champ d'application.

Cependant, l'étude de son manuel de référence est extrêmement ardue, et le langage s'avère très complexe, ce qui déjà devrait poser de très sérieux problèmes de formation des programmateurs.

Pour l'efficacité du code produit, on peut douter de la possibilité de l'obtenir si on analyse les concepts tels que : les types variables, les contrôles de sous-types, les conversions de types, la sémantique des opérations réelles, les génériques, les entrées-sorties par type et par élément, les mécanismes temps réel...

La couverture des besoins spécifiques devrait être bonne, cependant on peut avoir certains doutes sur les points suivants :

- adéquation des mécanismes temps réels aux besoins,
- possibilité d'écrire l'environnement d'exécution en ADA.

Pour la facilité d'écriture ou de modification des programmes, ADA a des éléments intéressants. Cependant :

- les mécanismes de structuration des applications multitâches sont trop complexes et difficiles à appréhender,
- les programmes peuvent devenir totalement incompréhensibles si on abuse de notions telles que "NEW TYPE", redéfinition d'opérateur, surcharge, "renommage"...
- l'apprentissage du langage sera difficile.

Enfin sur le critère de disponibilité d'une chaîne de production adéquate, il semble aujourd'hui que les compilateurs ADA seront très volumineux, très chers à développer, et demanderont pour s'exécuter des calculateurs d'une certaine taille. De plus il sera nécessaire d'avoir autour du compilateur un grand nombre d'outils dont certains spécifiques d'ADA.

Dans ces conditions, il ne sera pas simple de rendre le langage facilement accessible aux divers réalisateurs de logiciel militaire, et le coût de l'investissement à faire peut se montrer rédhibitoire.

10. OBJECTIFS DU LTR V3

10.1. Définition des objectifs

Les objectifs du LTR V3 ont été :

- préserver les acquis du LTR V2 en reprenant ses éléments de base,
- introduire des améliorations technologiques,
- faciliter son utilisation,
- permettre son utilisation effective.

10.2. *Préservation des acquis du LTR V2*

Afin de préserver les investissements faits sur LTR V2 et assurer une certaine continuité, les fondements du LTR V2 ont été conservés :

- structuration des applications,
- mécanismes temps réels,
- structure des programmes.

10.3. Améliorations technologiques

Un bon nombre d'éléments intéressants du PASCAL ont été repris :

- la notion de type, *les types structurés*,
- la syntaxe,
- l'imbrication des procédures.

De plus un certain nombre d'idées nouvelles ont été introduites : modularité et visibilité partielle, types paramétrés, définition d'opérateurs, accès à la programmation système.

10.4. Facilité d'utilisation

Pour la facilité d'utilisation on peut noter :

- la définition d'un atelier de programmation,

- la simplification et l'homogénéisation de la syntaxe,

- des compléments tels que des chaînes de caractères variables, des entrées-sorties sur fichiers séquentiels ou directs...

10.3. Possibilité d'utilisation effective

Un des critères de base dans la définition du LTR V3 a été de permettre une implémentation efficace. Pour cela une étude d'implémentation a été faite parallèlement à la définition du langage, et le langage a été laissé volontairement simple et limité pour tenir cet objectif. De plus la reprise de la structuration et des mécanismes temps réel du LTR V2 devrait garantir la possibilité d'avoir un environnement d'exécution ayant le même ordre de grandeur de taille.

Les besoins spécifiques doivent être aussi bien tenus qu'en LTR V2 puisqu'on a veillé à offrir les mêmes services et ajouter des facilités nouvelles. On retrouve en particulier les mêmes mécanismes temps réel et les mêmes principes de structuration d'application, mais avec plus de souplesse permettant une meilleure sécurité. De nouvelles facilités ont été ajoutées telles que les paramètres de dimensions variables, des chaînes de caractères variables et ordonnées, des entrées-sorties sur fichiers et des possibilités de programmation système.

La facilité d'écriture et de modification doit aussi être améliorée par rapport au LTR V2 par :

- l'introduction de types et d'une syntaxe proche du PASCAL,

- l'homogénéisation et la simplification de la syntaxe,

- l'élimination autant que possible d'ambiguïté du code (constantes, opérateurs...),

- le découpage de l'application en modules s'ajustant à la structure de l'application et limitant les visibilités au juste nécessaire,

- la définition d'un ensemble d'outils standards d'aide à l'écriture et à la mise au point des programmes (atelier logiciel LTR V3).

Les choix faits au niveau du langage doivent permettre d'obtenir une chaîne de production compacte et de coût raisonnable. Les études d'implémentation faites montrent que cela est possible, et la première implémentation du compilateur (pour microprocesseur 68000) est prévue sur un Exormacs (microprocesseur 68000) avec 256 kilooctets de mémoire et un petit disque. Le découpage du compilateur et son écriture dans un sous-ensemble de PASCAL "portable" sous système UNIX devraient permettre de réaliser facilement des générateurs de code pour diverses machines cibles et de porter le compilateur (et son atelier logiciel) sur d'autres calculateurs de production.

11. PRESENTATION DES PRINCIPAUX CONCEPTS DU LTR V3

11.1. Description d'application

Une application peut être composée :

- de blocs de données de communication,

- de bibliothèques de sous-programmes,

- de programmes activés par appel ou par interruption,

- d'un programme de départ (START PROCESS).

Un programme peut être composé de blocs de données et de sous-programmes dont un de type "process".

Les différentes parties de l'application sont décomposées en "MODULE", un "MODULE" comprenant :

- une interface décrivant les données et points d'entrée visibles de l'extérieur,

- des références aux interfaces d'autres modules ("USE"),

- un corps de module (sous-programmes et blocs de données),

- des directives d'implémentation.

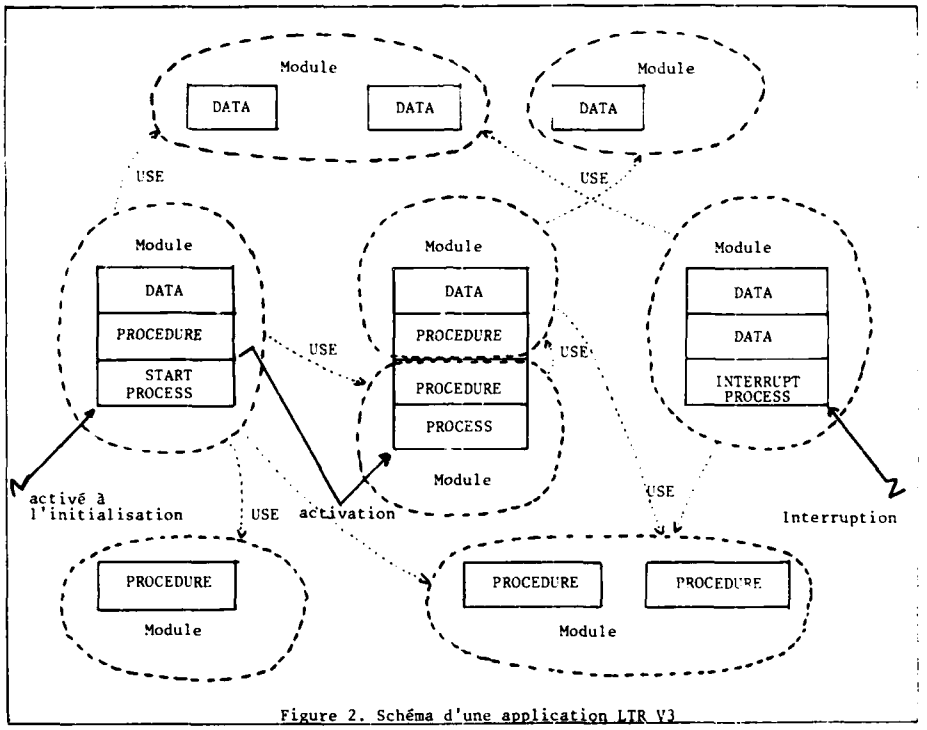


Figure 2. Schéma d'une application LTR V3

11.2. Sous-programmes

Un sous-programme comprend des déclarations et des traitements. Ce peut-être une "procédure", un "process", une fonction ou une définition d'opérateur.

La déclaration comprend les paramètres et leurs modes (en entrée, en sortie, en entrée-sortie et par valeur ou par adresse) et les cas d'exception.

Un sous-programme peut contenir des sous-programmes, un bloc de phrases et des handlers d'exceptions.

Les paramètres peuvent avoir des dimensions variables.

Le nom peut être surchargé si les types des paramètres sont différents.

Les opérateurs ne peuvent être définis que sur de nouveaux types.

11.3. Phrases

Les phrases simples peuvent être des affectations, des branchements, des appels de procédure, allocations-désallocations d'objets dynamiques, entrées-sorties formatées, entrées-sorties sur fichier séquentiel ou sélectif.

Les phrases composées permettent des structures alternatives (IF THEN... ELSIF... ELSE), sélectives (CASE), et répétitives (FOR... WHILE, REPEAT... UNTIL..., SEARCH...).

11.4. Phrases temps réel

Elles permettent :

- la création d'une tâche sur un programme,
- l'attente d'évènements ou de délais,
- l'activation d'évènements,

- l'accès à des sémaphores,

- le contrôle des interruptions,

- l'acquisition de l'heure temps réel.

11.5. Types d'objet

- types simples prédéfinis : entier, fixe, réel, booléen, caractère,

- types énumératifs ordonnés ou non ordonnés,

- types structurés (RECORD) avec variante,

- types tableaux à plusieurs dimensions (ARRAY),

- types référence à un type d'objet,

On peut définir des sous-types avec contrainte d'intervalle, de précision ou de discriminant de variante.

On peut également définir des types paramétrés contenant un tableau dont les bornes supérieures de dimension sont variables. Deux types paramétrés sont prédéfinis : logique (suite de bits) et STRING (chaîne de caractères variable).

11.6. Expressions

Un certain nombre d'opérateurs sont prédéfinis :

- comparaison par égalité ou inégalité quel que soit le type,

- pour types arithmétique : +, -, *, /, **, DIV, MOD,

- pour types booléens : NOT, AND, OR,

- pour types ordonnés (arithmétiques-énumératifs-caractère) : >, >=, <, <=,

- pour types logiques : CPL, LAND, LOR, & (concaténation),

- pour types STRING : & (concaténation),

Ces opérateurs peuvent être étendus sur de nouveaux types où ils ne sont pas définis par définition d'opérateur.

11.7. Facilités diverses

- toutes les constantes ont un type défini par leur syntaxe,

- on peut définir des constantes symboliques,

- on peut accéder à des tranches de tableaux à une dimension,

- il y a une bibliothèque de fonctions standards arithmétiques et trigonométriques,

- certains types peuvent être définis comme "opaques",

- à l'intérieur de modules "système" on dispose de possibilités de programmation système : type prédéfini WORDS, phrase WITH, entrées-sorties directes,

- le jeu de caractères est l'ASCII et est ordonné suivant l'ordre ASCII.

12. ATELIER DE PROGRAMMATION LTR V3

L'atelier de programmation doit comprendre des outils permettant d'aider à l'écriture et à la mise au point des programmes LTR V3.

Il doit comprendre :

- un chef d'atelier de programmation

- un bibliothécaire de texte source

- un gestionnaire d'objets

- un éditeur de texte syntaxique

- des préprocesseurs de texte source

- une souche de compilateur

- un générateur de code interprétable

- un interpréteur,

- des générateurs de code pour machines cibles,

- des éditeurs de liens et générateurs d'application pour machines cibles.

Cet atelier réutilisera si possible des outils standards existants, et sera pour le reste écrit en PASCAL (dans un sous-ensemble défini portable) sous système UNIX.

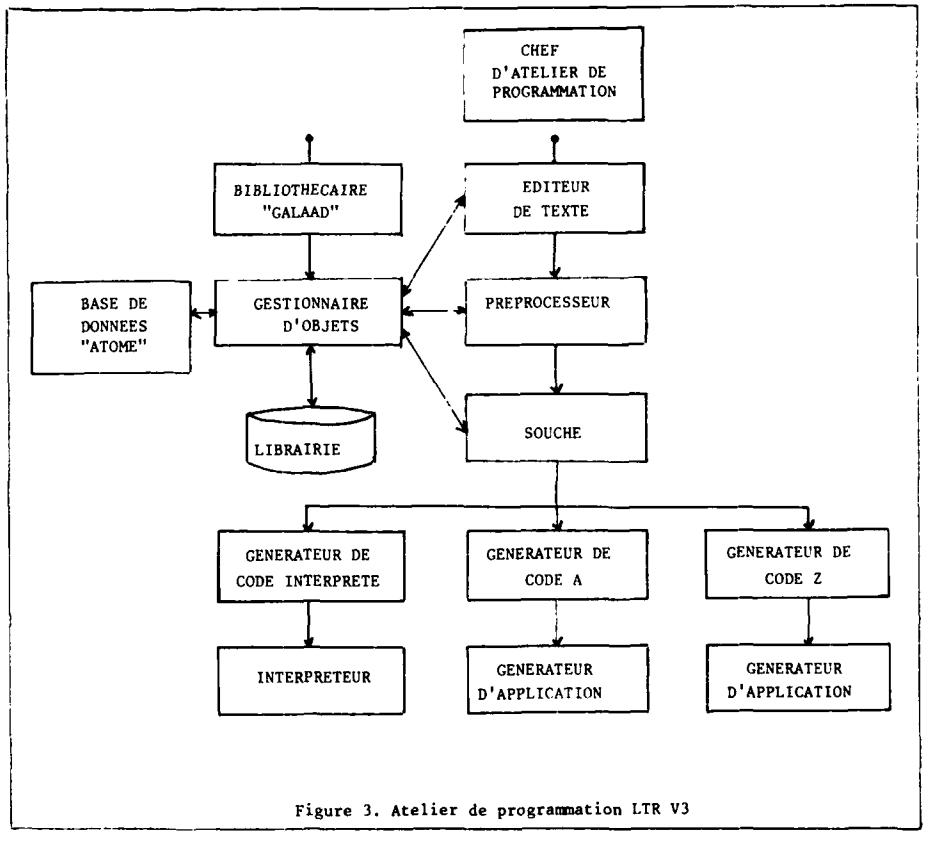


Figure 3. Atelier de programmation LTR V3

13. CONCLUSIONS

Afin de préserver l'acquis de la standardisation du LTR V2, une stratégie de rénovation du langage, appelée LTR V3, a été définie et étudiée. Cela semble être la meilleure solution apportant des garanties pour l'avenir. La phase de réalisation effective devrait maintenant commencer et fournir les premiers produits pour évaluations dans un délai de 2 ans. Parallèlement à cela, il faut maintenant définir les premiers produits opérationnels et les premières applications visées.

NOTA : La documentation sur le LTR peut être obtenue en écrivant à :

CELAR - 35170 BRUZ-FRANCE

OPERATIONAL FLIGHT PROGRAM DEVELOPMENT WITH A HIGHER ORDER LANGUAGE

R. E. Westbrook
L. L. Crews

Naval Weapons Center
China Lake, CA, USA

SUMMARY

This paper presents the problems and the future trends and solutions to many of the problems of developing Operational Flight Programs (OFP) for embedded computer systems. As OFPs become larger and more complex in nature the need for their development through structured design, higher order language (HOL) programming, and programming documentation standards supported by automated aids will become increasingly mandatory. Proper procedures and tools can remove much of the OFP housekeeping and manual effort while improving quality, reducing life-cycle cost, and reducing modification time to more readily respond to Fleet needs.

An HOL by itself will improve productivity during development and, in general, improve reliability of the software. But, coupled with an approach integrating (1) host execution and test, (2) instruction level target computer emulators (ILTCE) execution and test, and (3) target computer operation (under control of the simulated operational environment and test package), the OFP development time may be significantly shortened as well as providing substantial savings during the maintenance phase of the software life cycle.

This approach does not address the need for efficient compilers and computers that more closely execute the HOL operations. The fact that such HOLs as Ada and the Ada Programming Support Environment (APSE) have been defined and are being developed, will not negate the need for good software engineering methodologies.

1. INTRODUCTION

The development of large-scale software systems such as OFPs requires discipline in all of its stages. The long lifespan of a large software project makes mandatory maintenance of both the software and its documentation, and the requirement for a *central and disciplined configuration control mechanism*.

The requirements for large software systems such as OFPs can become extremely burdensome as the life of the project continues. Improper design, lack of maintenance, or the absence of configuration control can quickly lead to a software system that is virtually impossible to maintain, and certainly impossible to continue to upgrade and modify in a cost-effective manner.

One *major method* to help with these problems is to automate the required activities by using a modern HOL such as Ada. HOLs not only aid in structured design, increased productivity, and better documentation, but also ensures that many unstructured designs cannot exist.

A test bed for proving many of these concepts is the A-6E OFP which is being rewritten in an HOL with new design methodology. There will be enforced standardization by use of software tools through all stages of the software cycle.

2. BACKGROUND

Most of the present day OFPs were written many years ago. They are complex assembly language programs exhibiting many negative characteristics.

OFPs suffer from lack of HOL availability because no compilers generate code for the particular embedded computer, or the HOL compiler generated inefficient code both in terms of time and storage. Time and storage are so critical that even the assembly language OFPs are filled with areas of tricky code placed there to "save" a core location or "speed-up" some function.

During the long maintenance period (sometimes 15 to 20 years) whatever software structure existed initially is quickly lost. This may be blamed partly on the computer being out of time and out of storage when the system is delivered, and to some extent on the lack of proper modularization and control of module interfaces.

Most OFPs rely on very experienced key people to make even minor changes. This poses training problems and increases risk due to personnel turnover.

Software changes are made via unique, semi-manual support systems. Limited verification and validation is performed in unique facilities using actual avionics mounted in test stands simulating flight computer input/output conditions.

3. PROBLEMS

The OFPs are the heart of the aircraft avionics system and must interface to a multitude of avionic subsystems such as stores management, radars, electronic countermeasures, inertial navigation, and operator displays. The OFP must make decisions about many complex moding functions based upon the inputs of these subsystems. Due to these many different inputs, outputs, and modes, a detailed and complex software requirement document must be written before progressing to the OFP development stage. Historically, a *complete software requirements document* has not been developed, this creates a larger problem during the development and future stages of the software development cycle where requirements must be changed or modified.

OFPs are usually developed and documented in a nonformalized, technical data approach where groups of engineers develop and program each of their certain expertise areas. This type of development leads to poor documentation. Notes are usually kept by the engineer in his desk and may never be published in a formalized document. The end product is usually an unmodifiable and very costly product requiring highly technical engineers who must retrace and document the development cycle of the OFP in order to intelligently modify or maintain it. This type of software development increases the life-cycle cost considerably in the case of OFPs where the majority of the life-cycle time is spent in the operations and maintenance stage.

Many of the problems are attributed to avionics computer size limitations and code inefficiency. However, the reality of the situation is that many of the problems associated with the OFPs can be overcome by the use of HOLs, automated aids, and modern software programming and management techniques.

Testing of the OFPs is difficult due to the manner in which the software is developed. Also there are no software systems or tools that can be used for testing. It is almost impossible to test a program written in a fixed point assembly language that is in a black box, has a very minimum of peripherals, and hardly any capability for monitoring the software execution. Consequently, the major portion of OFP validation is performed using expensive flight testing.

During the total life-cycle of an OFP the maintenance phase is by far the most costly (RUBEY, R. J., 1978). The typical pattern for a U.S. Navy tactical aircraft is for the software to be "upgraded" on a 2-3 year cycle as the aircraft responds to the Navy's changing requirements. Very few errors are found and corrected during maintenance; most of the changes are requests for improvements or enhancements, new avionics, or new weapons for the aircraft. HOLs and the tools supporting HOL OFP development are critical if the Navy is to reduce the cost and improve responsiveness to change (GLASS, R. L., 1980).

4. TRENDS

There are many trends being developed today that will enhance and streamline the process of OFP development and maintenance. There is an emphasis on HOLs that support real-time environments. These in turn will lead to programming techniques, documentation standards, and automated aids that will improve the quality and lower life-cycle costs and modification time for future OFPs. Hardware size limitations are disappearing and software management is becoming prevalent with tools that aid in the process of software development. There are new structured design methodologies that achieve modularity, abstractness, and information hiding while still producing efficient software code. There are software support systems being developed, such as APSE, that provide tools for the entire development cycle. Software engineers are becoming more aware of the problem and are starting to use *many* of these new techniques and tools to develop OFPs.

The tools available for software development are numerous and can be beneficial to the engineer. Most tools currently support only one stage of the software development effort. Many of the new systems are promoting the concept of a software development system where the outputs of one stage, are the inputs to the next stage, and all input and output from each tool is stored in a central data base addressable by all tools.

One of the long-term prospects from using an HOL is the idea of reusable code where routines are like hardware chips. An engineer can collect the routines and build an avionics system without rebuilding each routine from scratch.

4.1 Examples of Future Trends

4.1.1 Software Engineering Environment (SEE)

The SEE must support the entire life-cycle of the system. From early requirements analysis through delivery of the OFP and future modifications, the SEE must support documentation, development, debug, test, configuration management, and quality assurance.

The SEE must support a common set of tool products to avoid being uniquely developed for each embedded system. Such tools act as cross compilers for target machines that also generate code for execution on the SEE host. If the SEE and its tools are portable, other projects could more easily host the SEE on their computer equipment. Cross training between projects is possible.

The SEE must be extensible to encompass new tools, methodologies, HOLs, targets, and hosts. This will encourage new projects to consider improvement rather than developing specialized SEEs. If the SEE's command language processor handles the interfaces to the host computers operating system, the SEE will appear the same to the user no matter which host computer is used.

4.1.2 Software Tools

As part of the SEE a collection of software tools is necessary to provide the support required throughout the project life-cycle. One of the major requirements is that the tools make use of a common data base; this is not done in most of today's environments.

4.1.3 Higher Order Languages

Many HOLs have been developed by the U.S. and other countries. Each have numerous strong and weak points. The important underlying motivation is to describe commands to a computer in a manner more understandable and more natural to the human programmer. Implicit in this is increased productivity, increased reliability, and a better way to describe the solution (a program) to other engineers.

4.1.4 Computer Hardware

Computer hardware existing today and qualified for use in tactical aircraft have two basic problems that need to be corrected before full HOL utilization is possible. Technology is making the physical memory space available, this problem should disappear quickly. However, the solution to the problem of executing HOL programs efficiently will not be completely solved with instruction execution speed-up, but will require computers that directly execute the HOL.

In order to make fully controllable testing possible with target computers additional control and monitoring interfaces are necessary for integrated use with host SEEs and simulation facilities.

4.1.5 Reuse of Code

Many of the functions performed by OFP are similar, if not identical, this leads to the idea of reuse of code. For code reuse to be practiced, HOLs must support several software engineering principles. There must be a very good concept of modularity. There must be the ability to hide implementation details and other information while providing the required interface at a sufficient level of abstraction to allow flexible integration into many programs. Requirements definition and documentation must be complete and accurate, and the code proven true.

5 A CONCEPT FOR HOL OFP DEVELOPMENT

There is little disagreement within the software engineering community that (1) if a sufficiently powerful HOL were available, (2) it was supported by a good SEE, and (3) it compiled concise and efficient code for each target computer, all OFPs would be written in that HOL.

An example of what constitutes a modern tactical HOL is given in U.S. Department of Defense Steelman Specification for Ada. Another example for ideas concerning programming support environments is given in the Pebleman Specification.

However, for software development of aircraft, missile and electronic warfare OFPs other concepts are practical even with today's languages and support environments. For instance, an HOL that would be used for analysis during the requirement and early design phase of OFP development should be the same one used during the actual OFP development (Figure 5.1). This would seem obvious except that many HOLs targeted for embedded computer systems (ECS) do not generate code for the host machine. Secondly, the opposite is true: most HOLs targeted for suitable host computers do not generate code for ECS. Manual compilation, although error prone, is preferable to not using an HOL.

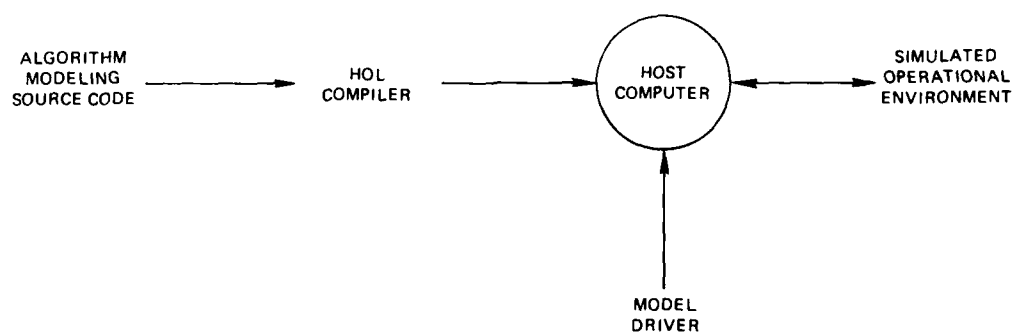


FIGURE 5.1 Algorithm Analysis: A Part of the OFP Development Process.

If the OFP is originally written for the host computer and interfaced with a simulated environment with proper test drivers, it would seem logical for an ILTCE interfaced with the same simulated environment and test drivers to do most of the development before any "real" hardware were part of the OFP development or test.

The third step would be to utilize the same source code cross compiled for the actual flight computer (there may be test hooks needed for the ILTCE) where the flight computer is interfaced with the host computer operational environment and test drivers.

There exists today several HOLs, some of which are suited to the OFP environment, that compile code from "identical" source code for host, ILTCE, and target computer. It is the use of an operational environment simulation and a set of test drivers integrated with the host executed OFP, followed by ILTCE executed OFP, and target computer executed OFP that allows for a structured stepwise OFP development and test (Figure 5.2). Because only limited hardware is utilized during the three phases of development, the real-world can be slowed to facilitate test probes and data monitoring and collection, increasing the development view into the entire operational environment.

This approach of course, requires an up front investment in software tools such as compilers and ILTCEs that might not be necessary otherwise. But the visibility, flexibility, and ease with which an OFP may be developed (as a set of modules coded in an HOL, integrated into a total program, and tested as modules that are integrated individually) offers a structured development approach where the code remains virtually unchanged from the design analysis phase through the operation test phase.

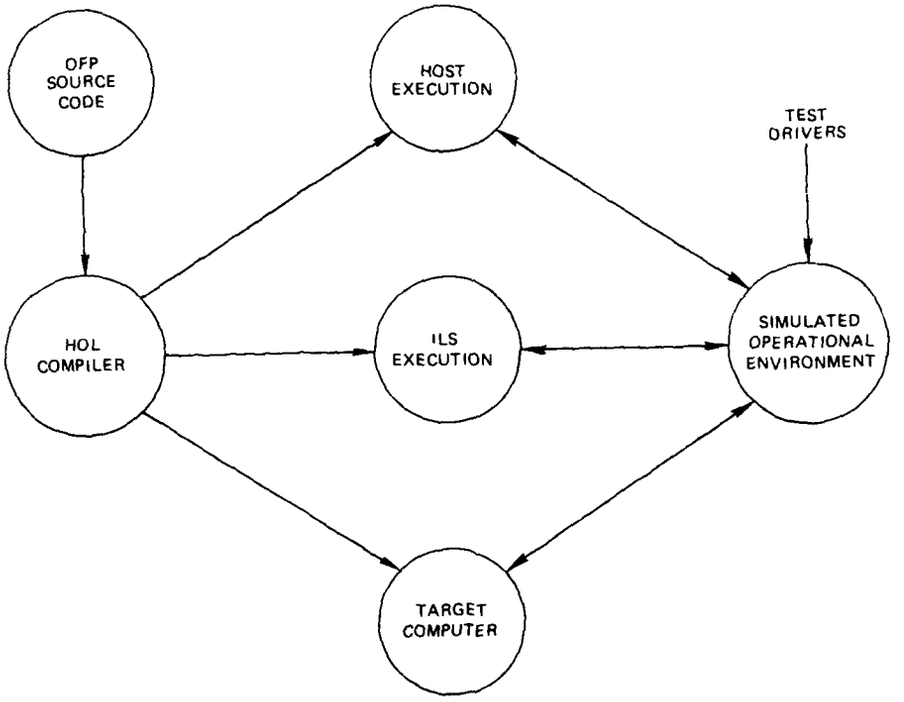


FIGURE 5.2 Stepwise OFP Development and Test.

6. EXAMPLES

DAIS The DAIS project is the first totally digital OFP written in an HOL. The verification was performed in a large digital computer with the simulation models before actually being loaded into the flight computers for final checkout. However, automated aids were not used for every stage of the software development cycle.

AN APPROACH TO A PORTABLE PASCAL LANGUAGE FOR DIFFERENT ONBOARD COMPUTER SYSTEMS

ST. REITZ
W. WIEMER

MESSERSCHMITT-BÖLKOW-BLOHM GMBH
Unternehmensbereich Apparate
Postfach 80 11 49
D - 8000 München 80

Abstract

In order to improve and to optimize software-development for microprocessors it is necessary to produce portable software, including vertical (different host computers) and horizontal (different target processors) portability.

In addition the method of structured programming should be applicable.

Programs, written in PASCAL, are developed and tested on different host-computers. To run the programs on the target microprocessors the generated intermediate code is converted into specific code.

The paper describes the objects of the implementation and way leading to a PASCAL Programming-System. The selected HOL is PASCAL. The choice was made for the P4-Code compiler written in PASCAL, so that the extensions can be easily inserted.

The extentions are orientated on the IEEE-Standard, presented in the IEEE MICRO (May 81, P 70).

Special realtime requirements are also planned.

The final Programming-System is divided into five parts:

1) Extended P4-Compiler
2) P4-Code Linkage
3) Optimizations and table-driven codegenerators
4) Target system simulator
5) Target operating system

The codegenerator consists of one package, which forms from the architecture description the table, which is used as input for the table-interpreter. Its output is the microprocessors'code. The following processors are used to run program:

a) AMD2901 Bit-Slice Processor
b) MC 68000
c) Intel 8086/8087
d) TI 9900

Optimization on the PASCAL-Source-Level is not planned. But the P4-Code sequences have to be optimized in order to produce fast and short codes.

TABLE OF CONTENTS

1. Development of microprocessor-assisted systems

Microprocessors are used increasingly to improve the efficiency and intelligence of technical systems. They are suitable for carrying out control activities and monitoring functions. Software development accounts for a considerable amount of work.

However, other than in hardware development, engineering methods have not yet experienced a general breakthrough in the development of software, and the coordination of software and hardware development thus causes considerable headaches. Moreover, many system designs are developed within the scope of multinational projects. This leads to additional communication problems and has an adverse effect on cost development.

The systematic application of software engineering methods and harmonization of hardware and software design methods and instruments constitute a first step in overcoming these problems. The improvement in communication between hardware and software designers and within project management is another goal of vital importance. Communication is closely linked with the documentation.

1.1 Development process

If we compare the conventional hardware design method and the software engineering procedures (Koch, 79) we recognize that they are based on the following common features:

- top down design
- modular system structure.

Since hardware as well as software projects cannot be implemented without a phase concept, we should start to improve the development process at this point. The preliminary hardware and software design phase can be implemented in parallel. The communication interfaces are described in detail in para. 1.3.

However, top down design and modular system structure are only applicable, and hence useful, if the user software is written in a HOL.

Harmonizing the development methods for the hardware and the software without, however, affecting their independence from one another must be aimed at when improving the development process. This implies that the processor-dependent features are reduced to a minimum in the user software. The life of a microprocessor has proved to be particularly short where technical systems require constant adaptation to the latest state of the art. This entails redesign and reimplementation, if the software is programmed in the assembler code only. The relationship between development effort and lifetime needs to be improved. A portable HOL source is used to provide decoupling of hardware and software.
To permit estimation of the costs for adaptation, it must become clear which parts of the user programs are not portable.

The time of development can be influenced considerably by suitable design of the programming system.

The programming system comprises all components and tests required to edit HOL programs.

1.2 Development tools

The software is designed and tested on the development computer with the aid of the programming system. The microprocessor software can be tested on a development system or on a host computer. The advantages and disadvantages of both media are illustrated below.

	Host computer	Development system
Protability: User software	yes	no
System software	yes	no
Debugging aids:		
Symbolic debugging	yes	yes
In-circuit emulation	no	yes
Validation	yes	possible
Software tools	many	few
Multiuser Communication (data, documentation)	very good	impossible
Scope of project	unlimited	limited
Shared resources	yes	no
Access to storage media	rapid	very slow, laborious

This illustrates that many arguments speak in favor of the host computer principle, in particular

- unlimited scope of project
- good communication
- portability of system software
- test methods.

However, the development system offers an additional test method, namely in-circuit emulation.

The host computer provides two methods of developing portable software:

- through cross assembler
- and cross compiler.

MBB developed the UMICAS cross assembler. Cross compilation described below is a preferable but more sophisticated and costly method.

1.3 Development phase documentation

The documentation is the link between project management, and hardware and software development.

Hence, the major part of the hardware and software design can be run in parallel (Fig. 1).

Operability of hardware and software can thus be tested by mere comparison of the test runs.

If, in addition, the user software is established in a structured HOL, self documentation is ensured.

2. Field of application

2.1 Definition of the language

As already indicated in chapter 1, the first step in improving the software quality consists in establishing the programs in a HOL.

In general, the disadvantages of a HOL comprise:

- longer execution times
- additional memory requirements.

These shortcomings can be offset by suitable optimizations and by improving the efficiency of the target processors.

A runtime and operating system is required for executing the translated user programs. Whereas for program development the operating system of the host computer is used, the object program is supported by the target operating system. This operating system can be written in a HOL, thus increasing the portability of the programming system.

ADA, PEARL and PASCAL are available as a possible choice for the selection of a HOL. On the multinational level, PASCAL is the only reliable language available.

The PASCAL compiler is written in PASCAL, so it is portable and suitable for extension.

Unfortunately, real-time language elements are not available; this lack can, however, be easily compensated for by adding language extensions.

2.2 Definition of the field of application

The definition of standard PASCAL is not fully sufficient for its use in all intended fields, covering:

- microprocessor application
- applications in control engineering
- description of operating systems.

Additional language extensions must be implemented in compliance with the grammatical characteristics. As some of the new language elements relate to pecularities of the microprocessor, the PASCAL language philosophy is no longer strictly adhered to.

The extensions are largely based on PASCAL compilers which have already been modified. Special language elements required for microprocessor application have been defined according to the IEEE standardization (IEEE, 81) proposal.

Based on the IEEE standard, the following additional characteristics are desirable for microprocessor application:

- direct memory access (PEEK, POKE)
- support of special input/output operations
- interrupt handling (DISARM, ARM)
- access to registers of the processor (MEMLOC, PUTREG, CALLEK)
- bit manipulation (type of data: BOOLEX)

Applications in control engineering call for

- a fixed point notation (fractionalized, for higher processing speeds)
- overflow correction.

As the extended PASCAL language shall serve to describe the flow of the control of processes and the operating system, new control structures (Fig. 2) have to be established.

The process constitutes the active element. The system takes over the coordination of the processes. As processes can be executed in parallel, the system must be supported by an operating system. Both system and process access routines which are defined externally and combined to form a unit.

When combined, the extended PASCAL language comprises the following basic units:

- system
- process
- unit
- procedure
- function.

All of these five basic units can be compiled separately, whereby procedures and functions can be combined into a unit constituting an intermediate code, or form separate assembler code programs.

Some of the extensions suggested here comply with ADA language elements:

unit -- package
process -- task
fixed point -- fractionalized.

3. Development of the PASCAL system

Below are the four main characteristics which determine the design and implementation of the PASCAL programming system:

- flexibility
- portability
- modularity
- efficiency.

3.1 System structure

The PASCAL source programs are analyzed by the P-compiler and translated into P-code programs. The basic units are linked in the subsequent linkage run. The code generator transform the compiled and linked program into the executable code with the aid of the tables built by the code table generator. For test purposes, the interpreter can already evaluate programs in the P-code language. Execution on the target system is controlled by an operating system and a monitor.

3.2 Qualitative system requirements

3.2.1 Flexibility

Flexibility is a yardstick for the capability of the system to undergo modifications. The impact of functional modifications on the system must, for instance, be reduced to a minimum. This requirement affects the modular structure of the system. System components which will be subject to frequent modifications must already be taken into account in the design phase. Therefore, all components accessing target processor-related information have to be regarded with particular care and must usefully combine the requirements of the user and the system developer.

The components receive all machine-related information via the general input interface. This interface must be designed to permit transmission of data to any processor without affecting the internal structure. System parts which exclusively use machine-related characteristics have to be specified as autonomous modules.

Hence, the system must be clearly split into machine dependent and non-dependent parts.

The user provides the following information to the programming system:

- the PASCAL source program
- linkage editor information
- information on the target system which is required to generate the tables.

The target system communicates with the user via the monitor. The user should provide all information which the programming system requires to generate any desired target processor code without modification. The procedure applied is as follows:

- machine-related information is transmitted by means of tables with a fixed architecture;
- abstraction levels are created with the result that the source language is not translated directly into the object code.

In consequence, modifications made on the target system side have no direct impact on the structure of the programming system.

The tables are broken down into groups of information. The individual parts are chained by means of references. Since the abstraction levels are formed as PASCAL is converted into the P-code (K.V. Nori, 76), a list has to be generated in the first instance enumerating the valid P-code instructions. These instructions serve to obtain the block lists which describe the architecture of the target machine. The block lists refer to the code blocks, which describe the instruction architecture of the target machine.

The above-described procedure serves to

- reduce the redundancy of the tables,
- delay the generation of the final bit pattern, so that optimizations can be inserted with the aim of improving the runtime.

Decoupling of table entries and architecture enables the user to set up his tables without knowing the architecture. For this purpose an additional system component is required which receives the relevant information by way of interactive communication and thereby checks the inputs.

All measures to improve flexibility are only useful if

- the selected intermediate language can be fully described by means of tables, and
- the command structure by means of generator instructions.

These two requirements must be regarded.

The characteristics of the P-code are as follows:

- the instruction set is fully defined (as for assembler code);
- information on the source program is lost;
  - o it is extremely difficult to identify control structures,
  - o the names of the data objects have been lost,
  - o compound data structures are completely resolved.

The P-code instructions define a simple stack mechanism without accumulator and index register. This mechanism can easily be mapped on the target machine.

The relatively small extent of the instruction set (- 100 instructions) seems to favor the code table approach. Although the P-code language is rather limited in its extent, it permits using the assembler instructions of the target processors despite of their much larger capacity and compactness, since P-code sequences sometimes can be represented by a single assembler instruction.

Prior to refering the second requirement, the following example shall serve to illustrate the principle described:

	Block list		Generator instruction
	.		
	.		
	.		
P-code instruction ---			
	MOVO30	---	LDI MVA basic pattern
	.		
	.		MOVEA
	.		OR MOD1Q address mode
			OR REG4Q register source
			OR REG1Z register target

The block list element MOVO30 refers to a list of generator instructions which, upon execution, form the bit pattern for the MC68000 instruction MOVEA.L A4, A1.

```
   MVA    =    7000
OR          OR
   MOD1Q  =    0008
OR          OR
   REG4Q  =    0004
OR          OR
   REG1Z  =    0200
-------------------
               720C
```

This method only applies to regular instruction architectures which are characterized by few exceptions. Unfortunately, not all target processors fulfill this condition. However, special generator instructions can be defined to cover these exceptions.

3.2.2 Portability

Portability is a measure expressing the transferability of a program from one software or hardware environment to another. A distinction is made between horizontal and vertical portability, the former signifying the portability of the user software with resepct to several different target processors, the latter meaning portability of the programming system with respect to different host systems.

Vertical and horizontal protability is accomplished automatically by implementing the measures described in para. 3.2.1.

Portability of the programming system is measurable, since the design makes a clear distinction between machine-dependent and non-dependent system parts.

3.2.3 Modularity

Modularity relates to both

- the basic language units and
- the system parts.

The basic units of the extended PASCAL language must be compilable and permit testing independently from one another, regardless of whether the basic unit exists as a P-code object or an assembler program. However, there are certain conventions which the assembler code must observe.

Modularity, or rather modular structure of the programming system is a basic condition for a portable and flexible system design. It means decompostion into autonomous functional substructures which only use a single input and an output interface, respectively, for external communication. If possible the internal functional structure should not be visible from outside. This leads to a considerable increase in system flexibility. The programming system is decomposed according to its functional structure.

3.2.4 Efficiency

Efficiency, i.e., low runtime and low memory requirements, are conditions that must be fulfilled by the user software when processed in the target system. Special design decisions affecting the runtime and memory requirements of the programming system have not been made.

Since the user programs are executed under real time conditions, the requirements to be met by the target system with respect to runtime and storage capacity are very high.

The first implementation of the system is planned to comprise two steps. The P-code programs are too comprehensive without compression and must be condensed by means of peephole optimization. At the same time, the runtime condition can be improved, within limits, by using special instructions of the target system. A number of P-code sequences can be replaced by such instructions. Peephole optimization is accomplished by means of specified transformation trees.

The runtime condition is systematically observed prior to the output of the object code. In this phase, many data transfers occurring between storage cells are intended to be replaced by register transfers.

PASCAL statements, such as the For statement, involving a long runtime, are thereby subject to particularly close observation.

General procedures do not exist currently for that phase, so that a final specification is still pending.

REFERENCE

G.R. Koch, 79 — Systematisches Softwareengineering für Mikrocomputer Elektronik Heft 21

K.V. Nori, u.a., 76, — The PASCAL -P- Compiler Implementation Notes, Bericht ETH Zürich, Zürich

N.N, 81, — A Proposal Standard for extending High-Level Languages, IEEE Micro

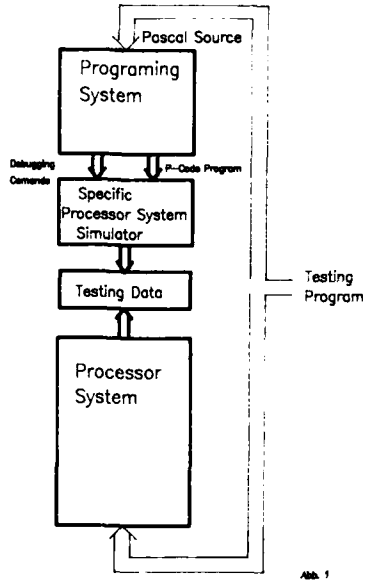


Abb. 1

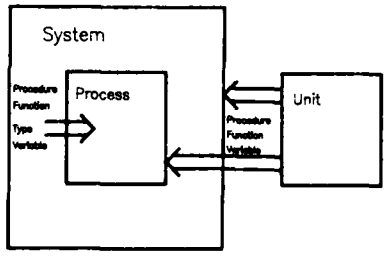


Abb. 2

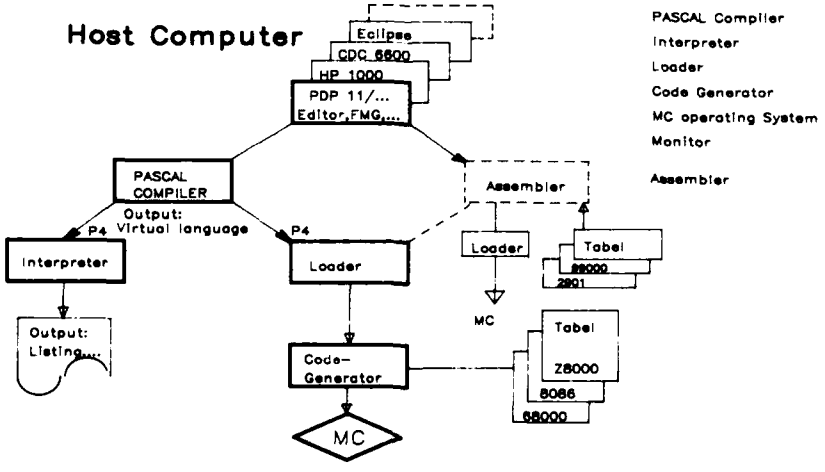


Abb.3 Portables Softwarekonzept von MBB

THE USE OF HIGH ORDER LANGUAGES ON MICROPROCESSORS

R.M. BOARDMAN

Marconi Avionics Limited
Elstree Way
Borehamwood
Herts.

SUMMARY

This paper deals with the use of High Order Languages on Microprocessors. It describes the special features of the software tools - compilers, loaders, etc. - and support environments which are highly desirable if the High Order Languages are to be used effectively.

It discusses the impact of High Order Languages on Programming and Documentation techniques and considers the implications for both Off Line and On Line Testing. It identifies and discusses certain disadvantages of these techniques.

The paper is concerned in particular with microprocessors which are embedded within larger systems and which are dedicated to specific tasks, for example display drivers or signal processing units. For the majority of systems these tasks are real time. For most avionics applications the microprocessors are fitted with the minimum amount of memory, usually a mixture of ROM and RAM, and have a minimum of interfaces with the outside world. These interfaces are frequently special purpose.

The practical application of a High Order Language to this type of microprocessor based project is described in the form of a Case Study.

1. HISTORICAL BACKGROUND

The advantages of using High Order Languages on conventional software systems are well known. Compared with Assembler Language programs, High Order Language programs are easier to write, easier to test, easier to document and easier to maintain.

It is surprising then that, until very recently, these advantages have been exploited rarely for microprocessor based software. This, however, is seen to be partly due to the type of problems that have been tackled, partly for historical reasons and partly due to the lack of suitable languages and compilers.

Early microprocessors were not very powerful. They had only a limited instruction set and could only address small amounts of memory. The tasks for which they were used were comparatively simple, requiring less than a thousand instructions and quite small amounts of data for their solution. The programs were frequently time critical and needed the precise timing of Assembler Language. Such programs did not immediately warrant a High Order Language solution.

The requirement to use these simple microprocessors was usually determined by the engineer who was responsible for designing a complete unit and who specified a microprocessor as a component in a larger piece of electronic equipment. It was customary for him to write the necessary programs himself in machine language and to debug them in the way he commissioned the rest of the equipment. Such an engineer, while expert in his own field, did not always appreciate the problems of software maintenance or the advantages of a High Order Language.

In addition, very few High Order Language compilers were marketed for microprocessors and those that were available did not always allow programs to be split into ROM and RAM areas.

2. CURRENT NEED FOR HIGH ORDER LANGUAGES

Modern microprocessors have extremely powerful instruction sets. They can address more than a hundred thousand bytes of store and are as fast as some minicomputers. Consequently they can be used to undertake large and complex tasks and so the volume and cost of microprocessor associated software is increasing rapidly.

The use of a High Order Language for program development is an important step towards containing costs and producing a more easily maintainable program.

High Order Languages make it easier to use techniques like structured design and modular programming and these in turn make it easier to produce larger programs, since they allow several programmers to work on the system at the same time.

High Order Languages discourage such techniques as writing self-modifying code or using coincidentally occurring binary patterns as constants, both tempting concepts to the 'novice' Machine Language programmer. These methods may save a few bytes of store but they make the system nearly unmaintainable in the absence of the original author.

Larger programs need better documentation. A well commented High Order Language listing is a useful part of a documentation package, but it is nothing like sufficient.

Documentation should start with a firm detailed specification, written and frozen before coding starts, and continue with a description of how the program is broken down into functional units and then into modules. It should be completed by a description of how the system is linked together and the tests that have to be carried out to prove that the software meets its specification.

All this is normal good programming practice. But microprocessor based systems need some additional items.

The functional specification should be extended to describe in detail the target system on which the program is to run. This description should include memory maps showing where ROM and RAM will be placed and descriptions of the interfaceswith special devices. This should cover information about features like status messages; reserved locations, timing constraints and error conditions.

The fact that some of this information will probably change during development is an important reason for including it in the specification rather than an excuse for leaving it out.

Some means for verifying and identifying the PROMs must also be specified, since one unlabelled PROM tends to look very like another. The minimum should be the inclusion of a checksum and the name, date and version number of the system.

Another extra item which should be included in the documentation package is a description of how to prepare a final system in ROM. This should include compilation options, information needed by the linker and instructions on how to blow the system into PROMs.

3. THE NEED FOR A HOST SYSTEM

Most microprocessors used in avionics systems are embedded in other equipments and have a minimum of memory and few standard interfaces. These target systems with their limited memory and lack of peripherals are totally unsuitable for running a compiler or developing large programs. Hence a host system is required. This system may be a development system supplied by the chip manufacturer or it may be a general purpose computer.

A manufacturer's development system usually consists of a microprocessor chip with some memory, attached to a floppy disc unit, a V.D.U. and a printer and supplied with a simple operating system which turns it into a minicomputer.

In some cases this minicomputer can be plugged into the target system thus gaining access to any special interfaces. Such systems are invaluable during final commissioning stages because they allow a programmer to find out why a system is misbehaving and to make and test changes quickly and easily. But they are expensive, they can support only a small number of programmers and are limited to the one microprocessor.

A general purpose computer, on the other hand, can provide filing and editing facilities for a number of programmers and can support several different types of microprocessor. However, it requires a cross-compiler for each microprocessor supported as well as a compiler which produces code which will run on the host system.

Choice of a host computer is limited by the need to transfer the code from the host to the target. Down-line-loading facilities can be awkward because they tie up the host computer and common media between equipment from different manufacturers are very rare.

4. SPECIAL FEATURES REQUIRED IN A HIGH ORDER LANGUAGE SYSTEM

Whether a Microprocessor Development System or a General Purpose computer is used for software development, there are certain special features which the High Order Language System will need. In the context of this paper High Order Language System includes the linker and loader as well as the language compiler itself.

These features fall into two categories. Features which enhance the ability to produce efficient code which will run on the target machine and features which will allow programs to be tested adequately on the host system.

To produce code which will run efficiently on a microprocessor the High Order Language must not be too "High Order". It must allow the programmer to use "assembler-like" data structures, for example BYTE ARRAYS, to carry out logical operations on single bytes, for example "AND", "OR","EXCLUSIVE OR" etc., and to use non-floating point arithmetic.

In order to drive the special interfaces the programmer must be able to address absolute locations and to include short sections in Assembler code in order to allow fast and precisely timed sequences of operations.

Where the target system uses ROM is must be possible to divide the memory used by the compiled program into three sections:- instructions, constants and variables, and also to be able to re-locate these sections anywhere within the address space of the microprocessor so that instructions and constants can be placed in ROM and variables placed in RAM. It must also be possible to connect blocks of instructions to the appropriate interrupts to form interrupt service routines.

The compiler should produce an Assembly Language listing of the code produced, which should include comments which link blocks of instructions to lines of High Order Language text. Such a listing is invaluable during the final commissioning stages when it is usually necessary to stop the program at particular points and to inspect the contents of variables.

To produce code which can be tested on the host system the High Order Language compiler must be able to include or exclude portions of source text. For example, it may be necessary to write characters into an absolute location to drive a particular display. However, on the host machine such a display will not exist and to write to an absolute location would be meaningless. Hence the instructions to write characters to this location must be replaced by instructions to write characters to a standard display. During the testing phase the program will be continually altered and recompiled either for the host or for the target system and it should be possible to choose either one set of instructions or the other without altering the program.

5. PROGRAMMING TECHNIQUES

Programming techniques follow normal good High Order Language programming practice but need to be extended to take account of the special features of microprocessors. These are the need to test the program on both the host and the target systems and the need to produce an efficient program.

The text replacement features mentioned in the previous section are used to bridge the gap between the host system and the target system. Code segments applicable only to the target system are replaced by equivalent High Order Language text which is meaningful on the host system and extra I/O statements used only on the host system are added while common modules remain unaltered.

Modular programming is very important on microprocessor systems because of the number of special interfaces and special devices used. Each special interface or device should have its own dedicated module which can be checked out as soon as the hardware is available whether the rest of the programs are ready or not and if one of these interfaces changes during the development only a single module need be altered.

Another good idea is to give hardware dependent features, such as absolute addresses, symbolic names which are linked to the absolute value only at one place. This has two advantages, firstly if the feature changes only one line of code need be altered and secondly a programmer trying to maintain the system in years to come is not left to puzzle over the meaning of a strange constant which seems to pop up all over the place without explanation.

Where a High Order Language is used in a microprocessor system it is important that the compiler is used as efficiently as possible. It frequently happens that there are several ways of writing programs in a given language to produce the same results. For example, data can be held in either one two-dimensional array or in several one-dimensional arrays. Frequently there is little to choose between the various solutions from a source code point of view but there are often large differences in the amount of code produced and in the running time of the resulting program. In the previous example each use of a two-dimensional array might generate a multiplication and an addition as well as an indexed load whereas the one-dimensional array will use one pointer to reach the item required so that no multiplication is needed.

Compilers frequently contain optimising features such as the evaluation of common subexpressions or the efficient use of registers but these features need to be understood if they are to produce worthwhile savings.

In order to avoid inefficient constructs it is necessary to know what code is generated by each construct and to choose a suitable compromise based on the nature of the problem. Similarly, if there is a choice of microprocessors able to do a given job the one with the 'best' compiler should be chosen. The fact that the microprocessor manufacturer has provided his chip with a variety of complicated and powerful instructions is no asset if the compiler does not use them. Many compilers use a surprisingly small subset of the order code.

6. OFF-LINE TESTING

A High Order Language System, with the features described in the previous sections, allows the programmer to test his software in a radically different environment from that in which it will eventually run. This has enormous advantages. For example, each part of the system can be tested, possibly in slow motion, just like any conventional non real-time program on the Host computer. Programs can be tested under precisely controlled conditions without any possibility of interference from imperfectly functioning hardware. Error conditions which cannot normally be produced on demand can be simulated and programs which would normally receive variable real-time data can be fed with precisely controlled, repeatable data.

An even bigger advantage is that large parts of the system can be tested without needing to use any special hardware rigs. This means that programmers and hardware development engineers do not hold each other up and that hardware and software development can proceed independently until very late in the project, which should give a big saving in both time and costs.

In many cases it should be possible to test the complete system off-line to the point where the only remaining errors are due to timing or to unexpected hardware features.

The off-line test system is not just a development tool. It is also a very useful maintenance tool. Any fault which is reported can be tried on the off-line system and if the fault is repeatable there, its diagnosis becomes easier because of the debugging facilities available. In addition, modifications to correct the fault can be tested under controlled conditions without recourse to a hardware rig and without the necessity of producing a set of 'PROMS'.

A set of files containing tests and expected results may be built up and used as a partial acceptance test for each modified version. The files should be enhanced and updated whenever a fault is cleared or a modification introduced, thus ensuring as far as possible, that clearing one fault does not induce another.

7. ON-LINE TESTING

Once the system has been tested exhaustively off-line it must be commissioned on-line. At this stage the only untested items should be the hardware interfaces, timing and possibly a small number of code inserts.

The use of a manufacturer's development system and a logic analyser are invaluable at this stage. The manufacturer's development system provides RAM instead of ROM and allows the use of extra input/output statements. This means that the program can easily be loaded, often from floppy disc, and that the systematic set of tests carried out off-line can be repeated on-line.

This use of the manufacturer's test system must be planned in advance. Firstly the ROM in the final system must be placed in a part of the address space that may be replaced by RAM in the test system and any memory mapped addresses must be reserved in the test system. Secondly, the High Order Language Input/Output routines which were used on the host machine must be duplicated on the test system so that the same procedure calls may be used. If timing constraints are such that Input/Output calls cannot be used then results should be stored in a circular buffer in memory and printed out after the system has completed its test run.

8. DISADVANTAGES

Any new technique brings with it some disadvantages and some constraints. The principal disadvantage of programs written in High Order Languages is that they tend to require more instructions to carry out a task and to run more slowly than those written in assembler language. This means that additional ROM must be provided and occasionally a more powerful processor. To test a program off-line as well as on-line means that the extra code has to be written, possibly as much as 25% of the total.

There are also constraints imposed upon the hardware design. For example, the part of the address space reserved for instructions must be contiguous and the time dependent features must be minimised However, many of these constraints are accepted good engineering practice already.

Finally, the programmers concerned must be familiar with both the High Order Language and the assembly language of the microprocessor.

9. PRACTICAL APPLICATION

The Author has recently been responsible for a microprocessor project using these techniques. The language used was CORAL 66 which is one of the standard languages for all British M.O.D. projects. The target system was a Motorola 6800 system and the host system a Texas 9900 system. The two CORAL compilers were both produced by the same manufacturer which minimised any differences between them. CORAL 66 has an official definition published by the Ministry of Defence. It includes most of the features mentioned including absolute addressing and fixed point arithmetic and the particular compilers used had conditional compilation features which make it very easy to include and exclude text portions.

Eight systems were produced, each occupying between four thousand and seven thousand bytes of ROM. They were written over a period of twentyone months by a team varying in size from one to four programmers and were documented to M.O.D. A.V.P. 7O standards.

The systems all had similar functions and had one interface in common. They were all written in a very modular fashion and a number of modules were common to all systems.

At the start of the project the flow of information through the complete group of systems was specified and from this the functions of each individual microprocessor were defined. At the same time a common format for all messages passing between processors was specified.

Next a specification for the first system was produced, giving details of the hardware to be used, all software interfaces, timing constraints, data to be held within the system and precautions to be taken against system malfunction.

This specification, which was used as a model for all the subsequent specifications, was a very good one and we were lucky to have a customer who produced such a comprehensive specification and then did not change it throughout the project.

Subsequent specifications appeared at regular intervals, always before we started work on the system concerned. Shortly after the first specification was agreed a test rig was agreed and we started work on the first system.

At the start we had no target system hardware and all initial work had to be done on the host system. We therefore designed a very simple overall framework which would run on either processor and started writing the least processor-dependent modules.

When the target system equipment did appear we had to transfer quite a lot of CORAL source text from one set of equipment to the other which we did by writing a special 6800 program which would read 9900 floppy discs. Once transferred the programs all compiled and ran to give the same results.

The first system, although by no means the largest, t ok the longest time since this included the time taken to understand the test rig. The remainder took varying lengths of time depending on their size and complexity.

A great deal of thought was given to the methods of testing the systems and the same method was used for all systems. It would have been quite impossible to test the systems with all possible combinations of inputs and hence three arbitrary objectives were laid down. These were:-

- All statements had to be obeyed at least once.
- All input messages would be tested with a zero and a non zero value in each field.
- All output messages would be produced with a zero and a non zero value in each field.

These were real-time systems and hence the test data had not only to have the correct numeric values but also had to be sent at the correct time.

To achieve this an 'on-line test system' was written which ran on the test rig and which fed data to the system under test and recorded the results. The inputs to the test system were files of data items, each data item consisting of a numeric value and the time it should be sent.

The 'on-line test system' took more effort than any single target system.

The test data files were also fed into the off-line test program which ran in the host computer and which tested the program in 'slow motion', producing a lot of diagnostic print out as well as the required results.

These methods have produced an exceptionally reliable group of programs which (with the documentation we have also written) I believe can be maintained for many years. If this is true the effort will have been worthwhile.

SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT

Mr R Dibble

Ferranti Computer Systems Limited

Ty Coch Way

Cwmbran

Gwent NP44 7XX

United Kingdom

SUMMARY

The MASCOT methodology was developed to contain increasing software costs and ensure the production of reliable software. Ferranti programmers have produced standard MASCOT products and developed several large real time applications using MASCOT. Their experiences are discussed in this paper.

A basic feature of the methodology is modularity, which produces benefits at all stages of software development, although the degree of decomposition required by MASCOT is a problem for most projects. MASCOT identifies three types of modules (Activities, Channels and Pools) and represents the design in a diagrammatic form (the ACP diagram) which is regarded as a useful design tool and an effective representation of that design. Formal definition of data and its access mechanisms is an improvement over existing practices but whether it significantly eases the problems of multiprocessor design is disputed.

Overall there is a price to be paid for the MASCOT method in terms of runtime overheads and we see how this problem was resolved by various projects. The advantages and disadvantages of MASCOT are discussed and related to avionic software requirements. The relevance of the design philosophy to the imminent arrival of Ada is considered.

1. INTRODUCTION

MASCOT is an acronym for a Modular Approach to Software Construction Operation and Test and as such is a design methodology supported by a programming system. It was developed in the United Kingdom during the 1970's at RSRE (The Royal Signals and Radar Establishment) with the specific aim of containing the trend towards increasing software costs and ensuring the production of reliable, maintainable software. An 'Official Definition of MASCOT' was first published in 1978 and was expanded into the Official Handbook of MASCOT (MASCOT OH - Reference 1) in 1980. MASCOT is now being used by many suppliers of real time software and in particular for defence applications. Since the Ministry of Defence is a major customer of FCSL (Ferranti Computer Systems Limited) it is not surprising that MASCOT has assumed an ever increasing importance for this company in recent years. We are, in fact, currently engaged in the development of several large real time applications using MASCOT and it is the experience gained on these projects which forms the basis of this paper.

The main areas to be addressed are:

(i) A short description of MASCOT.

(ii) An outline of FCSL involvement as a supplier of standard MASCOT software for its own range of computers, and as a user, in the production of real time applications software.

(iii) A discussion in terms of the claims made for the design methodology and the standard facilities provided, and an evaluation of the software development system and real time executive. The run time overheads incurred by the MASCOT approach are addressed and we see how projects have found it necessary to circumvent the rules in order to reduce these overheads.

(iv) With several projects in development sufficient material is now available to make it possible to compile various statistics for MASCOT based systems. These are presented together with some empirical estimating rules devised from them.

(v) The impact of the arrival of Ada as an international programming standard is assessed in order to evaluate the relevance of MASCOT as a design methodology for the new language.

(vi) The concluding section considers the advantages and disadvantages of MASCOT and relates this discussion to the particular requirements of avionic software.

My thanks are due to the representatives on the FCSL 'MASCOT Experiences Working Party' who, during the past year at meetings and in discussion papers have provided the information without which this paper could not have been written.

2. MASCOT

MASCOT is a method of designing real time computer systems which is supported by programming tools for use in realising such designs. The software structure of a real time system is defined in a formal manner which is

independent of both computer configuration and programming language. This structure is highly modular and is characterised by a close correspondence between the functional elements identified during design and the constructional elements of which the system is actually built. Each element has an explicit interface specification, a feature which adds considerable integrity combined with flexibility.

The MASCOT system consists of ACTIVITIES, which are the units of scheduling, and IDAs (Intercommunication Data Areas) which comprise data and their access mechanisms. Data which is shared by activities may only be accessed by means of access procedures which incorporate calls to the real time executive to achieve synchronisation and mutual exclusion of competing activities. Two types of IDA, designated CHANNELS and POOLS, are identified in the MASCOT OH. A channel provides uni-directional data flow and is further characterised by the consumption of its data by readers. The pool allows bi-directional data flow and the data is not consumed by readers but is amended by writers. Access to peripherals is achieved by HANDLERS and DRIVERS which are effectively access procedures which interface the peripheral device to a channel. During design, the system can be represented diagrammatically as a network diagram known as the ACP (Activity-Channel-Pool) DIAGRAM which shows all the system elements and their interconnections.

The real time MASCOT based system runs under an executive program called the KERNEL which controls the scheduling of activities and the handling of interrupts. A number of basic operations can be requested by calling kernel routines called PRIMITIVES. In particular, JOIN puts an activity on a queue for exclusive access to data and LEAVE is called to relinquish exclusive access to that data. Another primitive called DELAY is a timing primitive which allows an activity to discontinue processing for at least a specified period of time. No reference is made to the remaining primitives in this paper.

MASCOT provides the programming tools for constructing a system. Activity and IDA specifications (templates) are added to the MASCOT Construction Database using the ENROL facility. The actual system elements are produced using CREATE and built into a subsystem using the FORM facility. The real time system consists of a number of subsystems which can be controlled at run time by the functions START, TERMINATE, HALT, RESUME. If the subsystems comprising the target system are fixed at run-time the system is said to be FROZEN but some MASCOT implementations allow those subsystems to be changed on-line and in this case the system is said to be EVOLUTIONARY.

3. FCSL and MASCOT

Following trial implementations of MASCOT by the Ministry of Defence FCSL began initial studies in 1976 and were invited to join the MASCOT Suppliers Association which was formed, later that year, to aid the transfer of ideas to industry. Research continued and in 1979 the development of MASCOT software for our own computers commenced. In association with Software Sciences Limited, software for the Argus 700 and M700 computers was implemented and formally issued as 'MASCOT 700' in 1981. At the same time FCSL have developed MASCOT software for our other range of minicomputers (the FM1600 range) and also started work on several large real time applications which use MASCOT. These systems are targeted on various processors and range in size from less than 256Kbytes to over 1Mbyte. Various implementations of MASCOT are used in these applications and therefore a wide range of experience has been acquired.

The projects include:

(i) An Action Information Organisation (AIO) system for the Royal Navy which interfaces to weapons and sensors. The target system configuration is two FM1600E processors and over 1Mbyte of (private and shared) store. For program development a linked VAX 11/780 - FM1600E program generation system was used.

(ii) Intelligent 2-man consoles for the above system, targetted on two M700 processors and 256Kbytes of store. Program development was hosted on an Argus A700G Computer.

(iii) Conversion to MASCOT and multi-processors of an existing single processor non-MASCOT AIO and fire control system. The target is again twin M700 processors and 256Kbytes of store and program development was on an Argus 700G computer.

(iv) A Central Tactical Processor (CTP) for a mission avionics simulator interfacing with sensor simulators, keyboards and displays. The system was targetted on a PDP11/34 computer with 256Kbytes of store linked to the rest of the system by a PCL data highway. A PDP11/34 computer was also used for program development.

In addition to real time applications, the MASCOT approach has also been used by the MASCOT project team itself to implement a variety of support packages and the experience of this team has also been taken into consideration in this paper.

4. PROJECT EXPERIENCES

Let us consider MASCOT from the point of view of some of the claims made in relation to the MASCOT concept.

4.1 Modularity

This is certainly a basic feature of the MASCOT philosophy but it would also be true to say that we at FCSL have also learnt the need for modularity and that our existing design practices enforce almost as much modularity, although MASCOT has the additional feature of modularity of data areas (IDAs) which is formalized to a greater extent than our own existing practices. The modular approach will undoubtedly pay off during the implementation and maintenance phases. The correctness of the design stage should be easier to establish and there is also the additional benefit of improved visibility and control for the project manager.

The main problem has been to decide the degree of decomposition in a given design. Natural functional modules (ie Activities) seem to be small and therefore the activity scheduling and communication overheads will be high. Only one of the projects studied felt that they could afford the overheads and complexity of a true one activity - one function design. In areas such as keyboard processing and displays, all the remaining projects found it necessary to combine numbers of related functions into one activity. Often the principle applied is that decomposition will only go as far as is strictly necessary within the constraint of the need for parallel processing. While this certainly reduces the overheads it partly negates the point of using MASCOT and of course costs in terms of the loss of benefits of modularity (although projects would claim that these large activities had been given an internal structure to offset this).

MASCOT does recognise the problem of activity sizes and introduces the idea of decomposition of the system to the lowest level and then recomposition (recombining activities to form larger ones) until the final design is achieved. This was regarded as a somewhat idealistic approach, although it may be conceptually what happens. In practice it was thought more likely that the 'experience', 'flair' and 'creativity' of the design team, given the requirements and constraints of the systems design, are responsible for achieving a practical level of decomposition. Furthermore, we can conjecture that decomposition followed by recomposition to a given level may not produce the same result as decomposition directly to that level.

A further problem may arise with functional decomposition because previous design methodologies have favoured a tree structure whereas a MASCOT design is a flat network. Is the same method of decomposition valid? Functional blocks may be different because communications are different and in MASCOT there is the need to minimise communications through channels to reduce overheads.

It was felt that a MASCOT system was no more likely to retain its modularity throughout its life cycle than any other type of system. For example a fully decomposed design may require recomposition of certain activities to reduce store and load overheads, while if decomposition has not been taken to its lowest level then further decomposition may be required to meet system constraints (e.g Memory mapping or multi processor requirements). However it is recognised that it is important that maintenance and enhancement do not involve reviewing the total system design every time for each fault or modification.

4.2 The ACP Diagram

None of the systems considered here had produced an overall software design using an ACP diagram. Instead the systems had been partitioned into a number of tasks and ACP diagrams had been produced for each task or subtask. The loss of overall design visibility has been partially offset by the production of higher level design diagrams such as, for example, a TCP diagram where the T stands for Task (a small 'subsystem') and the diagram shows inter task communications only. The reluctance to produce an overall ACP diagram at the onset of the project stems from a belief that for large systems the ACP diagram is too large and too complex to be of use and diagrams drawn retrospectively tend to confirm this. The problem can be alleviated to a certain extent by mechanising the process such that the ACP network could be held as part of the database in an automated software development system. Even then, however, the overall picture will be lost if the design cannot be represented on, say, two sheets of lineprinter paper. The overall picture of the project which the single ACP diagram can give can provide vital information to the designer on, for example, incorrect system partitioning and critical areas of high connectivity and for the software manager the ACP diagram provides the information to help plan the system implementation.

Suggested solutions to the problem of ACP diagram complexity include the separation of the diagram into 2 components (AC and AP) or the production of the diagram incrementally as each task level design is completed.

Some reservations that exist about the ACP diagram must be attributable to lack of familiarity. Current designers have not, in the past, been required to show so much detail at such a high level of software design and previous representations of design have been in terms of control flow rather than data flow. Nevertheless the ACP diagram was generally regarded as a useful design tool and an effective way of representing that design for documentation purposes.

One problem with the ACP diagram is the restriction in the definition of IDAs to just two types. Some IDAs cannot readily be classified as either a Channel or a Pool and although MASCOT does not exclude other IDA types there is no standard representation available. Should a designer wish to show an object of intermediate characteristics on an ACP diagram he is forced to invent symbols with a consequent loss of standardisation.

4.3 Intercommunications Data Areas (IDAs)

Clearly it is a good idea to define data flow between modules and the formal nature of the IDA interface is an improvement over existing practice and ensures that the interface is specified early in design.

Access Procedure The concept of the Intercommunications Data Area (IDA) incorporates the encapsulation of data, with that data accessible only by procedures which form part of the IDA. The separation of code from the detailed data structure is a good idea, which is by no means new to FCSL. However the use of access procedures has previously never been a formal requirement of the methodology as it is in MASCOT. It was felt that the MASCOT Official Handbook does not place enough emphasis on the numbers of access procedures (400 or 500) that will be generated, for systems of the size and complexity of those considered in this paper. The management and documentation of access procedures is a significant task and great care is required to avoid the repetition of code. For future projects it is suggested that greater attention is given to the allocation of effort and timescales for the production of IDAs in the same way as activities. MASCOT statistics and estimating rules reproduced in section 5 should help in this task. The MASCOT OH recommendation that experienced programmers undertake IDA design had not been strictly followed, possibly because of the number of channels and pools, with their many access procedures, had proved to be such a large proportion of the system.

If we look at access procedures for channels we find that these were nearly always simple, sometimes being used for control only. The examples presented in the Official Handbook have been followed and standard channels used in some cases. There was no general agreement about the complexity of access procedures for pools. Sometimes these were non-trivial to the extent that the majority of the work in a given activity was performed in the access procedures. Such access procedures, which can be termed functional access procedures, were the subject of some discussion, with claims that they resulted in loss of modularity, loss of design visibility and indeed were not truly in the spirit of MASCOT. Projects were generally agreed that this was not the case. Functions were not 'hidden' in access procedures since they would not have appeared explicitly on the ACP diagram even if they had been included in the body of the activity. Since our standard procedures require that all access procedures are fully documented, all the required information was easily accessible to anyone having access to the software documentation. There was no loss in modularity and it is difficult to see the difference between including the function in an access routine, or in a service routine which would often be the alternative for functions such as, for example, axis conversions. It was thought that the MASCOT OD does not currently preclude functional access procedures although this may represent an evolution of the methodology from its original aims.

Another technique which can be used in access procedures is the block transfer. This was generally thought to be undesirable because of the loss of interface checking, although it can be argued that gains in efficiency would outweigh the limited type checking available in the CORAL language (this could not be said for an implementation language as strongly typed as Ada). It was noted with some dismay that we are encouraged to use this technique by the examples presented in the Official Handbook of MASCOT which show the use of block transfers.

Large Pools The problem of access to large pools such as a Main Track Table had been considered by each of the projects. For such a pool a choice has to be made between one or several control queues and if several control queues are adopted then there is nothing in MASCOT to prevent deadly embrace. The only safeguard is a procedural one which requires activities to always JOIN in the same order but, of course, manual procedures are far from infallible. As a comparison, it is worth noting that the FCSL Supervisor (real-time executive) for the FM1600 computers requires all data areas to be secured with one procedure call and prevents nested calls. Pools access is less of a problem if a fully cooperative scheduling algorithm is selected since an activity will not be rescheduled until it is voluntarily suspended and so is guaranteed protection when accessing data.

4.4 Multiprocessor Systems

The question of how well MASCOT handles multi CPU system design requires some discussion, particularly in relation to the ACP diagram and the concept of IDAs as interfaces.

In theory the whole system is shown on one ACP diagram so multiple instances do not occur. However, if the system is partitioned between two or more CPUs then more than one copy of an activity may be required. Since each instance is functionally the same it is misleading to give them unique names yet you do not want two items with the same name on the ACP diagram. The problem can be avoided by having one ACP diagram for each processor but this presupposes you know how the functions will be distributed between processors. Alternative high level design representations encounter the same problem and it seems that it is difficult to do 'top down design' unless you know some of the answers before you begin. In practice, aspects of the bottom level (hardware configuration, public libraries, MASCOT kernel implementation) are defined before the start of software design.

We have already stated that the formal nature of the ID and its early specification are an improvement over existing practice. As such this is bound to assist in implementing a multiprocessor system but whether MASCOT significantly eases the problems of multiprocessor design is debatable. Several of the systems considered in this paper are dual processor configurations. The overriding considerations in their design were the location of system functions, bus loading and system response. Once the partitioning was achieved using these criteria the MASCOT method was applied to individual processors. It would seem likely that the future enhancement of these systems will be more easily achieved because of the MASCOT design but this has yet to be shown in practice.

4.5 Portability of MASCOT Applications

The claims made for the portability of the MASCOT application have yet to be tested at FCSL although some design work carried out by the MASCOT project team on certain support packages had been implemented on various computers and it would seem, therefore, that the MASCOT design is probably portable except where implementation dependent features (handlers/drivers) are involved. It was generally felt that code portability is as (un)obtainable as it would be using any constructional philosophy given the amount of detail left to the implementor of the MASCOT machine (which is equally true of CORAL, the language in which most MASCOT applications are implemented and which itself exists in a number of dialects).

Certain advantages must accrue from the use of any standard methodology and these will hold equally true for MASCOT. Programmers and designers will be 'portable' between projects with the minimum of retraining and customers, engineers and managers should more easily be able to understand software documentation. These benefits should become more apparent with the second generation of MASCOT systems but will depend to some extent on the degree of standardisation that is achieved by the manufacturers of MASCOT systems and on the degree of adherence to the principles of MASCOT that individual projects achieve.

4.6 Prototyping

Prototyping, the idea of implementing the design of an embedded real time system on a large mainframe computer in order to check the correctness of that design, is a concept recommended by the MASCOT Official Handbook. It is however a practice that has not yet found favour among the MASCOT projects who felt there was little to be gained by prototyping, rather than developing software for the target machine immediately.

There was, perhaps, a reluctance to attempt quick try-out solutions and projects preferred to rely on rigorous design methods and systematic setting-to-work procedures to achieve the desired result. The cost effectiveness and indeed the practicality of prototyping large and complex systems was doubted. In such cases it was thought that the customer was unlikely to pay the cost of prototyping, as described in the MASCOT Official Handbook, and that this would rule it out in most cases.

4.7 Standard Facilities

MASCOT, the programming system, provides us with a number of standard facilities both during system construction and at run time.

Construction

The existence of a formal construction sequence based on the ENROL, CREATE and FORM facilities was regarded as a positive advantage to the implementation of a system. Generally an incremental approach to construction was adopted but one project had decided to always recreate the database from scratch (to simplify control). The FCSL MASCOT machines (for M700 and FM1600 computers) do not include an Evolutionary facility. This was the result of a deliberate decision related to the types of operational systems produced by our company for which the capability of dynamically varying the design of the software was not required. Further more the hardware needed to support such facilities (host computer or fast backing store) is not usually present. Such hardware would be present during the software development phase and the usefulness of evolutionary facilities during software testing was considered. However there would be great difficulty in controlling the build state of the system, and therefore associated Quality Assurance problems. For example, in the case of a system controlled from a VDU terminal, there would be no hard evidence of on-line deletions etc. made to an evolutionary system.

Run Time Facilities

In-line monitoring facilities provided by MASCOT had been used during testing (in particular, RECORD was used to obtain execution history in correct time sequence with primitive calls) and although there was some criticism of run time overheads of using Monitor, it was generally thought to be acceptable and to compare favourably with similar systems. It is not likely to be used for Trials Recording because of the special requirements (data rate, message sizes) of that activity.

Subsystem Control has been implemented in the FCSL MASCOT even though evolutionary capabilities are not. The control functions can be invoked in a frozen MASCOT system during software development and operationally. In the latter case START and TERMINATE had been used to reconfigure the system in response to hardware failures while, during software testing, subsystem control had been used for emulation and for fault diagnosis. The HALT and RESUME functions had not been used by any of the projects.

The scheduling of the real time activities and the synchronisation of data access is achieved by a run time executive called the MASCOT Kernel. The scheduling mode provided depends on the particular MASCOT implementation but Ferranti provide both pre-emptive and co-operative algorithms. The facility for having a user timer interrupt program has also been provided and is used to achieve cyclic stimulation, the Delay primitive being regarded as unsatisfactory for our type of system. Variable time slice length is provided but it was felt that it ought to be possible to allow time slice length to be infinite. Since the run-time overheads resulting from the use of the MASCOT methodology appear to be high, it is essential that scheduler and its primitives perform as efficiently as possible and to this end the use of in line code and microprogram should be considered. At least one manufacturer has partially implemented MASCOT kernel facilities in hardware for its own range of computers.

4.8 MASCOT System Constraints

So far we have looked at the facilities MASCOT offers in terms of the collected reactions of those projects who have experience in their use. To provide these facilities there is a price to pay in terms of run time overheads and this has constrained projects in their use of MASCOT such that they have found it necessary to circumvent either the letter or the spirit of MASCOT in order to minimise the overheads.

MASCOT Overheads

These are incurred in a number of ways:

Channels The concept of channels for passing data leads to the repetition of data throughout the system with extra processing required to carry out the message copying. Furthermore the channel and its control queues are always maintained even when not being used.

Access Procedures The action of a program in reading or writing a given data object has to be performed somewhere, but by incorporating the access mechanism into a separate procedure, overheads are incurred which may become significant in a large system. The large number of access procedures generated in these systems has already been noted, and if they are not carefully controlled additional penalties may result from duplication of code. The use of primitives to synchronise data access also imposes extra processing load on the system.

Scheduling MASCOT activities are non-terminating and therefore each activity requires its own stack in which context data may be stored. In a large system this will be a significant amount of data compared, for example, with the F1600 Supervisor (the real time executive for the Ferranti FM1600 computers) which stores only static data ($\leq$ 4 words) on a per activity basis with additional space for context data being allocated for each of the three priority levels.

4.9 MASCOT 'Avoidance'

Most projects, realising the potential overheads of using MASCOT, have to a greater or lesser extent made concessions to the necessity of minimising store and load. Some examples are given below:

Decomposition By not adopting the degree of decomposition that the 'spirit of MASCOT' would require, savings in both store and load can be achieved because the amount of message passing through channels is reduced as are the associated re-scheduling activities, and the number of stacks required.

Pool Access To reduce overheads when accessing pool data, MASCOT primitives were not used to synchronise data access. In one of the systems, since a fully co-operative scheduling algorithm had been adopted, mutual exclusion was guaranteed anyway. However, other systems relied on program priorities to secure data or accepted the possibility that some data (for display) might be instantaneously corrupted. In one system direct referencing of data objects was allowed within a subsystem, providing that the pool was internal to that subsystem. In this case the activities within the subsystem operated their own data access synchronisation without invoking MASCOT primitives.

Channels Projects had favoured simple channels to the extent that channels often contained no data but were simply a means of passing control. In many cases to save primitive calls activities 'JOIN' a control queue at initialisation and never 'LEAVE'.

Peripheral Interfaces This was an area where strict adherence to MASCOT principles sometimes led to unacceptable overheads. In particular we find that addresses rather than data are passed between activities.

For example, in an output situation, addresses rather than the actual data to be output, were queued. This reduced the overheads of message passing, kept buffers to a reasonable size and also allowed data to be updated while it was waiting for the peripheral to become available. In a second example, a peripheral control subsystem, the subsystem controlled the input of data from the peripheral to a generic IDA, whose address was supplied by the user activity when requesting a function of the subsystem. In this case, to increase the security of this approach the IDA header contained an address dependent pattern which could be checked by the subsystem to trap any attempt to pass an invalid address.

4.10 MASCOT Load

Some estimates of load due to MASCOT have been made and the figures obtained suggest that MASCOT imposes an overhead of 30% to 40%. Measurements have been made on the avionic simulator which produce a figure for the actual load of the MASCOT kernel of 35-40% but this is a hosted MASCOT system and this figure therefore includes the load due to the RSX11 operating system.

Allowing for possible errors in the estimated figures and for some debate as to what actually constitutes MASCOT kernel load, there seems to be no doubt that a real time MASCOT-based application will have a much higher scheduling overhead than previous systems.

5. MASCOT STATISTICS

5.1 With several large MASCOT applications in development, the raw material exists to permit the compilation of various statistics and to allow preliminary conclusions to be drawn regarding the form of a MASCOT based system. The size of the data sample from which the statistics were derived is shown in table 1.

	ACTIVITIES	CHANNELS	POOLS
System (i)	190	153	134
System (ii)	41	40	11
System (iii)	126	126	16
System (iv)	39	38	44
TOTAL	396	357	205

Table 1 : Overall Statistics

So far the investigation has been limited to channels and pools (IDAs) and the results are presented in figures 1 to 5. Currently a further exercise is in progress to collect statistics on activity sizes although no results are yet available.

Two initial warnings are necessary. Firstly the production of such data is a wholly manual process and as such subject to error. There may be small errors in the data but these will not invalidate the overall picture.

Secondly the reader will find it impossible to correlate various parts of the data which he might reasonably have expected to be able to do. The difficulty is that all the projects are incomplete (in some sense). There are dummy access routines, access routines whose declarations have been turned into comments

(presumably to be implemented properly in due course), channels with no readers, channels with no writers, and so on. In general as much data as possible was extracted from the material, thus for example a dummy access routine can be included in figure 4 but would not be relevant for figure 5. The disadvantage of this approach is that the data does not form a self consistent whole.

The distribution of various system factors can be compiled and figure 1 summarises these distributions giving the mean, maximum and minimum values for each. Given the extreme skew on these distributions it can be taken that the mode value will always be significantly different from the mean value. Figures 2 to 5 are the raw data from which figure 1 was derived, presented in tabular form except for figure 5 which was plotted as a histogram to illustrate the skew distribution.

Figures 2 and 3 examine the data component of the IDAs. The main problem here is that many IDAs are composite and include a number of discrete and overlaid data areas of various sizes, various message sizes, numbers of messages and so on. These complex IDAs were mostly ignored in figures 2 and 3. Both figures understate the extent of this problem and as a consequence may be in error.

Figure 4(a) is concerned with the basic connectivity of the ACP diagram. Four types of connection between activities and IDAs are identified.

(a) Read only

(b) Write only

(c) Write and read

(d) Total.

Type (c) were found to be necessary because some activities required both types of access to individual IDAs. Also in some cases an access routine which is essentially a write access also returns a value to the writer (indicating success or fail or etc) and thus incorporating some form of read access.

Type (d) indicates how many IDAs (of whatever type (a), (b) or (c)) are accessed by each activity, and vice versa.

Figure 4(b) has three components, the number of access routines (of the above types) per IDA, the analysis of the declarations for these routines in terms of parameters used, and lastly the number of procedure calls nested within the access routine bodies but excluding MASCOT primitives.

Figure 5 is simply a distribution of the size of access routine bodies in terms of Coral statements.

5.2 Empirical Estimating Rules

We can use the data presented in Figures 1 to 5 to derive a set of estimating rules for MASCOT systems. To avoid spurious precision, various numbers are rounded as appropriate. All rules deal with systemwide averages. The rules are, of course, no better than the data they are extracted from and the assumptions used in the process. It may be that the systems analysed may not be typical MASCOT systems. However they are fairly typical Ferranti systems and will provide a good basis for deriving estimates for future systems.

Since the rules are empirical there is a continuing need to collect more data as it becomes available and update them, meanwhile they are offered as the best available method of estimating the requirements for a MASCOT system.

Rule 1 The number of Channels in a system will equal the number of Activities.

Rule 2 A Channel will need 3 access procedures.

Rule 3 A Channel access procedure requires 6 CORAL statements.

Rule 4 A Channel requires, for all purposes, a workspace of 100 words.

Rule 5 A Channel will contain space for 8 messages of 11 words each of a single type.

(Note:-

Rules 4 and 5 are slightly misleading because of the extreme skew on the distributions. The typical channel will contain say 2 messages of 4 words. In terms of overall project estimates however that typical channel is not the one of interest.)

Rule 6 For the level of decomposition shown by the systems studied, the number of Pools needed will be half the number of Activities.

Rule 7 A Pool will need 11 access routines.

Rule 8 A Pool access routine needs 7 CORAL statements.

Rule 9 A Pool needs a workspace of 1000 words, for all purposes.

Rule 10 A Pool will contain 26 messages of 24 words.

(Note:-

In addition a skew effects, the Pool rules are also affected by the presence of composite pools. The typical Pool will contain say 4 entries of 4 words, and as for Channels this is not the appropriate data for systemwide estimates.)

Rule 11 From Rules 1 to 10, each system Activity will use, on average, 9 access routines needing 60 CORAL statements.

6. ADA AND MASCOT

In the next few years Ada compilers will become commercially available and the full impact of this new programming language will begin to be felt. Where will this leave MASCOT? Do we need to throw away all the hard earned experience in the use of MASCOT and learn a new set of techniques? Let us look at some of the attributes of Ada, at a high level, and compare them with MASCOT.

First consider Ada:-

(a) A programming language

(b) System construction facilities and a run time executive

(c) Aims at reliability, maintainability, portability of software

(d) Wide use in the defence industry

(e) No design methodology

(f) No defined documentation system

MASCOT, as we have already seen is:-

(a) A Design methodology

(b) System construction facilities and run time executive

(c) Language independent

(d) Aims at reliability, maintainability and portability of software

(e) Widely used in the defence industry

(f) Defined documentation scheme.

I think it can be seen, therefore, that although the system construction software and run time executive provided by MASCOT may not be needed (since they are already provided in the Ada Programming Support Environment (APSE)), the MASCOT design methodology and aims are complementary to Ada. It is also certain that, because Ada is such a powerful and complex programming tool, the use of Ada must be backed up by a sound design methodology.

How can the Ada language be used to implement a MASCOT design? Such a design is basically modular and two types of module are indentified. These are the ACTIVITY which is the unit of construction, scheduling and testing and the IDA (channel or pool) which comprises the data area and its access mechanisms. In Ada, TASKs are the units of scheduling and synchronisation, PACKAGES are the data and access mechanisms and, thus, mapping a MASCOT design onto Ada can be achieved as follows:-

```
            ACTIVITY -> TASK (type) with no entries

Either      :- CHANNEL/POOL -> TASK (type)

               with Access Mechanism -> entry

or          :- CHANNEL/POOL -> (generic) PACKAGE

               with Access Mechanism -> procedure
```

In the latter case the data can be protected as in MASCOT In a recent report for the United Kingdom's Department of Industry, the Augusta Consortium (of which FCSL is a member) reviewed available design methodologies (Reference 2). This report includes, among others, an example of a MASCOT design implemented in Ada.

Table 2 addresses several desirable features of MASCOT and relates them to Ada implementation facilities.

MASCOT OBJECTIVES	ADA FACILITIES
1. Controlled access to shared data	Packages
2. Visibility in ACP diagram	Separate compilation and tasks
3. Encapsulation of real time aspects	Packages, generics and tasks
4. Formal definition of interfaces	Separate compilation, strong typing, tasking model
5. Flexible system construction approach	NOT in Ada
6. Formal definition of kernel	Tasking model
7. Modularity	Separate compilation

Table 2 : MASCOT vs Ada

Most of the MASCOT objectives can be realised in Ada so it seems that the MASCOT design methodology is suitable for Ada. It is interesting to note that the flexible system construction approach (item 5), not realisable in Ada, is achieved by providing evolutionary facilities which have not been implemented in the FCSL MASCOT machines.

7. MASCOT ASSESSMENT

From the experience of our project teams it should be possible to make an assessment of the benefits and shortcomings of using the MASCOT approach to program development.

Standardisation One benefit that MASCOT can achieve is a standard approach to the design and implementation of real time systems. However, it will lose its credibility as a widely used standard if a variety of different implementations exist. Unfortunately the MASCOT Official Handbook was late in appearing and had to be kept vague so that it could accommodate the existing, different, MASCOT systems. This will, in turn, mean that further variations on the theme can be produced in the future. The standard is further eroded by MASCOT avoidance techniques which different projects are forced to use but which compromise the principles of modularity and protection which are the very basis of MASCOT.

Related to the concept of standardisation is the idea of portability of MASCOT applications and it was felt that, in practice, applications will be no more portable than other CORAL systems.

Overall Software Reliability This is clearly very important in the development of software for avionic systems and the principles of design visibility, functional modularity and data protection which are fundamental to MASCOT will help to ensure the production of reliable software and reduce the problems of maintenance and enhancement during the in-service lifetime of the system. MASCOT itself cannot, of course, guarantee the correctness of the design or implementation and must be backed up with project quality procedures. Since our projects are not controlled experiments it is difficult to make an objective assessment of existing MASCOT implementations. However the evidence from the software integration and trials phases, where these have taken place, is encouraging and these phases have proved relatively trouble free.

Store and Load By far the major problem encountered by the various projects was the store and load overheads associated with the MASCOT approach. We have seen that there is a large scheduling overhead and it is therefore critical that the means of scheduling (ie the MASCOT kernel) be optimised for efficiency. Store overheads are also incurred.

These overheads result, at best, in a loss of spare capacity, and with it the opportunity for software enhancement, but may result in the requirement for extra processors or extra store. Even though such hardware is becoming smaller and cheaper there is, nevertheless, a cost in terms of size, weight, complexity and heat dissipation, problems which are present in most systems, but are particularly acute in avionics.

MASCOT, of course, does not set out to produce the smallest most efficient implementation. Software quality is of paramount importance and the overheads must be accommodated if that quality is to be achieved. The costs of the overheads must be weighed against the cost of reliability and maintainability and a practical balance achieved. In essence the problem is similar to that encountered in the transition from low to high level language implementations and will perhaps be encountered again when current languages are superceded by Ada.

Avionic Rigs During the development of an avionic system there may be a requirement for a number of rigs designed to test various system functions. It is a definite advantage to be able to quickly and easily rebuild and reconfigure the target system software in response to faults or to changed requirements and the MASCOT system construction software and the modular approach required by the methodology provide the means

of achieving this. The Evolutionary capability, allowing the dynamic reconstruction of the target software, is also available although we believe that such a powerful facility should be treated with caution and may not be suitable for strictly controlled software development. However, as we have already seen, even in a frozen system, subsystem control can be used to reconfigure the system on line and provide fault diagnosis. MASCOT also provides in-line monitoring facilities which are a further aid to testing software on a development rig.

Multiprocessor Systems A system designed in a modular way with formally defined interfaces can more easily be partitioned between a number of processors and it follows that MASCOT is suitable for use in multiprocessor systems. Our experiences suggest that there is some difficulty in representing a multiprocessor system at the highest level of software design and that MASCOT is no more helpful than other methods with the problem of deciding how the software should be partitioned.

Management and Customer Aspects From the point of view of both management and customers the adoption of a widely used standard such as MASCOT offers an opportunity for a better understanding of the software contribution to a real time system and the standard terminology provides a common language for describing software. MASCOT offers design visibility through the ACP diagram which also provides information to assist in project planning. It has been noted that there are many more identifiable modules in MASCOT based systems, which increases the management task but also allows better control of software development. The use of MASCOT design methods may result in a change in our ideas on effort allocation to various phases of the development task. Extra effort may be expended during design but with easier integration and reduced maintenance leading hopefully to an overall reduction in the effort requirement.

There is a need on the customer's side for an understanding of the implications of using MASCOT. There is a natural expectation on the part of the customer that for each increase in processing power available he will see a corrresponding increase in the user facilities available. What MASCOT is saying is that we must use some of the advances in hardware technology to improve the quality of the software product and thereby curb the escalation in software costs and increase the reliability of complex . ` time computer systems.

8. CONCLUSIONS

The general project reaction to the use of MASCOT has been an agreement that many of the features of MASCOT are highly desirable (modularity, encapsulation of data and control of data access) and the formal construction system was a positive advantage. However, it was also frequently stated that the ideas were not new and that existing design practices within FCSL are as good as those in MASCOT. Proponents of MASCOT would say that MASCOT never claimed to be earth shatteringly innovative, but simply brings together a number of accepted techniques in a co-ordinated manner to form a standard approach to software development.

We conclude that the advent of this standard defined methodology and programming system is in principle a good thing but there are, in practice, some problems such as the acceptability of the store and load overheads. Analysis of statistics, such as those presented in this paper, will allow us to better evaluate the implications of using MASCOT in any particular application. Although the MASCOT programming system may not be needed if the Ada language and the APSE become universally accepted, MASCOT could have an ongoing role as a design methodology for Ada.

9. REFERENCES

Reference 1. Official Handbook of MASCOT
MASCOT Suppliers Association 1980

Reference 2. Ada based system development methodology
Study Report - Volume 1
United Kingdom Department of Industry 1981.

FIGURE 1 SUMMARY DATA

1a IDA DATA AREAS

		MINIMUM	MEAN	MAXIMUM
CHANNELS	SIZE	1	108.87	4506
	MESSAGE PER CHANNEL	0	5.74	160
	MESSAGE SIZES CHANNEL	0	10.42	120
	MESSAGE TYPES PER CHANNEL	0	1.14	13
POOLS	SIZE	1	1008.1	12120
	NUMBER OF ENTRIES	1	25.23	300
	ENTRY SIZE	1	23.30	582

1b. SUMMARY OF SYSTEM FACTORS

		MINIMUM	MEAN	MAXIMUM
CHANNELS PER ACTIVITY.	READ	1	1.11	5
	WRITE	1	2.49	88
	WRITE & READ			
	TOTAL	1	3.38	89
ACTIVITIES PER CHANNEL	READ	1	1.07	4
	WRITE	1	2.30	47
	WRITE & READ			
	TOTAL	1	3.61	48
POOLS PER ACTIVITY	READ	1	1.21	25
	WRITE	1	1.21	22
	WRITE & READ	1	0.92	8
	TOTAL	1	4.34	27
ACTIVITIES PER POOL	READ	1	1.45	37
	WRITE	1	1.53	23
	WRITE & READ	1	1.15	15
	TOTAL	1	5.36	86
ACCESS PROCEDURES PER CHANNEL	READ	1	1.19	3
	WRITE	1	1.57	7
	WRITE & READ			
	TOTAL	2	2.75	10
ACCESS PROCEDURES PER POOL	READ	1	4.49	39
	WRITE	0	3.31	34
	WRITE & READ	0	2.49	61
	TOTAL	2	10.21	81
PARAMETERS PER PROCEDURE CHANNEL	VALUE	0	0.82	7
	LOCATION	0	0.40	4
	TOTAL	0	1.20	7
PARAMETERS PER PROCEDURE POOL	VALUE	0	1.40	7
	LOCATION	0	0.59	6
	TOTAL	0	1.91	10

1c. CORAL STATEMENTS PER ACCESS PROCEDURE.

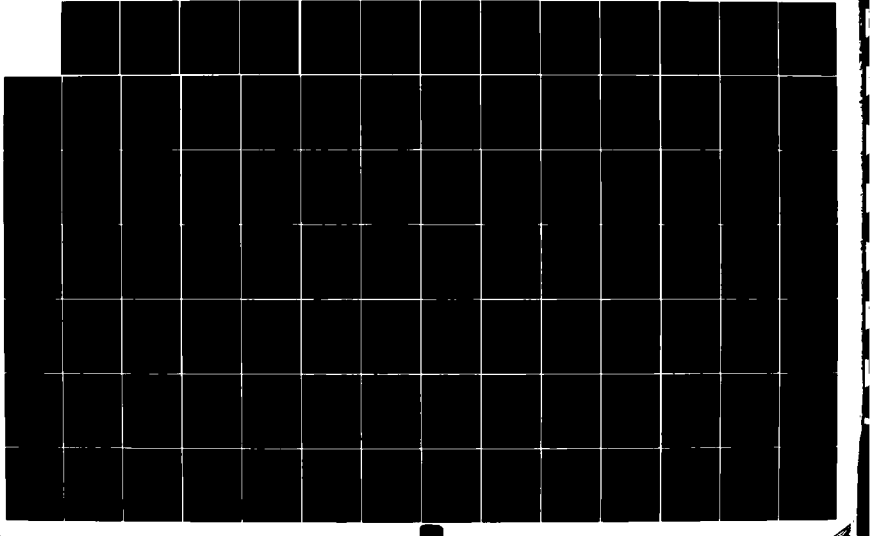
	MINIMUM	MEAN	MAXIMUM
CHANNELS	1	5.93	19
POOLS	0	6.88	80

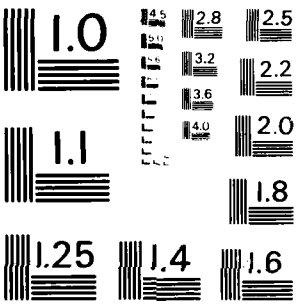
FIGURE 2 CHANNEL MESSAGE STATISTICS

CHANNEL DATA AREA SIZE		MESSAGES PER CHANNEL		MESSAGE SIZES		MESSAGE TYPES PER CHANNEL	
SIZE	Nº OF CHANS	Nº	Nº OF CHANS	SIZE	Nº OFF	TYPES	Nº OF CHANS
1	1	0	20	0	11	0	11
2	1	1	30	1	7	1	63
5	11	2	23	2	16	2	5
6	9	3	1	3	2	6	1
7	3	4	5	4	6	13	1
8	2	8	1	5	2		
9	2	10	1	6	3	mixed	3
10	1	14	1	7	1	Dummy	3
11	1	17	1	8	7		
12	1	20	1	9	1		
13	4	21	1	10	2		
14	2	30	1	11	5		
15	2	32	1	13	1		
16	3	100	1	19	1		
18	10	160	1	21	2		
21	1			26	1		
22	3	Special	1	28	1		
24	1	mixed	3	32	11		
26	2	Dummy	3	120	1		
28	1						
30	3			Special	1		
37	1			mixed	3		
39	1			Dummy	3		
41	1						
44	1						
50	1						
54	1						
65	1						
72	1						
75	1						
78	6						
80	1						
84	1						
86	1						
104	1						
105	1						
131	1						
144	2						
154	1						
176	1						
178	1						
181	1						
208	1						
232	1						
2066	1						
4506	1						

AD-A127 131 SOFTWARE FOR AVIONICS(U) ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT NEUILLY-SUR-SEINE (FRANCE) JAN 83 AGARD-CP-330 3/5

UNCLASSIFIED F/G 9/2 NL





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

FIGURE 3 POOL STATISTICS (DATA AREAS)

INDIVIDUAL POOL SIZES	
SIZE	SIZE
5@1	166
2@2	172
3	205
2@4	228
6	238
7	264
11	290
12	362
14	426
17	474
20	552
2@22	553
25	584
26	659
27	1077
31	1134
36	1305
38	1564
40	1586
43	2231
46	3030
66	4239
74	5820
84	7197
86	8022
92	8666
107	12120
122	
138	
142	

Nº OF ENTRIES PER POOL	
ENTRIES	POOLS
1	16
2	4
3	1
4	5
5	1
8	3
10	2
11	3
12	1
20	3
21	3
33	1
40	3
52	1
64	1
83	1
132	1
264	1
300	1

POOL ENTRY SIZES	
SIZE	Nº OFF POOLS
1	16
2	3
3	3
4	2
5	2
6	1
7	2
8	4
10	2
11	1
12	5
13	3
14	1
15	1
16	1
18	1
20	2
21	1
25	4
27	1
28	1
29	1
38	1
262	1
582	1

MANY POOLS ARE COMPOSITE WITH MIXED MESSAGE SIZES-VARYING NUMBERS OFF- VARIOUS DEGREES OF OVERLAY- AND IMPOSSIBLE TO CLASSIFY ON THE SIMPLE SCHEME ABOVE

FIGURE 4 RELATIVE FREQUENCIES

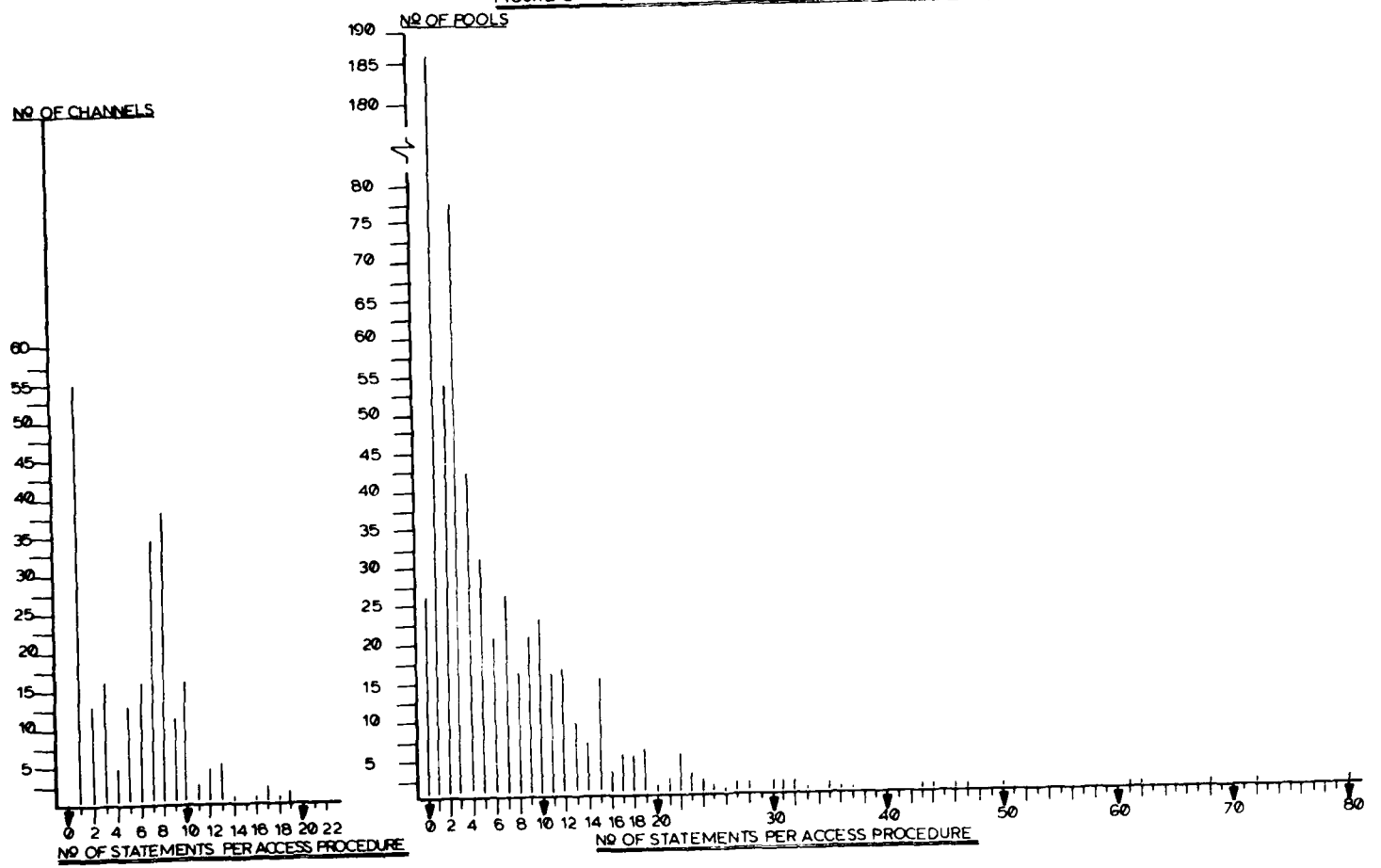
4(a)

NO	CHANNEL PER ACTIVITY			ACTIVITIES PER CHANNEL			POOLS PER ACTIVITY				ACTIVITIES PER POOL				NO
	READ	WRITE	TOTAL	READ	WRITE	TOTAL	READ	WRITE	W&R	TOTAL	READ	WRITE	W&R	TOTAL	
1	305	146	36	324	217	5	68	88	86	105	87	71	61	28	1
2	18	97	118	8	32	203	39	45	37	76	26	31	31	44	2
3	6	44	97	3	20	24	18	18	10	57	6	9	10	43	3
4	1	33	40	1	9	21	9	10	4	54	4	6	4	37	4
5	1	6	30		4	11	8	4	2	26	2	2	1	12	5
6		6	8		4	2	2		1	12	2	1	2	8	6
7		4	6		4	4				13	1	1	2	1	7
8		1	5		1	1			2	5	2	1	1	2	8
9		2	1			1					3	1		5	9
10			2		2	1				6		1		4	10
11		1	2		1	3				1		1		2	11
12					2	1							1	2	12
13						1				1		1		1	13
14		1			1									1	14
15			1			1				1		1	1	1	15
16		1												1	16
17			1		1	1@18						1		2	17
18						1@23								2	18
19						1@25					1				19
20						1@26									20
					1@24	1@27								1@21	
														1@22	
					2@26	3@30								1@24	
														2@27	
					2@29	1@43								1@28	
														1@31	
					2@42	1@46				1@25				1@56	
		1@88	1@89		1@47	1@48	1@25	1@22		1@27	1@37	1@23		1@86	

4(b)

NO	ACCESS PROCEDURES PER CHANNEL			ACCESS PROCEDURES PER POOL				PARAMS PER PROC (CHANNEL)			PARAMS PER PROC (POOL)			NESTED PROC OF FUNC CALLS		NO
	READ	WRITE	TOTAL	READ	WRITE	W&R	TOTAL	VALUE	LOC	TOTAL	VALUE	LOC	TOTAL	CHANN	POOL	
0	4				20	11		102	154	60	176	404	80	183	415	0
1	72	60		7	10	13		77	60	89	242	212	191	49	130	1
2	16	22	58	14	7	12	5	30	12	57	187	71	205	7	35	2
3	3	10	17	28	15		9	7		11	68	17	122		24	3
4		2	11	5	5	5	8	2	1	2	25	2	65		14	4
5			8	3	1	1	4	2		2	10		12		5	5
6				3	3	3	14			1	1	1	4		4	6
7		1		1	1		8	1		1	2		3		2	7
8				1	2	1	1						1		2	8
9							3								1	9
10			1		1		2						1		1	10
11				1			1									11
12				1			1								1	12
13							2								1	13
14							2									14
15							1									15
16				1			1									16
17																17
18				1											1	18
19							1@20									19
20						1	1@21								1	20
							1@50									
					1@21		1@54									
				1@21	1@25		1@76									
				1@39	1@34	1@61	1@81								1@23	

FIGURE 5 Nº OF CORAL STATEMENTS PER ACCESS PROCEDURE



SAFETY CRITICAL FAST-REAL-TIME SYSTEMS

by

B. Güsmann*)
O.F. Nielsen
R. Hansen
MESSERSCHMITT-BÖLKOW-BLOHM GMBH
Aircraft Division
8000 München 80, Postfach

SUMMARY

The development of advanced military aircraft requires large embedded digital systems and digital test equipment for performance enhancement. Examples at MBB are CCV III, an eight computer test rig for evaluating next-generation fly-by-wire systems and cross software test systems (CSTS) for verifying safety critical airborne software. Typical cycle times of such systems range from 10 to 60 msec. Such systems impose restrictions on software design and development tools, especially on required High Order Language Tools. Four languages, Fortran, Pearl, "C" and Pascal were evaluated for use in the CCV III system and CSTS. Finally "C" was chosen to implement both systems with very satisfactory results. A comment will be given with respect to convert "C"-programmed systems to ADA in the future.

1. INTRODUCTION

The continuously growing part of local and central digital data processing is significant for flight guidance and control systems of advanced military aircraft and the related testsystems. The task of the software in a Fly-By-Wire (FBW) System is to control the aircraft as well as to detect and to handle hard-/software errors. That means: guidance and control software has a key function as a safety critical item during the whole mission from take off to landing. The important role of onboard software for an entire project is known for instance from the ALCM competition (AWST, March 1980) and the certification of the DC-9 Super 80 (AWST, Sept. 1980).

A special problem of flight control systems is the short cycle time of the control loop of 10 - 60 msec. These systems are called Fast-Real-Time (FRT) systems in contrast to other control systems where the cycle time is measured in seconds. In nuclear power plants the time from realizing temperature trouble until starting the emergency shutdown procedure may be 3 seconds (Fetsch, Gmeiner, Voges, 1981).

The software design and development process has to consider the special requirements of safety critical FRT systems (part 3). The advantages of High Order Languages (HOL) for general software development are well known. In part 4 of this paper the languages Fortran, Pearl, "C" and Pascal are evaluated for FRT-systems. Finally "C" has been chosen for the development of the CCV III-test rig and a cross software test system. Both systems are described in part 2. Part 5 comments on the conversion from "C"-programmed systems to ADA.

2. 2 FRT SYSTEMS

Software for digital FBW systems as realized by MBB in the instable CCV II-F104G testbed (Beh, AGARD CP No. 260) has been programmed in assembler up to now. CCV III is a MBB test rig for FBW guidance and control systems of next-generation instable military aircraft, and one purpose of the eight-computer-rig was testing the suitability of HOL to implement such systems. As shown in Fig.1, CCV III is a closed loop simulation with MIL-STD-1553 B data bus link. The computer stations are:

- o Three PDP 11/34 used as guidance and control computers (FF_1, FF_2, FF_3)
- o One PDP 11/34 used as device simulator for non-existing devices
- o One PDP 11/34 used as cockpit terminal
- o One PDP 11/45 used as symbol generator for cockpit displays
- o One VAX 11/780 used as simulator of CCV II-F104G
- o One SABRE X used as data record/reproduce system
- o One µP-Buscontroller

The PDP 11/34's are the testbeds for HOL evaluation. The FF-computers have an additional DMA-link for data exchange; the closed loop cycle time is 30 msec. The CCV II-F104G simulation has been chosen to compare system efficiency directly with the CCV II-F104G test aircraft.

*) present address of B. Güsmann: LITEF, D-7800 Freiburg

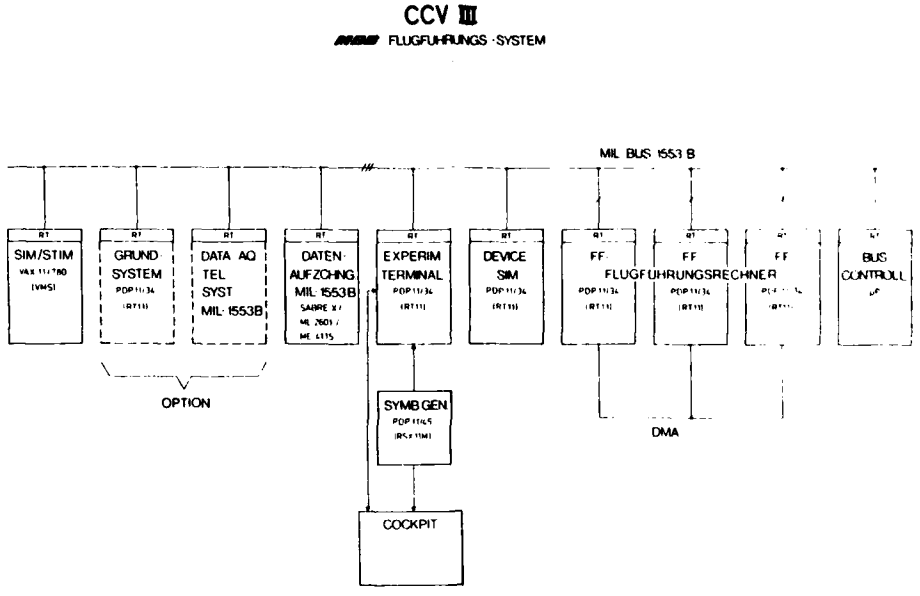


Fig.1 CCV III

The second system is a cross software test system for parallel testing of safety critical software as shown in Fig.2. The PDP 11/70 stimulates the test unit, computes the safety critical software in parallel, and compares the results.

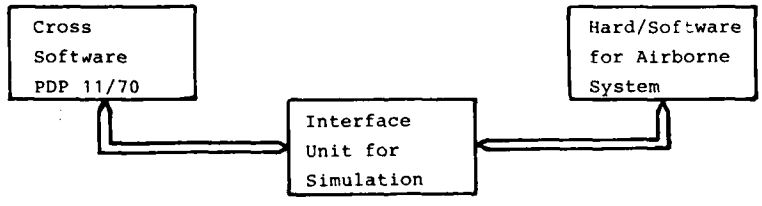
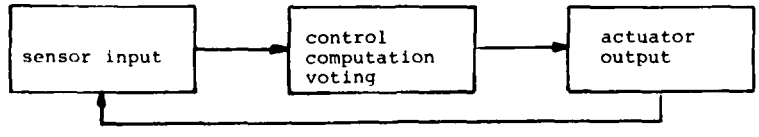


Fig. 2 Cross Test System

3. REQUIREMENTS FOR THE DEVELOPMENT OF FRT SOFTWARE

Digital control of instable aircraft is a highly safety critical item. The well known advantages of HOL will simplify program development, however, HOLs which do not generate readable assembler code cannot be relied on to guarantee system safety. The typical loop is



No multitasking capability is necessary.

For standardization there must be

- only one language for the entire system.

This is required for application programming as well as for systems programming. A standardization of systems programming is also recommended in civil aircraft (ARINC Characteristics 701, 1979).

The requirements for testing, validation and quality assurance aspects are:

- software with simple modular structure. This guarantees that there is no data conflict (jamming) on the communication lines.
- tight task-to-CPU dedication in a multiprocessor system
- machine-level debuggable code
- no global optimization

In the near future the available hardware for onboard computation will not be sufficiently powerful to perform without difficulties all the desirable workload for strap down-, air data-, control law computation, control and redundancy management, I/O-handling etc. Therefore we need:

- fast code
- compact code
- stand alone capability

The final requirement for cost efficiency is:

- commercially available compiler for micro-computer systems.

4. HOL FOR PRESENT FRT SYSTEMS

To implement the two systems described in part 2, four high order languages have been considered: Fortran, Pearl, Pascal, "C" (all are commercially available). The first three languages are known for application programming, "C", although more than 10 years old, has become well known in Europe during the last 3 years as a quasi-standard for 16 bit microcomputer implementation; UNIX is written in "C" (Kernighan, B., W., Ritchie, D., M., 1978).

The language test for FRT-applications was performed in several steps. Since the runtime measurements depend strongly on the quality of the available compilers, the arguments in this part are therefore restricted to language inherent features.

(a) the first step was to convert and code the F104G-control program including I/O-routines using the four languages. The computer for all tests was a PDP 11/34. The software implementation of the mathematical part looked very similar in each language with the restriction of Fortran's lack of modern structures. As an example, Fig.3 illustrates the "C"-program for matrix * vector multiplication: y = Ax.

```
# define N 100
main  (  )
{       register i, j;
        int a [N] [N] , x [N], y [N] ;
        for (i = 0; i< N; i++)
          {  y [i] = 0;
             for (j = 0; j<N; j++)
                 y [i] = y [i] + a [i][j] * x [j] ;
          }
}
```

Fig. 3

The runtime of the mathematical part was also similar for each language, differences appeared in the I/O-time and the amount of runtime support.

Pearl is a process language with a general I/O-concept. At the time of the test, the runtime support was not modular enough for compact and fast code. Therefore Pearl and the poorly structured Fortran were not further investigated;

(b) the second step was the implementation of a typical part of the redundancy philosophy including DMA-communication. The necessary assembler support for Pascal turned out to be four times greater than for "C";

(c) as third step the Tornado control law was implemented by "C" and a new Pascal-version;

(d) several small test programs were written and the generated assembler codes were compared.

The detailed results are given in "Programming of Fast-Real-Time Systems" (Güsmann, B., Hansen, R., 1981). With respect to those features of "C" and Pascal, which could be compared directly, the 1981-tests showed an advantage of "C" concerning code size and runtime. Meanwhile for the PDP 11-family the OMSI Pascal II is available and this should reduce the code size and runtime advantage of "C" on PDP 11 for application programs.

Furthermore the tests demonstrated the importance of systems programming for FRT-systems. At the beginning of an implementation the ratio between system software and application software is:

system software : application software $\sim$ 2 : 1

It is the system software which makes I/O fast and safe. In this area "C" offers better language features than Pascal which was proved by step (b). Another advantage of "C" is the standardization of available compilers based on the same book (Kernighan, B., W., Ritchie, D., M., 1978) whereas there are a number of Pascal dialects.

Fig. 4 shows a list of hardware orientated "C"-operators which have no equivalents in Standard-Pascal (Jensen, K., Wirth, N., 1978).

++	increment	&	bitwise and
--	decrement	^	bitwise excl. or
>>	bit shift right	|	bitwise incl. or
<<	bit shift left	~	one's complement

Fig. 4

These operators are very useful for driver implementation, especially for A/D and D/A conversion, and for masking bit messages from the MIL-STD-1553B bus in CCV III.

Other features of "C" are register variables as shown in Fig. 3. The register declaration is a hint to the compiler to place variables in registers for fast computation. The last instruction line of Fig. 3 can also be written as

```
y [i] += a [i] [j] * x [j] ;
```

E_1 op = E_2 is equivalent to $E_1 = E_1$ op E_2, but E_1 has to be evaluated only once.

For I/O-handling the access to hardware addresses is indispensable. This is achieved via the pointer concept:

```
int      * p;
p  =  0777570;
putchar (*p) ;
```

These instructions transfer the contents of address 777570 to output. The Pascal pointer concept, on the other hand, is more restrictive, as well as the whole I/O-concept.

Finally, separate compilation and compiler control directives such as "include"-statements as required in MIL-STD-1679, facilitate the handling of large program systems. These program features are not required for Pascal. The test results are characterized by an efficient code generated by the "C"-compiler with a small running time overhead.

"C" generated assembler code $< 1.2 *$ hand generated assembler code

The result of the HOL test is that systems implementation is an important factor for FRT systems. All four languages showed similar performance for mathematical programming, but for I/O and systems programming, "C" has evident advantages. "C" has been chosen for the implementation of both MBB-systems described in part 2.

Most of all coding could be done by "C", only a small part of assembler support was necessary. The ratio for the two systems is:

"C"-Code: Assembler $\geq$ 95% : 5%

5. ADA

Most people who work with several software systems and have programmed with different HOL feel the demand for HOL standardization. Standardization of onboard software is one of the requirements in part 3. The enormous effort to promote ADA as a standard for embedded systems, and the possible life cycle of 20 years for weapon systems, raises the question of converting "C"-programmed systems to ADA-programmed systems. Here are some comments concerning guidance and control systems.

Formal transition:

- "C" and ADA strongly support modularity, programs can have similar structure in both languages.

- The programming of mathematics is similar in both languages.

- Systems programming will look different because ADA has a different pointer concept and does not have the hardware orientated operators of Fig.4. Access to hardware addresses in ADA is given by the 'use at' address specification and address attribute. Efficient byte / word-masking and handling can be done by writing routines using in-line-assembler which is supported by ADA. "C" has no inline-assembler feature and this was not missed during CCV III implementation because all operations could be formulated in "C".

Practical transition:

- The ADA-compilers must be examined with respect to the readability of the generated object-code. Can quality assurance be accomplished without transparency on object code level? At the moment for ADA programs designed for general user (e.g. industrial projects) patches will still be necessary (Bennett, P., A., 1982).
- One of the ADA design goals was safe programming, but even ADA requires careful design and cannot prevent program errors, for instance errors based on multiple declaration of names in different program levels (Ref. Man. ADA, 8.3, 8.4).
- "C"-generated code will be a measure of system-efficiency. It is not yet predictable how fast ADA-generated code will be and whether faster CPU's will compensate runtime overhead or whether faster CPU's will be used for greater onboard workloads.

The decision of changing-over to ADA-programmed safety critical FRT systems requires a workload analysis for future systems and a test with existing ADA-compilers analogous to part 4.

The items which influence the choice of "C", ADA, assembler parts are

- compiler availability
- code efficiency
- code transparency (assembler readability)
- faster CPU, faster memory
- greater onboard workload
- methods of validation
- acceptance by Q.A.

6. CONCLUSION

Four languages, Fortran, Pearl, "C" and Pascal have been tested at MBB for the implementation of safety critical FRT-systems. "C" has been chosen for implementation because it fulfils the requirements of part 3:

- one language for the entire system
- readable assembler is generated, no global optimization
- compact and fast code
- the CCV III programs are running stand-alone
- "C" is available for M 68000, Z 8000, 8086 and others.

The implementation of the CCV III-rig has been finished in Dec. 81. Since that time the rig has been tested by F104G pilots and demonstrated to visitors without any software errors being detected. This demonstrates the reliability of "C".

"C" is recommended for present FRT-systems. For future FRT systems practical tests and experiences with ADA-compilers are needed in order to evaluate ADA as the appropriate language choice.

REFERENCES

- Arinc Characteristics 701, "Flight Control Computer System", Aeronautical Radio Inc., Annapolis
- Aviation Week & Space Technology, March 31, 1980, "Software Key to ALCM Choice"
- Aviation Week & Space Technology, Sept. 22, 1980, "First DC-9 Super 80..."
- Beh, Korte, Löbert — "Stability and Control Aspects of the CCV-F 104 G", AGARD Conf. Proc. No. 260
- Bennett, P., A., — 1982, in P. Reid: "Keynote Address and Industrial Viewpoint", ADA UK NEWS JAN
- Fetsch, F., Gmeiner,L., Voges, U., — 1981, "Entwurf eines hochzuverlässigen redundanten Mikrorechnernetzes", Informatik-Fachberichte 50, Springer
- Güsmann, B., Hansen, R., — "Programming of Fast-Real-Time Systems", München, MBB/S/PUB 45
- Jensen, K., Wirth, N., — 1978, "Pascal", Springer

- Kernighan, B., W., Ritchie, D., M., 1978, "The C Programming Language", Prentice-Hall

- MIL-STD-1679, 1978, "Weapon System Software Development"

- Reference Manual for the ADA Programming Language, Proposed Standard, 1980, United States DOD

Acknowledgements

The research on CCV III was fully supported by the German Bundesministerium der Verteidigung.

USABILITY OF MILITARY STANDARDS FOR THE MAINTENANCE OF EMBEDDED COMPUTER SOFTWARE[1]

Norman F. Schneidewind
Naval Postgraduate School
Monterey, CA 93940 U.S.A.

SUMMARY

Several military software standards were examined and evaluated with respect to their applicability and usability for maintaining embedded computer software. These standards included the following: Department of the Navy Tactical Digital System Documentation Standards, SECNAVINST 3560.1; MIL-STD 1679, Navy Military Standard for Weapon System Development; and Weapon Specification 8506. These standards were discussed from three standpoints: (1) the degree to which they support the use of newer software development technologies (e.g., requirements analysis methodologies) for improving software maintenance; (2) the effect of the microcomputer and its software development environment on the application of these standards; and (3) the extent to which these standards enhance traceability (tracing the various levels of related documentation). These aspects required a reevaluation of the applicability of software standards. A recommendation is made to use the A7-E Aircraft software redesign project as a model for improving (1) and (3) in the three standards. Item (2) was judged to be not relevant to the development of software standards.

1. INTRODUCTION

This paper addresses the question of how useful military software standards are for maintaining embedded computer software. Our discussion builds on previous studies of this topic (SCHNEIDEWIND, N.F., Feb. 1982) which involved an analysis of the following United States Navy publications:

- Military Standard (MIL-STD) 1679;
- Weapons Specification (WS) 8506; and
- Tactical Digital Systems Documentation Standards, SECNAVINST 3560.1.

(Note: It is recognized that, technically, only the first document is a standard. For ease of exposition, all three are referred to as "standards" in this paper.) The question posed by the previous research study was: Could these standards, accompanied by basic program documentation, such as a listing, provide adequate quidance for a new programmer to maintain software, such as that found in the Trident Command and Control Subsystem? These standards were reviewed with respect to the following criteria:

- design approaches for achieving good maintainability;
- specification and documentation requirements for achieving good maintainability; and
- testing approaches for achieving good maintainability.

With some significant qualifications, it was concluded that these standards were adequate for maintenance purposes. However, it was pointed out that these standards were developed for use in design and not for maintenance specifically. (The interested reader may find the details in the references which have been cited.)

Now, an excellent standard would recognize the linkage between software design and maintenance and would specify design practices that are conducive to maintenance. The problem seems to be that standards of the type which have been referenced were developed prior to the time when maintenance was recognized as an important phase of the software life cycle and prior to the realization that maintainability must be designed into the software. Software standards should be revised to reflect this important concept. Also, advances in software requirements analysis and design methodologies, coupled with the significant use of microcomputers in embedded computer systems, have led to the need to update military software standards to reflect the realities of newer design and programming environments. Improvement in design approach enhances maintainability; the use of microcomputers, on the other hand, presents new problems for the software development agency due to the limited software development tools which are available in many microcomputer software development facilities. However, it is an open question as to whether the increased use of microcomputers for embedded systems is aiding or retarding the production of maintainable software. Although many microcomputer software production facilities are low-level, oriented to assembly language programming, the trend is for microcomputer software to be developed on larger host machines, using elaborate program development tools on an interactive basis, and down loaded to a development system and eventually to the target machine (ZIEGLER, S., Feb. 1981). Also to be noted is the trend toward moving system functions out of software and into hardware (KAHN, K.C., Feb. 1981). This trend may lead in the future to more emphasis on chip certification and less on software validation. As contrasted to advances in software design and programming methodology, it is not clear that software standards should be significantly changed just because computers get smaller and programming environments change. The desirable objectives of, for example, producing code which can be changed without upsetting the rest of the system, remains valid independent of the particular form, size, speed or configuration of the hardware-software system.

---

[1]The research reported in this paper was sponsored by the Command and Control Systems Maintenance Agency, U.S. Navy, Newport, RI.

Transcending the aspects of improved software design methodology and changing computer technology, is the need to trace both errors in the software and design decisions (which many times are related to errors) to the pertinent technical information. The need for traceability is independent of software design methodology and the particular computer technology which is used in a system. However, proper use of methodology and technology can greatly improve traceability, particularly with regard to identifying the effects of changes to the software.

Within the context of the question posed at the beginning of this section, we examine the usability of military software standards in the ensuing sections with respect to the following areas:

- requirements analysis methodologies;
- microcomputer software development; and
- traceability.

We conclude with recommendations concerning the effective utilization of these standards in an environment of changing methodology and technology.

2. EFFECTS OF REQUIREMENTS ANALYSIS SYSTEMS ON SOFTWARE STANDARDS

One of the major efforts to improve the quality of software has focused on the development of formal software requirements analysis methodologies (ALFORD, M.W., Jan. 1977; BELL, T.E., Jan. 1977; ROSS, D.T., Jan. 1977; TEICHROEW D., Jan. 1977). Objectives of these and related systems are the following:

- improved quality of documentation with regard to precision, consistency and completeness;
- formal methods of specifying requirements, usually involving the use of a language or format for expressing requirements (LISKOV, B.H., Mar. 1975); and
- separation of system functions so that related functions appear in the same module and unrelated functions appear in different modules, resulting in the creation of independent modules (MYERS, G.J., 1978).

A major ingredient of requirements analysis methodologies is computer-aided analysis, consisting of the following components: a language for expressing requirements; a data base for storing requirements and specifications; an analysis and retrieval system for checking requirements consistency and completeness; and various types of graphics terminal and hardcopy outputs (BELL, T.E., Jan. 1977). The emphasis of these systems is a language aimed at achieving formulism and consistency of expressing requirements. These methodologies do not address to a great extent strategies for translating requirements into a software design.

An effort where design strategies and requirements analysis techniques are directed toward confining the effects of changes to software (hence, improving maintainibility) is the project of the Naval Research Laboratory (BRITTON, K.H., Dec. 1981; HENINGER, K.L. Nov. 1978, Jan. 1980; PARKER, R.A., Nov. 1980) to rewrite the software for the A-7E Aircraft, using the principles of information hiding, separation of concerns and abstract interfaces (PARNAS, D.L., Dec. 1972; PARNAS D.L., May 1978; HESTOR, S.D., Oct. 1981; BRITTON, K.H., May 1981). Central to this effort is the method for decomposing modules. The method proposed by Parnas (PARNAS, D.L., Dec. 1972) is the following:

- every module in the decomposition process is characterized by design decisions which are hidden from all other modules (the information hiding principle); this criterion does not decompose the system into modules on the basis of the time sequence of processing the modules; and
- elements that are likely to change are identified and incorporated into separate modules (device interface modules) in order to minimize the effects of device changes on user modules.

Myers (MYERS, 1978), among others, has sounded a similar theme. He recommends that modules should be partitioned so that relationships between elements within a module are maximized and relationships between modules are minimized. In Myer's terms, this results in high module strength and weak module coupling, and leads to the desirable result of module independence. A major objective of these approaches is to reduce the ripple effect of software changes (that is, the effect on modules external to the changed module).

2.1 STATUS OF STANDARDS RELATIVE TO REQUIREMENTS ANALYSIS SYSTEMS

What is the status of the standards (1679, 8506, 3560.1) relative to specifying the use of requirements analysis methodologies and design techniques (e.g., methods for module decomposition)? MIL-STD-1679, Section 5.2, states that the design shall be a hierarchical structure with the highest level of control logic residing at the top of the hierarchy and computation functions residing at the lower levels. As stated by Myers (MYERS, 1978), the objective is not simply partitioning modules into a hierarchy, but partitioning so that each module is as independent of other modules as possible. This procedure will result in confining the effects of change and, hence, make the software more maintainable. In addition, as indicated by Schneidewind (SCHNEIDEWIND, N.F., Feb. 1982, NPS-54-82-002), although the change in reporting and control procedures specified by 1679 is excellent in Section 5.11.2, the coverage is inadequate regarding separation of software functions by anticipated degree of change. With regard to WS 8506, it makes brief mention of describing major functions and the dependency among functions in Section 5.2. This reference does not elaborate on why or how the information is to be used (SCHNEIDEWIND, N.F., Feb. 1982, NPS-54-82-002). An important use of the information would be for decomposing a system into modules and for the related purpose of achieving module independence. The situation is worse in the case of SECNAVINST 3560.1. This document is notable for the great amount of detail presented pertaining to function and interface descriptions, data

exchange, program resource budgets, etc. (SCHNEIDEWIND, N.F., Feb. 1982, NPS-59-82-003). However, there is an absence of material dealing with designing for change.

In addition to the above gaps in coverage, the standards pre-date the use of software requirements analysis systems, such as those described by Alford (ALFORD, M.M., Jan 1977). Naturally, the older the standard, the more obsolete it is relative to advances in requirements analysis methodologies and software design. So these remarks should not be construed as criticisms of the standards, but as indications of the need to consider updating the standards for the purpose of bringing them into line with software engineering techniques which look quite promising. It is necessary to emphasize that methods proposed by Parnas, Myers, Teichroew and others have not reached the stage of accepted practice by a large segment of the software engineering community. For one thing, there must be further demonstration of improvements in software maintainability on large systems before these procedures will graduate to the status of standard practice. However, we contend that the approach in standards development should be to lead rather than to follow developments in software engineering. There is obviously more risk associated with this policy, but its successful implementation can prevent a standard from being obsolete before it is even issued.

In summary, with regard to the areas of requirements analysis and software design, the standards are weak and in need of upgrading to include the following requirements:

- ° decomposition of a system into modules for the purpose of confining the effects of software changes by using techniques such as:
  - hiding device characteristics from user programs and,
  - designing modules so that related elements are contained in the same module and unrelated elements are contained in different modules (high module strength and low module coupling); and
- ° employment of a requirements analysis system for the purposes of:
  - standardizing the language in which requirements are stated, and
  - providing computer-aided tools for storing, analyzing and retrieving requirements to ensure consistency and completeness.

3. THE EFFECT OF MICROCOMPUTERS ON STANDARDS DEVELOPMENT

As suggested in Section 1, standards development should be independent of the characteristics of the hardware and software employed in an application. A standard should require the developer to employ sound software engineering practices. These techniques become 'sound' by evolving from theory to standard practice through a process of proposal, debate, demonstration, use, concensus and acceptance. This process should not be influenced by whether, for example, an application is implemented in a centralized main frame or a distributed microcomputer system. The aspect that is affected by the choice of technology is the ability of the developer to meet the standard (e.g., conforming to a requirement for using structured programming with assembly language versus high level language). For example, if crude program development and test tools are used for implementing microcomputer software, or any software for that matter, traceability will be difficult to achieve. More will be said about traceability in a later section.

Concern about the peculiarities of microcomputer software development may evaperate in the near future. As mentioned in Section 1, the trend in microcomputer software development is toward using the types of tools which have been used for some time in the minicomputer and mainframe areas. For example, if we compare two publications which are only one year apart (OGDIN, C.A., 1980) and (MARKOWITZ, R., Feb. 1981), we find that the former, in describing microcomputer programming environments, stresses hexidecimal coding, prototyping systems, computer evaluation kits, portable front panels, single board computers and microcomputer development systems. In contrast, Markowitz's article describes the architecture of the iAPX 286 in terms of memory management, segmented memory, protection, and various priviledge levels - characteristics of large machines. Zeigler (ZEIGLER, S., Feb. 1981) talks about extensions to the ADA language which INTEL has developed in support of the 432 architecture.

None of the standards makes reference to the use of microcomputers. This would obviously be the case for 3560.1 and 8506, since their publication pre-dated significant use of microcomputers. Although 1679 was published during the era of microcomputers, mentioning the use of this technology in the standard would have been inappropriate. As stated by Cooper (COOPER, J., Aug. 1981), in describing the development of 1679, the single most important rule of a MIL-STD is that it can only specify what is required, not how to satisfy a requirement. However, it must be noted that 1679 does not seem to be entirely faithful to this rule, since it calls for the use of structured programming constructs (Section 5.3), and top down design and high order languages (Section 5.5), as examples of many "how to" provisions.

Based on the reasons given in this section, we conclude that the standards should not be modified to incorporate provisions that deal with the development of microcomputer software for embedded computer systems.

4. APPROACHES FOR IMPROVING TRACEABILITY OF STANDARDS

A prerequisite for achieving traceability and, hence, maintainability is to have planned for the software to change when it was designed and to have designed the software correspondingly, so that changes can be easily traced through the documentation in order to identify the relevant inputs, outputs, data base, modes of operation, conditions, etc. The A-7E Aircraft documentation (HENINGER, K.L., Nov. 1978) does a good job of providing traceability because it was designed with change in mind. Some of the formats which are useful for achieving traceability are the following:

- Event Tables which relate modes, events, and actions;
- Condition Tables which relate modes, conditions and actions or values; and
- Selector Tables which relate modes and mutually exclusive characteristics of modes.

In the above, the following meanings apply:

- mode - system state;
- condition - expression whose value is true or false and characterizes the system for a measurable time;
- event - a condition which changes from true to false or vice versa at a specific moment in time;
- action - evaluation of a function; and
- value - expression or output data item value.

In this system of documentation, if an event (e.g., ground distance to a reference point) were to change when the radar update mode is entered, it would be possible to ascertain the fact that this combination affects an action taken by the pilot relative to cursor enable (output data item). In general, design decisions, module decomposition and module dependencies are made explicit in the system of software design and documentation. Other useful aspects of the documentation include a dictionary of commonly used terms and a section dealing with subsets - a part of the system which is isolatable from the total system, performs part of the services provided by the total system and uses less computer resources than the total system. One of the ideas of subsets is to be able to reassemble a smaller system and thereby save on resources if the entire system is not utilized (e.g., a particular weapon is not available or used).

Weaknesses in this system seem to lie in the areas of tracking changes in outputs to inputs and data bases, where applicable, and, in some cases, lack of clarity of definitions as they relate to various tables. Also, although we subscribe to the objectives of information hiding, which provides the underpinning of the A-7E project, it is our opinion that the design procedures and terminology which are necessary to implement information hiding could be confusing to software engineers. We prefer to think of the process of requirements analysis as making requirements explicit, rather than hiding certain ones, and to only embody a requirement in a module when the requirement is relevant to that module; otherwise, the requirement is implemented in a module where it is relevant. Requirements which are common to two or more modules are contained within a separate module, rather than within the given modules themselves. Additionally, requirements which are likely to change should be quarantined and placed in a limited number of modules rather than being spread across many modules.

Nevertheless, on the whole, the A-7E Aircraft project and a related project (HESTER, S.D., Oct. 1981) would provide an excellent model for revising the standards to incorporate software design practices which specifically address the need to account for future change to the software. This would be a particularly powerful approach if coupled with the use of one of the requirements analysis systems (ROSS, D.T., Jan. 1977) for providing computer-aided requirements analysis tools and for supporting the requirements analysis which must precede the software design.

The primary author of MIL-STD 1679 (COOPER, J., Aug. 1981) feels that with only two years use, it would be premature to revise it. However, neither 1679 nor the other standards are strong in the vital area of traceability (SCHNEIDEWIND, N.F., Feb. 1982). We feel, therefore, that because the volatility of software is so great and affects maintenance so significantly, that a standard must explicitly provide for change in the design process in order to achieve traceability in the maintenance phase. Note that this characteristic of a standard is not the same thing as a change control procedure, which is a part of 1679. To use a medical analogy, the recommended approach involves using preventive medicine early in the life of a system in order to avoid emergency surgery at a later date.

5. CONCLUSIONS AND RECOMMENDATION

Three important areas relative to software standards have been considered which potentially impact on software maintainability:

(1) requirements analysis and software design methodologies;
(2) microcomputer software; and
(3) traceability.

It is concluded that (1) and (3) should be improved in SECNAVINST 3560.1, WS8506 and MIL-STD 1679 and that (2) is not appropriate for inclusion in a standard.

Furthermore, it is recommended that A7-E Aircraft software redesign project be used as a model for improving the standards relative to (1) and (3).

6. REFERENCES

Alford, M.W., Jan. 1977, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 60-69.

Bell, T.E., Bixley, D.C. and Dyer, M.E., Jan. 1977, "An Extendable Approach to Computer-Aided Software, Requirements Engineering", Jan. 1977, IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, pp. 49-60.

Britton, K.H. and Parnas, D.L., Dec. 1981, "A-7E Software Module Guide", NRL Memorandum Report 4702, Naval Research Laboratory, Washington, D.C.

Britton, Kathryn Heninger, Parker Alan R. and Parnas, David L., Mar. 1981, "A Procedure for Designing Abstract Interfaces for Device Interface, Modules", Proceedings of the 5th International Conference on Software Engineering, San Diego, CA pp. 195-204.

Cooper, Jack, Aug. 1981, "Development of MIL-STD-1679, Software Engineering Standards Application Workshop", San Francisco, CA, pp. 139-143.

Heninger, Kathryn, L., Kallander, John W. and Shore, John E., Nov. 1978, "Software Requirements for the A-7E Aircraft", NRL Memorandum Report 3876, Naval Research Laboratory, Washington, D.C.

Heninger, K.L., Jan. 1980, "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications", IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, pp. 2-13.

Hester, S.D., Parnas, D.L. and Ulter, D.F., Oct. 1981, "Using Documentation as a Software Design Medium", The Bell Systems Technical Journal, Vol. 60, No. 8, pp. 1941-1977.

Kahn, Kevin C. and Pollack, Fred, Feb. 1981, "An Extensible Operating System for the Intel 432", Digest of Papers, Spring COMPCON 81, San Francisco, CA., pp. 398-404.

Liskov, Barbara, H. and Zelles, Stephen N., Mar. 1975, "Specification Techniques for Data Abstractions", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 7-19.

Markowitz, R., Feb. 1981, "Software Impact on Microcomputer Architecture: A Case Study", Digest of Papers, Spring, COMPCON 81, San Francisco, CA., pp. 40-48.

MIL-STD-1679 (NAVY), Dec. 1978, "Military Standard, Weapon System Software Development", Department of Defense, Washington, D.C.

Mosak, Allan, Feb. 1982, "Structured Programming Can Be Applied to Microprocessor, Even by Novices: A Review of Structured Microprocessor Programming", IEEE MICRO, Vol. 2, No. 1, pp. 63-71.

Myers, Glenford Jr., 1978, "Composite Structured Design", Van Nostrand Reinhold Company, New York, N.Y.

Ogdin, Carol, Anne, 1980, "Microcomputer Management and Programming", Prentice-Hall, Englewood Cliffs, N.J.

Parker, Robert A. et al., Nov. 1980, "Abstract Interface Specifications for the A-7E Device Interface Module", NRL Memorandum Report 4385, Naval Research Laboratory, Washington, D.C.

Parnas, D.L., Dec. 1971, "On the Criteria To Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058.

Parnas, David L., May 1978, "Designing Software for Ease of Extension and Contraction", Proceedings of the 3rd International Conference on Software Engineering, Atlanta, GA, pp. 264-270.

Ross, D.T. and Schoman, K.E., Jr., Jan. 1977, "Structured Analysis for Requirements Definition", IEEE Transactions of Software Engineering, Vol. SE-3, No. 1, pp. 6-15.

Schneidewind, N.F., Feb. 1982, "Software Maintenance: Improvement Through Better Development Standards and Documentation", Naval Postgraduate School, NPS-54-82-002, Monterey, CA.

Schneidewind, N.S., Feb. 1982, "Evaluation of SECNAVINST 3560.1 Tactical Digital Systems Documentation Standard For Software Maintenance", Naval Postgraduate School, NPS-54-82-003, Monterey, CA.

Schneidewind, N.F., Feb. 1982, "Evaluation of Maintainability Enhancement For TCP/TSF Revision 6.0 Update .20", Naval Postgraduate School, NPS-54-82-004, Monterey, CA.

SECNAVINST 3560.1, 8 Aug. 1974, "Tactical Digital Systems Documentation Standard", Department of the Navy, Office of the Secretary, Washington, D.C.

Teichroew D. and Hershey, E.A., III, Jan. 1977, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing", IEEE Transactions on Software Engineering, Vol. SE-3, Nov. 1, pp. 41-48.

Weapons Specification, WS-8506, Rev 1, Nov. 1971, Requirements for Digital Computer Program Documentation, Naval Ordinance Systems Command, Department of the Navy, Washington, D.C.

Zeigler, Stephen, et. al., Feb. 1981, "The Intel 432 Ada Programming Environenment", Digest of Papers, Spring COMPCON 81, San Francisco, CA., pp. 405-410.

SOFTWARE CONFIGURATION MANAGEMENT

AT WORK

Dr.ing. Jan Tore Pedersen
A/S Kongsberg Vaapenfabrikk
Postbox 25
N-3601 KONGSBERG
Norway

1. A/S KONGSBERG VAAPENFABRIKK

A/S Kongsberg Vaapenfabrikk is a Norwegian company founded in 1814. Initially it was set up to produce hand-arms for the Norwegian army. Today we have quite a variety of products in the areas of mechanical engineering, electronics, and computer systems. Defence systems account for approximately 50% of our revenue. In 1981, the total revenue was in the order of 350 mill. US dollars.

Our computer based products are turn-key systems in the following application areas:

- Mechanical engineering, Computer-design and production
- Cartography, from stereoscopic pictures to maps
- Supervision and control of oil production platforms and hydro power plants
- Dynamic positioning of vessels. IP - systems for anti-collision control and machine-room supervision of ships
- Maritime traffic control
- Simulators
- Defence navigation systems
- Fire control
- Command, Control, and Information Systems

Our defence systems and all other process control systems are implemented as "embedded systems", using our own computers, the KS-500 and the KS-900 (coming shortly).

We have 250 programmers (performing all tasks from specification through coding and testing) in software development and production.

2. SOFTWARE DEVELOPMENT AND PRODUCTION

Kongsberg does not market "off-the-shelf" software systems. Our deliveries of software systems require significant adaptions in order to satisfy the individual customer requirements. Such adaptions are either modifications to existing programs, or development of new ones. One consequence of our major customization effort is that we end up with several versions of the same programs. In order to keep this under control, we need special archiving and system generation tools (we are using UNIX as a basis for our software engineering environment).

More importantly, however, we need special control mechanisms in order to ensure the quality of each delivery, its progress, and its economy. The control mechanism must also be adaptable to the task at hand so that we don't exercise the same bureaucracy for a "trivial" production task with minimum risk and the development of a complex system where the risk is significant.

The most important aspect of any software project is to specify the user requirements. Quality, or ability to satisfy the user needs, requirements, and expectations, is virtually impossible to achieve unless proper requirements specifications are defined. Many will say that this is obvious, but it is a statistical fact that we make most of our errors in the early stages of a project. In addition, the system defects created by these early errors are the most expensive ones to remove after the system is installed. (Ref. 1)

We also have the interesting paradox, illustrated in Figure 1, that even if we are spending the effort in developing a proper user requirement specification (URS), conformance between the developed software system and the URS is the last thing we can verify. In addition, when we are able to do the verification, it is normally time for delivery, and no time for fixing bugs. Consequently, we need a method of working

which results in a continuous validation of the quality of the system as it progresses from initial specification to the finally tested code.

Economically, we find that the most expensive errors, or defects, to remove are those which occur after the system is installed and is supposed to be operational. In order to prevent defects from being part of the system when delivered, we need a controlled development and production process which focuses upon the orderly production of specific intermediate results, or baselines.

3. KONGSBERG'S SOFTWARE CONFIGURATION MANAGEMENT CONCEPT

We started in 1976 on the process of controlling our software activities. Our approach is based on the ideas of conventional Configuration Management (CM), and obeys the principles of:

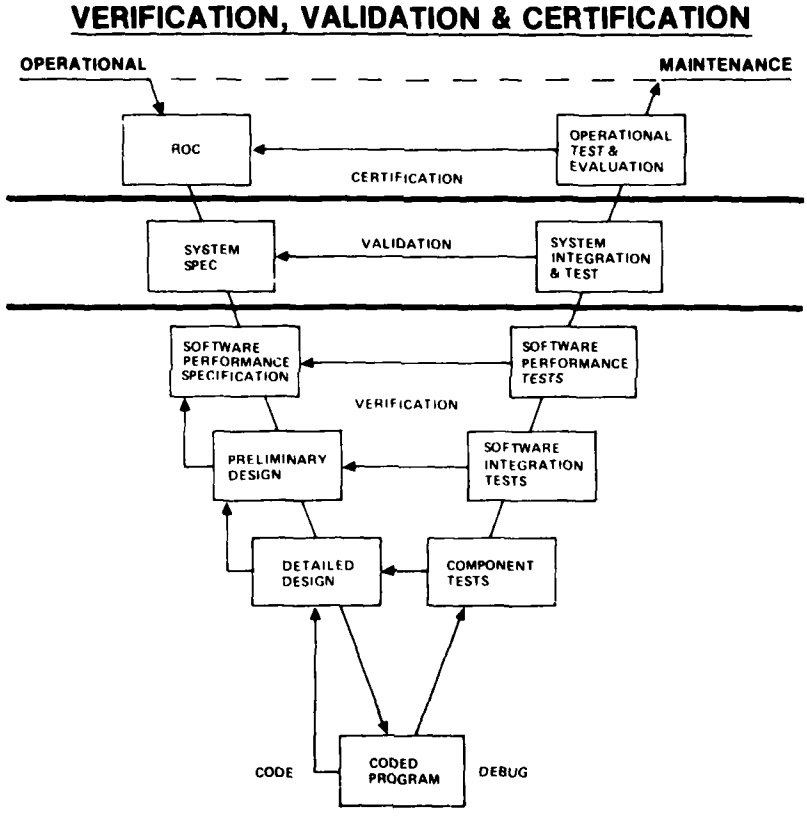


Figure 1. Verifying Software Quality (ref 2)

IDENTIFICATION	-	What does the system consist of?
CONTROL	-	How are change proposals handled correctly?
STATUS ACCOUNTING	-	What changes are really made to the system?
REVISION	-	Will the system under development really satisfy the documented requirements?

Our model for development and production of software systems is shown in Figure 2. It separates the process into specific baselines. The handbook that each project manager receives at the start of a new project, contains a description of all baseline documents which are to be produced as part of the project. The document sequence, the manner in which they are to be produced, the contents requirement for each baseline document, and how and by whom they are to be reviewed and approved. The approval is done through a formal inspection process. When the document is approved, it is formally signed by the approval authorities. Approval authorities are assigned to each document at the start of a new project.

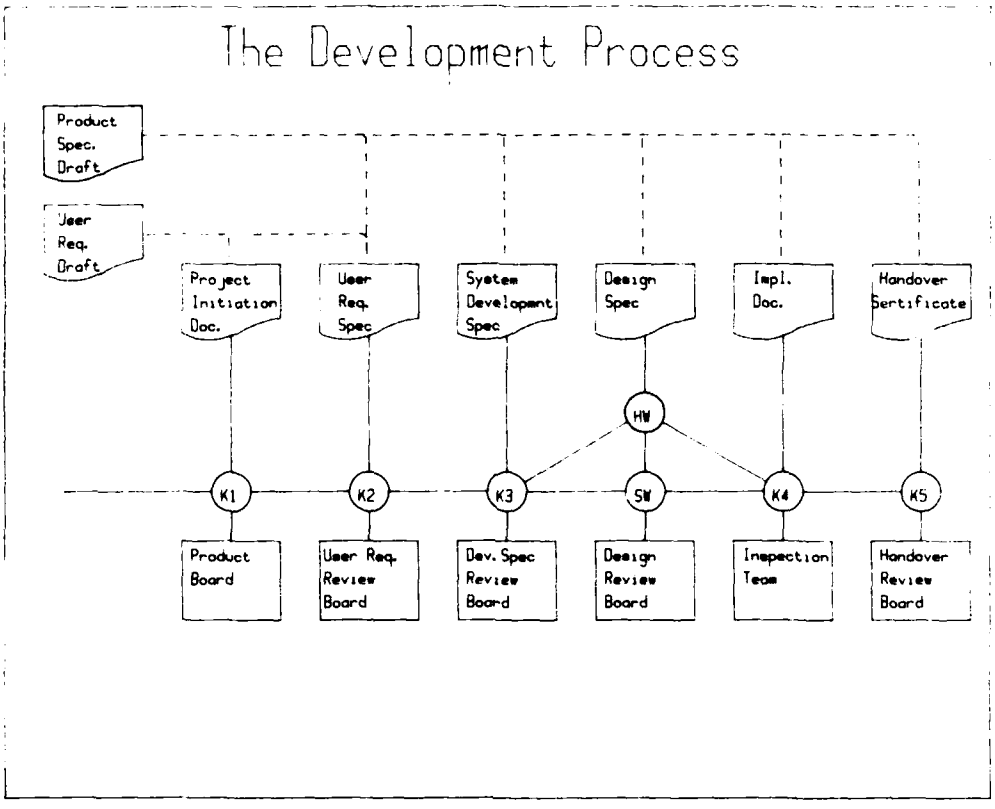


Figure 2. The development model for one of Kongsberg's divisions.

Now that the document is approved, it cannot be changed without the approval of and a new signature from the original approval authorities. If a change is made, the working procedures ensure that all relevant project personnel are informed.

The signature of baseline documents marks a significant milestone in each project. A milestone is not passed unless the relevant baseline document is signed. This formalism simplifies project supervision, because there is now a single, "measurable" criteria for determining the project status, provided all baseline documents are represented in the project plan.

To illustrate the level of detail in the description of the baseline documents, here is the list of contents in our User Requirement Specification as it is specified in the handbook:

1. Scope: introduction to the document
2. Document history
3. Referenced documents
4. Application area
5. User functions
6. Outline user interface
7. Non-functional requirements
8. Quality measures
9. Product safety requirements

The handbook contains guidelines for how to formulate each paragraph.

4. OUR EXPERIENCE

4.1 What does it take to introduce new methodologies?

For more than five years we have introduced Software Configuration Management procedures in various groups within our company, with mixed success.

Our first effort was to prepare a set of production procedures for our highest volume software product. It was a very successful effort, both technically and economically. All the engineers in the initial production group became very enthusiastic as results were achieved.

When the first effort was finished, we believed that the success would sell itself and that other production groups would grasp the new procedures eagerly. But alas ....

It took us more than two years to experience what went wrong after the initial effort. In those two years we spent approximately 200.000 dollars without getting a measurable effect.

Eventually we did an analysis of what went wrong, and we found that three criteria have to be satisfied in a specific group where new methods of working are to be introduced:

- The establishment of the new procedures must be accepted and actively supported by the group management. Some of the staff may in the beginning feel that their workload will increase. The management, who will know that the change will have an overall positive effect, must take the responsibility for the introduction of the procedures, and for ensuring that the staff are motivated to use them.

- Support is a prerequisite.

  Learning a new method of working will always require an effort from each individual. In a normal working environment, where there is a constant struggle to keep up with delivery schedules, there will be a natural resistance to change. Only by allowing for special support resources which have the sole task of preparing the new setup such that "it is easier to follow than the old one", is the establishment guaranteed to be successful.
  Introducing new method is expensive, but our experience is that it is worth the expense.

- The level of ambition must be right.

  Learning is done in steps, and there are limits to the size of the steps a person can take. If we are to improve our methods of working, we must allow ourselves to take one step at a time. What is most important is to find a set of simple guidelines, procedures, or tools which soon become obviously useful for all parties involved.

  The wrong approach may cause a negative reaction such that the people involved spend more time "cheating" the system than they do doing productive work.

After having learnt these lessons, we have introduced our CM procedures into new departments, and so far the new efforts have been successful. We now experience that our projects are better controlled with regard to quality, time schedule and economy.

4.2 How do programmers and customers react?

A natural question is: How do programmers react when confronted with such bureaucracy?

Our answer is that they tend to regard it as a "life insurance":

- The formalized process requiring signatures on each baseline document motivated the participation of users or customers in the early specification phases. When a "customer" has signed the URS, the programmer knows that if he delivers a system according to that specification, he has done his job properly. If there still are differences between what the customer now wants and what the system does, it is because the customer changed his mind without it being reported in the proper fashion, and the programmer is not to be blamed.

- The separation of a project into specific baselines promotes working effort to be spread out over the whole project. Human beings work best under certain pressure. Passing a baseline creates such a pressure. Thus it is made easier to keep schedules. They also have a better feel for what the project status really is.

- Focusing on the baseline documents (with checklists for document content, how they shall be prepared, and how they shall be approved) gives a much better basis for planning new projects. New experiences are continuously being incorporated into the handbook, such that it contains our accumulated experience. This approach makes it easier to estimate project efforts and makes it easier to start elaborating on a new document since all non-technical issues are taken care of in the handbook.

Then, what about the "customer" reactions? Our experience has been surprisingly positive both with internal and external customers.

We expected a problem with having the customer representatives sign the specifications. It was never so for us. In circumstances where the original specifications turned out to be wrong, we either were given an extra order to change the product, or the customer just adapted himself to our system. In our times without formal specifications, we always had to pay the bill if the customer was not satisfied.

We now feel a strong movement within our group of large customers, both industrial and military, to require a formal quality assurance and configuration management system from us as a supplier. In doing so, they take upon themselves the responsibility of accepting their share of the project, which is to have a URS properly developed and designed.

Under circumstances where it is impossible to have a customer signature, let that not be an excuse for not writing a requirement specification. Without a URS, the project will be without a proper foundation.

5. THE MANAGERS ARE RESPONSIBLE

The software configuration management system we have established at Kongsberg utilizes ideas and experiences from other types of industrial and engineering business. It has proved itself very efficient in the cases where it was properly introduced.

Based on the requirements for increased software quality and profitability (productivity), the managers responsible for software activities must now require that programming is developed into an industrial, professional discipline, with all the necessary implications. I do not believe that programmers will do that on their own.

6. REFERENCES

1. Boehm,B.W. 1977: "State-of the art Advances in Software Reliability and Measurement" International Software Management Conference.

2. Munson,J.B.1977: "System Acquisition Guidebook Program for the U.S. Air Force" International Software Management Conference.

CONFIGURATION MANAGEMENT AND THE ADA PROGRAMMING SUPPORT ENVIRONMENT

Kevin J. Pulford
Marconi Avionics Ltd.,
Elstree Way,
Borehamwood,
Herts,
WD6 IRX

SUMMARY

It is the aim of software development environments to increase the efficiency with which software is produced. One such environment is the Ada Programming Support Environment (APSE) initiated by the U.S. Department of Defence. These environments are a great benefit to programmers making some of their tasks much easier. They also offer great opportunities to monitor and control software development. This in its turn will affect the way that projects are organised and run, and it will affect project personnel"s jobs to varying extents. The way that projects will be affected by the adoption of an APSE is explored in this paper by considering the way that Configuration Management can be implemented in an APSE.

1. INTRODUCTION

In recent years there has been a growing interest in software development environments. With the development of software engineering as an area of study and as a discipline the idea of centralising and enforcing the discipline with the aid of the computer has become increasingly attractive. Much of this interest was sparked by the development of the UNIX operating system (KERNINGHAM B.", 1981) and the Programmer's Work Bench (IVIE, E.L. 1977). What this work demonstrated was that by putting the right features into an operating system, the development of software was made a lot simpler and easier. This has been born out to a large extent by the fact that the UNIX operating system has been adopted on a wide range of mini-computers and that there are a number of operating systems being marketed which bear a strong resemblence to UNIX.

More recently much interest has centred around the Ada language and its associated environment (BUXTON, J. N. 1980). The development of Ada was initiated by the U.S. Department of Defence as an attempt to improve productivity in real-time military systems. The broad requirements for the Ada Programming Support Environment (APSE) are given in the STONEMAN document (BUXTON, J. N. 1980). Interest in the Ada development is strong on both sides of the Atlantic. This is shown by the fact that the U.S. Army and Air Force, the U.K. Ministry of Defence and Department of Industry and the EEC are all funding the development of an environment which will satisfy the STONEMAN requirements.

It is obvious that the adoption of an environment by a project may mean that its working practices will change. They do not have to change, but to gain most benefit from the environment the way the work and people are organised will need to be examined. To assess some of these implications in this paper the area of Configuration Management will be taken as an example of an area that will be affected. The sort of things done in Configuration Management, how project personnel interact with Configuration Management and how these can change with the adoption of a development environment such as APSE will be considered.

This paper starts by describing the sort of activities which take place in Configuration Management so as to set the scene and define a common terminology. It then goes onto give an overview of the relevant features of the STONEMAN requirements. This leads onto a discussion of how the APSE can support Configuration Management and finally a discussion of the implications of using APSE for Configuration Management.

2. CONFIGURATION MANAGEMENT

2.1. Configuration Management Activities

The description given here of Configuration Management activities is loosely based on the terminology given in (BERSOFF, E. H. ....1979). Obviously practices will vary in detail from company to company and even between projects in the same company, but the main ideas will be essentially the same.

In (BERSOFF, E.H. 1979) four component activities are identified within Configuration Management. These are:-

a) Configuration Identification

b) Configuration Control

c) Configuration Status Accounting

d) Configuration Auditing

Configuration Identification is concerned with labelling the bits of the software as it evolves and how these bits of software are related together to produce a version of the system. The most elementary item to be identified is termed a "Software Configuration Item" (SCI). In a development plan there will be a number of baselines. These are planned ahead and are set as milestones in the plans. Such baselines can consist of such objectives as "all major software components functionally specified", "all software modules designed" or "all software modules tested". Generally these baselines will appear as a hierarchy of SCI's. Configuration Identification allows the state of the system to be described and also,by reference to the baselines,what it is that is being aimed at.

Configuration Control. During the development of a system the software is bound to change for one of the following reasons:-

a) The software does not meet its requirements in all respects.

b) It has been found necessary to change the hardware

c) The software requirement is changed

d) There are problems with project plan schedules

e) A way of saving costs is needed.

There must be some mechanism to regulate changes to ensure they are incorporated in a controlled way and that no changes are overlooked. It must take into account the implications of a change to other aspects of the system. These implications may be associated with functional, economic or schedule aspects of the system. In the practice of Configuration Control there are three basic components.

a) A formal mechanism for raising, approving (or disapproving) and checking proposals.

b) A system of documentation to support the change mechanism.

c) Procedures for controlling the application of changes to the system.

Configuration Status Accounting provides a mechanism to maintain a record of how the system evolved and the current state of the system design relative to published baseline documents and written agreements. This actively involves recording all the SCI's and their related data including changes. It thus supplies data for the other three functions.

Configuration Auditing checks on the relation of the current status of the system to baselines and system requirements. This includes verifying that what is planned to be in a baseline is actually present. It validates that no design decisions in the current baseline has compromised the customer's requirements. It also provides a mechanism for the update and control of baselines.

There are a number of formal mechanisms which interface between the quality activities and the programmer and the customer. The first of these mechanisms is Registration. When a programmer has completed a design specification or a software module, which will be part of the baseline, he passes these onto the Configuration Management mechanisms. This process is formalised as registration. Before items are accepted for registration there will be a check that all the items required for registration are present. Then responsibility for these objects passes from the programmer to Configuration Management. To change these objects after registration means going through a formal change procedure. Thus registration marks the entry of an SCI into the formal change control mechanism of Configuration Control.

A second interface mechanism is the Design Review. Here a group of interested parties meet to review a document or software design or whatever constitutes the SCI in the baseline. A design review meeting may include customer representatives. The review assesses the SCI to see if it satisfies its requirements and is well designed. The review may put forward change proposals which are noted as design review actions. Thus the design review activity supports the auditing function of Configuration Management.

The final interface mechanism is the Configuration Control Board. This is a group of people who assess change proposals and decide whether a change should be adopted and at what point in the development schedule it should be incorporated. They will consider the impact of the change on the requirement costs schedule and on the quality of the system. Thus it is a mechanism for supporting the change control mechanism of Configuration Control.

The view of Configuration Management given above is rather idealistic and in practice the application is often simplified in a number of ways. Design reviews can be informal where no strict minutes are taken, but note is taken of review actions. Often there is no formal Configuration Control Board and the software project management will decide whether to incorporate a change. At the detail design levels design reviews maybe less formal. For instance at a requirements review there may be representatives of the customer, while at a review of module design the most senior person may be only a section leader. To shorten timescales the design may be started before the requirement has been properly reviewed. The decisions when, where and how Configuration Management procedures are applied in the different aspects of a project is a mangement decision which must be based on a careful value judgement.

2.2. Project Roles and Configuration Management.

Having described various activities involved in Configuration Management the way Configuration Management affects the work of project personnel can now be considered. Again a slightly idealistic view has been taken, but it is hoped that there is sufficient reality left to maintain credence. Although, on a project of any size there will be many different jobs, only four roles will be considered here. The various jobs are assumed to consist of elements from the four basic roles. The four roles are:-

a) Programmer

b) Quality Engineer

c) Software Project Manager

d) Librarian

Under this hypothesis, for instance, a Section Leader or Chief Programmer can be considered to combine the roles of programmer and project manager. He will still do some design and look at code, but will also manage his team in an analogous way to that in which a project manager runs a project.

The programmer's role is central to a software project. It is his output that forms the product and is the thing that is controlled by Configuration Management. A programmer's basic sequence of actions is shown in Fig.1. This is an idealised view; in practice some of the reviews may be informal or even not be present at all. The programmer will go through this sequence a number of times and under different circumstances. He will design the software from scratch based on a requirement: he will modify an already existing design based on a set of changes approved by the Configuration Control Board or he will integrate several modules. The requirements and approved changes received by the programmer will be under the control of Configuration Management and will thus be issued via an official issuing mechanism. When modifying or integrating software the programmer will also be issued with code also via the Configuration Management mechanism. While the programmer is developing the software he will have versions which are outside Configuration Management Control. When an item is finished and approved by a design review it is then passed into Configuration Management control via the registration mechanism. The registration of things like code and binary often involves producing copies on some physical medium such as paper tape for lodging with the librarian.

Conventionally the quality engineer's role is to advise on quality matters and monitor quality practices used by the project. Here we extend this role to include actually carrying out the quality procedures as well.

The quality engineer's role is not normally as highly structured as the programmer's. His attention can be on several parts of the project at once, at several levels and over a number of versions of the system.

Early in the project he will be involved in looking at the project plans and proposed quality procedures. In particular in the Configuration Management area he will check that the baselines have been defined adequately and that there is proper identification of SCI's. Also at this stage he will be involved in the design and implementation of the Configuration Control mechanisms.

During the project a recurring action of the quality engineer is his participation in design reviews. At these reviews he is responsible for recording the review actions and for raising change proposals for the Configuration Control Board.

In preparing for the Configuration Control Board the quality engineer must ensure that the right set of change proposals are put before the board and that the implications of the changes are compiled. He may need help from other project personnel to complete the compilation of the implications.

After design reviews and Configuration Control Board the quality engineer will check that all the resulting actions are propagated through the project and are properly carried out.

Another important activity involving the quality engineer is the quality audit. Here the customer or possibly the company itself will check to see that there are proper quality procedures set up for the project and that they are being followed adequately. The auditor will want to know what the procedures are and will want evidence that they are being followed. Part of this evidence is supplied by the Configuration Management records and comparing them with the actual software produced.

It is the librarian's job to administer the software library project. Every project needs some central position to store its software and related documents. This is normally supplied by the software project library. The librarian will accept items for logging in the library and be responsible for indexing and classifying all the items in the library.

As part of the indexing activity the librarian will keep a list of software identifiers used in the project and act as a source of new identifiers. The acceptance of items into the library is via registration. During registration the librarian will ensure that all the items that should be present for the SCI and baseline are present and all of the changes that are supposed to be incorporated have been. The librarian will then update his records to reflect the completion of an SCI in a baseline. He will issue copies of reference items of code, binaries as well as documents. He is also responsible for the distribution of change information to relevant project personnel. In addition the librarian is responsible for the safe keeping of items in the library and is thus responsible for taking copies of items or what ever action is necessary to ensure that the information in the items is not corrupted.

The Software Project Manager monitors the project's progress and ensures that there is a proper supply of resources to complete the project on time and within the budget.

To monitor progress he needs to know where he is and where he is making for. He can use the Configuration Management data to help him. The baselines are where he is making for, and the fraction to which each baseline is complete is an indication of where he is. Of course, this is only part of the picture and he will ask his programmers for progress on the non-Configuration Managed items they are currently working on.

3. SOFTWARE DEVELOPMENT ENVIRONMENT

A very general definition of a development environment is that it provides a context for the orderly, rational evolution of software systems. It does this via:-

a) A flexible system for the storage and retrieval of code, binary and all related project data.

b) A set of software tools to manipulate the data. Examples of tools are editors, compilers, text formatters, report generators, etc.

c) A friendly user interface to allow the user to interact with the system in a simple and natural way to achieve his aims.

The advantages claimed for this approach are:-

a) Better manageability of the development process. Since a large part of the project data is in the data storage of the environment this makes it easier to control.

b) Improved visibility of the development process for both Management and Quality. Again the central storage of data makes it easier to see what has and has not been done.

c) Easier enforcement of project standards since it should be possible to produce tools to scan sections of data to highlight any deviations from standards.

d) Better integration of the development process through all project phases. Since the data is held centrally the transfer of data is made much simpler.

e) Reduction of clerical tasks. Since the data is available on the computer, the collection, transcription and transmission task associated with paper based project control systems is reduced.

Overall these advantages are expected to lead to improved quality and more easily maintained software and lower overall life-cycle costs.

The Ada Programming Support Environment (APSE) is intended to support the development of systems written in the Ada language. The emphasis in the environment is the development of software for real-time embedded computers. The broad requirements given in the STONEMAN document are aimed at providing cost effective support for the whole of the software life-cycle and are as follows. The tool set must be integrated. Each tool should carry out a simple task, but must be easily combined to allow more complex tasks to be carried out. There should be a database to store all project data. It must be possible to assign descriptive attributes to objects in the database and create relationships between objects. It must also be possible to group objects in the database in several ways. The requirement calls for a host independent User Interface so that whatever host the system is implemented on, a user will always have common useful set of facilities.

The logical structure as described in STONEMAN is shown in Fig.2. The part outside the host facilities is termed the kernal APSE (KAPSE) and provides the host independent user interface. It also provides a host independent interface for the tools in the environment tool set. A part of the KAPSE is involved in providing the database facilities. Thus both tools and users can access the database via the KAPSE interface. The environment also includes the concept of a minimal APSE (MAPSE). This is defined as the minimum useful APSE. The MAPSE contains a basic tool set as shown in Fig.2.

The smallest separate identifiable collection of information on the database is termed an object. It must have three mandatory types of attributes which are category, access rights and history. The object category indicates the type of data in the object e.g. code, binary, text. The access rights attribute indicates how an object was derived e.g. what files were edited and what commands were used in the edit or what version of compiler was used. The history mechanism appears to be based on similar ideas to those in the Source Code Control System of programmer's work bench (ROCHIND, M. J. 1975). It must be possible to add further attribute types for any object type and to record relationships between objects in the database. The relationship facility will for instance, allow an object containing Ada code to be related to its corresponding binary and its design documentation.

When developing a design or piece of code it may go through several versions before being completed and accepted. The result aimed at is always the same and is always refered to by the same name. The STONEMAN document covers this by allowing an abstract object to be given the name of the product being developed and then requires that one can indicate different versions of the abstract object.

4. CONFIGURATION MANAGEMENT AND THE ASPE

4.1. Implementation of Configuration Management in the APSE.

To see how Configuration Management can be aided by the APSE an' how the Configuration Management aids can be implemented on the APSE, a particular implementation strategy will be described. This strategy is only sketched out here to show some of the possibilities. In reality there will be more detail to be worked out before a full Configuration Management system could be implemented on the APSE to suit a particular company and project. The MAPSE does require a Configuration Management tool, but the minimum requirements for this, given in the STONEMAN document, is that it must allow interrogation of the history attribute of any object in the database and it must provide managerial control over the persistence of objects in the database. This is only a very small and minor aspect of Configuration Management.

The requirements on the APSE to be able to implement Configuration Management are:-

a) To be able to store baselines, SCI's and changes on the database.

b) To be able to relate SCI's in a tree to the baselines and to relate changes to the SCI's and baselines they affect.

c) To be able to relate the baselines and SCI's to the code, binaries etc., stored on the database.

d) To be able to store references to the documents which form part of a baseline and relate these to their relevant baselines and SCI's.

e) To be able to control access rights to an object and change these as the status of the object changes e.g. when a module of software is registered the database object containing the code and binaries of the module have their access rights changed so that they cannot be deleted.

f) To be able to write tools that allow the Configuration Management data on the database to be updated in a natural way. The tools must also be able to create and modify the relationship between data objects to reflect the evolution of the system and its baselines, and also to reflect the change control mechanism.

g) To be able to write tools that produce reports about the data to support the Configuration Management process. This will require such reports as lists of the SCI's in a baseline, a list of items completed in a baseline, a list of changes related to a baselines and which changes are awaiting approval. There are obviously many more possibilities.

The requirements a,b,c and d in principle are satisfied by the STONEMAN requirement for a database object. The requirement e) should be satisfied by the access control attribute, but there is not enough detail to confirm that it is adequate. An example of a possible scheme of database objects is shown in Fig.3. This diagram depicts the records types and the relationships between them. This scheme proposes only 3 object types; one to represent baselines, one to represent SCI's and one to represent changes. The SCI object will be an abstract object because it will come in a number of versions. The baseline must be related to the SCI's assigned to that baseline. The SCI's must be able to be incorporated in a tree. Also the SCI must be related to the particular aspect of the SCI which make up the baseline e.g. documents, code, binary and test results. These could be the actual objects in the programmer's area. Thus there is a direct link between the Configuration Management area and a programmer's area. In other words there is potentially direct control of the software being developed.

Changes must be related to the baselines they affect and to the SCI's they affect. There are two relations between change and SCI; one representing the SCI version to which the change is applied and the other representing the SCI version resulting from the change. The change objects will need an attribute to indicate whether the changes have been approved or not. It is possible to extend this model to include objects representing Design Reviews and Configuration Control Boards. The design review can then be linked to the changes they have raised and the Configuration Control Board to the changes that were or are to be considered at the Configuration Control Board.

It can be seen that this simple scheme will support the requirements in f and g. It is possible to hold baselines and their related SCI's. It is possible to hold changes and relate these to the SCI's and baselines.

The following tools could be supported on this scheme and provide assistance to project Configuration Management.

a) A tool to input and store data on baselines and SCI's

b) A registration tool to accept SCI's for registration. It will check items presented against the requirements of the baselines and set the access rights on the relevant files. It will also update the baseline to indicate that the SCI has been registered.

c) A change control tool which would input change information,create a change object and link it to the relevant SCI's and baselines. It could accept data allocating a change to a design review and Configuration Control Board and connect the change object to the relevant design review and Configuration Control Board objects. It would also be possible to reflect the approval status of the change by altering an attribute of the change object. It will distribute the contents of the change documents to the attendees of Configuration Control Board and Design Review and to the relevant programmers using the mailing system. (See e below.)

d) An identification control tool which will keep a register of project identifiers and issue identifiers on request.

e) A mailing tool that maintains a set of distribution lists and issues nominated documents to the people on a specified distribution list upon request. This mailing system might simply print the requisite number of copies adding one of the addresses on the distribution list to each copy or alternatively it might flag the availability of the document to the addressee's user.

f) A baseline listing tool to produce listings of baselines and SCI's. These listings will include the SCI's, completion status and related changes with their approval status.

g) A formal issuing tool which will issue working versions of library items. It will check on the user's access rights to the item before issuing it. It could include a checking mechanism if the output is to some hard copy device, such as paper tape, to ensure the correct output of the item.

4.1. Effect of APSE on Configuration Management Activities

Having discussed a possible implementation of tools to support Configuration Management on the APSE it is now possible to see how these impact on the Configuration Management activities. First consider the way each of four component activities of Configuration Management, identified earlier, are affected.

In Configuration Identification a lot of the clerical administration task will be relieved by the identification control tool. When choosing an identification system at the start of the project any limitations imposed by the environments objects naming and version naming must be taken into account. With a carefully designed environment there should be little or no restriction on choosing naming conventions.

In Configuration Control the formal mechanism will still be needed, but the way it is implemented may well change. It will be possible to use the computer to capture directly data on change proposals from the proposer. These could be sent to the appropriate people via the mailing tool. The change control tool can then be used to monitor the progress of each change through the formal mechanism. The tool could also ensure that no change is forgotten and can be used to identify quickly if there is a hold up in a change and where it is being delayed. Upon approval of any change the system can also ensure that the programmer is aware of any change and that it is being applied to the right version of the software.

In Status Accounting most of the data could be on the computer. Here enquiries can be made directly rather than by going through a project librarian. In a development environment it is possible that the job of project librarian could disappear.

In Configuration Auditing the support from the environment will be to control the update of baselines. The assessment of SCI's to determine if they still meet the requirements must still be done by humans.

The formal interface mechanisms of Registration, Design Reviews and Configuration Control Baords are supported by environment tools. The registration data for the registration tool could be captured directly by computer with the programmer in dialogue with the computer; the system correlating his replies with its record of baseline and changes and finally setting the access rights on the registered objects to stop them being changed. It is conceivable that the whole process could be completely automated, but it is felt desirable that there ought to be at least one other human check before accepting objects for registration to verify that the programmer is indeed delivering the right product.

The Design Review and Configuration Control Board meetings can be supported by the distribution and recording of the inputs and results of these meetings. Again using the change control and mailing tool this is fairly easily achieved.

The affect on the programmer is to reduce his clerical tasks. When he starts a design, the text of the requirement document could well be on the database. In this case he may simply list the document for himself and any changes that must be incorporated to achive his baseline. Thus he will be reasonably sure that he has all the current changes. Once has has produced the design document this again can be held on the database. Upon acceptance, the access rights may be set to stop the text being changed. Changes required will be compiled in terms of change requests. The progress of changes can be monitored through the change mechanism by the programmer. A similar process is carried out when the code has been produced and tested. If the programmer is modifying or intergrating code then he will be allowed to copy the relevant version of the code for himself to work on by setting appropriate access rights.

The quality engineer not only has the Configuration Management data on the computer, but he can directly control the programmer's access to controlled data objects. With the APSE, the quality engineer can administer Design Reviews and Confiugration Control Boards more easily. The Design Review actions can be embodied in change objects and related to the baselines and SCI's that they affect. When it comes to Configuration Control Boards the relevant list of changes to be considered by that particular meeting can be easily listed. The list of adjacent SCI's can also be made to aid determining the implications of the change. The distribution of meeting inputs and the propogation of changes to the system through the project can be greatly assisted by a computer mailing system.

As noted earlier the registration of documents is a lot simpler. Also many of the tasks done by the project librarian could disappear.

The project manager's role is not affected except that he gets his data more rapidly, it is more upto-data and of better quality than a manual system. He may have better control over the development, since he has a greater visibility of what is being developed. In planning the project the project manager will have to take into account the different procedures that will be needed if he is going to use an APSE.

The use of an environment may change the things an external auditor will look at. The tools used to manipulate the Configuration Management data will embody the quality procedures. Thus the auditor may want to look at the way these tools work and even at the code. At another extreme the customer may insist that particular Configuration Management tools are used on his project. Financial files on computers are audited using computers. The customer may want to impose a similar audit on the Configuration Management data on the APSE. This may especially be the case if the customer has insisted that a particular Configuration Management suite of programs is used.

5. CONCLUSIONS

It is obvious that the Configuration Management tools will be developed and probably be available soon after the early APSE's are delivered. A project that uses an APSE does not have to use Configuration Management tools. But it is an obvious advantage to adopt Configuration Management tools from the project manager's and quality engineers points of view.

In this paper we have considered only one aspect of project work in relation to the APSE. Other aspects will also be affected by APSE. What can be said with confidence is that the use of APSE will change the

the way a project is run. This must be planned for.

Although many of the advantages of the system and the facilities supplied are focussed on the programmer, it is the author's feeling that the main justification of the APSE is in the management aspects of the project. The greater visibility and control ensured with an APSE are a great advantage in any project and this is the area where a lot of the cost benefits will arise.

6. REFERENCES

BUXTON, J. N. 1980 "Requirements for Ada Programming Support Environments - STONEMAN" Department of Defence.

BERSOFF, E. H.: HENDERSON, V.D.: SIEGAL S.G., 1979. "Software Configuration Management - A Tutorial" COMPUTER January 1979.

IVIE, E.L. 1977 "The Programmer's Workbench - A machine for Software Development" Comm of A.C.M. Vol.20 No.10 October 1977.

KERNINGHAM, B.W.; MASHEY, J.R.: 1981, "The Unix Programming Environment" Computer April 1981.

ROCHKIND, M.J. 1975 "The Source Code Control System" IEE Trans. Software Eng. Vol SE-1 No.4. Dec 1975.

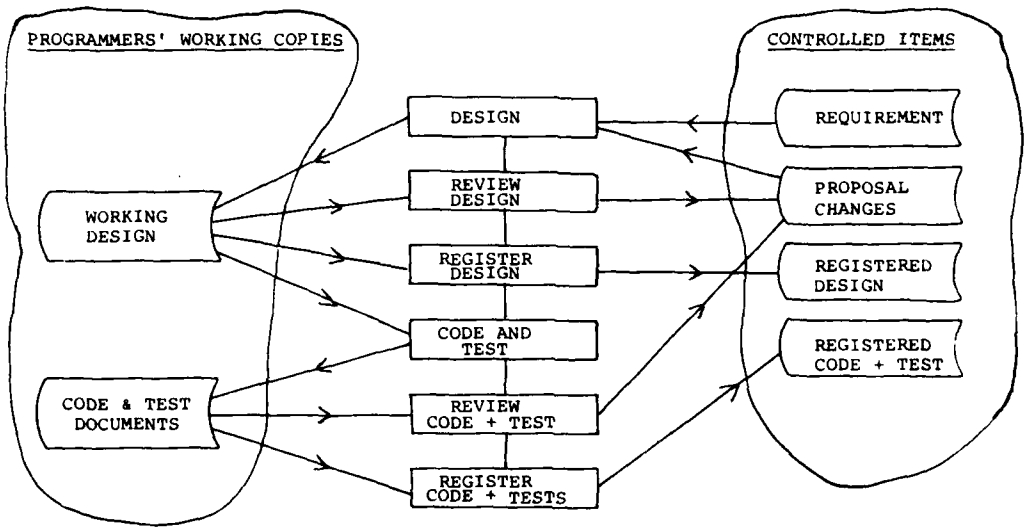


FIGURE 1 SEQUENCE OF PROGRAMMER ACTIONS

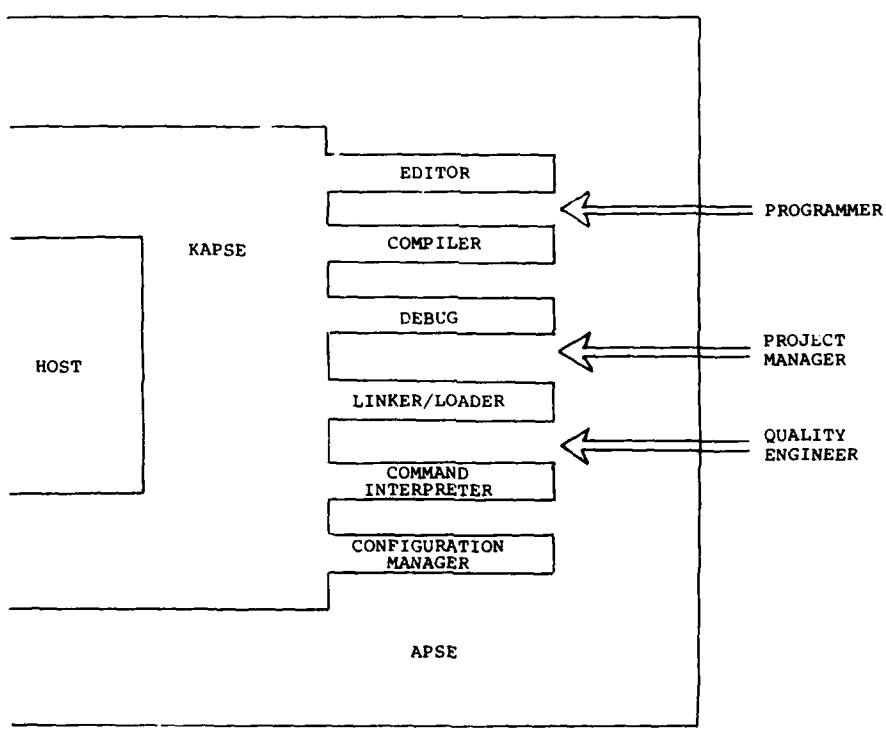


FIGURE 2 STRUCTURE OF APSE

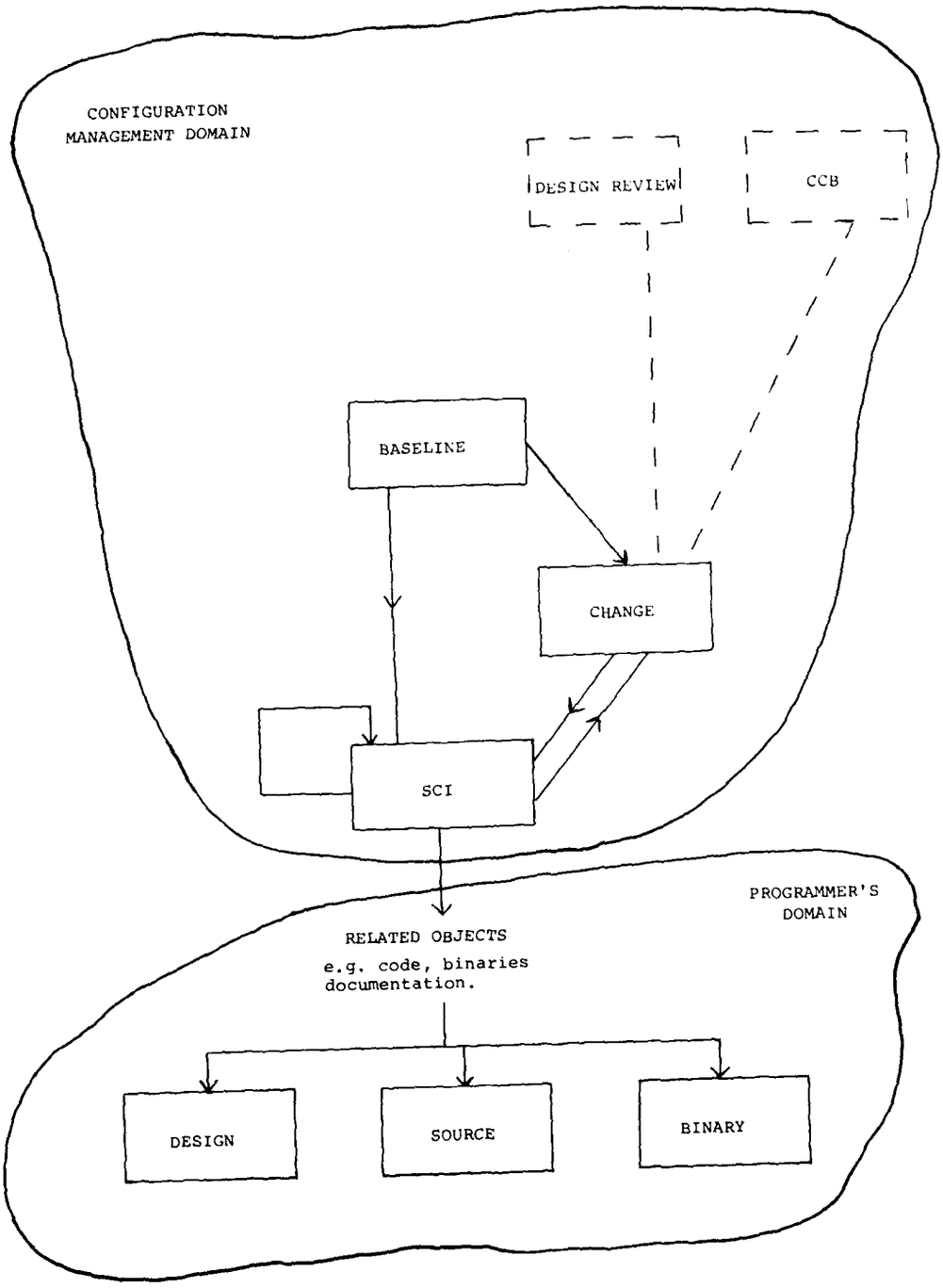


FIGURE 3 POSSIBLE DATABASE CM SCHEME

SOFTWARE FAULT TOLERANCE

FOR REAL-TIME AVIONICS SYSTEMS

T. Anderson*
University of Newcastle upon Tyne
Newcastle upon Tyne, England.

J. C. Knight**
NASA Langley Research Center
Hampton, Virginia, USA.

SUMMARY

Avionics systems have very high reliability requirements and are therefore prime candidates for the inclusion of fault tolerance techniques. In order to provide tolerance to software faults, some form of state restoration is usually advocated as a means of recovery. State restoration can be very expensive for systems which utilize concurrent processes. The concurrency present in most avionics systems and the further difficulties introduced by timing constraints imply that providing tolerance for software faults may be inordinately expensive or complex. This paper asserts that this is not the case, and proposes a straightforward pragmatic approach to software fault tolerance which is believed to be applicable to many real-time avionics systems. A classification system for software errors is presented together with approaches to recovery and continued service for each error type.

1 Introduction

Digital avionics systems typically operate in real time. This means that inputs may be expected and/or outputs must be generated according to some real-time schedule. For example, an avionics system may send commands to control surfaces every tenth of a second of real time. The requirement for operation in real time presents difficulties over and above those normally encountered in programming. For example, a real-time program may successfully produce the output demanded by its specification but fail to do so within the imposed real-time deadline.

Avionics systems must be extremely reliable. There are two approaches to the construction of software which must exhibit behavior that is highly reliable (that is, complying with its specifications most of the time). Avizienis (Avizienis, 1976) called these fault intolerance and fault tolerance. Fault intolerance, embraces all the various techniques which try to ensure that software contains no faults. Fault tolerant software incorporates techniques which attempt to ensure that service is maintained by coping with the faults which remain after the application of all possible fault avoidance measures.

This report discusses the application of fault tolerance techniques to real-time avionics software. A practical approach to software fault tolerance is presented which can be applied relatively easily and can use existing hardware. Using this approach, systems can be constructed which will continue to provide adequate responses in real time under circumstances where faults in the software would normally cause a loss of service.

Previous work in the area of fault tolerant real-time systems has been reported by others (Campbell et al, 1979; Hecht, 1976; Kopetz, 1974). It is likely that certain military real-time systems have made some use of software fault tolerance but in most cases, such systems have not been described in the open literature. One exception is SAFEGUARD (Gawron, 1975).

2 Principles of Fault Tolerance

Detailed discussions of the general principles of software fault tolerance may be found elsewhere (Anderson et al, 1979; Randell et al, 1978). Only a brief overview is given here.

The following terminology will be adopted:

1. a FAILURE occurs whenever the external behavior of a system does not conform to that prescribed by the system specification,

2. an ERROR (more accurately known as an erroneous state) is a state of the system which, in the absence of any corrective action by the system, could lead to a failure which would not be attributed to any event subsequent to the error,

3. a FAULT is the adjudged cause of an error.

---

* Research performed unnder NASA contract numbers NAS1-14101 and NAS1-1 4472 while in residence at ICASE, NASA Langley Research Center, Hampton, Va.
** Present address: Deptartment of Applied Mathematics and Computer Science, Science, University of Virginia, Charlottesville, Va.

The term fault will thus be used to refer to any defect in a system which could generate an erroneous state (for example, a defective hardware component or a "bug" in a program).

Fault tolerance techniques can usually be divided into four constituent phases. They are:

1. ERROR DETECTION. In order to tolerate a fault its effects must first be detected. Clearly, this can only be achieved by performing checks to determine whether any erroneous situation has arisen.

2. DAMAGE ASSESSMENT. Having detected that the system is in error, it will usually be necessary to identify how much of the state of the system has been corrupted.

3. ERROR RECOVERY. Probably the most important aspect of fault tolerance is the provision of an effective means of transforming an erroneous state of the system into a well defined and error free state. Methods for achieving this transformation can sometimes make good use of the information retained in the erroneous state, but it can be more secure to simply discard the erroneous state and reset the system to some prior state (a recovery point).

4. CONTINUED SERVICE. In order to enable the system to continue to provide the service required by its specification, further action may be needed to ensure that the fault whose effects have been obviated does not immediately recur and thus ruin the whole approach. Unless the fault was transient and will not recur in any case, it must either be rectified or circumvented.

The occurrence of errors in software is unpredictable. They do not arise through component degradation (as in hardware) but have the characteristics of design faults. For this reason, the techniques adopted for the four component strategies described above must operate in as general a way as possible. Thus, it is advocated that error detection should be achieved by checking that the system is functioning acceptably. It is not suggested that the more conventional approach of checking for specific malfunctions should be discarded, but that negative checks of this type should be supplemented by positive acceptability checks.

In an unanticipated error situation, an automated exploratory approach to damage assessment would be difficult. It is appropriate to base decisions about the extent of damage on assumptions of how the system is structured and the apparent severity of the error.

A similar approach to error recovery entails mistrusting any of the state information considered to be damaged and avoiding the use of recovery techniques which rely on such information. In order to recover from the unpredictable situations which can ensue from design faults, it is necessary to adopt the more drastic alternative of replacing all suspected parts of the system state together with any other parts which must be replaced for consistency. This may involve substantial processing and consequent delay.

Finally, in order to achieve continued service after recovery has taken place, some means of preventing a repetition of the original fault must be found. It will be necessary to obtain some estimate of the location of a software fault so that the module containing the fault can be replaced by a stand-by spare. Given the nature of software faults, it is clear that the spare module must be of independent design.

3 Fault Tolerance in Concurrent Systems

While considerable success has been achieved in devising mechanisms to provide fault tolerance in the software of sequential systems, difficulties arise when systems of communicating concurrent processes are considered, particularly if real-time constraints are imposed. Suggestions in this more difficult area have involved major assumptions about the nature of the concurrency in the system (Campbell et al, 1979; Kim, 1978; Randell, 1975; Russell, 1975; Shrivastava, 1979; Shrivastava et al, 1978).

The basic problem is that if processes can communicate at will, then whenever one process establishes a recovery point (for state restoration purposes) it is advisable for all other processes to do the same. Thus the processes have to be synchronized. If this is not done, system-wide consistent state restoration may only be possible by rolling back the activity of the system to an arbitrarily earlier point in time. This is the Domino Effect (Randell, 1975). All of the above approaches are aimed at avoiding the heavy overhead incurred with large numbers of recovery points (and associated synchronization) or extensive rollback.

The process structure of real-time systems contains many synchronization points which are usually associated with timing constraints. Synchronization points occur within the process structure where a subset of the processes are synchronized, and at frame boundaries where all of the processes are synchronized. In fact, much of the synchronization of processes in a real-time system stems from the need to synchronize with the external environment, rather than from any inherent needs of the processes themselves. Thus much more synchronization occurs than would be found in concurrent

systems that do not operate in real time. This means that although real-time systems are concurrent, they have a characteristic which is highly desirable if recovery points are to be provided without excessive overhead - the provision of fault tolerance need not involve any changes to the process structure. Such systems are particularly amenable to the application of a modified form of the CONVERSATION technique (Randell, 1975).

A set of processes which participate in a conversation may communicate freely among themselves, but with no other processes. Processes may enter the conversation at different times but, on entry, each must establish a recovery point. All processes must leave the conversation at the same time since if an error is detected in any participant, every process in the conversation must restore its recovery point and try again. If the conversation structure is used to provide recoverability in a general concurrent system, the necessary state restoration can be automated using a recovery cache (Horning et al, 1974), which is a form of mechanised incremental checkpoint. Although this is conceptually straightforward, if a recovery cache is not supported in the underlying machine then extensive processing will be necessary to simulate its operation. Presently available computers do not provide a hardware recovery cache although an experimental PDP 11 with a recovery cache is being built (Lee et al, 1979). Except in particularly simple cases, the overhead of a software recovery cache is prohibitive.

A particularly simple form of conversation occurs when processes enter the conversation upon creation and leave upon termination. Practical real-time systems frequently have characteristics which allow much simpler recovery provided this restricted form of conversation is used for process communication. Specifically, full state restoration need not be attempted since, in practice, a great deal is usually known about the system state when processes synchronize. The repetitive nature of a real-time system dictates that its state at a given synchronization point will be very similar on each frame. No data, or very little, is generated which is used or modified from frame to frame. Recovery mechanisms can therefore be based on re-establishing the processes as they normally appear when initiated in a frame and then ensuring that any frame specific data has its correct values. In view of the limited amount of data which is frame specific, the recovery required involves little more than a reset. As such, hardware assistance is not essential. Assuming all code and constants to be in a read only memory, the reset procedure can be simply and adequately handled in software.

4 Error Classification

For the purpose of discussing the recovery mechanisms which have to be applied, errors will be classified according to a set of definitions. This classification is basically with respect to the apparent seriousness of the situation arising from a fault. The definitions are:

1. INTERNAL error - an error that can be adequately handled by the process responsible for the system being in error.

2. EXTERNAL error - an error that cannot be adequately handled by the process responsible for the system being in error, but whose effects are limited to that process.

3. PERVASIVE error - an error that cannot be adequately handled by the process responsible for the system being in error, and such that other processes generate errors not directly attributable to their own faults.

The incidence of errors will be classified according to the following definitions:

1. PERSISTENT - an error is persistent if the frequency of occurrence of the associated fault exceeds some predetermined threshold.

2. TRANSIENT - an error is transient if it is not persistent.

Given this classification scheme for errors, it is necessary to be able to determine which class an error falls into once it has been detected. This enables appropriate recovery techniques to be employed reflecting the extent of the damage incurred by the system. In practice, a classification can only be attempted and, in general, it will be impossible to classify all errors correctly. For example, an error could occur which was in fact pervasive, but if the consequent damage to the other processes was not detected, then this pervasive error would be indistinguishable from an external error. There is nothing that can be done about this problem. Some form of recovery will be invoked even when an error is wrongly classified and this may still be sufficient to ensure continued service from the system.

5 Error Detection and Damage Assessment

Errors detected by hardware are usually signalled by the generation of an interrupt, but the signaling of software detected errors can take many forms; for example a flag could be set, a branch instruction executed leading to an error handler, or an interrupt deliberately generated. It is assumed in the following discussion that whenever an error

is detected control passes automatically to a system error handler.

On being invoked the error handler must make a determination of the extent of the damage to the system state, and then initiate appropriate error recovery measures. In the approach proposed in this paper, damage to the system is implicitly assessed by the error handler classifying each error as being internal, external, or pervasive. For external and pervasive errors, the recovery technique applied is also dependent on whether the error is deemed to be transient or persistent.

In order to classify errors with reasonable accuracy, it will be necessary for the error handler to retain information concerning the error history of processes in the system. No information need be maintained for internal errors since such errors are considered to be completely localized difficulties for which the recovery applied by the process involved is adequate.

Whether an error in a process can be considered internal or not will be very system dependent. The error handler makes this determination based on whether this particular error is one for which processes are permitted to attempt local recovery and whether the process in which the error occurred has the means of attempting local recovery. If local recovery is available, permitted, and apparently successful the error is classified as internal. Otherwise, the error will be dealt with as an external or pervasive error.

An error can be suspected to be pervasive if multiple non-internal errors occur in a single frame. A persistent external error is suggested if an external error recurs frequently in a particular process. Frequent recurrence of pervasive errors indicates a persistent pervasive error. Quantification of "multiple" and "frequent" in the above yields a well-defined classification algorithm for use by the error handler.

It is suggested here that if an external error has occurred in a frame, then any further occurrence of a non-internal error in that frame should be classified as a pervasive error. It is preferrable for the error handler to err, if at all, on the side of caution.

Consideration of existing systems suggests that a less rigid approach can be adopted toward determining the persistence of an external error. A straightforward frequency test seems appropriate; for example, an external error in process P could be considered persistent if an external error in P had occurred either in each of the n previous frames, or in p of the q previous frames (where n, p, and q are integers selected by the system designer). A stricter version of the same test might be considered necessary to detect recurring pervasive errors.

6 Recovery and Continued Service

6.1 Internal Errors

Recovery from internal errors is only attempted for those errors for which explicit provision has been made in the system design. Techniques for internal recovery by a process include 'ad hoc' repair as a part of a local exception handler (Goodenough, 1975) such as a PL/I 'ON' unit, or a more general approach such as the systematic state restoration employed by recovery blocks. It is inappropriate to discuss the response to internal errors because in any given set of circumstances, recovery is highly dependent on the structure of the individual processes involved.

6.2 External Errors

Recovery from external errors is assumed to be provided by state restoration using a simple reset mechanism for the process involved. As discussed in section 3, this can be easily achieved in software.

Three general approaches to recovery and continued service are possible following the detection of an external error. They are:

1. No special processing. The error is ignored and the system continues trying to provide service.
2. Provision of behaviour that is acceptable in the short term but is inferior to that intended from the process in which the error is deemed to have occurred.
3. Provision of behaviour equivalent to the intended behaviour of the process in which the error is deemed to have occurred.

Approach 1 could be considered for processes which are not critical but for no others. It is not recommended even under these circumstances since there is always the danger

that an untreated error could have unanticipated side effects.

Approach 2 is essentially the use of recovery blocks as proposed by Hecht (Hecht, 1976). Although it was suggested in the context of timing errors, this approach is equally applicable to other external errors. Essentially, the occurrence of a fault in a primary process is handled by the execution of an alternate providing degraded service. It is interesting to note that several simple alternates are possible. In particular, in real-time systems with short frame times it is often acceptable to re-use the outputs of the previous frame as the outputs for the frame in which the error occurred. This is known as the "skip-frame" strategy. Another possibility is some form of extrapolation based on data from several previous frames.

Approach 3 is similar to approach 2 but assumes that non-degraded outputs must be generated on every frame regardless of the occurrence of faults. In practice this approach will be required only rarely in the treatment of transient external errors. Most real-time systems seem able to operate acceptably despite momentary degradation of service and, if an external error is truly transient, approach 2 will often be appropriate. If an external error is persistent, repeated use of approach 2 will almost certainly lead to a state which constitutes a system failure at some point. For example, repeated use of the skip-frame strategy amounts to the system repeatedly ignoring changes in the external environment. The primary intent of most real-time systems is prompt response to changes in the external environment.

Hecht (Hecht, 1976) has proposed the design of a real-time executive which will remove a defective process from the system and replace it by a new version. Using the model and error classification scheme proposed here, this amounts to responding to a persistent external error by replacing the relevant process with a substitute. This substitute is completely equivalent in its interfaces to the rest of the system but is constructed differently so that, hopefully, it will not become erroneous under the circumstances which caused the original process to become erroneous.

Thus, provision of continued service depends on whether the error is transient or persistent. Both types can occur and so provision must be made for both. This suggests that every process should be supplemented by at least one degraded service alternate to cope with transient external errors and another version of the primary process to cope with persistent external errors.

6.3 Pervasive Errors

Pervasive errors are the most serious of the error classes. The fact that the error is pervasive means that, in the absence of fault tolerance, total system failure is very likely even if errors have only been detected in supposedly noncritical processes. So much damage has probably been done that critical processes will almost certainly enter erroneous states.

Strategies are limited by the gravity of the situation. The error will be classified initially as transient and the only practical approach to continued service is to use the simple skip-frame strategy discussed above. The system complexity required if an attempt is to be made to execute more extensive alternates for many processes is almost certainly unacceptable. If the error is indeed transient then this strategy is probably adequate anyway.

If the error turns out to be persistent and pervasive then it is extremely unlikely that the system will be able to provide any acceptable service. To all intents and purposes, it has failed. Treatment of the error during its initial transient classification will have attempted to ensure that acceptable service was maintained but such treatment cannot continue. The only viable automatic treatment for persistent pervasive errors is complete replacement of the software. If provisions for recovery and continued service have been made for external errors, there will be a second version of each process available and the replacement of each process by the second version amounts to total software replacement. Once again, recovery can be handled by a simple reset.

7 Conclusion

A general approach to fault tolerance in real-time systems has been presented and it has been suggested that these systems often have characteristics which make them particularly amenable to the inclusion of fault tolerance.

There is a cost associated with the provision of fault tolerance and it may be substantial. If two versions of a primary process are to be provided they must both receive equal care and attention in their preparation. This could more than double the total cost of the software. It must be remembered that in such critical systems as commercial air transports, the software cost is not a substantial portion of the total development cost. Copies of the software for additional aircraft cost nothing and so, for an entire fleet, the cost of producing high-quality, fault-tolerant software may be insignificant compared to the total cost of producing the aircraft. Irrespective of the

cost, in many cases the need for the utmost reliability dictates the need for fault tolerant systems.

References

1. Anderson, T., Lee, P. A., and Shrivastava, S. K.: "System Fault Tolerance," 1979, In: Anderson, T. and Randell, B. (eds.) *Computing Systems Reliability: An advanced course.* Cambridge University Press, pp. 153-210.

2. Avizienis, A.: "Fault Tolerant Systems," 1976, IEEE Transactions on Computers, Vol. C-25, No. 12, pp. 1304-1312.

3. Campbell, R. H., Horton, K., and Belford, G. G.: "Simulations of a Fault Tolerant Deadline Mechanism," 1979, Digest of Papers, FTCS-9, Madison, pp. 95-101.

4. Gawron, L. J.: "System Error Control," Bell System Technical Journal , Vol. 54, Special Supplement on Safeguard, 1975.

5. Goodenough, J. B., "Exception Handling: Issues and a Proposed Notation," 1975, Comm. ACM, Vol. 18, No. 12, pp. 683-696.

6. Hecht, H.: "Fault Tolerant Software for Real-Time Applications," 1976, Computing Surveys, Vol. 8, No. 4, pp. 391-407.

7. Horning, J. J., et al.: "A Program Structure for Error Detection and Recovery," 1974, In: Gelenbe, E. and Kaiser, C. (eds.) Operating Systems: Proc. Int. Symp. held at Rocquencourt. Lecture Notes in Computer Science 16, pp. 171-187, Springer-Verlag, Berlin.

8. Kim, K. H., "An Approach to Programmer-Transparent Coordination of Recovering Parallel Processes and Its Efficient Implementation Rules," 1978, Proc. Int. Conf. on Parallel Processing, Detroit, pp. 58-68.

9. Kopetz, H., "Software Redundancy in Real-Time Systems," 1974, IFIP Congress 74, Stockholm, North Holland, Amsterdam, pp. 182-186.

10. Lee, P. A., Ghani, N., and Heron, K.: *"A Recovery Cache for the PDP-11,"* 1979, Digest of Papers, FTCS-9, Madison, pp. 3-8.

11. Randell, B.: "System Structure for Software Fault Tolerance," 1975, IEEE Trans. on Software Engineering, Vol. SE-1, No. 2, pp. 220-232.

12. Randell, B., Lee, P. A., and Treleaven, P. C.: "Reliability Issues in Computing System Design," 1978, Computing Surveys, Vol. 10, No. 2, pp. 123-165.

13. Russell, D. L.: "Process Backup in Consumer-Producer Systems," 1975, Proc. Sixth Symp. on Operating System Principles, pp. 151-157.

14. Shrivastava, S. K.: "Concurrent Pascal with Backward Error Recovery," 1979, Software: Practice and Experience, Vol. 9, pp. 1001-1020 and 1021-1033.

15. Shrivastava, S. K. and Banatre, J.-P.: "Reliable Resource Allocation Between Unreliable Processes," 1978, IEEE Trans. on Software Engineering, Vol. SE-4, No. 3, pp. 230-241.

ELECTRONIC WARFARE SOFTWARE

Rudy L. Shaw
Air Force Wright Aeronautical Laboratories
Wright-Patterson AFB Ohio

SUMMARY

The development and maintenance of software for computer based Electronic Warfare (EW) systems has become a major problem and represents a significant percentage of the overall life cycle costs. This paper partially summarizes a three year effort which looked at various software cost reduction techniques within the framework of EW processing requirements. EW processing characterization was the initial task undertaken in this study and it will be summarized in this paper. The intent of the characterization study was to provide a data base for the following investigations, which also will be summarized in this paper: (1) The investigation of the compatibility of instruction sets, which are used or have a near term possibility of being used in EW systems, to the EW processing tasks; (2) The investigation of High Order Languages and software structuring efficiencies based upon bench marks which characterized major EW functions.

1. INTRODUCTION

Electronic Warfare (EW) systems represent the most demanding processing environment to be encountered anywhere. These systems must operate in near real time on an extremely high data rate environment with a high degree of reliability. In addition, these systems must be configured to be easily reprogrammed and maintainable. The development and maintenance of software for these systems has become a major problem and represents a significant percentage of the overall life cycle system costs. In 1978, the Avionics Laboratory Electronic Warfare Division of the Air Force Wright Aeronautical Laboratories (AFWAL) at Wright-Patterson Air Force Base embarked upon a program with Systems Consultants, Incorporated to investigate EW software. This investigation looked at various software cost reduction techniques within the framework of EW processing requirements. The data presented in this paper is primarily extracted from the AFWAL technical report (Ziesig, D., June 1981) resulting from this study.

The characterization of EW processing was the initial task undertaken in this study. As a data base for this characterization, six EW systems were selected based upon various stages of development and available data. Three systems were operational and one system each was in engineering development, advanced development and exploratory development. As such the characterization was based upon a wide range of EW processing techniques. The characterization study had three primary objectives, two of which will be summarized in this paper.

The first objective was to characterize EW software to the extent necessary to generate an ideal EW instruction set (CALEW) which incorporated the needs of the EW programmer. The basic cost reduction assumption here is that by the proper choice of the instruction set architecture, significant economies in memory size and program execution, higher visibility of the data structures and algorithms, clearer more concise programs will result. The intent of generating the CALEW was not to recommend a new instruction set, but rather to have a basis on which presently available or proposed instruction sets could be compared. This characterization and instruction set comparison will be presented in this paper.

The second objective was to generate benchmarks which were representative of EW processing and use these benchmarks to evaluate the effects of using a High Order Language (HOL) and structured programming in EW systems. The benefits resulting from the application of a HOL to system development are many. One cost benefit estimate, resulting from programming in a HOL, is typically based upon the theory that the number of lines of source code per unit time is independent of the language used (Doty, D., February 1977). If this is indeed true, using a typical expansion rate of four machine code statements per HOL source statement, the resulting cost impact could be 4:1. However, there is an obvious penalty to be paid in the form of increased storage requirements and increased program execution time resulting from the HOL inefficiency. Another area which has potential to reduce system development and maintenance costs is the use of Modern Programming Practices (MPP) such as structured code, structured design, programming support libraries, and chief programming teams. The results of work in this area indicate that the total person hours required for developing and testing software can be substantially reduced when MPP are applied (Curtis, B., March 1980 and Milliman, P., February 1980). However, there are increased storage requirements and run time inefficiencies to be encountered, related to using HOL and structured code in light of EW requirements were pursued in this study and will be summarized in this paper.

2. EW PROCESSING CHARACTERIZATION AND INSTRUCTION SET COMPARISON

The characterization of EW software was approached from many different aspects. The first aspect was a thorough analysis of EW software data structures. This review covered 197 data structures and can be summarized as follows. The primary data structure is the table which is implemented as a linked list. There are numerous tables within a program which are linked together in order to resolve ambiguities. The reasoning behind the predominant use of tables is the ease of reprogramming the system. The statistical distribution of field widths among the data structures is shown in table 1. Note the large number of fields which are only one bit and the large number less than 8 bits. This is consistent with the real world processing of EW systems where items of interest such as pulse width, scan type, etc require small fields for characterization. It is of interest to also note that of the 41% of the fields which were full words, the majority of these fields were pointers to other tables and files. The fields within the tables were also tightly packed irregardless of the instruction set support for manipulating partial fields. It appears that an instruction set which supports bit manipulations is essential for EW processing.

FIELD LENGTH	DISTRIBUTION
1 BIT	26.6%
LESS THAN 8 BITS	45.6%
8 BITS	9
LESS THAN 16 BITS	58.6%
FULL WORD FIELDS	41.4

TABLE 1. DISTRIBUTION OF FIELD LENGTHS

INSTRUCTION CATEGORY	DISTRIBUTION
LOAD REGISTER	29.4%
STORE REGISTER	13.5
BRANCH	18.9
BRANCH TO SUB	5.1
REGISTER ADD/SUB	6.3%
I/O CONTROL	2.9%
REGISTER LOGICAL	3.8
COMPARES	9.8%
TOTAL	89.6%

TABLE 2. DISTRIBUTION OF INSTRUCTION TYPES

Another aspect of EW software characterization consisted of a static analysis of instruction use. This analysis consisted of simply counting the instruction usage in the programs. The results of the static analysis are shown in table 2. As can be seen from this table, approximately 43% of the instructions used were concerned with moving data. A flexible load/store instruction set is essential for EW systems. Also, note the large number of branches. Branching is primarily based upon logically testing a variable (i.e. a variable is less than, larger than, or between some set limits).

The variables used primarily range from -8 to +7 or 1 to 4 bits in length. These small numbers represent data items such as the number of hits, the available jammer units, the number of active threats, etc. The predominant use of small numbers again emphasizes the need for an instruction set which is efficient in manipulating small numbers. Note also in table 2., the small use of arithmetic instruction which consisted primarily of adds/subtracts. EW processing is primarily logic oriented and sophisticated arithmetic manipulation is not used to a great extent. However, this characterization may require modification as the need arises to process more exotic emitters.

Another aspect of the characterization process, consisted of searching the operational programs for groups of instructions which were indicative of the type of processing involved in EW systems, but were not implemented because of lack of instruction set support. This analysis coupled with the previous characterization approaches allowed the characteristic sequences shown in table 3 to be generated. This table is self explanatory and summarizes the primary attributes of EW software.

OBSERVED SEQUENCES	CHARACTERISTICS
BRANCH BASED UPON MEM LOC G OR NOT G BRANCH BASED UPON RESULT OF LOGICAL OPERATION OR COMPARISON	DECISION ORIENTED
SET A = B CLEAR FLAGS FEW ARITHMETIC FORMULAS HEAVY USE OF SMALL NUMBERS	NON-COMPUTATIONAL
INCREMENT/DECREMENT POINTERS SMALL LOCALIZED LOOPS	TABLE DRIVEN
SET/RESET/TEST A BIT MANIPULATE PARTIAL WORD FIELDS	BIT ORIENTED

TABLE 3. EW PROCESSING CHARACTERISTICS

After the EW processing characterization effort, an instruction set was generated which covered the features necessary for real time processing, in general, and focused specifically on EW. The final CALEW consisted of 110 instructions. After CALEW was generated, it was sent to companies which are specifically involved in EW systems and EW programmers were requested to rate from 1 to 10, the value of each instruction in the instruction set. This data was then averaged for each instruction and used as a multiplier to ultimately compare instruction sets. Ten instruction sets were selected for comparison. They were the instruction sets included in the characterization study and instruction sets which have near term potential of being used in EW systems. Each instruction in the CALEW was compared against each instruction in the comparison set. If the comparison set did not have the instruction capability, it was given a 0 for this instruction. If the comparison set possessed the instruction, that instruction was given the value of the multiplier. The results of this comparison are shown in table 4. Note that MIL STD 1750 ranks highest among those instruction sets tested. In particular, it surpassed the competitors by its high degree of data manipulation capabilities, compare features, and efficiency in handling small numbers.

PROCESSORS	% OF CALEW
1. MIL-STD 1750	81.6
2. AN/UYK-20	61.5
3. PDP 11/70	50.1
4. ATAC 16M	45.3
5. 4-PI	42.5
6. ROLM 1602A	42.3
7. TI 2520	40.4
8. LC 4516D	32.8
9. MAPS	32.4
10. ROLM 1601	25.7

TABLE 4. PROCESSOR RANKING

3. BENCHMARK SELECTION, HOL AND STRUCTURED PROGRAMMING ANALYSIS

During the EW processing characterization study benchmark algorithms were selected which characterized the major EW functions. The benchmarks selected are listed in table 5 by name and function. These benchmarks with the exception of DATAMAN and PURGE were selected as they existed in the six data base systems and as such, these benchmarks did not use structured programm techniques. The benchmarks DATAMAN and PURGE were selected to exercise the manipulation of tightly packed data, exercise the use of global data and linked list operations. These operations are typical of all EW systems and are not readily extractable from any one. The PRIDE 1 benchmark is a special case which will be discussed later. The selected benchmarks were extensively documented to assure a minimum of ambiguity and interpretation difficulties. Documentation typically consisted of 25-30 pages per 120 line benchmark. This is for more extensive than the 25-30 page per 1000 lines of code typically expected (ref 2). The documentation consisted of verbal descriptions, symbol dictionary, unstructured flow chart, data structure description, and structured algorithm description.

AFLOOK	- PARAMETER MATCHING - LINKED-LIST PROCESSING (SUBSIDIARY LISTS AND POINTERS) - PARTIAL WORD MANIPULATION - PRIORITIZATION
DATAMAN	- USE OF GLOBAL DATA - TIGHTLY PACKED, COMPLEX DATA STRUCTURE MANIPULATION
EID	- PARAMETER MATCHING - LINKED LIST PROCESSING (POINTERS) - PARTIAL WORD MANIPULATION
EXD	- SIGNAL DETECTION (INTERRUPT HANDLING) - PARAMETER MATCHING - USE OF FLAGS - LINKED LIST PROCESSING - ERROR DETECTION
EXEC	- SCHEDULING - PARAMETER MATCHING - FLAG DETECTION AND SETTING
PRIDE 1 and PRIDE 2	- PARAMETER MATCHING - TABLE UPDATES - BINARY SEARCH
PURGE	- LINKED LIST ADDITIONS AND DELETIONS

TABLE 5. BENCHMARKS

The HOL's and respective assembly languages (AL) selected for testing were JOVIAL (J73), AN/AYK15A (MIL STD 1750); CMS 2, ULTRA; and FORTRAN, MACRO-11. Benchmarks were also coded in ADA in order to assess any language difficulties, but because of comiler unavailability, no quantitative data was collected. The benchmarks selected were distributed to programmers proficient in the languages under consideration. All HOL benchmarks were coded using structured programming techniques and all AL programs,with the exception of PURGE, DATAMAN, and PRIDE 1, were coded from the unstructured flow charts. The PRIDE 1 benchmark was reconstructed from the PRIDE 2 implementation by analyzing the PRIDE 2 function and redesigning the original PRIDE 2 algorithm. The AL version of PRIDE 1 was coded using structured programming techniques only.

The number of lines of source code required for each benchmark is shown in table 6. In general, the amount of HOL source code required was 57% of the amount of AL source code required. Specifically, the JOVIAL implementation required 23% less code than the AN/AYK15A and the FORTRAN implementation required 51% less

source code than MACRO. The CMS2 sample is small and no conclusions are inferred. Note than the benchmarks implemented in JOVIAL required 35% more lines of code than those implemented in FORTRAN. This is attributable to the detailed data definitions required in JOVIAL compared with the numerous default conditions in FORTRAN. It is also worthwhile to note than the benchmarks as implemented in AN/AYK15A required 23% less lines of code than those implemented in MACRO. This supports the conclusion arrived at earlier that MIL STD 1750 highly supports EW processing. During the benchmark programming the programmers were required to keep track of their programming time. In general, it took twice as long to implement AL programs as it did the HOL benchmarks. Since, in general, the amount of AL source code was approximately twice the amount of HOL source code, the amount of time required to code is consistent with the theory that the programming time per line of code is independent of the language used.

BENCHMARK	J73	15A	FOR	MAC	CMS	ULT
AFLOOK	132	128	99	267	124	187
EID	53	80	35	158	67	94
EXD	167	213	109	182		
EXEC	110	121	62	236		
PRIDE 2	163	159	102	178		
PURGE	48	36	42	91	49	53
DATAMAN	76	229	41	194	79	141
PRIDE 1	190	198	128	267		

TABLE 6. LINES OF SOURCE CODE

BENCHMARK	J73	15A	FOR	MAC	CMS	ULT
AFLOOK	956	530	1313	764	1508	832
EID	386	236	663	270	432	408
EXD	1146	718	1282	758		
EXEC	502	368	740	424		
PRIDE 2	642	462	832	534		
PURGE	252	160	262	204	384	220
DATAMAN	862	674	933	1196	668	572
PRIDE 1	814	594	1186	646		

TABLE 7. MEMORY USE OF IN BYTES

The memory usage in bytes for the benchmarks is shown in table 7. In general, HOL benchmarks required 33% more storage than the AL versions. By comparing the memory use contributed by those benchmarks which used structuring in both the HOL and AL implementations with those benchmarks, which used structured coding in HOL and unstructured coding in AL, an evaluation of the effects of HOL implementation and structuring can be derived. The benchmarks, PURGE, DATAMAN, and PRIDE 1 were implemented using structured programming techniques in both the HOL and AL versions. For these benchmarks, JOVIAL implementations required 26% more bytes of storage than the MACRO-11 versions. For those benchmarks which used structured HOL and unstructured AL, JOVIAL required 36% more storage than the AN/AYK15A and FORTRAN required 23% more storage than MACRO-11. The code expansion contributed by structured coding then appeared to be around 10%.

Executable instructions for each benchmark were accumulated and execution times estimated to derive total benchmark execution times. These tabulations are shown in table 8. Since different execution times were used for each machine, no comparisons among machines would be relevant, but comparisons of language effects and structuring effects within a machnie is relevant. The comparison of structured HOL to unstructured AL shows an increase in execution time of 42% for JOVIAL compared to AN/AYK15A, and 45% for FORTRAN compared to MACRO. The comparison of structured HOL versus structured AL shows an increase in execution lines of 28% for both JOVIAL vs AN/AYK15A and FORTRAN vs MACRO-11. The increased run time attributed to structured coding then is approximately 14% for JOVIAL and 17% for FORTRAN.

BENCHMARK	J73	15A	FOR	MAC	CMS	TLT
AFLOOK	569	314	492	281	641	310
EID	217	134	249	91	194	171
EXD	748	396	511	314		
EXEC	285	194	320	178		
PRIDE 2	431	264	393	221		
PURGE	131	99	113	79	197	89
DATAMAN	513	414	474	429	221	226
PRIDE 1	524	331	544	306		

TABLE 8. EXECUTION TIMES OF BENCHMARKS

4. CONCLUSION

The characterization of EW software provided insight into the capabilities the instruction set used to program EW system should possess. Of those instruction sets presently used in EW systems or those which have potential of being used in EW systems, MIL STD 1750 appears to have the capabilities most compatible with EW processing. It is recommended that MIL STD 1750 be adopted for use in such systems, if assembly language programming is desired.

Evaluating HOL and structuring, in light of the EW processing requirements, has given insight into the specific impact these areas will have on storage requirements and execution speed. In general, it appears as if EW systems could presently anticipate increases in storage requirements, due to HOL inefficiencies, of approximately 25%. Of course, this number is a function of the specific compiler and could decrease as better compiler optimization is implemented. It would also appear that 10% more storage would be required when structured coding is implemented. The effects of structuring on execution time appears to be in the area of 14% increase. And the price to be paid in execution speed due to implementation in an HOL appears to be around 40%. Of course, these inefficiencies are also a function of the compiler and are subject to change. Using these numbers as rules of thumb, the EW system designer now has some insight to the price to be paid for using HOL and structuring, but they should in no sense be used exclusively in determining whether or not a system will utilize these techniques.

5. REFERENCES

1. Curtis, B., Sheppard, S., Kruesi, E., March 1980, "Evaluation of Software Life Cycle Data from the PAVE PAWS Project," RADC-TR-80-28.

2. Doty, D., Nelson, P., Stewart, K., February 1977, "Software Cost Estimation Study," RADC-TR-77-220.

3. Milliman, P., Curtis, B., February 1980, "A Matched Project Evaluation of Modern Programming Practices," RADC-TR-80-6.

4. Ziesig, D., Scheer, L., Perry, J., June 1981, "Electronic Warfare Software Study," AFWAL-TR-81-1006.

DISCUSSION FROM AVIONICS PANEL FALL 1982 MEETING ON
SOFTWARE FOR AVIONICS

Session 3 : SOFTWARE DESIGN AND DEVELOPMENT PROCESS - Chmn B. Mirailles (FR)

Paper Nr. 13 - THE IMPACTS OF STANDARDIZATION ON AVIONIC SOFTWARE
Presented by - Dr. D. E. Sundstrom
No Questions

Paper Nr. 14 - ADA STATUS AND OUTLOOK
Presented by - L/Cdr J. F. Kramer
Speaker - G. Lejeune
Comment - ADA is anticipated to be used by the US Air Force for "Low Risk Programs". Can you define those programs more clearly.

Response - What I said is that ADA will be used by the US Air Force for "Low Risk Programs" until adequate compilers and an environment is available. I define "Low Risk Programs" as those where the programming language choice will not be on the critical path of a program, where a production quality compiler for the program computer is not required until after 1985 or 86, and where the program is not too large and complex. The Air Force may attempt to do one or more parallel Jovial/ADA developments to gain experience without having the implementation language be the critical path item.

Paper Nr. 14 - ADA STATUS AND OUTLOOK
Presented by - L/Cdr J. F. Kramer
Speaker - H. Schaaff
Comment - Will there be any subsets of ADA allowed? If not, how will that be achieved?

Response - No subsets will be allowed. This will be achieved by the validation capability. This does not mean that you can not choose to implement project management constraints on what you let your programmers use, particularly the inexperienced programmers. It also does not mean that a particular job code generator and run time can not be optimized to execute a particular application language area and a certain portion of the language better than the rest of the language. But an ADA computer must implement the whole language to be called an ADA compiler.

Paper Nr. 14 - ADA STATUS AND OUTLOOK
Presented by - L/Cdr J. F. Kramer
Speaker - M. Kleinschmidt
Comment - Do you have a concept for the validation of ADA compilers?
Is there a well defined procedure?

Response - There are over 1400 routines designed to check the presence of features and the absence of non-standard features. The tests are both compile time and run time tests designed to check for the presence or absence of legal programs; link/load rejection of illegal programs and self testing. The tests are publicly available, but will be run at certification time by validation personnel with their own parameters input to the test.

Paper Nr. 14 - ADA STATUS AND OUTLOOK
Presented by - L/Cdr J. F. Kramer
Speaker - H. von Groote
Comment - You were mentioning that today preliminary compilers are available. Do you know of experiences gained in applications of ADA with these compilers and would you like to comment on them?

Response - I know of one non-imbedded computer application for a parts inventory, accounting, and payroll using the telesoft compiler on a Motorola 68000 microprocessor. Their experiences for the first part of the system which the customer is using and has accepted are described in the MAy, June 1982 ADA letters (Vol. 1, Number 4). They have now delivered the second increment for a total of 70,000 lines and still feel the same. I quote from the article. "To the ADA detractors who claim the language is too large, too complex, too limited, too slow, etc., we say (resoundingly) "POPPYCOCK!". There is another user of the Intellimac Computer at the Naval Weapons Center, Dehlgren, who is using it for modeling and rapid prototyping and who is also very positive and whose only regret is the features not yet available like generics. I think users are indicating that ADA is a good language.

Paper Nr. 15 - STANDARDISATION DU LTR POUR CALCULATEURS EMBARQUES - LE PRESENT ET LE FUTUR
Presented by - ICA De Montcheuil
No Questions

Paper Nr. 16 - USE OF HIGH ORDER LANGUAGE FOR OFP PROGRAMMING WITH EMPHASIS ON THE USE OF ADA
Presented by - R. Westbrook
No Questions

Paper Nr. 17 - AN APPROACH TO A PORTABLE PASCAL LANGUAGE FOR DIFFERENT ONBOARD COMPUTER SYSTEMS
Presented by - Dr. H. Wiemer
Speaker - D. M. Weiss
Comment - Could you describe the method used to specify the target micro-processor to the code table generator?

Response - The target microprocessor, to be precise the target assembly instructions, are described by a sequence of generator instructions. By executing them the related bit (code)

patterns are generated.

Paper Nr. 17 - AN APPROACH TO A PORTABLE PASCAL LANGUAGE FOR DIFFERENT ONBOARD COMPUTER SYSTEMS
Presented by - H. Weimer
Speaker - W. Fraedrich
Comment - You said that for political reasons PASCAL was selected. In which programs/projects are you using the described/presented features?

Response - The described features are almost in the development phase, therefore no programs/projects exist, to which the dedicated system is applied.

Paper Nr. 17 - AN APPROACH TO A PORTABLE PASCAL LANGUAGE FOR DIFFERENT ONBOARD COMPUTER SYSTEMS
Presented by - Dr. H. Wiemer
Speaker - W. I. van Meurs
Comment - The topic you discussed handles sequential PASCAL. To cope with realtime applications have you considered applying your methodology to concurrent PASCAL? An additional advantage hereof is the independence of any manufacture-supplied operating system on the target, since CP is based on a virtual machine implemented by a small kernel/interpreter running on the target machine.

Response - Considering the possible realtime features we have taken a look at concurrent PASCAL. We only adopted the datatype semaphore. The monitor concept was omitted.

Paper Nr. 18 - USE OF HIGH ORDER LANGUAGES ON MICRO-PROCESSORS
Presented by - R. M. Boardman
Speaker - E. J. Dowling
Comment - You mentioned that you had a requirement that during testing, all statements should be executed at least once. Did you have any automatic tools to help decide when this had been achieved?

Response - We had no requirement to obey all statements at least once during testing. This was an objective set by myself as project manager. We did not use any automatic tools to help us achieve this.

Paper Nr. 18 - USE OF HIGH ORDER LANGUAGES ON MICRO-PROCESSORS
Presented by - R. M. Boardman
Speaker - N. P. H. Haigh
Comment - You mentioned that the HOL should have an "Assembler-Code Insert" capability, and also that the programmer should be familiar with the assembly language of the target micro. Would you comment on the lack of generality this might introduce?

Response - Short sections in assembler language were necessary to meet the timing constraints in the target system. However in order to test the software on the host system we wrote equivalent HOL text for each code segment. If a new processor were used this text could be used in place of the assembler sections. It has been tested on the host and if the new processor could meet the timing constraints it would work.

Paper Nr. 19 - SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT
Presented by - R. Dibble
Speaker - Dr. H. R. Simpson
Comment - Can you separate out the overheads due to CORAL from those attributable to MASCOT?

Response - The estimates quoted in the paper compare MASCOT systems with other CORAL based systems and should therefore be wholly attributable to the use of MASCOT methods.

Paper Nr. 19 - SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT
Presented by - R. Dibble
Speaker - Dr. T. G. Swann
Comment - In our somewhat limited experience of MASCOT we too have found high store-space overheads. But these were largely due to deficiencies in the compiler, which was unable to produce re-entrant, shared, code. The remaining overheads were only the normal overheads needed to communicate between code modules. we would have had to have these even without MASCOT. Can you comment?

Response - The overheads resulting from deficiencies in the compiler are separate from those due to the methodology. Ferranti MASCOT systems are supported by CORAL compilers that do allow re-entrancy. If you choose to communicate between modules as in MASCOT then you will incur a certain level of overheads. These may well be greater than might be incurred if you use an alternative method of design. However that is not a criticism of the MASCOT method, but a statement that S/W quality (in terms of reliability, maintainability, etc.) is not obtained without cost.

Paper Nr. 19 - SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT
Presented by - R. Dibble
Speaker - Dr. A. A. Callaway
Comment - You had considerable experience with the CORAL Mascot combination for building real-time systems.
Do you think we are actually going to see any significant advantages in moving to ADA? In other words is it going to bias anything, or are we going just to be fashionable?

Response - They will buy either facility in 1980 language rather than in 1970 language, extra

facilities, and I suspect in the short term it will cause you a lot of heart-ache, but maybe in the long term it will be worth it.

Paper Nr. 19 - SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT
Presented by - R. Dibble
Speaker - M. J. Looney
Comment - It was stated in your paper that the top level decomposition did not actually use MASCOT techniques.
At what stage were they introduced?
Do you think that this might in any way have had an effect on the final overheads?
Do you consider that the proposals put forward by Dr. H. Simpson in his paper will make the use of MASCOT for large systems easier?

Response - For the projects discussed in this paper, the ACP diagram was not used at the top level of software design but was used at the task level (Level 2). ACP diagrams have been produced retrospectively for each system at the top level of S/W design and, in the particular system (system 1) with which I was associated, did not show up the need for any design changes. With regard to the decomposition process, MASCOT overheads would seem to be a function of the final level of decomposition. To terminate this process at a high level, to reduce overheads would seem to negate the MASCOT aims at modularity and design visibility. I do not think we would have ended up at a different level of decomposition if we had used the ACP diagram at the highest level.
With regard to the final question. Dr. Simpson's presentation was my first contact with these new proposals and I would wish to reserve judgement until I have had a chance to study his paper (which was not included in the Conference Preprints).

Paper Nr. 19 - SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT
Presented by - R. Dibble
Speaker - M. J. Looney
Comment - The prospects of several implementations of MASCOT existing appears to disturb you. Why should MASCOT be limited to one version, we have already heard of numerous compilers for ADA and several APSE systems?

Response - I am not disturbed by the existence of several MASCOT systems. What I said in my presentation was that the existence of <u>different</u> implementations might undermine the MASCOT aims of portability and standardization. If the handbook is not sufficiently precise in its definition of MASCOT this could happen.
Taking your own example, in ADA great efforts are being made to allow only standard ADA compilers to exist.

Additional Comment by M. Looney

Due to comments originating from MOD(UK) contractors using MASCOT, a MOD working party was set up to look into how the firms were using it. The working party, of which I was chairman, held talks with ten different organizations, representing both users and users/suppliers about their experiences in using MASCOT to develop software systems.
A summary report on the findings of the Working Party has recently been issued. I will be glad to send a copy to anyone interested if they give me their name and address, or contact me at ASWE.

Paper Nr. 19 - SOFTWARE DESIGN AND DEVELOPMENT USING MASCOT
Presented by - R. Dibble
Speaker - Dr. D. J. Martin
Comment - The size of 960 words per activity sounds large in comparison to good programming practice. Does MASCOT drive you towards larger modules?

Response - As I stated in the presentation. The activity size estimate was based on a limited sample and I hope to be able to refine this figure in due course. We would certainly agree that for a functional decomposition the resulting functional modules would be small. Several projects had halted the decomposition process at a higher level to reduce the number of activities and hence MASCOT overheads. So it does seem that for current systems the use of MASCOT will result in larger basic software modules.

Paper Nr. 20 - SAFETY CRITICAL FAST-REAL-TIME SYSTEMS
Presented by - Dr. B. Gusmann
Speaker - Dr. W. J. Cullyer
Comment - Should the whole "C" language be used or are there constructions which should be barred?

Response - None of the constructions we used proved to be dangerous. However, we did not examine all the constructions in the language.

Paper Nr. 21 - USEABILITY OF MILITARY STANDARDS FOR THE MAINTENANCE OF EMBEDDED COMPUTER SOFTWARE
Presented by - Prof. N. Schneidewind
Speaker - J. P. Sudworth
Comment - Standards are only meaningful if they are enforceable. Does not this imply that we should incorporate into the standards we impose on suppliers, the testing methods and tools we propose using to establish that delivered software conforms to the standard. Are adequate tools available?

Response - Yes, testing methods and tools should be included in the standard. A good idea!

Paper Nr. 21 - USEABILITY OF MILITARY STANDARDS FOR THE MAINTENANCE OF EMBEDDED COMPUTER SOFTWARE
Presented by - Prof. N. Schneidewind
Speaker - Capt. J. Astley
Comment - Since MIL-STD-1679 is to an extent itself modular in that it normally calls out the associated DIDs (Data Item Descriptions) which define documentation form and contents, would it not be more appropriate to put current analysis methodologies in these subservient and more easily modified DIDs rather than the main body of the standard?

Response - An interesting idea. My first reaction was that using a DID would save a lot of work in achieving the goal of updating a standard. upon further reflection, I concluded that this would not be a good idea because relatively few people read DIDs as compared to those who read the standard itself. In order to give the new parts of a standard wide distribution, the updates should appear in the standard itself.

Paper Nr. 21 - USEABILITY OF MILITARY STANDARDS FOR THE MAINTENANCE OF EMBEDDED COMPUTER SOFTWARE
Presented by - Prof. N. Schneidewind
Speaker - H. R. Simpson
Comment - You said that when considering standards, method is more important than hardware technology. I agree with this. However, method should take some account of target environment. One important new feature of computing technology is the use of distributed processing systems. Would you not agree that methods need to change to cope with this, particularly by placing greater emphasis on parallelism rather than changing to methods too strongly based on sequential concepts?

Response - No.

Comment - Supplementary Point (briefly mentioned in author's talk)
The concept of mode can be difficult to apply in distributed processing systems due to information propagation delays. Consequently formal techniques which make use of system modes may be flawed when applied at the level at which system operation takes place at several processing sites executing concurrently.

Response - One of the tenents of a standard is that it should specify what is required but not "how" to achieve the requirement. If the suggestions were implemented this principle would be violated. A new methodology for analyzing distributed systems would be appropriate for inclusion in a standard; the technology of distributed systems would not be appropriate for inclusion in a standard.

Paper Nr. 22 - SOFTWARE CONFIGURATION MANAGEMENT AT WORK
Presented by - Dr. Jan T. Pedersen
Speaker - S. Oxman
Comment - The paper ended at certification. What about post-certification field audit?

Response - Since in Norway, we normally maintain all our software, even for defence customers, we normally have control of the maintenance process. Our types of procedures are adherred to also in the post-development phase of a software system, thereby implicitly achieving some form of post-certification field-audit. However, we have not specifically focussed on exactly that issue.

Paper Nr. 22 - SOFTWARE CONFIGURATION MANAGEMENT AT WORK
Presented by - Dr. Jan T. Pedersen
Speaker - A. Cameron
Comment - Your paper has discussed mainly the ways in which the Kingsberg staff are controlled in order to produce a high-quality product. Please indicate the manner in which the customer is controlled to prevent unnecessary interference and disturbance of the Kingsberg staff.

Response - Customers are intensely informed about our methods for controlling software projects. When they are convinced that we are able to perform, they limit themselves to participate in scheduled reviews and progress report meetings.

Paper Nr. 23 - CONFIGURATION MANAGEMENT AND THE ADA PROGRAMMING SUPPORT ENVIRONMENT
Presented by - Chf. Eng. K. Pulford
Speaker - A. R. Runnalls
Comment - I think you will agree that there is no need to wait for ADA to introduce automated configuration control along the lines you suggest; indeed our experience is that if the facilities of a standard commercial operating system like UNIX are exploited, the necessary tools can be produced with remarkably little effort. One area where we have found difficulty, however, is in dealing with diagrams. Have you any proposals or comments in this regard?

Response - Yes, this is true but there are two points that must be added. Firstly the APSE has a general database which allows all data, including not only configuration management data but other project data, to be accessed in a uniform way. This is not strictly true for UNIX. Secondly with this growing prevalence of APSEs the sorts of problems described in this paper will become more common.
As far as storing diagrams, there are already many techniques in CAD for storing diagrams so it is certainly feasible to do this and certainly desirable if one is also storing the contents of

the document..

Paper Nr. 23 - CONFIGURATION MANAGEMENT AND THE ADA PROGRAMMING SUPPORT ENVIRONMENT
Presented by - Chf. Eng. K. Pulford
Speaker - T. F. Kensey
Comment - Configuration Management covers wider aspects than put forward in this paper. Has the author considered the various resource aspects regarding the cost of implementation in terms of core occupancy, computer time loading, cost of implementation, or impact upon delivery?

Response - The author has not considered the problems of the resources required to support such a system which are really outside the scope of the paper. But the author is worried about the resources that will be require to support an APSE.

Paper Nr. 23 - CONFIGURATION MANAGEMENT AND THE ADA PROGRAMMING SUPPORT ENVIRONMENT
Presented by - Chf. Eng. K. Pulford
Speaker - Wg. Cdr. S. Barker
Comment - To what extent are the facilities you describe a necessary component of the APSE?

Response - All the tools described are not necessarily part of the APSE. It would still be possible to develop a project on the APSE using normal configuration.

Paper Nr. 23 - CONFIGURATION MANAGEMENT AND THE ADA PROGRAMMING SUPPORT ENVIRONMENT
Presented by - Chf. Eng. K. Pulford
Speaker - E. J. Dowling
Comment - As a comment, rather than a question, there will of course be a large number of APSEs and quite a lot of MAPSEs and KAPSEs. It will only be possible to easily transport configuration management (and other) tools if there is some attempt to standardize database interfaces, etc.

Response - Agreed. But the lack of portability does not affect the conclusions of the paper.

Paper Nr. 24 - SOFTWARE FAULT TOLERANCE FOR REAL-TIME AVIONIC SYSTEMS
Presented by - Dr. John Knight
Speaker - B. Malcolm
Comment - We must pay for fault tolerance through effort spent on additional code and complexity, which in turn might have an interesting effect on overall reliability. Given your reluctance to reduce effort spent on fault-intolerance are you saying that this is a way of adding additional resources to a project when some kind of limit has been reached with what can be achieved in fault-intolerancy or could it be that there might be a balance between the two within a given budget?

Response - In critical applications where the digital system has to remain operational, reliability is the key issue. I would prefer to see fault tolerance used solely to improve reliability. I can see the argument being used that software fault tolerance can be used to improve reliability within a fixed budget by reducing the attention paid to fault intolerance. I think it is too early to tell whether this is appropriate.

Paper Nr. 24 - SOFTWARE FAULT TOLERANCE FOR REAL-TIME AVIONIC SYSTEMS
Presented by - Dr. John Knight
Speaker - Dr. J. Martin
Comment - You suggest that fault tolerance should be an addendum to fault intolerance. I shall outline a case where we have used "fault tolerance" as the basis for producing a high integrity system.

Paper Nr. 25 - Cancelled

Paper Nr. 26 - ELECTRONIC WARFARE SOFTWARE
Presented by - R. Shaw
No Questions

AN EIGHT POINT TESTING STRATEGY FOR REAL TIME SOFTWARE

R.E. WILSON
N. HIGSON

Marconi Avionics Limited
Elstree Way
Borehamwood
Herts.

SUMMARY

This paper describes a strategy for testing real time modular software systems. It gives an eight point strategy with its structure, its objectives and documentation considerations, and includes as an appendix, a glossary of terms used in the paper.

The paper considers the responsibilities of programming staff using such a strategy and the problems entailed in re-testing as result of errors detected at higher levels of testing.

1. INTRODUCTION

Software testing of a real time modular software system is often the least structured aspect of development. This paper describes an eight point strategy which distinguishes between bureau machine testing and specialised system testing environments (fig. 1). It considers the impact of correcting errors detected at high level of testing and the experience of using such a strategy on a real time system. The paper concludes with a glossary of software component terms used.

The strategy described here presumes the design of a real time system to be decomposed (fig. 2) into software 'module' units that do useful work by manipulating parameters or data areas. The concept includes the 'process' which is a unit of software with its associated data areas that is recognised by an operating system so has known attributes within the total system, and the 'suite' in which groups of processes provide system facilities.

2. TESTING STRATEGY

The strategy is based upon the assumptions that a software system is designed from a specification and that the top level design sub-divides into progressively smaller software components which with data areas do useful work within the system. The software components are implemented in code and by their build-up and testing to larger and larger components a completely tested system is generated.

The testing strategy, which has 8 points, requires two testing environments namely, a bureau machine environment and System Testing environment with specialised hardware.

2.1 Bureau Testing

The first 4 points of the strategy, require test of the software on a bureau machine environment with normal software development aids to support the programming language in use and all such testing within the control of a commercial Operating System.

A fundamental principle of the strategy in the bureau enviroment is that a module in the hierarchy structure has input and output criteria which remain constant no matter how much of the structure is present during a test. Therefore, it will be seen that the input criteria used during a Point 1 test for a top level module are the same as for a Point 4 test for the module.

2.1.1 Point 1: Module Test

This testing applies to all modules whatever their hierarchy position in the design. Any lower hierarchy chore called by the module under test is replaced by a test stub and its test harness controls the input and output information.

The aims of this test are:

(i) to test rigorously a module especially with all exeptional conditions (including error situation)

(ii) to show that the input/output criteria meet the design requirements.

This test is carried out by the programmer who implemented the module, and is responsible for:-

(a) Coding the module
(b) Defining input and output criteria for the module
(c) Coding a test harness
(d) Coding test stubs if required.

On completion of this point, all information pertaining to the test should be registered in a software library. This information should contain test harness, test stubs, input and output data and running instructions such as Job Control Language details. At this registration, documentation for the chore in the form of a functional flow diagram and test information should be provided. See Fig. 3 for Module Structure, and Fig. 4 for Module under test.

2.1.2 Point 2: Small Hierarchy Module Test

This is the first stage of integration in that tested modules are grouped together in their hierarchy sequence. The number of modules grouped will depend upon their size, complexity and function.

The aim of this test stage is:

(i) Integration test: the compilation of the module should test for correct use of common data areas, detection of name clashes and provide the beginning of the link command details.

(ii) Module interface test: by replacing test stubs by the actual modules, the test shows if the module interfaces are correct.

(iii) Independent test: to provide an independent check that the original test criteria of all modules were valid and complete.

The tester should be a senior programmer, ideally with little involvement in the production of the modules.

The input and output criteria for this stage of testing are the same as those for the top module when it was Point 1 tested.

This testing does not generate more test results but confirms those from Point 1. However, documentary proof that the test has been completed should be registered and signed by the tester.

2.1.3 Point 3: Module Group Test

This stage of testing is an extension of the Point 2 but with different aims.

The aims of this test are:

(i) to test functions instead of compilable units.

(ii) to test that valid data is processed correctly.

This means that some modules will not have all their low level modules or test stubs compiled since only those modules needed to provide a particular function are compiled. Tests at this stage should not aim to testing all combinations of error conditions for the input criteria, but test that valid data is processed correctly, the correct files are manipulated and control returns properly. Error conditions should have been tested at Point 1 or Point 2 for such functions. This level of testing is the responsibility of the Team Leader who should provide documentary evidence of this test such as hardcopy output from the test run. Module group testing should, of course, be carried out for all functions.

2.1.4 Point 4: Partial System Test

This test is the summation of Point 3 tests. All modules for a particular task are integrated including the control chore for the whole task. The task is tested as a unit with test stubs to other tasks and the input and output criterion is the sum of the Point 3 criteria. At this stage the testing should be restricted to simple functions with minimal exceptional conditions and the testing should be supervised by a Team Leader.

The aim of this point is:

(i) to produce a tested task ready to be transferred to a specialised test environment of the next stage.

Documentary evidence should be provided by the Team Leader that this stage is complete for a task. Since the task will be transferred out of a Software Generation team's control, the Team Leader should ensure that all testing aspects and documentation for the task are complete and any amendments that have arisen from testing have been applied to the registered information.

2.2 System Testing

The software generated and testing during the first 4 points of the strategy produce tasks that have been tested on a bureau machine running under an Operating System and therefore, the testing has been static testing of the software components with no "real-time" considerations. The latter 4 points require facilities which allow the software to be run in real time with representative hardware interfaces.

The testing of the software for the first 4 points will have been the responsibility of a Software Generation Team(s) but that for the last 4 points should be the responsibility of an Integration Team. This team should be independent of the original generation teams and they should be familiar with the specialised hardware required for System Testing.

The bureau testing will have produced software components that correspond to a design task but before such components can be integrated further, they may need to be re-compiled into a process so that they interface directly with an Operating System.

During System Testing, documentary evidence of tests should be produced since many of the tests will not produce hardcopy results. A test certification document should be raised by the Integration team for any test stating the test environment, aim of the test and results. The certificate should clearly note the hardware configuration since failure during system testing can be attributable to hardware and software.

2.2.1 Point 5: Integration Testing

This test comprises the grouping of a number of processes from the suite which perform a function and testing them as a single unit. This testing requires an environment in which processes can function directly with a resident Operating System but the interface between the processes can be monitored.

A test harness is needed which gives access to the inter process data files and can monitor the control flow between processes and Operating System.

The input associated with special hardware interfaces should be controlled by the test harness so that simulated data can be loaded into an input buffer, processed by the software under test, and the software's output buffer cleared by the test harness.

During this testing, the processes should be run as close as possible to their final configuration but access to other processes not under test will need to be replaced by test stubs.

The aims of Point 5 testing are:-

(i) to test that transferrance from bureau testing to System testing is successful

(ii) that processes are providing the required functions.

See Fig. 5 for Point 5 testing environment.

2.2.2 Point 6: Suite Testing

This testing is an extension of Point 5 testing in that the test harness is replaced by software to interface to real hardware and all the processes associated with a suite are combined for this test. Since a suite is a design grouping of tasks, this test will still need test stubs to other suites not under test.

The aims of this test are:-

(i) to test hardware and software interfaces.

(ii) to run the test in real time with simple tests to exercise the software.

Since this test is run in real time with special hardware, the documentary evidence of a test with hardware and software configuration is essential.

See Fig. 6 for Point 6 testing environment.

2.2.3 Point 7: Facility Testing

A number of suites are combined for this test so that test stubs to other suites are replaced by software components. The combination of suites should be selected to provide system facilities and the test data used for the previous stage used for this testing.

The aims of this test are:-

(i) to run all tests in real time

(ii) to repeat Point 6 tests to check for System integrity, particularly access modes of data areas

(iii) to test functions that require more than one suite.

2.2.4 Point 8: System Testing

All suites of the system are combined and the whole system tested by re-running the Point 7 facility test data.

The aims of this test are:-

(i) to run the whole system as realistically as possible to its final requirements.

(ii) to test that Facility tests of Point 7 are still valid.

(iii) to test more complex inter-suite functions particularly little used ones or complex functions.

(iv) to prepare the system for Acceptance testing.

3. PROBLEM OF RE-TESTING IN A LARGE SYSTEM

When the system has satisfactorily completed the Point 8 testing, all information for the system should be registered in a software library. This includes link commands, test harnesses and job control language details that have been developed during the System Testing stages.

A bonded system should be produced for Acceptance tests. Errors detected during acceptance testing and system testing should be brought to the attention of the software generation team. Assuming the error needs software correction, then the generation team should edit and re-register the revised chores.

This raises the question as to how much re-testing should be done to check the correction. To test minor alterations through all 8 stages of this strategy would be impractical and so the correction should be classed into either code only change, data area change or code/data change. Depending upon the class the amount of re-testing can be judged.

3.1 Code only change

The incorrect chores should be edited and registered and all changed chores should be subject to Point 1 testing. This may mean changes to the test harness. A re-run of the Point 4 testing to give a confidence check for the process should be made and then a new system generated and subjected to Point 8 testing.

3.2 Data area change

If there is no code change with data area change, then the system should be re-generated and the system subjected to Point 8 testing.

3.3 Code and Data area changes

Such alterations can have far reaching effects upon a system. However, by good design and implementation, the extension of a data area can be minimised and the use of access routines to data areas so that the code changes are restricted to a few chores can reduce the effect of such data area changes.

Within the strategy, the revised chores should be subjected to Point 1 testing. A change effecting a single process can be tested by Point 4 and finally the system testing (Point 8) but a change with wider implications can be tested by sampling testing using Point 3 testing and Point 6 testing if more than one process has changed.

There are no hard and fast rules for re-testing but a good integration testing environment will quickly show if a minor change at chore level has cured a fault and has or has not introduced new ones.

4. BENEFITS OF THE TESTING STRATEGY

This strategy has been developed on a large system. The system resulted in approximately ½ megabyte of real time software comprising 16 processes produced from over 300 chores. When Point 8 testing was started, it took less than a week to complete the link commands and generate a system and the first generation produced a system stable enough to start preliminary investigation into system testing without it frequently crashing.

The benefits from using such a strategy are:

(a) it provided a means of controlling the testing of a system which was created by a large software team

(b) it gave a clear distinction between bureau testing by small programming teams and system testing by an Integration team

(c) the linkage commands generated at each successive point were closer to the final system requirements so that the privileged access needs of code and data could be checked at each Point.

The strategy discussed in this paper involves software staff following a well defined structure and producing evidence of testing at each stage.

The end result is a much better tested product.

APPENDIX: SOFTWARE COMPONENTS USED IN THE TESTING STRATEGY

A1.1 The software components considered in this paper are given below. The suite, task and chore are design concept components.

A1.2 Suite

In a design, a suite is defined as a logical grouping of software tasks which participate in a common function. For example, the software tasks required to provide an Operator Interface of input commands and output display information. In a system's implementation, the majority of the tasks equate to a "process" and so a suite is a collection of processes providing a facility of the system.

A1.3 Task

This is a design component which is dedicated to the performance of a particular function that is identifiable as a major part of an operational requirement. For example, the software required to log hardware "Built-inTest" information when it is reported to the system.

A1.4 Chore

This is the software component in the hierarchy structure for the lowest design level. Its functions range from control chores which have enough logic to control other chores in the correct sequence to chores that do some useful work such as transferring data between files. A chore is the fundamental building block for the system and so its rigorous testing is essential to this strategy. A chore must be a testable unit with input and output criteria.

A1.5 The Basic Code Unit, Module, Process and Integration are Implementation components.

A1.6 Basic Coded Unit (BCU)

This corresponds directly to a chore, it comprises the code for a chore and it has an input and output specification. This input/output data may be via parameters as in a procedure call or via data areas.

A1.7 Module

This is defined as the BCU of a chore with any local data areas required to support the function of that chore. When a chore is under test, the test harness calls, in fact, the module associated with the chore.

A1.8 Process

Modules are built up as defined by the design up to tasks and a task that is controlled by a resident operating system is called a "process", some times called a "program". Since the operating system recognises a process then it has known attributes within the total system. For example, its read/write access within the system can be limited to maintain system integrity.

A1.9 Integration

Within the context of this paper, integration means the linking of tested software components to produce processes or the total system.

A1.10 The test stub and test harness are Software Test Tool components.

A1.11 Test stub

For a module to complete its function, it may require access to lower hierarchy modules. The module under test would either pass parameters or set data before the call and on completion of its function, the lower module returns control. A test stub is sufficient software to emulate the lower level module so that the output of the calling module can be monitored. It should be capable of replying with the correct data so that the module under test cannot distinguish between the stub and the real module.

A1.12 Test Harness

This is sufficient software to handle the following:

(a) Input data. The setting up of initial conditions, fixed data or parameters. It may interface to an operator for the input of variable data if required.

(b) Output data. The storage of output data from the module, displaying of data areas on entering or on completion as required.

(c) Test stubs. Providing the stubs and monitoring the information passed between modules.

(d) Lower Hierarchy modules. A facility to replace test stubs by tested modules.

Point Number	Input	Test Stage	Output	Responsibility
1	Start criteria (parameters)	Module Tests (BCU + data)	End Criteria	Programmer
2	Start criteria (parameters)	Small Module Hierarchy Tests. 2 or more modules	End Criteria (parameters)	Senior Programmer
3	Message structure or data file	Group Module Test (functions)	Specified Results	Team Leader
4	Sum of Group Test Module data	Sub-systems Modules (suite/tasks)	Specified action	Team Leader
5	Simulation Test Data	Integration testing	Specified action	Integration Team
6		Suite testing		
7		Facility testing		
8		System testing		

FIG. 1. TEST POINT CRITERIA AND RESPONSIBILITIES

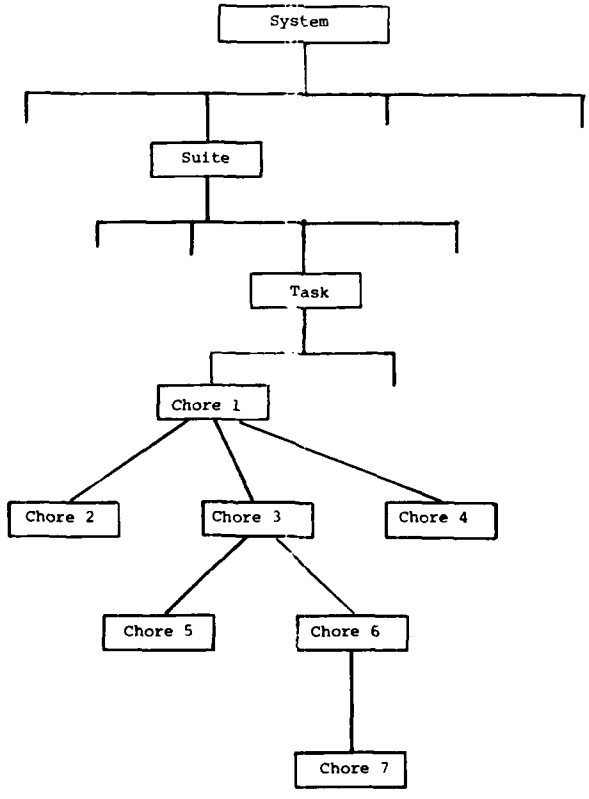


FIG. 2. HIERARCHY STRUCTURE OF SYSTEM

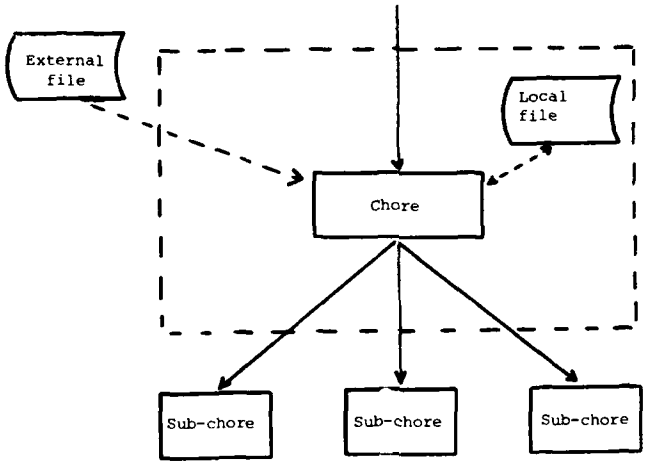


FIG. 3. MODULE STRUCTURE

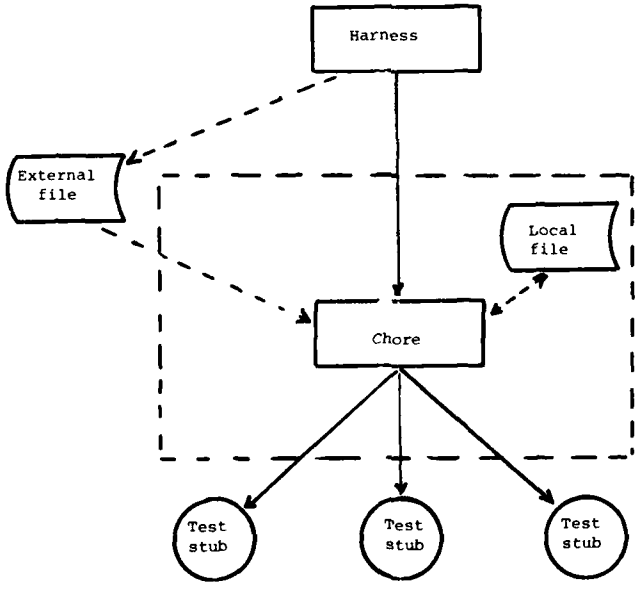


FIG. 4. TEST HARNESS STRUCTURE

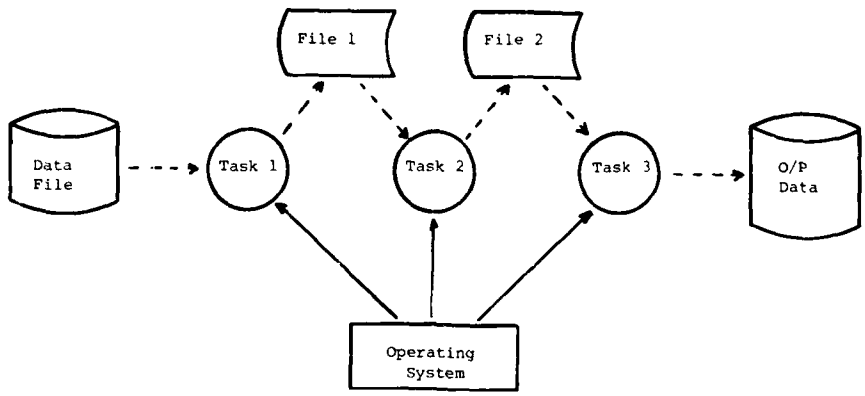


FIG. 5. EXAMPLE OF POINT 5 TESTING

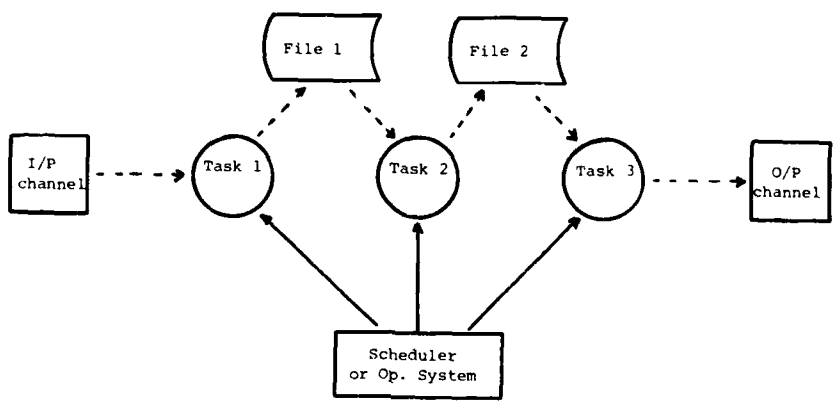


FIG. 6. EXAMPLE OF POINT 6 TESTING

TORNADO FLIGHT CONTROL SOFTWARE VALIDATION: METHODOLOGY AND TOOLS

by

Roberto PELISSERO
AERITALIA - Gruppo Sistemi Avionici ed Equipaggiamenti
10072 CASELLE (Torino) - Italy

ABSTRACT

In the development, integration and ground testing of avionic systems real time techniques are essential for software validation and verification especially when flight control systems are concerned.

The aim of the presentation is to describe methods and tools adopted in the Tornado project to perform at ground the following activities:

- Software confidence testing
- Investigation of failure effect
- Performance prediction
- Assessment and read across between simulated and in flight behaviour.

The real time facility in use is based on an integrated hardware/software system that has been ad-hoc designed to allow closed loop testing of the Tornado TF/AFDS Subsystem with particular emphasis to the Autopilot. The real equipments of interest are installed on an avionic rig and the flight conditions are obtained via a data flow to/from an external computing facility which performs both the acquisition/stimulation functions and the various simulations (aircraft, engine, sensors, etc.).

The validity of the results achieved by the presented facility is confirmed by repetition of manoeuvres performed during previous flights.

In addition an overview of other possible applications of such a facility is summarized.

1. INTRODUCTION

The utilization of an avionic system rig has been demonstrated very useful during the 1970s in aircraft development.
The rig system purpose is to provide the capability of performing avionic system integration. The simplest arrangement is obtained by using a bench reproducing the aircraft electrical wiring: this makes it possible to test individual equipments and to connect each other the various avionic units.
A second step is to handle the avionic signal flow by a computer able to make static and dynamic data stimulation and acquisition. Such an arrangement allows to perform:

- hardware/software integration
- system investigation
- flight back-up
- support for avionic system changes.

When an automatic flight control system, as the Tornado one (Terrain Following/Autopilot and Flight Director System), is to be considered, the use of open loop tests is not sufficiently representative of the real behaviour due to the nature of the system itself.
A real time simulation system with hardware in the loop is the most flexible and realistic tool to perform tests and analysis on Tornado TF/AFDS subsystem in order to provide necessary information to clear the various operating modes and their combinations. As a closed loop simulation system is very closely representing the actual environment of operation of the real system, it can be used to perform all the activities for the clearance recommendations and flight envelopes definition.
The closed loop capability is achieved adding to the data stimulation and acquisition system previously mentioned the aircraft dynamics simulation. The two functions, i.e. signal handling and simulation, can be performed on a single computer only if its characteristics, in terms of speed and usable program space, are able to satisfy the testing requirements, otherwise a two computer solution is preferable.
The testing objective of a rig having the real time closed loop capability (Closed Loop System) is to prove the performance and the hardware and software integrity under normal and abnormal operating conditions of the system to be tested. For the automatic flight controls this includes also:

- verification of correct implementation of the control laws and mode/failure logics
- performance evaluation with tests equivalent to the ones carried out in flight

- failure analysis
- software validation

The set up of a facility simulating in real time the behaviour of the controlled aircraft allows to perform this kind of activities efficiently and cost effectively.

2 SYSTEM TO BE TESTED

The Tornado Autopilot and Flight Director System (AFDS) is a digital integrated autopilot, flight director and autothrottle system which has been developed and now is being produced by Marconi Avionics and Aeritalia - Gruppo Sistemi Avionici ed Equipaggiamenti.
The AFDS has been designed to provide automatic control of the aircraft in the longitudinal and lateral axes in various operating modes (autopilot) and to provide the necessary information to the pilot instrumentation so that monitor of automatic performance or manual aircraft guidance shall be possible. The autopilot controls the aircraft by means of manoeuvre demand signals to a triplex fly-by-wire primary flight control system which is called Command and Stability Augmentation System (CSAS). In addition an "autothrottle mode" facility allows calibrated airspeed hold, via engine thrust control, and a "pitch autotrim" facility maintains, via autotrim motor control, the pilot stick in the trimmed position.

The Autopilot system is duplex to satisfy safety requirements and to have flight director facility available in case of autopilot malfunction. The two computers (AFDC1 and AFDC2), which are self monitored, have identical computing and only differ in the output interface circuits; the former computer provides triplex analogue pitch rate and roll rate autopilot demands to the CSAS, whilst the latter provides signals to F.D. displays, autothrottle and autotrim facilities.
The two computers are continuously cross monitored: in the event of a failure of either AFDS computer, mode computations results are applied to the output interface of computer 2 as F.D. information only.
The system moreover contains:

- a control panel allowing the pilot to select the operating modes, to perform the preflight and first line procedures and to see failure indications
- an autothrottle actuator for the thrust control
- pitch and roll stick force sensors giving information to the computers of the overcoming of prefixed stick force thresholds. (This enables the pilot to take up the control of the aircraft overriding autopilot operation just making the correct effort on the control stick).

The redundancy philosophy applies also to the AFDS input signals; from this point of view they are divided in the following groups:

- signals for which failure protection is provided by manoeuvre limits within the control laws
- signals which are fully monitored at source
- signals, coming from different sources, which are monitored within the computers.

The AFDS operating modes allow control in pitch, roll and thrust. As the control always acts on both pitch and roll axes it is not allowed to select a mode on a single axis. The mode selection is performed according to compatible modes combinations, predefined in the mode and failure logic system.
The operating modes are the following:

- Attitude/Heading hold mode (basic mode)
- Heading acquisition mode
- Track acquisition mode
- Barometric altitude hold mode
- Autothrottle mode
- Automatic approach mode
- Terrain following mode
- Radar height hold mode

Other AFDS facilities are:

- CAS datum adjust that can be used in autothrottle mode
- Autotrim, for the longitudinal modes
- Automatic steering override (ASO), for basic modes
- Stick Force Cut Out (SFCO), for other modes combinations
- Instinctive Cut Out (ICO), for all the modes

The AFDS during its operation is interfaced with the following equipments:

- Command and Stability Augmentation System (CSAS)
- Inertial Navigator (IN)
- Secondary Attitude and Heading Reference (SAHR)
- Air Data Computer (ADC)
- Main Computer (MC)
- Horizontal Situation Indicator (HSI)
- Radar Altimeter

- Terrain Following Radar (TFR)
- Head-up Display (HUD)
- Attitude Director Indicator (ADI)
- Approach aids

HUD and ADI receive AFDS signals, CSAS receives/provides signals from/to AFDS; all the other systems can be considered as AFDS data sources. (Fig. 1).

The signals interchanged with the other equipments are of various types:

- serial digital
- analogue
- synchro
- discretes.

3. TESTING FACILITY

In accordance with the trend usually followed during the 1970s, the three companies (MBB, BAe, AIT) of Panavia Consortium during the development phase of the Tornado aircraft used avionic rigs with the capability to test individual equipments and all the integrated avionic system. Each company had specific tasks to be performed and, as a consequence, every rig was set up differently to obtain the required capabilities.
To cover some involvements in automatic flight control area, the AERITALIA rig system, that initially was just used as a Flight Back-up Rig, was improved with the capability to perform dynamic real time closed loop testing. The present AERITALIA Closed Loop System is based on three main items:

- avionic integration rig
- data handling computer
- simulation computer

The real control loop is essentially composed by:

- Autopilot and Flight Director System (AFDS)
- Command and Stability Augmentation System (CSAS)
- Sensors: Inertial Navigator (IN), Secondary Attitude Heading Reference (SAHR), Air Data Computer (ADC) and their controls
- Main Computer (MC) and its controls and displays
- Displays: Head-up Display (HUD), Attitude Director Indicator (ADI)
- Control panels, manual commands
- Aircraft and environmental conditions.

The behaviour of this real system is reproduced by fitting the avionic rig with the equipments (AFDS, sensors, displays, control panels) and closing the loop through software models of CSAS and aircraft dynamics. (Fig. 2).
The avionic equipments are connected as on the aircraft: their input/output signals are controlled by the data handling computer via special interfaces and through dedicated patch panels. The operators can use the various equipment control panels and keyboards as in normal aircraft operations. The manual aircraft control is performed by the use of a dedicated hardware/software system which is representative of stick, trim and throttle functions with the limitation that artificial q-feel at present is not implemented.

The two computers are connected physically via a standard high speed communication interface and logically by two jobs operating respectively on the data handling computer and on the simulation computer. The operational data flow in the system is fulfilled according to the following steps:

- the output signals of the avionic equipments are acquired by the data handling computer and sent to the simulation computer
- these data are used as input signals for the software models by the simulation computer which computes the avionic equipments inputs to be sent to the data handling computer
- these computation results are used by the data handling computer to stimulate the avionic equipments that consequently produce new data to be acquired.

All the process is monitorized not only, as obvious, by the various aircraft displays, but also by additional measurement devices and by using video display/line printer and plotter for alphanumeric and graphic representation respectively.
A complete evaluation of the test is done performing alphanumeric and graphic replay of the relevant signals recorded on the appropriate mass memory devices (disk/magnetic tape) during the test itself.

The data handling computer is a DEC PDP 11/45 with peripherals and terminals (like disk units, magnetic tape units, teletype, video display unit, paper tape reader/punch unit, plotter, line printer, card reader) and some standard and special purpose interfaces, able to handle the various kinds of signals used in Tornado avionic system, i.e.:

- serial digital signals
- parallel digital signals

- binary coded decimal signals
- synchro signals
- discrete signals
- analog signals

The simulation computer is a DEC PDP 11/60 with limited peripherals and terminals (disk units, magnetic tape unit, line printer, video display terminals).

The physical connection between the two computers is obtained using a standard DEC intercommunication interface. (DMC11).

The closed loop system makes use of special software on both the computers.
The data handling computer is provided with the so called Closed Loop Data Acquisition and Stimulation System (CLDASS) which was developed by AIT starting from the Data Acquisition and Simulation System (DASS) designed by MBB, with AIT contribution, for open loop testing purposes (hardware software integration, system investigations, flight back-up, support for avionic system changes).
The CLDASS allows real time testing and off-line replay.
The real time functions are (Fig. 3):

- Recording on magnetic tape and/or disk of:
  . data acquired from the avionic rig
  . data received from the simulation computer
- Monitoring on video display or line printer and/or plotter of:
  . data acquired from the avionic rig
  . data received from the simulation computer
- Dynamic stimulation with:
  . data stored on magnetic tape and/or disk
  . data received from the simulation computer
- Substitution of serial digital data with:
  . data stored on magnetic tape and/or disk
  . data received from the simulation computer
- Data transmission to the simulation computer
- Event processing and subsequent action
- On line commands
  (for control of test, monitoring and stimulation).

The off-line functions allow to replay the recorded data on:

- line printer or video display (alphanumeric replay)
- plotter (graphic replay)

and to make use of all the necessary utilities including system generation and avionic data bank management.
In addition to data transfer handling, the most important difference between DASS and CLDASS is that DASS basic functions are rig data acquisition and rig data stimulation, while in CLDASS monitoring, recording, stimulation and substitution apply not only to avionic rig data but also to simulation computer data. For the rest the two systems are philosophically similar even if CLDASS was completely rewritten. The following concepts apply:

- operation under DEC Disk Operating System (DOS)
- use of Fortran IV and Macro 11 Assembler languages
- use of modular programming techniques
- use of special purpose device handlers to meet the real-time testing requirements
- use of test oriented file format
- use of system priority structure
- software organization in line with rig testing activities
- availability of a test oriented language

The software implemented on the simulation computer includes some programs running under the DEC RSX11-M operating system.
The simulation computer software (Fig. 4) consists of three main systems:

- aerodynamic data handling programs
- tests preparation interactive program
- real time simulation program

whose functions are detailed in the following.

The three parts have been designed and developed with the purpose that the limited resources of the machine (speed and word length) should not affect the ability to perform real time testing and that, at the same time, the computer should be utilized at the maximum of its hardware capabilities. During the project, in fact, the strongest constraints were:

- the need that the simulation program running time is shorter than a prefixed one, in order to maintain the global cycle time at a value enabling a realistic operation
- the address limitation of a 16 bit computer just allowing 32 K words program when overlay techniques can not be used for time consuming reasons.

The "Aerodynamic data handling programs" system collects a set of programs allowing the aerodynamic data management with the capability of selecting a certain desired portion of the whole flight envelope. Its functions are:

- data selection in accordance with the test to be performed
- data generation in accordance with the aircraft configuration
- data interpolation for intermediate situations

The "Tests preparation interactive program" system provides all the data/information necessary to the test, acting on previously selected data and on the basis of a man-machine dialogue. Its functions are:

- selected data formatting
- trim conditions computation
- fixed parameters computation
- logic commands interpretation and formatting.

The "real time simulation program" is the most important software running on the simulation computer because it is used during the real time test.
It contains the software models of all the non avionic items like aircraft dynamics, engine, control/stability augmentation system, environmental conditions and, in addition, of some avionic sensors so that the use of either real or simulated sensors is allowed.
For the aircraft simulation a six degree of freedom non-linear model, expressed with forces and momenta equations methods, is used. The simulation program has been written mainly in DEC Fortran IV plus; only some particular routines and a special purpose DMC11 device handler are written in DEC Macro 11, as requested by real time optimization. The data in the program are obviously in floating point format; so every data exchange with PDP 11/45, where the information is in Tornado avionic format, is in conjunction with the appropriate conversion.

The program, which was developed using computer oriented real time techniques, allows the following:

- starting phase
- avionic data acquisition from data handling computer
- avionic data conversion into floating point format
- atmospheric conditions computation
- CSAS and actuators output computation
- aircraft dynamics equations integration
- airframe/engine parameters computation
- sensors outputs computation (simulated sensors)
- sensors stimuli computation (real sensors)
- failure/disturbances generation
- data conversion into avionic format
- data transmission to data handling computer

The facility is very flexible and simple to use (Fig. 5). The first thing to do when a particular test has been decided is of course to install the equipments on the rig including the proper software and mission data. If, for any reason, an important equipment is not available it will be properly simulated. After rig and connections set up, a list of all the involved parameters is to be prepared considering which signals are to be used as stimuli and which signals are to be monitored and/or recorded.
Then the following have to be defined:

- test operations sequence
- correspondence between avionic and CLDASS channels
- sampling and resolution time of every function (the sampling time is the period of the operation, the resolution time is the interval in which the operation must be completed).
- maximum foreseen running time (the test is automatically stopped).

After that, the physical connections can be made on the various patch panels and the appropriate CLDASS instructions can be written.
At this point it is possible to start the operation on the data handling computer with the test program instructions interpretation.
On the other computer the test preparation has to be performed following the various preliminary steps. The programs related to management of aerodynamic data are not to be runned every time as the same portion of flight envelope is used for several tests and large files archives can be arranged. On the contrary, the interactive program is normally runned before every test, this has not to be considered a limitation or a problem due to program execution rapidity and to possibility of organizing the resulting data files into appropriate archives.
According to the operator's answers in the dialogue the appropriate values are arranged in two files to be used as inputs and initialization by the real time simulation program.

The real time closed loop process, during its cyclical part, is governed by the programmable real time clock of data handling computer which beats the global cycle time. However the data handling computer in order to be enabled, after test program instructions interpretation, needs to receive from the simulation computer a first data message for equipment initialization.
The test actually starts after adequate PDP 11/45 keyboard command: the data flow between the two computers follows predefined transmission/reception sequences with continuous verifications of transfer operations status to always guarantee use of updated and coherent data. For instance on the PDP 11/60 the program checks if the reception of a new set of data is completed before using them in the next computing cycle.
During test execution it is possible to follow its proceeding both on the various avionic rig displays and PDP 11/45 monitoring devices and to interfere properly with the on line commands facility.
The real time test can be stopped by the appropriate PDP 11/45 keyboard command: the two jobs running on simulation and data handling computers finish correctly and every interface buffers are resetted. The same happens if the predefined end time is reached.
The data set collected during the test on magnetic tape or disk can be evaluated using the off-line replay program.

4. FLIGHT CONTROLS TESTING ACTIVITY

4.1. Tests typology

When the various hardware units are available the need for an integration rig and especially for a facility with closed loop capability becomes more and more evident. The Closed Loop System allows testing of all the system functions with very high level of flexibility and accuracy.
The availability of such a facility is useful in any phase of an aircraft project because it makes it possible to:

- display and test all the functions and logics against system specification
- detect incompatibilities or failures at any level
- study and verify corrective actions
- anticipate the system in-flight behaviour
- reproduce in-flight situations
- produce results for the flight clearances
- perform training activities for company and customer personnel

Rig testing is of course performed in parallel with the flight trials: the comparison between flight test results and real time simulation is a very important feature which makes essential the permanent Closed Loop System support during the various phases of development and in-service.

After all the acceptance test procedures of the various avionic units have been accomplished and all the required integration tests have been finished it is possible to start the real time simulation activities.
The first one is of course the validation of the facility itself: this is carried out in separate steps by comparing the obtained outputs with the various off-line simulation results, the development flight simulator data and, as final step, the flight trials traces.

The Closed Loop System gives the opportunity to perform a software confidence testing on the program implemented in the automatic flight control system computers.
Appropriate test procedures must be prepared to verify the correct implementation of control laws and of mode compatibility/failure logics. All the mode combinations are tested and the rig results are compared against the expected one. The various limits, thresholds, switches and selectors are checked to verify if the corresponding actions are in the desired direction.

The mode and failure logic is proven by simulation of mishandling actions and various types of external malfunctions including lack of the sensors and also of the power supplies.

In case the system is extended to include also the primary flight control system in hardware, even the integration between automatic and stability systems could be checked: all the possible aspects to be verified before flights, including interface malfunctions and possible hardover failures, can be covered with a high level of confidence.

The performance analysis tests are to be intended as a validation activity more than a verification one. The investigation on automatic flight control system is extended by exploration of a very large number of flight conditions covering as much as possible of the complete flight envelope including several aircraft configurations.
The performance verification, or software validation, has the purpose to control whether the specification is met, i.e. whether the software is working as expected. The tests must cover all the autopilot operating modes in all the possible combinations.
The presence of the other avionic equipments gives the possibility to perform the tests in the same context of the real mission. The crew can participate to the tests acting as during real flights.

The performance tests have the following purposes:

- predict the in-flight behaviour giving more information to the test pilots and to the system people
- investigate the flight envelope portions not explored during prototypes flight trials
- evaluate the obtained performances against the specification requirements
- study possible software modifications in case the performances are not considered satisfactory
- clear the various modes and their combinations
- provide, in conjunction with flight trials activity, clearance recommendations and flight envelopes definition with respect to specification requirements.

The rig testing objective is not only to prove the performance and the hardware and software integrity under normal operating conditions, but also in abnormal situations, in fact due to the extreme flexibility and easy use of the Closed Loop System, a large variety of tests can be carried out in failure analysis area.
The main test objective is to check the failures detection by AFDS and the aircraft recovery analyzing their effects on the subsystem functions and on the aircraft motion.
The failures to be investigated can be summarized as follows:

- lanes failure including open and short circuit of interfaces
- interface failures including hardovers, specific variations and power fails
- component failures

For the open circuit failures it is required to investigate the effects of interruption of data flow on digital channels or cut of analog lanes in connection with various equipments which may dialogue with AFDS.
Similar investigations are requested for short circuit failures.
The hardover failures examination leads to define exactly the authority limits and the flight envelope that can be allowed in all operating modes in any situation. (As previously mentioned this activity can be really performed only when it is possible to integrate also the primary flight control system).
In the category of the so called "specific variation failures" the following types of signals failures are considered:

- failure to zero
- step failure
- failure to last value
- ramp failure

These failures are to be applied to every significant parameters involved in the control system in every operating mode and situation.
Power fails investigation includes power change over and power interrupt of the aircraft power generators.
The in box failure simulation represents the deepest investigation as far as the failures are concerned. The related tests are to be defined on the basis of all the safety studies and of all the results of the other failure tests.
Moreover failure analysis activities comprise investigations on sensors data tolerances and mishandling operations.
The tests on sensor data tolerances are performed to check the influence of various signals tolerances upon aircraft performances and system protection. The tests control the correct implementation and the adequate definition of the various monitor thresholds of the system.
Mishandling tests are performed to check influence of deviation from normal operational procedures on aircraft motion and system functions.
Finally the last thing to be carried out is the investigation of AFDS disengage characteristics in normal operation, emergency and failure conditions.

A collateral, but not less important, activity is represented by the continuous support to the flight trials. This support extends from answering the test pilots questions to reproducing strange in-flight occurrences.
With the Closed Loop System is in fact possible to repeat flight manoeuvres, therefore this allows deeper investigations on any events occurred during real flights.
The read across between in-flight and real time simulation results at last leads to a very high level of confidence in making the necessary assessment on safety and performances.
All the above mentioned rig activities together with the flight tests allow to give the various in service clearances.

The Closed Loop System is also important for training activities: it is a very efficient tool for crew familiarization with the various mission operations and procedures and it is also very useful to train technicians and engineers.

4.2. AERITALIA involvement

The Aeritalia Closed Loop System was developed in 1977-79 as a back-up of their own AFDS flight trials and to support on opportunity basis the MBB rigs. Since 1980 Aeritalia have been giving contribution to the various clearances for in-service activities especially in the following areas:

- performance prediction and evaluation
- interface failures effect testing
- read across between real and simulated flights behaviour
- crew and engineering training

The facility was validated using all the kind of results available within Tornado program: flight simulator, off-line models, actual flight trials data. (Fig. 6 a + c show a comparison between in-flight and rig time histories).

Performing the above mentioned activities the various hardware and software capabilities of the Closed Loop System are fully exercised.
For instance very large use of "substitution" is made as this function allows to replace a direct link between two equipments by a connection through computers and thus to substitute the desired signals by the simulated ones before the stimulation. (An example is given in Fig. 7 a+b).
For failures simulation few special words were arranged: each bit can be modified by use of appropriate on line command in order to inject various failure conditions. In addition for open circuit failures relays actioned by computers are also used. All this to allow failure injection exactly when requested by the operators and to record the failure command together with the other parameters.

The performance rig testing activity is devoted to the so called cruise modes in all their possible combinations: attitude/heading hold mode, barometric altitude hold mode, mach hold mode, heading acquisition mode, track acquisition mode and, in addition, autothrottle mode. (Some typical parameters are shown in Fig. 8 a+d).
The mode performances have been evaluated not only against the specification requirements but also considering the system behaviour taking into account the operating missions requirements in all their complexity. This is carried out examining in addition to the typical autopilot and aircraft dynamics parameters, also some other ones not considered by the AFDS specification (because without influence on the autopilot itself), but having a certain importance with respect to the evaluation of the system behaviour. For instance the rapidity of certain manoeuvres is to be evaluated: time to reach foreseen values during manoeuvres (e.g.: maximum bank angle in heading acquisition mode) and when the manoeuvre purpose has been accomplished. (e.g.: zero bank angle after the desired heading has been acquired in heading acquisition mode). All what above to assess the suitability to the mission success of the various AFDS modes.

The facility has been continuously used as a valid support to the flight trials: every phoenomenum encountered in flight was examined reproducing the same conditions to discover its cause. The investigations are often let easier to be made by the possibility of obtaining dynamically transfer functions between two internal points of the system provided that they are accessible.
The manual controls implemented on the Aeritalia rig system allow to drive the aircraft and so to test also flight performances.

In conclusion the Aeritalia Closed Loop System is able to perform all the possible AFDS cruise mode testing activities with the following limitations:

- the address computer limitation does not allow to change aircraft configuration during a rig trial (it is necessary to stop the test and re-start with a new data package)
- the lack of stick forces simulation does not allow to test realistically ASO ed SFCO facilities for which at present just the electrical signals are simulated.

5. FURTHER AVIONICS RIG TESTING ACTIVITIES

The Aeritalia Closed Loop System, which was developed mainly for activities related to Autopilot and Flight Director System, allows to perform several testing activities in other avionic areas; it is in fact, an efficient tool for the verification and the validation of every kind of avionic software. Therefore studies and tests have been carried out on the navigation/attack areas with particular attention to air-to-air and air-to-ground attack procedures.
In this kind of tests the bench is fitted with the necessary equipments, the Main Computer is loaded with the Operational Flight Program under consideration and with the appropriate mission data.
The loop includes not only the sensors/displays used during AFDS testing but also all the equipments involved in the nav/attack mission with their own displays, control panels, etc. The AFDS itself is not so important, in fact it is not used at all in air-to-air attacks while can be used in air-to-ground attacks.
On the contrary, in these nav/attack simulations it is essential to drive the aircraft manually using pilot stick and throttles.
Similarly to AFDS testing activities also for the nav/attack system the following have to be carried out:

- software confidence testing
- performance prediction and evaluation
- failure assessment on failure effects
- flight trials support.

Moreover also software maintenance activities have to be considered.

Among the activities recently carried out in Aeritalia the one related to air-to-air attacks can be summarized as an example. Air-to-air testing requires installation on the bench (or simulation on PDP 11/60) of radar sensor, Store Management System in addition to the equipments used for autopilot activities while the presence of AFDS is not relevant. The real time program to be used for this testing contains also the target motion simulation. A symbol, representative of target position with respect to the aircraft, is visualized on Head-Up Display; the type of movement of this synthetic target is chosen by appropriate answers in the dialogue of the interactive preparation program.

During software development or for software maintenance purposes it is moreover useful to have accurate simulation model of the software to be implemented within the on-board computer. The models can be inserted in the real time program or better in an appropriate off-line program for their evaluation.
The software model validation is carried out using off-line procedures.
After this validation the software is implemented in the MC: the real time testing must then be performed and successively the results evaluation is made using the same off-line procedures.
If unsatisfactory conclusions are obtained, to discover the MC software faults, the software model can be stimulated by the same input as used for the actually implemented software so that a comparison between the two sets of output is allowed.

As previously mentioned the air-to-air attack manoeuvres are to be performed manually. Since test pilots are not always available for this kind of activity and being quite difficult for engineers to repeat the various manoeuvres in the same way, special pilot software models were also added in the real time simulation program.
The pilot models were prepared taking into account literature examples and traces obtained in various manned simulation trials.
The pilot model acts in answer to HUD information and assures repetitive and exact attack manoeuvres. So it is possible to completely validate the air-to-air attack software both for not manoeuvring and for manoeuvring targets.

6. CONCLUSIONS

The Aeritalia Closed Loop System is a facility that is continuously updated and upgraded both hardware and software speaking.
The goal is to gain the capability to simulate every kind of mission of the Tornado aircraft.
The upgrading activity is accomplished by the addition of new functions or by the improvement of the existing ones normally following operator's requirements or system changes.
In next future the facility will be used for:

- AFDS cruise mode performance evaluation to produce necessary documentation for complete in-service clearance
- AFDS auto-approach mode performance evaluation in normal condition and under failures
- Air-to-ground attack procedure investigations
- Failure analysis.

In addition the facility will be prepared to be able to perform software maintenance activities.
Moreover due to the experience gained on this kind of facilities in the Tornado program and due also to the high quality of the results, it is Aeritalia intention to continue to follow this approach in future for avionic system development. In particular for the ongoing project of the new close air support aircraft, the AERITALIA-AERMACCHI-EMBRAER AM-X it was decided to design a rig with at least the capabilities of Tornado one. The AM-X rig will take profit of all the improvements in computer technology and in operating systems design.

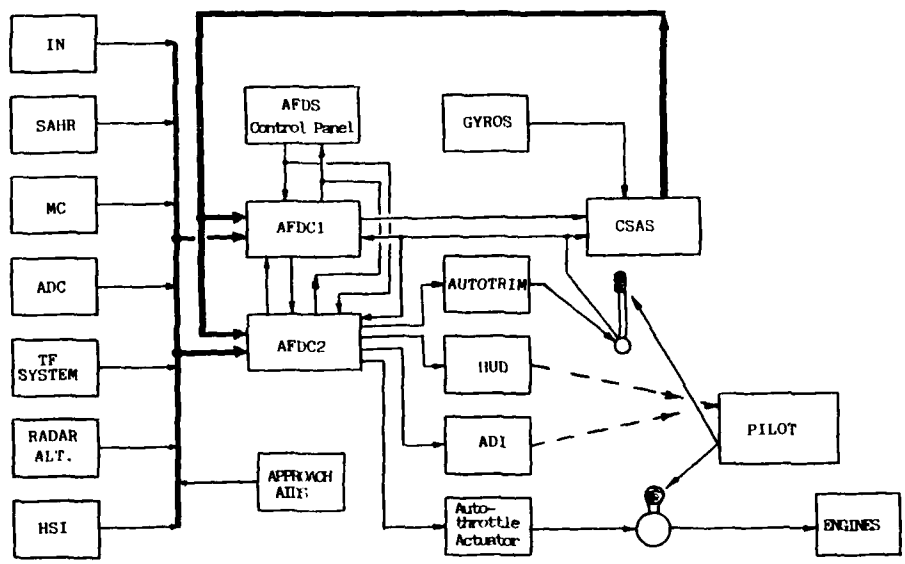


FIG. 1 - AUTOMATIC FLIGHT CONTROL LOOP BLOCK DIAGRAM

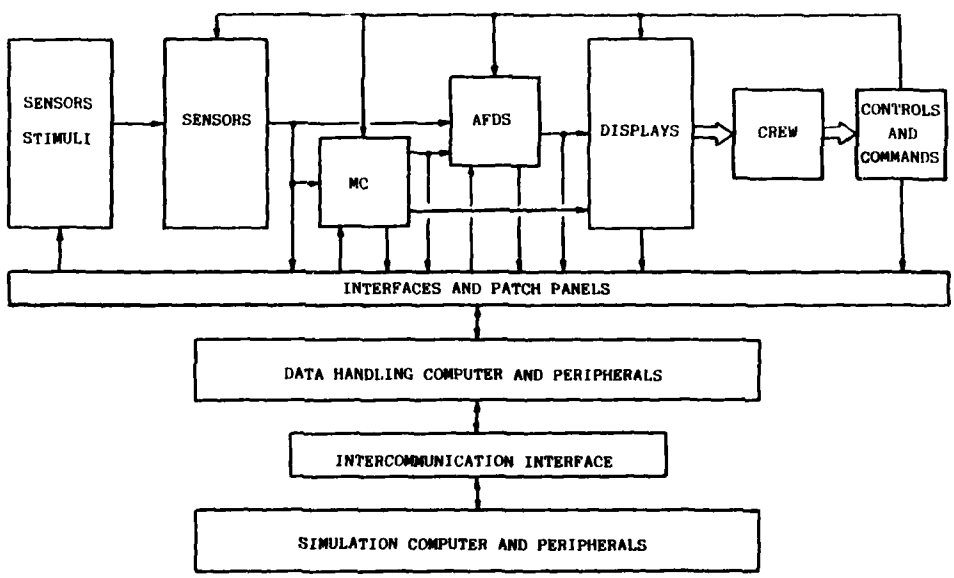


FIG. 2 - CLOSED LOOP SYSTEM SET UP

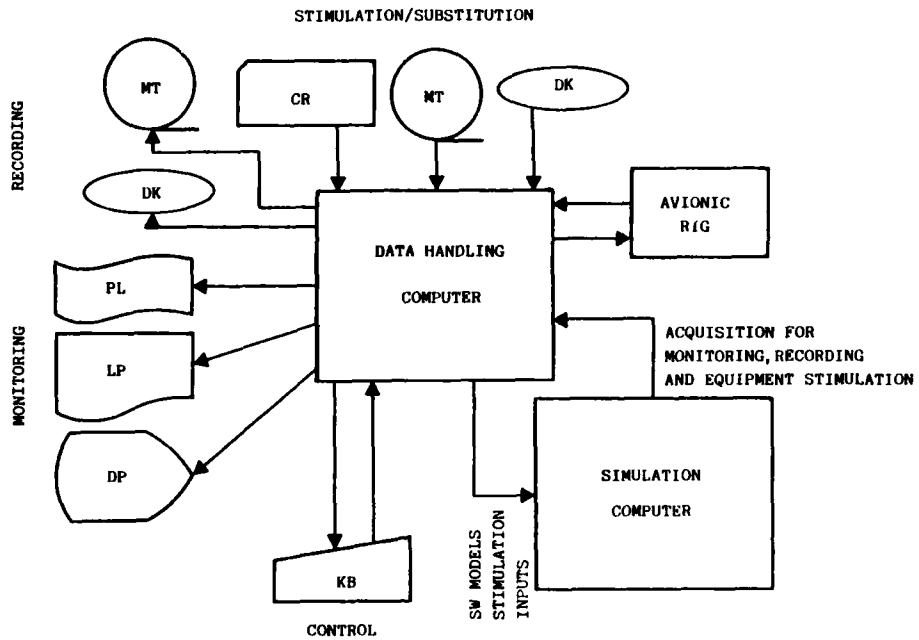


FIG. 3 - CLDASS REAL TIME FUNCTIONS

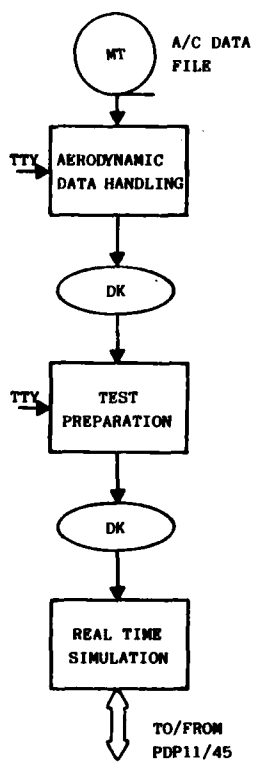


FIG. 4 - SIMULATION SYSTEM

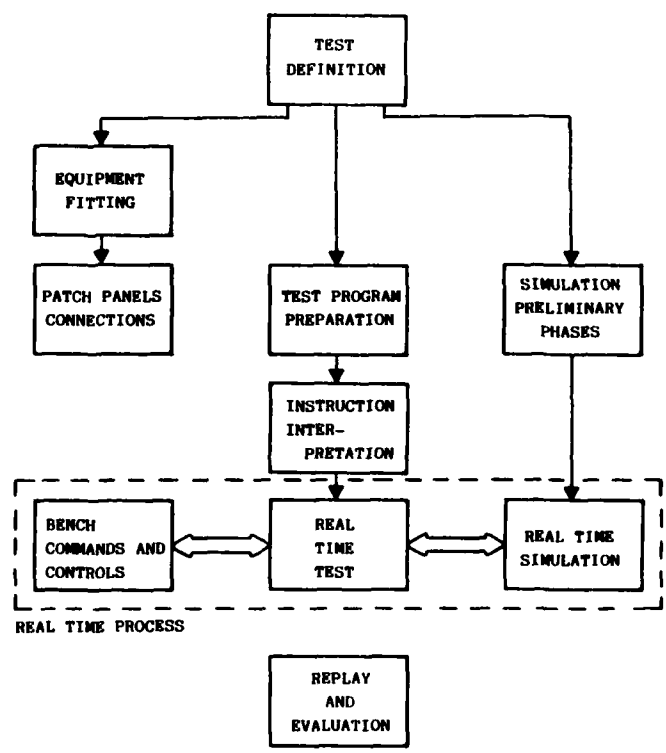
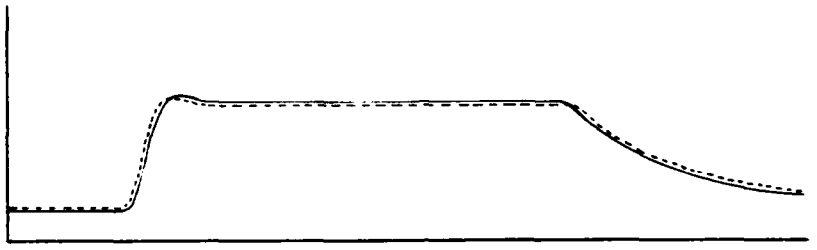
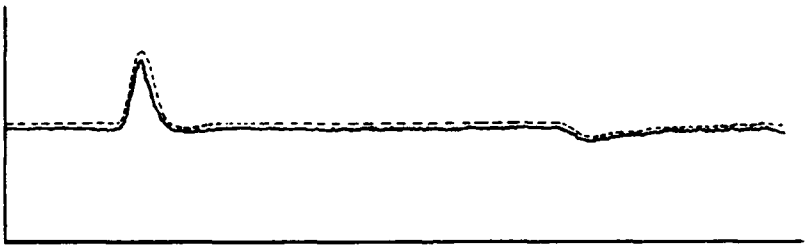


FIG. 5 - CLOSED LOOP TESTING: OPERATIONS SEQUENCE

a - PITCH ANGLE

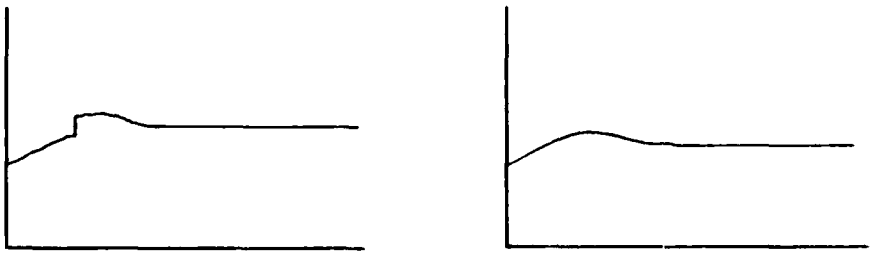


b - BANK ANGLE



c. ROLL RATE

FIG. 6 - COMPARISON BETWEEN RIG AND IN-FLIGHT RESULTS



a - SIMULATED PARAMETER

b - TRUE PARAMETER

FIG. 7 - INTERFACE FAILURES: SUBSTITUTION EXAMPLE

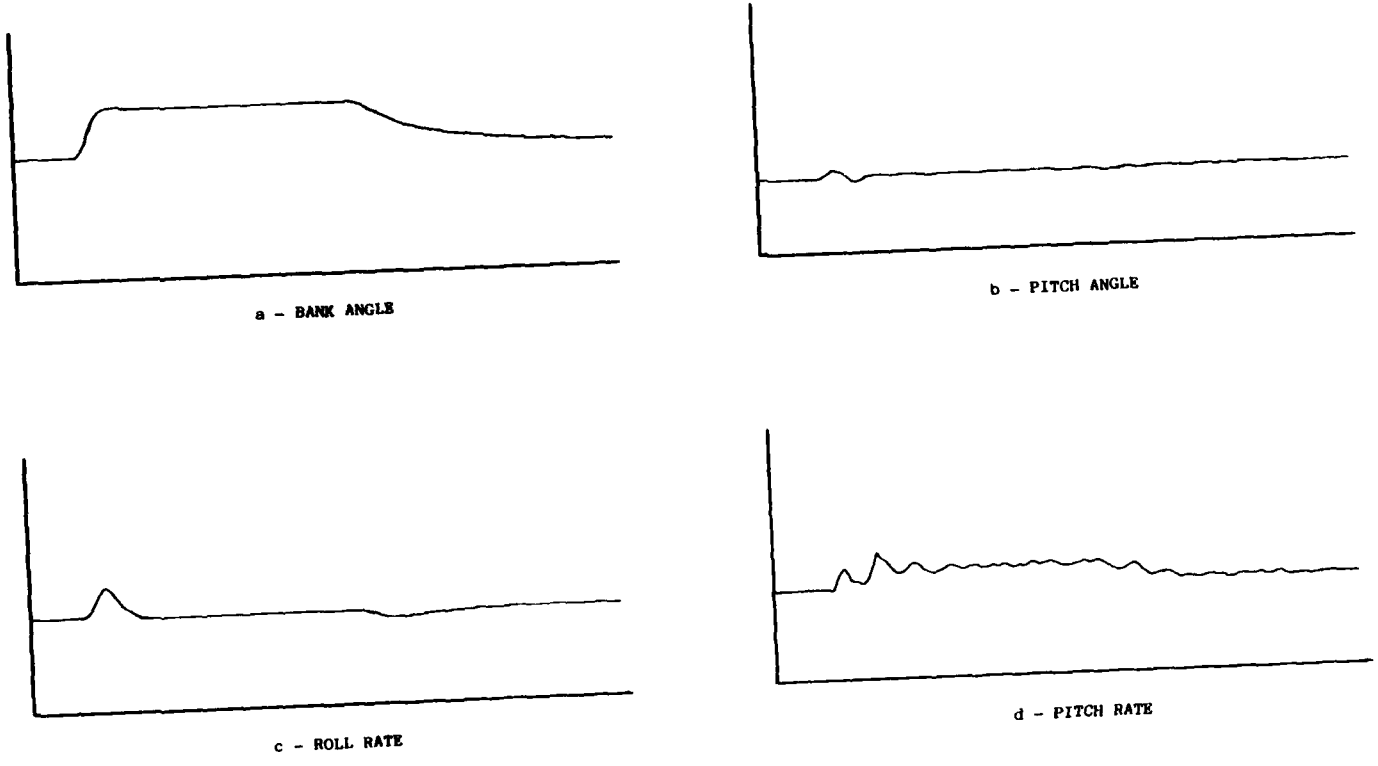


FIG. 8 - PERFORMANCE EVALUATION: TYPICAL TRACES

APPLICATIONS OF NETWORK MODELING AND ANALYSIS TO SYSTEM VALIDATION AND VERIFICATION

Gary M. Sundberg
Tracor, Inc.
65 West street Road
Warminster, Pennsylvania 18974

SUMMARY

Historically, Software V&V efforts have been undergoing an evolution. Initially V&V meant matching coding to specifications, and tracing specifications to an analytical basis. There were no well-defined techniques or tools for accomplishing these tasks. Most V&V efforts took the form of either physical or empirical testing. Gradually these evolved some standardized flow charting techniques and automated aids for code tracing and simulation. Then there came a number of relatively sophisticated automated tools which could be used to evaluate test provedures as well as test the subject program. Recently a system approach, applying generic system analysis techniques to the software problem, has been used. Network Logic Modeling and Analysis is such a system tool.

Network Logic Modeling and Analysis (NLMA) is a manual analytic process derived from network analysis and Boolean logic. Its main purpose is to verify and validate complex systems or concepts at any stage of development. The NLMA technique has been used as a V&V tool by several (6) Navy projects, thus establishing a record on which to base its effectiveness as a V&V tool and to provide specific examples of applications.

The purpose of this paper is to:

1. Emphasize the importance of applying V&V techniques early in any development effort and continue their use throughout the project life cycle.

2. Describe Network Logic Modeling and Analysis.

3. Provide specific examples from a project which used NLMA showing applications at each phase of the life cycle and types of discrepancies detected.

As more and more new systems are developed based on computer and software technology the more apparent it becomes that applying techniques and methods used in the acquisition of hardware based systems are not always adequate for the acquisition of software systems. One of the biggest areas which require change is in the concept of V&V. Because the quality of the hardware system was based on industry or government standards for such easily defined characteristicsas strength, power, dimensions, etc., V&V usually took the form of physical tests conducted near the end of the development cycle. However, such easily defined characteristics do not exist for software; in fact what constitutes quality in software is a subject that could fill volumes. It is the intent of this paper to show how to achieve quality rather than to provide a precise definition. With this goal in mind the first item to emphasize is that the old idea of physical tests as the only method to determine quality must be abandoned when applied to software.

To help understand why, we should first define the types of software errors. Simply stated, there are two types of software errors: performance and logic. The first are performance errors. These errors are commonly referred to as "bugs" and are usually program errors that are easy to detect because the program either stops running or produces obvious mistakes. The second type of error is logic errors. These errors fail independent of space and time. Logic errors can be made during the definition state of a program by incorrectly stating or omitting requirements; in the design stage by not satisfying the requirement in the definition specification or by incorrectly representing the design; and also in the construction stage by incorrect implementation of the design. Because logic errors are usually discovered late they become very expensive to fix, running as high as 50% of the total program cost. Performance errors are usually the last to be made (during coding) and the first to be discovered (during initial testing). Even though this type of error constitutes nearly 80% of the errors in any program they usually require only 20% of the funds spent to correct the programs. Logic errors are usually the first to be made and the last to be discovered.

It should be apparent from this discussion that the early detection of logic errors can have a tremendous pay off in both quality and cost. Therefore, the idea of testing must be expanded to include all phases of the developmental Life Cycle. The "testing" of software before it is coded and compliled is actually a test for reliability and a check for testability. The "tests" are evaluations of the product at each phase for correctness, completeness, consistency, i.e., reliability. Such testing ensures a firm product and a consistent program structure at each phase of the development. Network Logic Modeling and Analysis was developed to provide a testing tool and structured approach to perform this testing.

The foundation of Network Logic is built upon principles of heirarchy and the dependencies existing among all elements contained within a hierarchy. Among dependencies and elements there exists ordering, derivation and interplays. Elements, when combined, form a scheme for a system or concept whose dynamic behavior may then be challenged or measured against any arbitrary criterion of choice, such as performance, reliability or effectiveness.

Every major system acquisition begins with a concept that originates to serve an objective or goal. The goal generates requirements or needs that are to be satisfied in order to achieve the intended goal. Requirements give rise to functions or tasks that must be performed, which in turn evolve into subfunctions. As all functions and subfunctions are evoked, decisions are needed to be made to determine how the concept or system is to be mechanized with combinations of hardware, software, and human elements so that functions (tasks) can be performed. With all the requirements satisfied then the objective or goal can be achieved.

In essense, Network Logic Modeling Analysis is an ordered process of deductive reasoning that organizes all appropriate elements of a system or concept into their respective places and locations within the hierarchy. The hierarchy representing the system can then be verified analytically to determine if it is capable of achieving the goal. The network models the hierarchy and determines how the concept or system is organized, and the logic determines the exact nature of the interdependency existing among all elements. Recursive and iterative features as well as interfacing facets are developed. From these, the network identifies the correlation and cross-correlation of functions and subfunctions and clarifies how they are mechanized. In doing so, visibility and insight are obtained into the organization of the system interfaces, interconnections and common functions. After a system is so defined, the analysis can be used for a number of applications. From a critical viewpoint of management control, determinations can be made of system adequacy.

Before continuing, a brief description of a system life cycle's phases and program management should be included. The generic life cycle for computer programs consists of the following phases:

- Conceptual - System operational Identifier (Requirements)
- Definition - Program Performance Specification (PPS) (Design)
- Development - Program Design Specification (PDS) (Coding)
- Integration - Testing
- Operational - Life Cycle support

The management and control responsibility for these phases is greatly facilitated through the establishment of baselines. These baselines serve as technical references from which the individual elements become operational functions and form the basic system for which configuration control is established. The four major configuration baselines are the following:

- Functional
- Allocated
- Product
- Operational

The Functional Baseline is established through definitions and descriptions contained in a high level document such as a Prime Item Development Specification.

The Allocated Baseline is defined by the IDS, CPPS, PDS and DBD. This baseline should be confirmed by audit prior to certification.

The Product Baseline identification of the program is described by the approved Test Plans and Procedures, Operator Manual/System Operator Manuals, Command and Staff Manual, Tapes, Decks, and Listings. The Product Baseline provides the necessary information for procurement, integration, and acceptance of the program for subsequent versions.

Due to the shift of responsibility, funding and accounting for software in Combat Systems when the production phase ends and the software program becomes operational, the Operational Baseline is established.

Network Logic Modeling Analysis provides a visual representation of the Functional, Allocated, and Product Baselines and becomes a baseline document itself. Each function and subfunction is rigorously defined, thus enabling an accurate identification of function design and the interrelationship of hardware, software, and operational elements. During the NLMA development, each function will undergo a comprehensive verification and validation process with all discrepancies to the program and/or its supporting documentation identified.

With the model developed the NLMA will provide a graphic representation of the Product Baseline. This baseline is a valuable configuration management tool enabling the program manager to maintain control over the implementation of changes to the Operational program and evaluate the program trouble reports written against it.

Because Network Logic Modeling Analysis was developed a system tool emphasis was placed on keeping the language simple so that it could be understood by systems personnel and individuals other than computer specialists. The symbols used were also simplified to avoid confusion or a lengthy learning period.

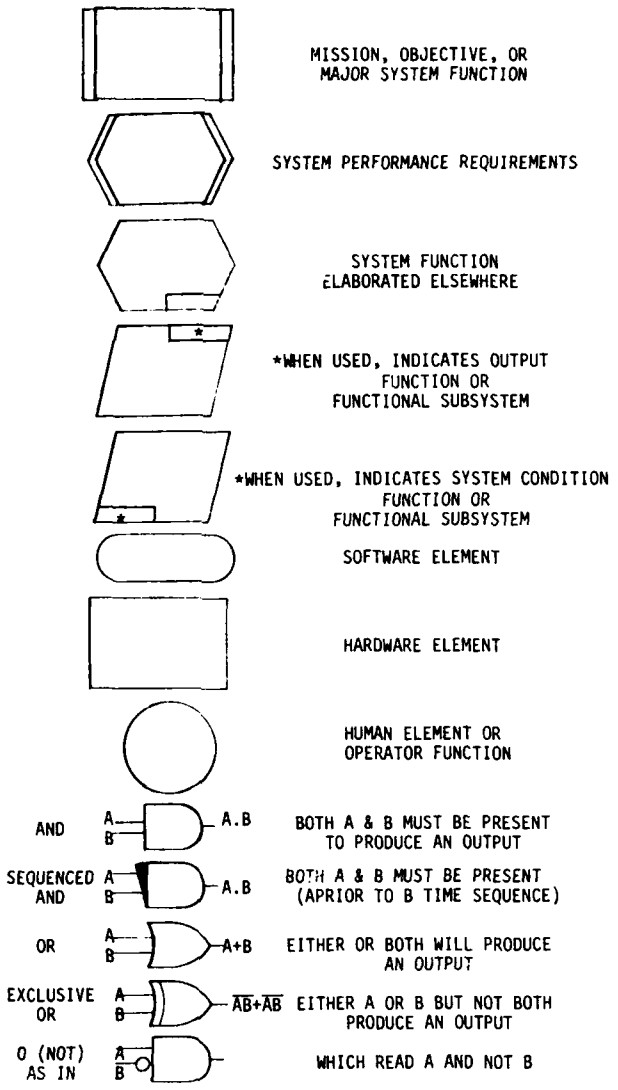
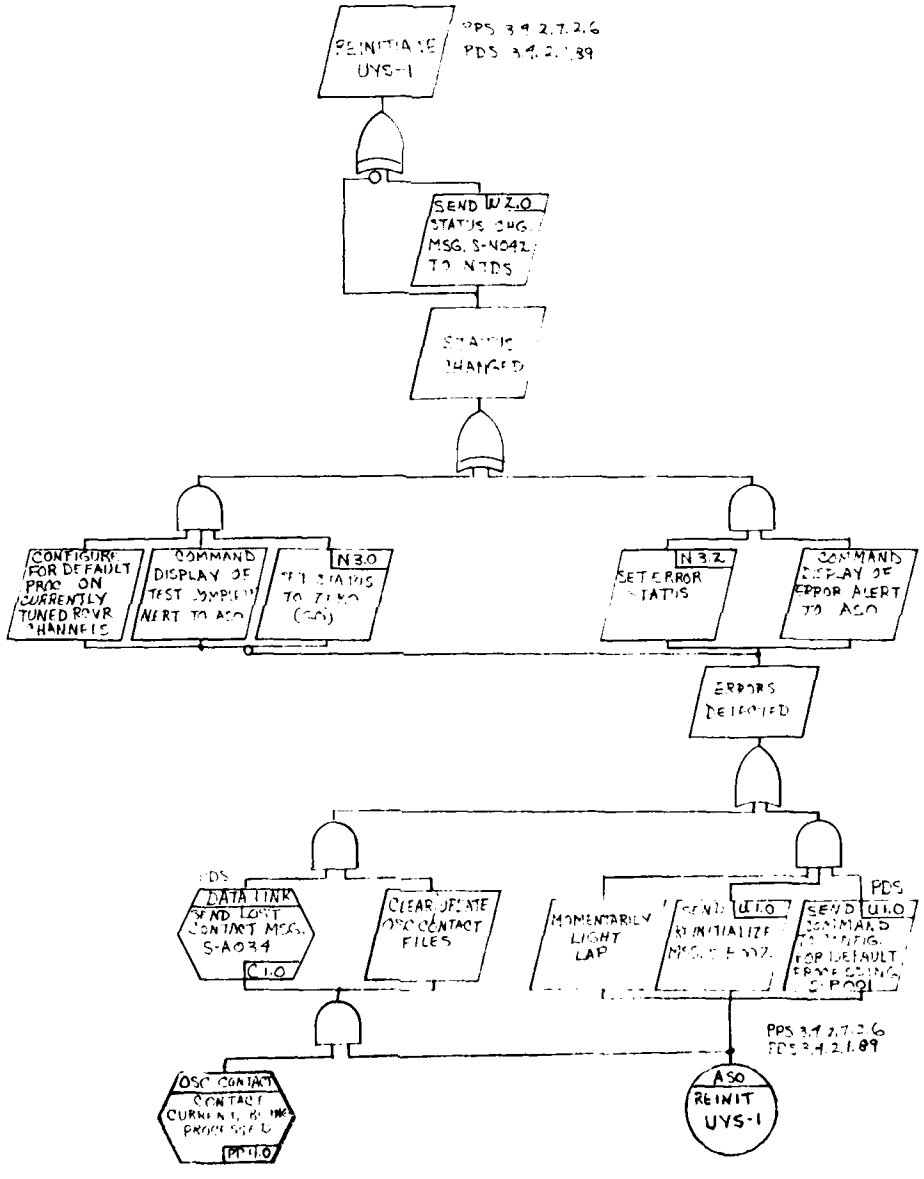


Figure 1. - Network Logic Symbology

The Network Logic symbology utilized is shown in Figure 1. The interpretation of logic gates requires strict adherence to rules of logic. These rules determine the exact nature of inter-dependency existing among system elements. By following those rules, rigor is achieved during the analysis. While an interpretive process governs the initial selection of appropriate logic, subsequent tests for validity determine whether the logic chosen correctly represents the behavior of system elements within the hierarchy.

Figure 2. is an example of a completed subfunction and is presented to illustrate how an NLM chart is "read". The subfunction is Re-Initiate UYS-1 (Acoustic Processor). The function is described and modeled based on PPS Reference 3.4.2.7.2.6 and has been verified to the PDS level, references 3.4.2.1.89. Starting at the bottom, the processing is initiated by the ASO operator selecting the REINIT UYS-1 switch. This action will cause the LAP (a light indicating active or activated switches) to light momentarily, an interf ce messages to reinitialize the processor is sent (S-P 002) and a command to configure for default processing is sent (S-P 001). The processing of these command (outputs) are used as inputs on chart UI 0. Additionally, if contact data is currently being processed (described on chart PP 4.0) a lost contact message (S-A 034) is sent and the OSC contact files are cleared. The processing to up-link this message is described in C 1.0. The PDS beside two of the symbols indicate that this processing was specified in the PDS only and not by the PPS. They are included because the represent important information. The next step in the processing was to determine if the REINIT results in detection of errors. There are two mutually exclusive paths. If errors are detected an error status is set (output to N 3.2) and an alert is displayed to the operator. If an error is not detected the system is

configured for default processing, a test complete alert is displayed and the UYS-1 status is set to go. Regardless of which path was taken the next step is to determine if the UYS-1 status has changed. If it has it is indicated (output) in an S-N 042 message and sent to the NTDS; if not, no further processing is performed.

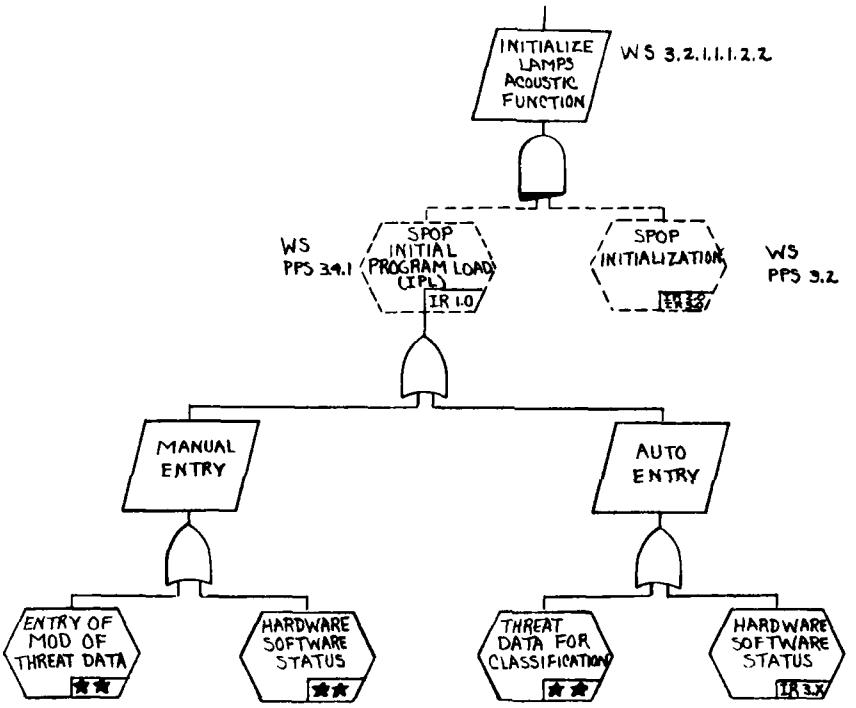


By introducing rigor into the analysis of complex systems, Network Logic contributes to the effectiveness and decisive control of weapon system software development. This extremely sensitive area has, until recently, evaded rigorous control partially due to the undeveloped state of software design and control methodology and partially to lack of understanding of software acquisition teams on the peculiarities of software generation. With these shortcomings, software proliferation has been permitted to evade rigor and to provide results that are generally unpredictable undisciplined and unstable. To relieve this problem, Network Logic forces software to assume its proper and appropriate place within the system heirarchy. It is treated as an essential constituent satisfying functional requirements. Its role in functional interplays is exposed to scrutiny and is compelled to undergo tests for validity and adequacy. Its structure within the hierarchy is defined, organized, and focused in perspective with all other elements. Because of its logical orientation and emphasis of software requirements and can be used to control the acquisition and generation of software.

As stated previously NLMA was designed for application at each phase of the life cycle. The following describes a few of the specific applications and examples to highlight how requirements and design problems were identified and the model used.

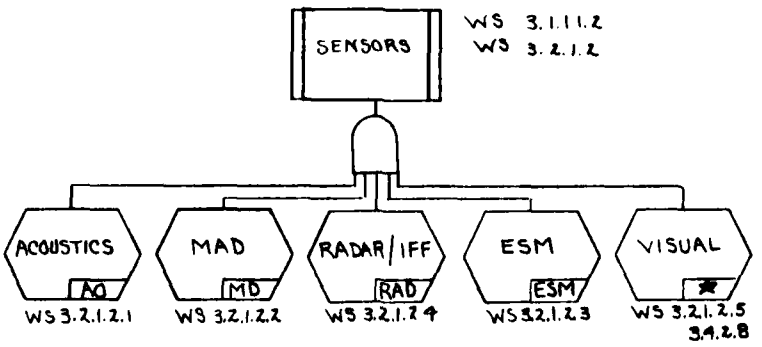
1. CONCEPTUAL

During the conceptual phase the system top level document is modeled and its requirements evaluated. The following analysis methods are used at this state. the first is visual inspection for any obvious ommissions. The second is an examination to testability, because vague or untestable requirements will leave validity of the product in doubt. The Third and most useful at this level is to "run" a scenario of intended use, using the model to verify that all elements required to achieve system objectives are provided for. The example below is from the Weapon System Specification for the LAMPS MARK III System. One of the first tasks required by the system's scenario was to load and initialize all of the subsystems.



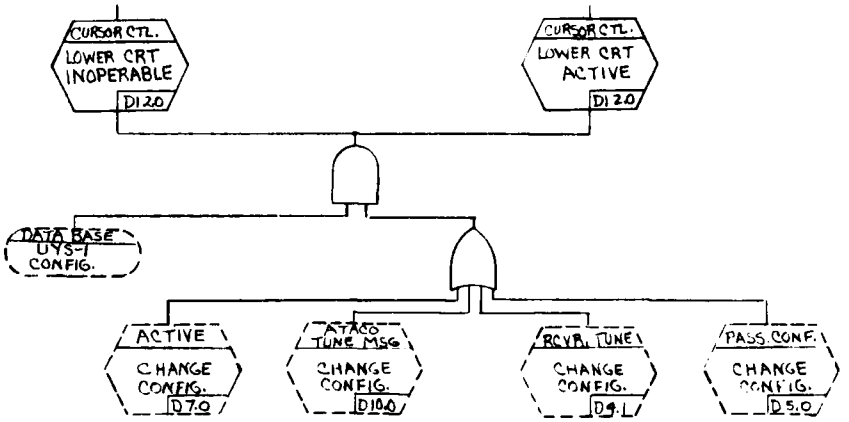
This exapmple presents two problems. The dashed symbol and lack of a WS reference indicates that a program load and initialization requirement was not contained in the Weapon System Specification. The PPS references are added to the support including those on the chart. Detailed descriptions of these functions can be found on Chart Numbers IR 1.0, 2.0 and 3.0. The second and more serious problem is indicated by the double stars located on the charts in the reference blocks. This indicates that these requirements are not met by the next level of documentation (PPS). This represents Top Level Requirements that will not be performed by the system and would be considered a major discrepancy.

The next example also shows a Top Level Requirement which also has a symbol without any reference to a lower level of processing. Upon first observation it seems reasonable that a visual sensor would not be found in the software specification. However, during analysis of the system and incoming interface message was found to have a word to indicate a visual contact. The contact tableau, which is where all the other information in the message was displayed, did not provide for processing this word. This was also considered a major discrepancy.



2. DEFINITION

During the definition phase the Computer Program Performance Specifications (CPPS) is modeled. The model is then analyzed for compatibility, traceability, mechanization, completeness, and evaluation of functions. In the example below the four dashed symbols connected by the "OR" gate represent different subfunctions which all specified stimulating the processing shown on this chart. The CPPS paragraph which describes the remainder of the processing shown on the chart did not list these functions as input or describe them as stimuli. All dashed symbols represent functions or processing that was not specified by the documentation but is included as a result of the analysis. All dashes should be accompanied by references to support their inclusion (not always the same document).



In the next example, the CPPS stated that the processing shown was stimulated by the receipt of a test message return. This message is actually the last in a chain of interface messages as described in the next level of documentation (PDS). The dashed symbol to set the status to go in data base was also referenced in the PDS. All the dashed blocks are shown to represent the complete function and aid in program visability. This would be considered a minor documentation error. (See example below.)

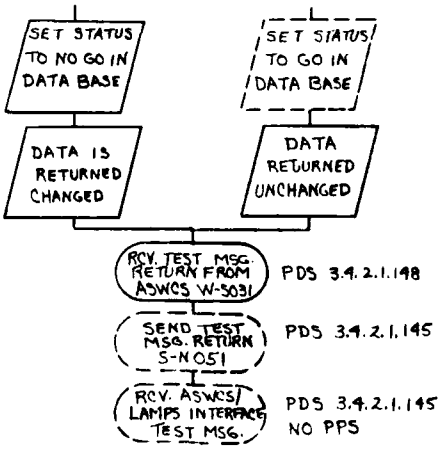


Figure 2. models a complete subfunction that resulted in the following observations. The first is that the abnormal processing (dashed symbols) is not specified by the CPPS. The lack of references for these dashed symbols means that these are educated guesses on the part of the analyst but cannot be supported by other documentation. Second, is the dashed symbol shown with a reference number which had to be added because it is part of the processing, supported by the reference, and was an important element for this function. Third, the reference which has been circled was not specified in the CPPS. This was added later to aid the chart user. Last, the function was evaluated. This function is basically an emergency operation; as such, subjecting the processing to the 40 track limitations was evaluated as poor definition and should instead be put on a priority basis.

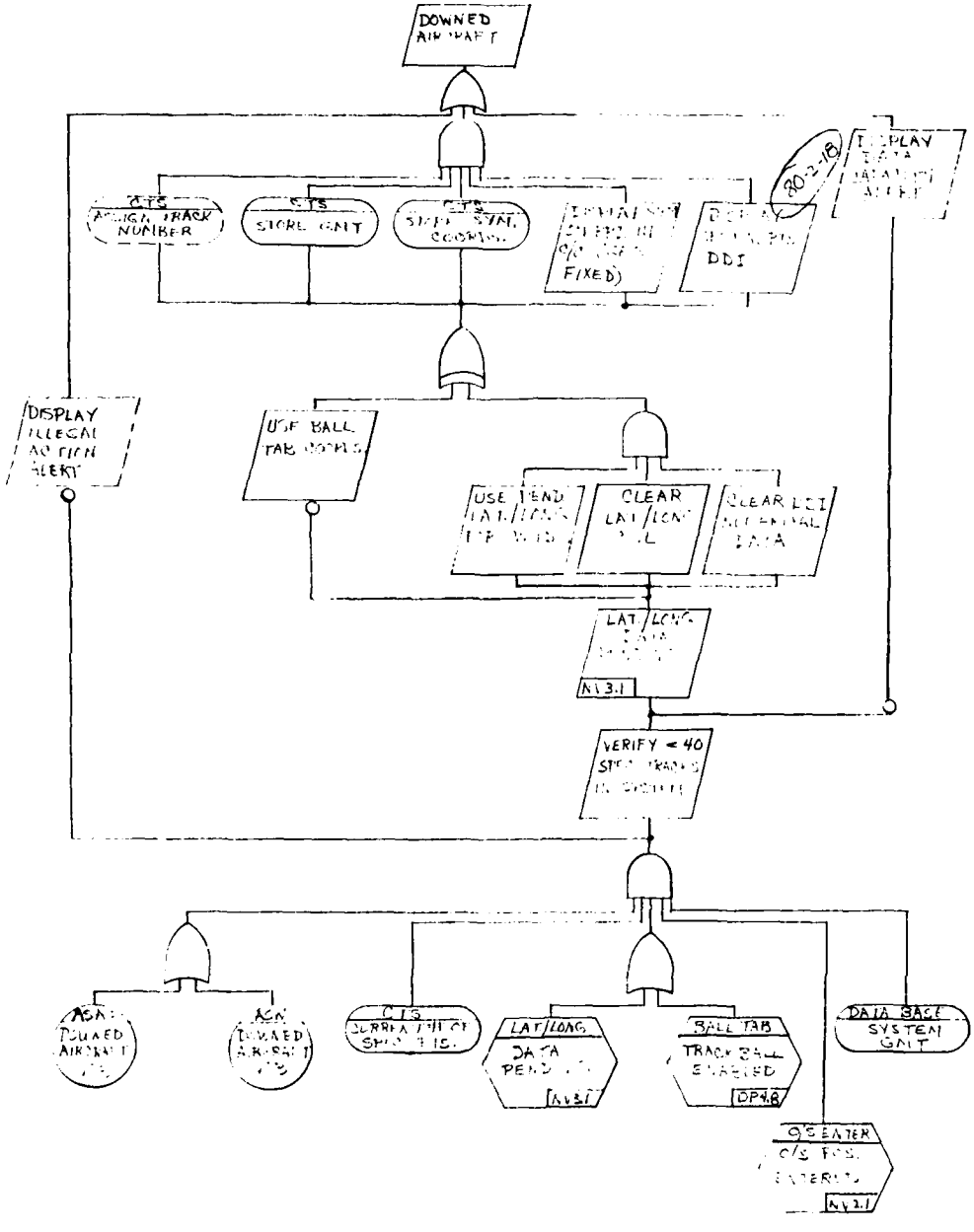
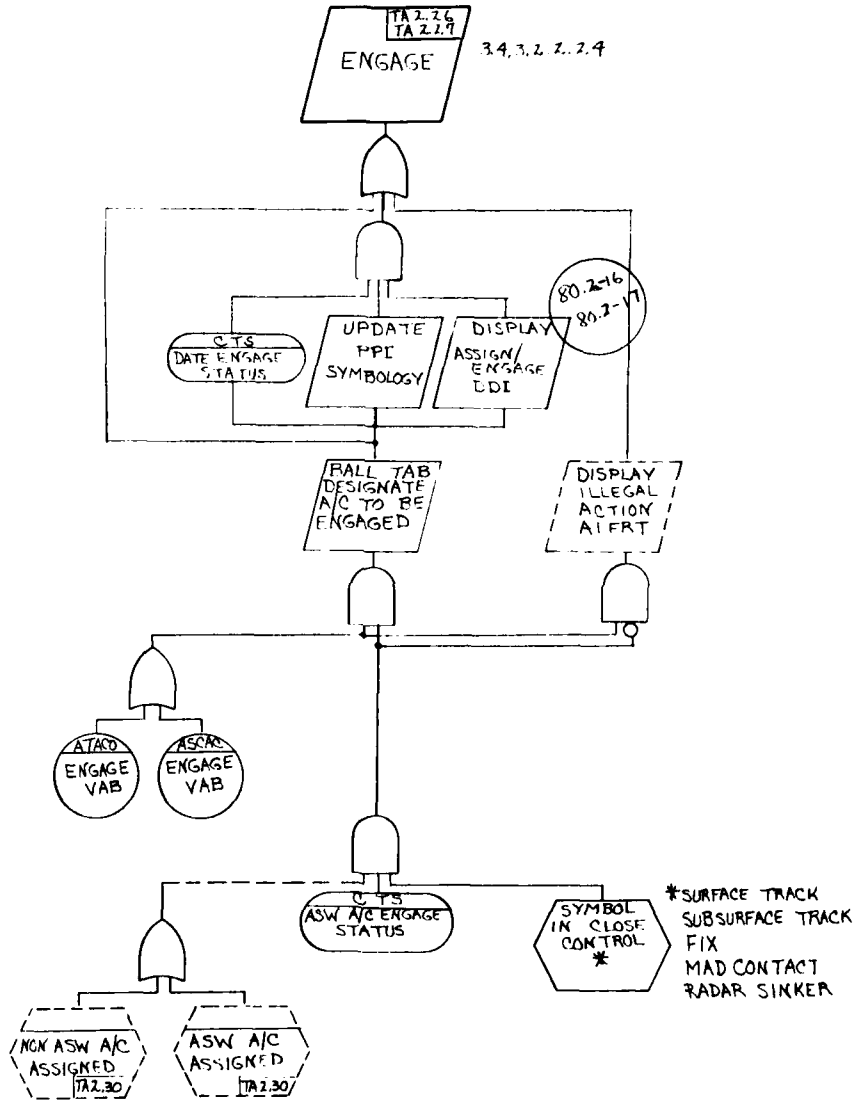
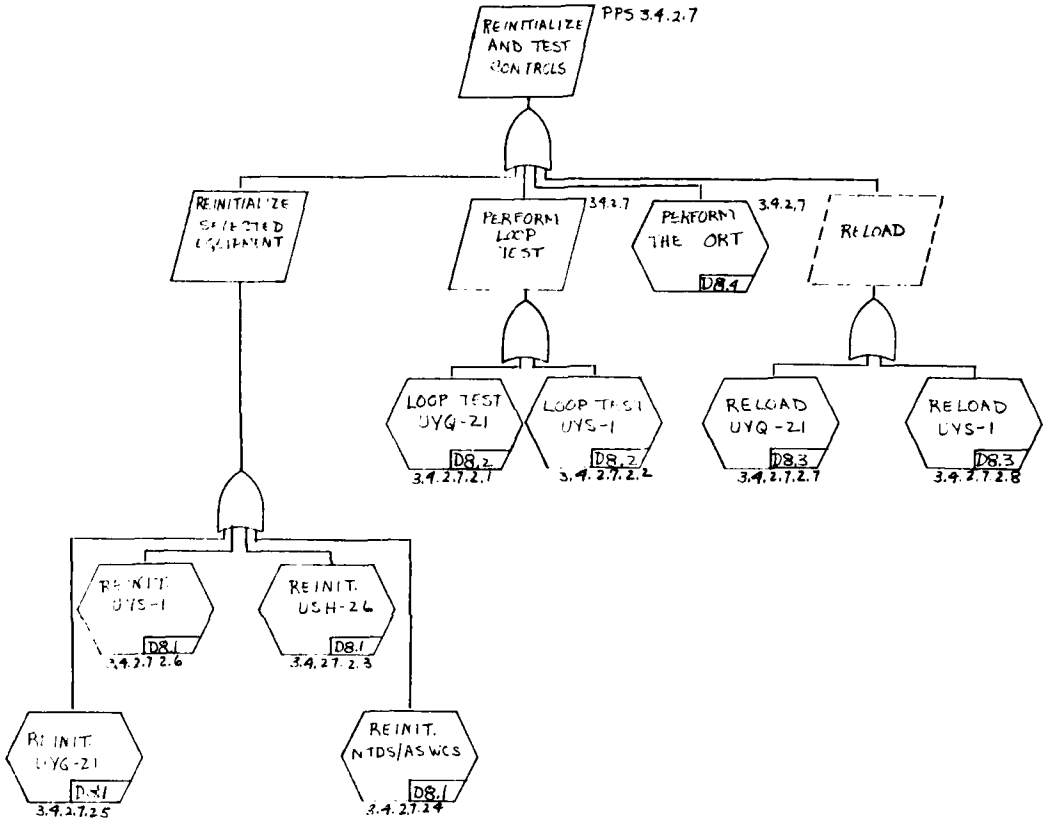


Figure 2. - Downed Aircraft Function

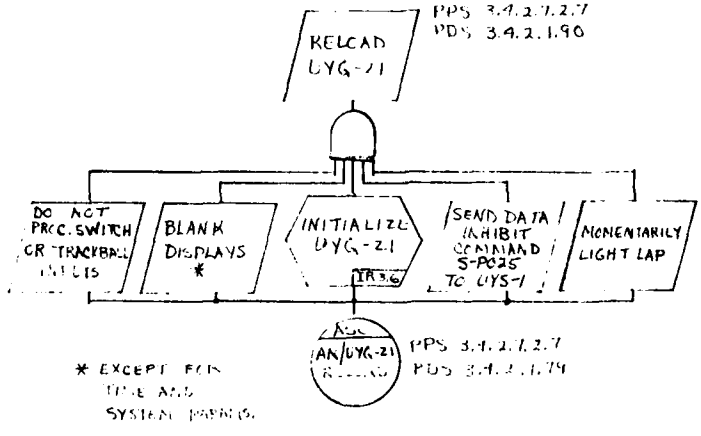
In the next example the references that are circled were added as the result of the analysis. The implication is that there are two Assign/Engage displays. The processing described displaying only one. Further analysis discovered that the first display is put up if the indicated symbols are in close control. The second would be put up if the ASW aircraft were in close control. This would be a major discrepancy because it represents incomplete design. The output references indicate that this function's processing output is used as an input to Charts TA 2.26 and TA 2.27.



Most of the preceeding examples are discrepancies uncovered while developing the model. Once the model is complete it can be used to evaluate overall design and interaction of functions. When weak design or conflicts exist, tests can be developed to determine the impact of these findings. In the example below, three subfunctions of Reinitialize and Test Controls Functions require the use of the USH-26 tape unit. They are REINIT USH-26, Reload UYQ-21 and Reload UYS-1.



Each of the functions were examined to see what measures had been taken to preclude interferences from one with the others. As shown below the Reload UYQ-21 subfunctions provide a condition not to process any other switches or trackball when selected. The other two did not provide the same provision. A simple test was written to alternately select the three switches in all possible combinations. When the REINIT USH-26 was selected followed by a Reload UYS-1, the system "crashed" which required a complete system reload and initialization to recover. In addition to uncovering this problem a simple "fix" could also be recommended, add the Reload UYQ-21 provision to not process switch to the other two functions.

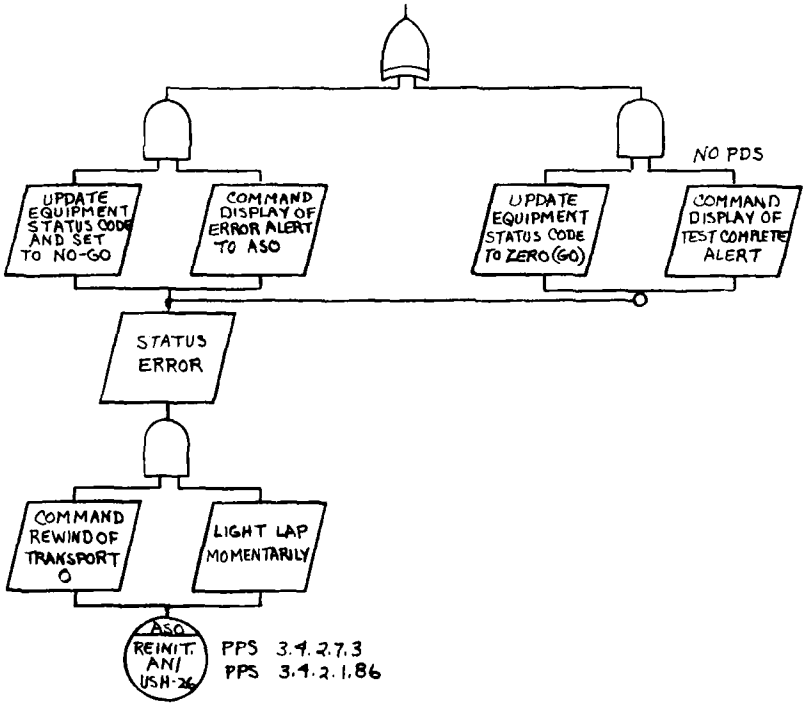


3. DEVELOPMENT

During the development phase the model can be used to develop or verify Test Plans and Procedures, to track other documents (IDS, PDS, etc.), to evaluate Software Change Request (SCR), or Engineering Change Proposal (ECP), and to continue program evaluations and program weakness studies. This type of analysis usually requires more than one chart to give a more complete example of this type of processing analysis. All of the above mentioned applications were found to be very useful and did provide significant results. Due to the limited number of pages allocated for each paper, more specific examples of these analysis could not be included.

4. INTEGRATION

During the integration phase, analysis can be expanded to include Program Trouble Report (PTR) evaluation and Fault Diagnosis. Using the Model to evaluate PTR's has uncovered four basic problems. The first was when the PTR writer did not have a proper understanding of how the program was designed to work. Second, the problem was unique to the laboratory environment and should not be included in an operational configuration. Third, if corrected by the PTR, a bigger problem could occur. Many times this type of problem was due to poor design and should have been submitted as an SCR rather than a PTR. The fourth was PTR's that did not contain enough data to evaluate the problem. The fourth problem was most evident during fault diagnosis. In the example below, the PTR states two problems: 1) "Test Complete" information alert did not appear, and 2) CMTU Tapes did not rewind.



First of all the PTR is stated incorrectly. If the Tape Unit did not rewind this would have constituted an error condition. As indicated by the exclusive "OR" gate, the test complete alert is displayed only if the processing is error-free. Going through the fault diagnosis process is the best way to identify information which should have been included in the PTR. Starting with the switch depression (circle).

1.	Did the LAP Light momentarily?	NO - possible hardware switch problem.
2.	Was Rewind Command Set?	NO - possible keyset switch code problem.
3.	Was Tape Unit On-Line?	NO - could be a design error with software working correctly. CPPS does not specify processing if no return is received and there is no time event associated with the processing.
4.	Tape Unit Did Send Return (Error or Error-Free)	NO - See 3. or command message could be in error. YES - and neither aler displayed could indicate system status monitoring error.

As part of the Fault Diagnosis a simple test procedure could be written to answer these questions to help isolate the possible causes of this problem.

5. OPERATIONAL

During the operational phase the program comes under a Life Cycle Maintenance Plan. Application of the model to Life Cycle Maintenance involves management edification and program control. Effective management control requires an intimate knowledge of system architecture, capabilities, and complexities. In this contest, the ability to compare configuration changes, evaluate change proposals, or commit to system growth or contraction can first be evaluated in the model to determine what ripple effects can develop throughout the entire system. One of the simplest illustrations to observe a ripple effect is to arbitrarily cross out selected key elements in the model and note what capabilities are affected or lost and to what degree those losses degrade the system. A cost exchange trade-off is helpful in determining how significant, sensitive, or attractive any proposed change is in terms of program pay-off.

During Life Cycle Maintenance, situations often occur where small fixes appear to solve local problems only to find out later that they produce other unwanted problems elsewhere in the system. The Model, applied to the evaluation of program fixes, can easily predetermine system impact and avoid unwanted problem proliferation. Using the model as a continuing system engineering tool should produce important cost savings and help maintain schedule integrity by substituting analytic methods which are effective and relatively inexpensive for trial and error methods whose results are generally unknown in advance.

CONCLUSIONS

Notwithstanding the preceding discussion, no single technique can provide a guarantee to an error-free, high quality software configuration. However, if a discipline such as Network Logic Modeling Analysis, a technique that provides uniform applicability at all phases of development without massive expenditures, is applied, a significant reduction in errors and costs can be realized. The following summarizes the advantages of NLMA.

Advantages of Network Logic Modeling Analysis

VISIBILITY

- AIDS UNDERSTANDING OF FUNCTION OPERATION
- SHOWS RELATIONSHIP OF OPERATOR/HARDWARE INTERFACES
- SINGLE POINT FOR REFERENCES WS/PPS/TP/SOM/IDS
- VISUAL REFERENCE TO EVALUATE PTR CORRECTNESS
- VISUALIZE SCR IMPACT

TRACEABILITY

- TRACE UP TO REQUIREMENT
- TRACE OUTPUTS TO OTHER SUBFUNCTIONS
- TRACE IDS MESSAGES ALL THE WAY TO "OTHER SYSTEM FUNCTIONS"
- TRACE BACK TO INPUT ORIGIN
- EVALUATE RIPPLE EFFECT

VALIDITY

- ENSURES ALL INPUTS PROVIDED
- ENSURES ALL PROCESSING CLOSED LOOP
- ENSURES OUTPUTS PROVIDED
- ENSURES COMPLETE DOCUMENTATION
- QUICK, EASY REFERENCE FOR FUNCTION OPERATION

CONFIGURATION MANAGEMENT

- ESTABLISHES VALID BASELINES (PPS/TP/DOM, ETC.)
- EASY EVALUATION OF ECP/SCR's
  - RIPPLE (INTERNAL-EXTERNAL)
  - DOCUMENTATION
- PTR EVALUATION - WEED OUT BAD, INCORRECT, INCOMPLETE, ETC.
- FAULT DIAGNOSTICS

IDA - LANGUAGE DE TEST DU LOGICIEL ET OUTILS ASSOCIES

IDA - SOFTWARE TEST LANGAGE AND RELATED TOOLS

G. LAMARCHE et P. TAILLIBERT

Electronique Serge Dassault (ESD)
55, Quai Carnot
92214 SAINT CLOUD
Tél. : 602.70.17 / 602.50.00

RESUME :

Ce document décrit les résultats d'une étude ayant pour but de définir un ensemble de moyens permettant d'informatiser les opérations de test des logiciels temps réel. Cette étude a abouti à la définition d'un langage de test dont les principales caractéristiques sont tout d'abord présentées. L'article évoque ensuite les différents problèmes pouvant être rencontrés lors de l'utilisation d'un tel langage et les solutions apportées par l'étude (définition d'un macro-langage et d'une bibliothèque d'outils standard). Le dernier chapitre est consacré à la description d'un outil de test particulièrement adapté au cas des logiciels comportant des processus parallèles.

MOTS CLES :

Test de logiciel, langage de test, logiciel temps réel, test en temps réel, processus parallèles, réseaux de Pétri.

I - INTRODUCTION

Malgré les efforts de recherche entrepris depuis un certain nombre d'années sur les étapes de spécification, de conception et de codage, le test des programmes continue d'occuper une place importante dans le processus de développement des logiciels.

Ce point est particulièrement sensible dans le cas des logiciels avioniques pour lesquels le caractère aléatoire des événements commandant l'exécution des différents processus et les contraintes de temps pesant sur ces logiciels imposent de longues et coûteuses opérations de test pour atteindre la fiabilité habituellement requise pour ce type de programme.

Il nous semble donc justifié de proposer un ensemble de moyens destinés à améliorer les conditions d'exécution des opérations de test des logiciels temps réel et à en diminuer les coûts.

1.1. Objectifs généraux de l'étude

La présente étude, menée sous contrat de l'Agence De l'Informatique (ADI) a permis de définir les moyens nécessaires pour aboutir à une spécification plus formalisée et à une automatisation plus poussée des opérations de test des logiciels temps réel.

Ce qui dans cette étude est désigné par test du logiciel peut être défini comme un ensemble d'opérations ayant pour but de comparer le comportement de ce logiciel à un comportement de référence ; la formalisation de ce comportement et des différentes opérations de test permet d'opérer sur des références mieux spécifiées et plus complexes qu'avec les méthodes informelles.

L'automatisation quant à elle, diminue les risques d'erreur humaine et permet l'application répétée de la totalité des tests d'un programme à chaque correction ou modification (tests de non régression). Elle favorise également la mise en oeuvre des techniques d'évaluation de la couverture des tests.

Trois critères principaux ont été retenus pour la conduite de l'étude :

- la généralité, supposant l'indépendance des moyens de test par rapport aux langages de programmation des programmes testés et à la machine sur laquelle ils s'exécutent.

- la portabilité, nécessitant que les outils soient définis hors de toute hypothèse d'implantation autre que celle de pouvoir s'exécuter sur les mini-ordinateurs classiques.

- la simplicité, évitant l'introduction de contraintes d'utilisation par rapport aux conditions actuelles de déroule ment des tests.

Enfin, bien que le but visé par cette étude concerne le test à proprement parler (détection des erreurs) et non pas la mise au point des programmes (identification de la cause de l'anomalie et correction), l'utilisation d'IDA facilitera notablement les opérations de mise au point lorsqu'une erreur aura été détectée (remise automatique en condition d'erreur, écriture d'un programme spécial de diagnostic ...).

1.2. Principaux résultats

L'étude se place dans le cas où les tests sont commandés à partir d'un autre calculateur que le calculateur cible ; cette approche permet la conduite des tests en temps réel c'est-à-dire sans perturbation du programme à tester. Une interface matérielle, connectée sur le bus interne du calculateur sous test, permet d'effectuer les observations nécessaires aux tests des programmes.

L'étude a permis de définir :

* Un langage de test permettant de décrire de manière formelle et standardisée les opérations qui sont habituellement exécutées lors du test d'un programme. Ces opérations peuvent être regroupées en trois grandes fonctions :

  - commande du programme testé, permettant l'exécution de tout ou partie de celui-ci sur un jeu de stimuli d'entrée.

  - mesure du programme testé par observation directe ou enregistrement des valeurs produites par ce programme.

  - vérification par comparaison des valeurs mesurées aux valeurs de référence ou plus généralement, du comportement observé à un comportement de référence.

* Une bibliothèque d'outils de test réalisant, à partir des primitives du langage, des fonctions de test plus évoluées. Elle comporte en particulier de puissants outils d'enregistrement ou de modèlisation du comportement du programme sous test.

* Une interface standard entre machine de test et machine sous test ayant pour but de faciliter l'implantation du système sur un matériel donné.

* Un macro-langage a été défini pour faciliter l'utilisation des outils de la bibliothèque et pour réaliser l'adaptation du langage de test (général) aux langages utilisés pour l'écriture des programmes à tester et aux processeurs de ces langages.

Le chapitre 2 présente les caractéristiques principales du langage de test, le chapitre 3 évoque les problèmes posés par la mise en oeuvre d'un tel langage et les solutions prévues pour les résoudre. Le chapitre 4 décrit un outil de test permettant le contrôle d'un programme par rapport à un modèle décrit à l'aide d'un réseau de Pétri.

II - LE LANGAGE DE TEST

Un langage de test doit offrir des possibilités algorithmiques adaptées aux traitements les plus fréquemment rencontrés dans les opérations de test ; mais, et c'est là un aspect important, il doit permettre la description et la manipulation d'objets extérieurs au programme de test lui-même (données et procédures du programme à tester).

2.1. Description des objets du programme à tester

Les objets du programme à tester se répartissent en deux catégories : les variables et les points de contrôle.

a) Les variables sont décrites par la structure de la donnée mais également par les indications nécessaires pour y accéder ("adresse" et procédé de lecture ou d'écriture). Pour cela, un type "variable-testée" permet d'indiquer :

- le type de la donnée,
- une procédure d'accès en lecture décrivant les opérations nécessaires pour acquérir la valeur de cette variable,
- une procédure d'accès en écriture.

On définit ainsi une application de la représentation de ce type, de la machine testée vers la machine de test. Un paramètre supplémentaire (attribut adresse) peut être précisé à la déclaration d'une telle variable et référencé dans les procédures d'accès. Ainsi par exemple, toutes les variables entières basées par rapport à l'adresse "PROC1" peuvent être décrites par le type ci-dessous :

```
type  entier-proc-1  is   tested-var
  integer (32) ;
              reading is
                        -- Procédure permettant l'
                        -- accès en lecture aux
                        -- objets du type
              end   reading ;

              writing is
                        -- Procédure d'accès en
                        -- écriture

              end   writing
                        end tested-var ;
```

Les objets de ce type sont déclarés par :

```
$TOTO at 20, $TITI at 100 : entier-proc-1 ;
```

- le caractère $ permet de distinguer les objets du programme de test de ceux du programme sous test.
- 20 et 100 sont les "attributs adresse" des objets déclarés.

b) Les points de contrôle : un point de contrôle permet de décrire certains points particuliers de la structure de contrôle d'un programme (étiquette, début de procédure, de bloc, numéro d'instruction ...) . Il peut être utilisé comme point de lancement, point de surveillance ou point d'arrêt d'un programme ; dans ces deux derniers cas, le passage du programme sous test devant un tel point peut engendrer l'activation d'un évènement dans le programme de test ou l'arrêt du calculateur sous test.

La description de ces objets se fait par l'intermédiaire d'un type "point de contrôle" où l'on indique :

- un prologue décrivant la séquence d'opérations à réaliser lorsque le point de contrôle est utilisé comme point de départ (y compris le passage éventuel de paramètre au programme sous test).

- un détecteur précisant les opérations nécessaires pour utiliser ce point de contrôle comme point de surveillance ou d'arrêt (action sur le matériel ou modification du code du programme sous test).

Comme dans le cas des variables la déclaration d'un tel objet est accompagnée d'une constante entière "attribut adresse" qui contribue au calcul de l'adresse du point de contrôle.

2.2. Manipulation des objets du programme à tester

Elle se fait par l'intermédiaire des expressions, des fonctions de conversion ou des primitives d'interface.

2.2.1. Expressions

Les objets du programme testé peuvent apparaitre dans une expression au même titre que ceux du programme de test ; les opérateurs du langage (+, -, x, /, =) pouvant être redéfinis pour les nouveaux types éventuellement introduits.

Exemple :
A : = B + $C
- $C étant un objet du programme sous test

2.2.2. Conversions

La redéfinition des opérateurs est une opération relativement lourde et peut être évitée par la définition de fonctions de conversion vers un type du programme de test et ainsi utiliser directement les opérateurs classiques.
Afin de faciliter cette opération, la fonction de conversion porte le nom du type cible, l'objet à convertir étant passé en paramètre. Ainsi dans l'expression suivante :

A : = B + FLOAT ($C) ;

FLOAT ($C) représente le résultat de la conversion de $C (lui même d'un type réel du programme sous test) dans le type prédéfini FLOAT.

2.2.3. Interface standardisée

La référence à un objet du programme sous test dans une expression provoque l'exécution de la procédure reading ou writing définie dans le type correspondant. Il en est de même pour le prologue ou le détecteur des points de contrôle. L'écriture de ces procédures nécessite la manipulation de l'interface entre les deux machines. Une telle manipulation est également nécessaire lorsque l'utilisateur souhaite qu'un événement soit activé lors de l'arrivée du programme sous test dans un état donné.

Afin d'assurer la portabilité des systèmes de test, l'interface entre machine de test et machine sous test a été standardisée. Elle se compose de procédures, définies par leur spécification externe, que chaque implémentation doit réaliser en tenant compte du matériel de l'installation. L'implémentation de ces procédures peut se faire en particulier grâce à un sous-ensemble du langage de test (sous-ensemble d'interface) ne comprenant pas les constructions qui manipulent les objets de la machine sous test.

Cette interface comprend en particulier :

- copie de mémoire à mémoire
- démarrage et arrêt CPU
- pose de point de surveillance et d'arrêt
- contrôle d'accès à la mémoire.

2.3. Instructions facilitant l'expression des tests de logiciel

Un certain nombre de constructions ont été introduites dans le langage afin de faciliter l'expression des opérations de test les plus fréquemment rencontrées. Parmi celles-ci figurent :

- l'itérateur
- la gestion du temps.

2.3.1. Itérateur

L'observation des techniques utilisées pour effectuer les tests d'une unité de programme avant son intégration (test unitaire) fait apparaître l'importance de la structure de contrôle répétitive. En effet, le test consiste dans la plupart des cas à exécuter un grand nombre de fois le programme pour différentes conditions et à comparer le comportement observé au comportement prévu ; conditions et comportement de référence se traduisent très souvent par des tableaux de valeurs sur lesquels porte la répétition. Cette constatation a conduit à prévoir une construction spéciale appelée "itérateur", [LIS 77] permettant d'engendrer une succession de valeurs sur lesquelles porte l'itération sans être contraint de stocker ces valeurs au préalable dans un tableau approprié. Cette construction est analogue à une fonction, opérant sur des variables rémanentes, initialisée à l'entrée d'une boucle et délivrant une nouvelle valeur à chaque appel. Elle présente également l'avantage d'améliorer la structuration des programmes de test.

Exemple :

```
Type donnée-test is
   record
   ENTR : integer ;-- valeur d'entrée
   SORT : integer ;-- valeur de sortie
   end record ;
r : donnée-test ;
iterator valeur-test (paramètres formels) yields donnée-test is
                -- détermination de l'ensemble des couples
                -- (ENTR, SORT) à raison d'un couple par
                -- activation.
end valeur-test ;
for r in    valeur-test (paramètres effectifs)
                loop
                -- initialisation du test avec la valeur ENTR
                -- exécution
                -- comparaison de la valeur de sortie à SORT
                end loop ;
```

2.3.2. Gestion du temps

Le langage de test étant principalement destiné à la vérification de logiciels temps réel, il est nécessaire qu'il dispose d'outils pratiques de gestion du temps ; ce besoin se fait plus particulièrement sentir lorsque l'utilisateur désire commander ou observer l'environnement de la machine sous test. La précision des outils habituellement implémentés peut s'avérer insuffisante compte tenu du fait que ces opérations sont soumises à l'aléa du "scheduling" logiciel. C'est pourquoi les constructions agissant sur le temps peuvent être utilisées en deux modes distincts :

- le mode "normal" où la synchronisation est réalisée par les mécanismes habituels de scheduling.

- le mode "précis" où au contraire l'opération est réalisée après une attente active garantissant ainsi une meilleure précision.

Une instruction du langage permet de préciser :

- la date de début de l'action associée
- sa périodicité
- une clause de fin de répétition (durée ou condition)
- une clause de précision

Exemple :

```
at time 10 every 0.4 sec during 3 sec schedule ;
```

III - MISE EN OEUVRE

L'utilisation du langage de test qui vient d'être défini conduirait à certaines difficultés s'il était utilisé seul. En effet, il obligerait l'utilisateur à redéclarer tous les objets du programme sous test qu'il désire manipuler ; d'autre part, on peut constater que le langage ne comporte pas de fonctions de test très élaborées, celles qui y figurent devant plutôt être considérées comme un ensemble de "briques" à partir desquelles peuvent se construire des outils plus évolués. Ces difficultés ont été résolues par la définition d'un macro-langage, d'une bibliothèque d'outils standard et de quelques règles d'implémentation.

3.1. Le macro-langage

Un système de test "réaliste" ne doit pas obliger son utilisateur à redéclarer tous les objets du programme à tester ; cette opération ayant été faite lors de l'étape de codage, il est souhaitable que ces objets soient connus "implicitement" dans le programme de test.

Pour aboutir à un tel résultat tout en restant indépendant du langage de programmation utilisé et de ses processeurs, un niveau de langage supplémentaire a été introduit afin d'aider l'utilisateur à déclarer les objets du programme sous test. Ce macro-langage est muni d'opérations d'entrée-sortie lui permettant d'accéder aux différentes tables produites par les processeurs du langage de programmation et si nécessaire au texte source du programme à tester. Il autorise la génération d'instructions représentant les déclarations souhaitées établies à partir des informations recueillies dans les tables.
Il est ainsi possible pour un utilisateur particulier (soit homme système, soit simplement premier utilisateur) de définir pour chaque couple langage - processeur un ensemble de procédures que l'utilisateur final pourra utiliser pour déclarer les objets et les manipuler avec pratiquement la même facilité que si l'existence de ces objets était implicite.

3.2. Bibliothèque d'outils standard

Elle a pour rôle de fournir un certain nombre d'outils réalisés à l'aide du langage de test et du macro langage et couvrant une bonne partie des besoins les plus courants. Elle permet ainsi de réduire notablement le temps à consacrer au développement des programmes de test. Elle sert également de structure d'accueil aux outils de test plus spécifiques d'une méthodologie ou d'une application.

La bibliothèque standard comprend en particulier :

* une macro-instruction d'exécution autorisant le lancement du programme sous test en un point de contrôle particulier ou au début d'une procédure (il est possible dans ce cas de fournir des paramètres effectifs à la procédure). Cette macro-instruction permet de spécifier un invariant sous la forme d'une expression qui est évaluée avant puis après l'exécution et dont la variation entraine un diagnostic d'erreur. Enfin, il est également possible de demander le calcul d'une variation et de préciser une durée limite.

* un outil de simulation d'instruction utilisable lorsque certaines instructions machine ne peuvent pas être exécutées (entrées-sorties non câblées) ou que certaines procédures ne sont pas encore au point. Il est possible, grâce à cet outil de substituer l'exécution d'une procédure du programme de test aux instructions ou procédures absentes.

* un enregistreur ayant pour rôle de mémoriser pendant l'exécution du programme un certain nombre d'informations. Le test consiste alors à dépouiller l'enregistrement ainsi obtenu. L'utilisateur doit préciser quand enregistrer (point de contrôle, accès à un domaine ..), quoi enregistrer et quand cesser l'enregistrement.

* un outil de modèlisation permettant de contrôler le comportement d'un programme par rapport à un modèle décrit à l'aide d'un réseau de Pétri. Cet outil est décrit en détail au chapitre 4.

3.3. Implémentation

Il peut être lourd ou difficile de spécifier a priori de manière formelle et détaillée l'ensemble des opérations de test à effectuer sur un programme.

En conséquence, il est utile qu'une implémentation d'IDA prévoit des possibilités d'élaboration interactive et progressive des programmes de test de telle sorte que l'utilisateur, partant d'un canevas défini a priori, puisse disposer en fin de test d'un programme complet qui pourra être réexécuté automatiquement chaque fois que nécessaire.
Sans pour autant offrir toutes les possibilités d'un traducteur incrémental, le système permettra à l'utilisateur d'effectuer l'adjonction de certaines instructions ou de compléter le domaine d'itération d'une boucle de test.
Ces possibilités d'élaboration interactive portent également sur les fichiers manipulés par les programmes de test. A cet effet, il est possible de prendre le contrôle en des points particuliers, et de compléter le jeu d'essai.

IV - UTILISATION DE LA TECHNIQUE DE L'OBSERVATEUR POUR LE TEST DES PROGRAMMES TEMPS REEL

Ce chapitre décrit, à titre d'exemple, l'un des outils de test en temps réel défini au cours de l'étude.

4.1. Objectif

Il s'agissait d'offrir à l'utilisateur les moyens permettant le contrôle sans perturbation du programme sous test :

- du comportement des processus parallèles (synchronisation, partage des ressources...)
- des contraintes de date ou de durée d'exécution.

Dans l'état actuel de disponibilité des outils de test, le second point peut être partiellement pris en compte grâce à l'outillage utilisé habituellement pour les tests du matériel (analyseur logique, oscilloscope,...).
Par contre, en ce qui concerne le test du comportement des processus parallèles, le programmeur se trouve réellement démuni. Ainsi la simple vérification du *fait que deux* séquences ne s'exécutent jamais simultanément ne pourra être effectuée qu'après une mise en oeuvre laborieuse d'un équipement mal adapté (analyseur logique "performant").

4.2. Principe

Il s'appuie sur le concept d'observateur [AYA 79-2] et est schématisé par la figure 1 ; il met en jeu :

- le programme sous test ;
- un modèle décrivant le comportement de référence faisant l'objet du test ;
- un ensemble de connexions entre programme et modèle ;
- un "contrôleur" chargé de faire évoluer le modèle parallèlement au programme tout en s'assurant de la validité de son évolution.

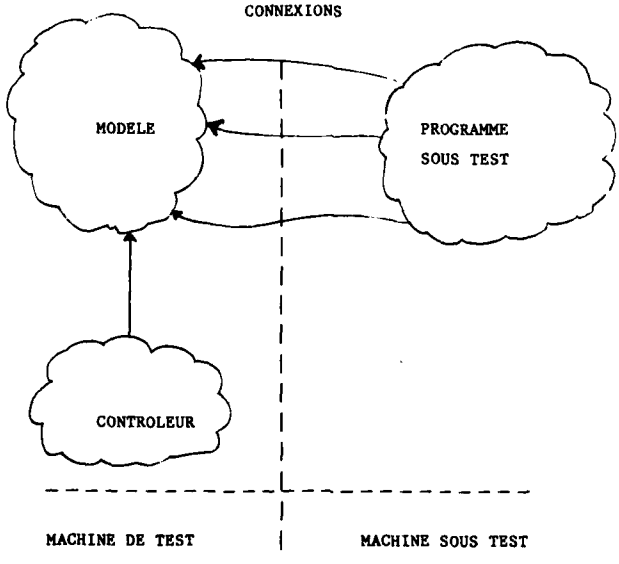


Figure 1

* Le programme testé est inchangé par rapport au programme définitif et aucune perturbation n'est induite par le contrôleur ; il est nécessaire pour cela de disposer d'une interface particulière d'observation entre la machine de test et la machine testée.
Ce point sera développé au paragraphe 4.4.

* Le modèle est une représentation d'une certaine partie du programme correspondant au comportement dont on veut tester l'implémentation.

* Les connexions permettent d'établir une correspondance entre programme et modèle afin que le "contrôleur" puisse vérifier qu'ils évoluent de manière cohérente. Les connexions sont réalisées grâce à des points de contrôle de telle sorte qu'à chaque passage du programme devant un tel point un signal soit émis vers le contrôleur pour que celui-ci s'assure que l'état courant du modèle est cohérent avec l'évolution constatée du programme.

* Le contrôleur a pour rôle de comparer l'évolution du programme à l'état du modèle et de faire évoluer celui-ci en conséquence. Il ne dépend bien entendu que du type de modèle utilisé et non de chaque réalisation.

4.3. Exemple

Il est parfois nécesaire, dans les programmes temps réel, de s'assurer que deux séquences ne s'exécutent jamais simultanément. Ce peut être le cas, par exemple si l'une d'entre elle acquiert une donnée utilisée par l'autre. La cohérence de la donnée nécessite une exclusion stricte entre les deux séquences. Cette règle pourra être modélisée à l'aide d'un réseau de Pétri et chaque transition associée à un point caractéristique des séquences en exclusion (figure 2.) Le contrôleur, activé à chaque passage du programme devant le point de contrôle, s'assurera que les jetons sont dans les bonnes places au bon moment et les fera évoluer en conséquence.

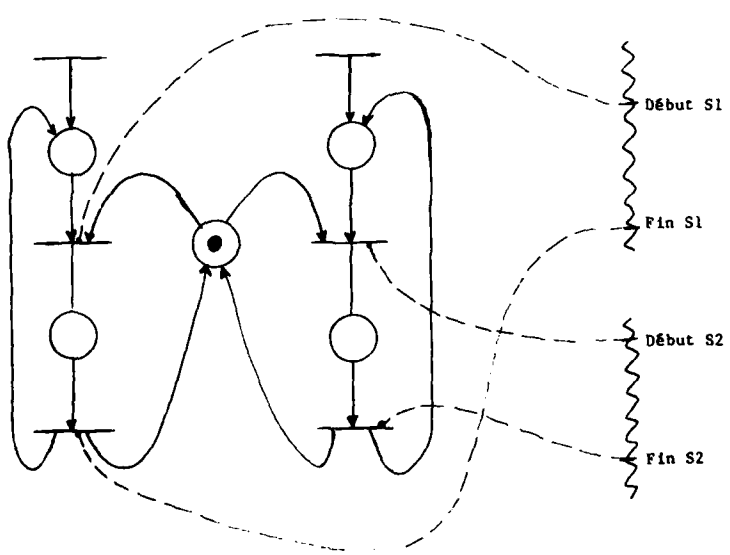


Figure 2

Remarque : Les réseaux utilisés resteront en général simples ; en effet, il ne s'agit pas de modéliser tout le programme (comme lorsque l'on désire prouver l'absence de blocage) mais uniquement le comportement à contrôler. Ainsi pour s'assurer qu'aucune perte d'interruption n'intervient il suffira de décrire le réseau ci-dessous.

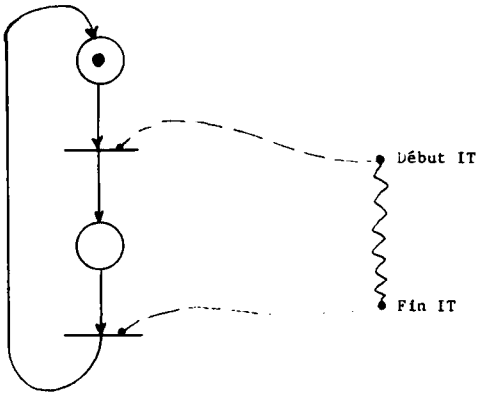


Figure 3

Enfin, l'outil proposé dans le système IDA permet le contrôle des temps inter-transitions.

4.4. Implémentation

Afin de ne pas perturber le programme sous test, l'interface entre machine de test et machine sous test se limite à une observation du Bus interne de la machine testée et des quelques signaux permettant d'identifier les informations y circulant.

Une telle interface permet d'effectuer malgré ces restrictions :

- la détection des points de contrôle et leur datation ;
- la détection des modifications intervenant dans certains emplacements mémoire désignés au préalable.

Cependant, la prise en compte de tels événements par la machine de test n'est pas possible en temps réel. (Ceux-ci pouvant apparaître ponctuellement de manière très rapprochée) ; une telle prise en compte n'est d'ailleurs pas utile, les contrôles pouvant être effectués en différé pourvu que la chronologie d'apparition des événements soit respectée. En conséquence machine de test et machine sous test sont désynchronisées ; une file d'attente contenant les événements observés (points de contrôle ou modifications mémoire)et leur date d'apparition permet d'"adapter" la charge de travail de la machine de test (figure 4).

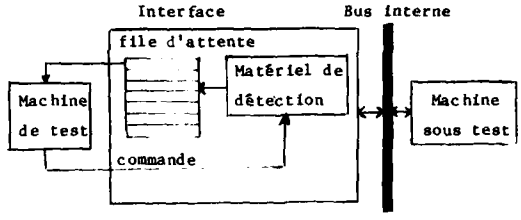


Figure 4

Le "matériel de détection" est réalisé grâce à un comparateur dynamique formé d'une mémoire RAM d'une capacité d'adressage égale à celle de la machine testée et d'une longueur de mot de 1 bit.

Une première phase - avant le test lui-même - consiste à programmer cette mémoire aux adresses pour lesquelles une détection est souhaitée. Les techniques actuelles d'intégration permettent d'envisager un tel comparateur avec un minimum de matériel.

Remarque

Dans le cas où la machine sous test est multiprocesseur chaque processeur est muni d'une interface de détection, la chronologie des événements enregistrés étant restituée grâce aux datations mémorisées dans chaque file d'attente.

4.5. Utilisation

L'outil qui vient d'être décrit permet d'effectuer des tests de niveau élevé et cela sans perturber le programme testé.

La puissance des tests est liée directement à la puissance des modèles proposés. L'absence de perturbation est, pour sa part, due au fait que seules des observations fugitives de la machine testée permettent de mettre en oeuvre la méthode.

L'utilisation de cet outil, et en particulier de la modélisation par réseau de Pétri, pourra être rendue aisée par une interface homme machine bien adaptée (interface graphique dans la mesure du possible). D'autre part les réseaux à décrire restent simples compte tenu du fait que seul le comportement à tester doit être modélisé et non pas l'ensemble de l'application comme cela est le cas pour effectuer des analyses statiques sur ces modèles.

Enfin on peut remarquer que cet outil peut rendre de grands services pour la mise au point des programmes lorsque, une erreur ayant été détectée, de longues manipulations sont nécessaires pour en trouver la cause.

V - CONCLUSIONS

Les "moyens de test" qui viennent d'être présentés (langage de test, macro-langage et bibliothèque d'outils) permettent d'envisager la conduite des tests des logiciels à haute sécurité sous un jour nouveau :

- pratique systématique des tests de non-régression,
- application de jeux d'essais longs et complexes grâce à l'automatisation introduite par les outils,
- contrôle de comportements difficilement observables jusqu'ici grâce à la puissance de modélisation disponible.

L'indépendance de ces moyens par rapport aux langages de programmation et aux calculateurs cibles a été étudiée afin de minimiser les coûts d'adaptation à chaque utilisation particulière.

Un interpréteur d'un sous-ensemble du langage de test a été réalisé à l'Electronique Serge Dassault et est en exploitation (système LOTUS [VIE 81]).

Enfin, une baie de mise en oeuvre, munie d'une interface matérielle remplissant les fonctions décrites précédemment, est en cours de développement pour un calculateur de l'ESD.

REFERENCES

[AYA 79-1] JM. AYACHE
A Methodology for specifying Control in Electronic Switching Systems
International Switching Symposium
PARIS MAI 79

[AYA 79-2] AYACHE-AZEMA-DIAZ
Observer : a concept for on-line detection of control errors in concurrent systems
9 th International symposium on fault tolerant computing
MADISON JUIN 79

[ESD 82-1] ELECTRONIQUE SERGE DASSAULT
IDA - Manuel de référence du langage de test Document ESD NE 39 I86
Mars I982

[ESD 82-2] ELECTRONIQUE SERGE DASSAULT
IDA - Manuel de référence du macro-langage
Document ESD NE 39 428 Mars I982

[ESD 82 -3] ELECTRONIQUE SERGE DASSAULT
IDA - Ebauche de spécification externe d'outils Document ESD NE 39 522 Mars I982

[HAL 81] N. HALBWACHS, P.CASPI
Modélisation et validation de systèmes temporisés discrets
Journée d'étude AFCET
Mars I98I

[LIS 77] LISKOV-SYNDER-ATKINSON-CHAFFERT
Abstraction mechanisms in CLU
Communications of the ACM
Aout 1977

[PER 78] R. PERRET
Présentation de LASICO. Langage de simulation et de commande
Journées BIGRE
NOVEMBRE 78

[SIF 79] J. SIFAKIS
"Le contrôle des système asynchrones: concepts, propriétés, analyse statique"
Thèse es-sciences, USMG - INPG
GRENOBLE Janvier 79

[VAL 76] M. VALETTE
Sur la description, l'analyse et la validation des systèmes de commande parrallèle.
Thèse de Docteur d'état
Université Paul Sabatier
TOULOUSE - Novembre I976

[VIE 81] G. VIENNET - J.C. SEGUIN - M.ESTEVENY
Manuel de référence du système LOTUS
Document EMD
JUIN 81.

SOFTWARE VERIFICATION OF A CIVIL AVIONIC AHR SYSTEM

Dr. Michael Kleinschmidt and Dr. Norbert Sandner
Litton Technische Werke - der Hellige GmbH
Lörracher Straße 18
D-7800 Freiburg

SUMMARY

The trend in civil aviation towards highly integrated digital avionics systems implies a new and thorough set of procedures for the generation and verification of the associated software. The objectives of these procedures are to raise the quality of the software product and to reduce the expenses for development and maintenance. For the development of an Attitude and Heading Reference System in strapdown technology, new methods and tools are described following the general guidelines set up by the aviation industry and its associates.

These methods and tools used in the process of validation were accepted by aviation certification authorities. The certified avionic system has been successfully operational in a transport aircraft, Airbus A300, since May '82 with no software errors having been detected. For both methods and tools, extension is in progress to enhance the advantages measured in the current project.

1. INTRODUCTION

The theory of software testing and validation has been under discuss for several years. Unfortunately, most of the contributions have only forced the maturity of the commercial and purely computational part of that field, neglecting the particular problem of verifying and validating software used in advanced technical applications.

For those applications, several avionic groups (users and suppliers) have developed standards for Software Design and Verification derived from the general testing techniques. These standards describe the requirements in general form and must be tailored to fit any individual application.

1.1 LTR 81 - LITEF ATTITUDE AND HEADING REFERENCE SYSTEM

LITTON Technische Werke in Freiburg (LITEF) is a company which has been building military and civil navigation systems since 1962. The modern design of those systems in advanced digital technology attaches a large portion of the systems' logic to computer programs.

The LTR 81 System is the first civil avionic product to be built at LITEF in all digital technology with a medium-sized computer program (about 20K of computer words in assembler language). LTR 81 is an attitude and heading reference system for use in transport aircraft similar to the A300. It is designed in strapdown technology using 2 two-degrees of freedom, dry-tuned rotor gyros, and 3 linear pendulous accelerometers to comply with ARINC 705 specifications. The dynamic attitude is measured to plus and minus 0.25 degrees and the heading to within 2 degrees. The instruments are combined with a dual 16-bit microprocessor set, common semiconductor memory, and the required interface electronics. All computer programs are written in assembler language using the facilities of implemented high order commands.

The LTR 81 system is classified as "flight critical" in the Airbus A300 installation (forward facing cockpit configuration), as it is the only reference for both attitude and heading during automatic precision instrument approaches and landings. In addition, LTR 81 is the basis for a product line of AHR systems currently under development at LITEF for various civil and military applications.

1.2 GENERAL VERIFICATION REQUIREMENTS

The purpose of the software verification is to ensure and document that all system requirements (product specifications) and software requirements have been achieved and that the software development standards have been followed, thus assuring that the software performs all intended functions and does not perform any unintended function. Furthermore, it will ensure that the final product is easy to maintain and that its documentation is easy to understand. The "criticality" of the system has a strong impact on the design, programming, verification, and documentation of the software.

The verification is a formal process of analysis and testing of the flight critical software including all applicable documentation. The general criteria are defined by the primary customer of this particular system. These requirements are similar to the recommendations as described in document DO-178 by the RTCA and other papers for avionic software.

Using the described documents as general guidelines, new methods and tools have been developed at LITEF. These methods cover all customer requirements, as well as additional LITEF requirements for maintenance.

All verification actions are depicted in Figure 1 in relation to other development activities (definition and design).

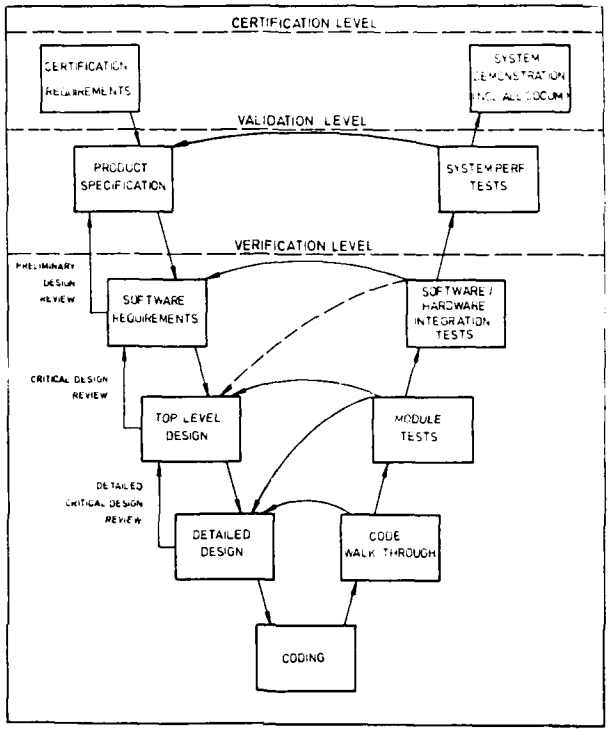


Figure 1: Development Actions including Design and Verification

2. TOOLS FOR SOFTWARE VERIFICATION

The complete process of software design and verification is controlled and documented by a Software Development Support System (SDSS) using an ONYX C8002 minicomputer with a UNIX operating system. The individual parts of the SDSS (see Figure 2) have been defined and developed at LITEF. The methods and tools allow applications in various civil or military projects.

The complete implementation includes:

- tools for the control of
  - software development
  - software verification
  - configuration control
  - change control
- documentation for software design and software verification

This paper is restricted to the tools and methods for software verification.

2.1 CONTROL OF THE VERIFICATION PROCESS

The main items which control the process of software verification with respect to the completeness of the tests and documentation and those which facilitate maintenance are listed below.

2.1.1 ANALYSIS OF THE SOFTWARE REQUIREMENTS

A formal review is performed and documented to analyze the Software Requirements for system level tests. The review is performed by system test engineers with the aid of system and hardware designers. It defines and documents in a standardized form all testable items of the system, as described in the Software Requirements. These results are the basis for establishing the corresponding Test Evaluation Matrix.

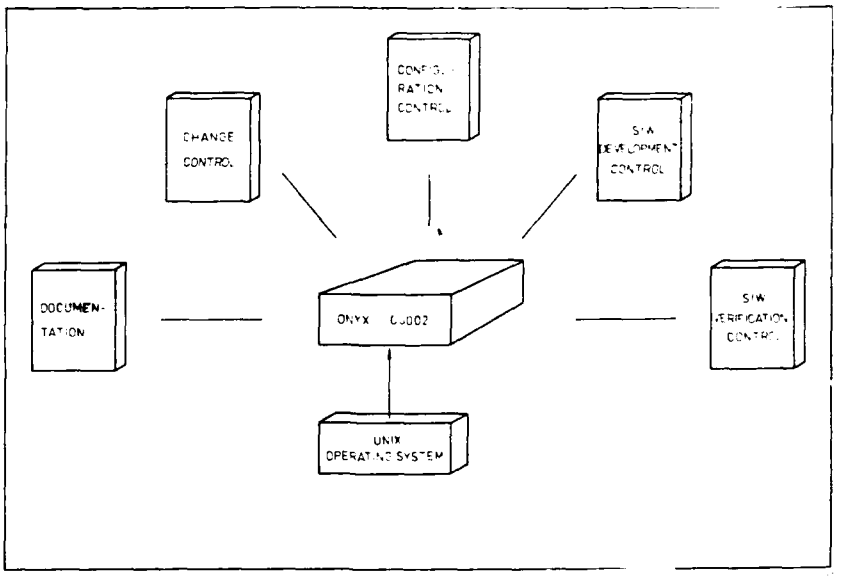


Figure 2: Software Development Support System

2.1.2 ANALYSIS OF SOFTWARE DESIGN

A similar review as described above is performed to define software tests at module level. The formal analysis of the Top Level Design and the Module Description yields a basis for the corresponding Test Evaluation Matrix.

2.1.3 GENERATION OF TEST EVALUATION MATRICES

Figure 3 shows by example of system level tests a scheme of the Test Evaluation Matrix. The Software Requirements Manual is the basic reference document used for system level tests. Thus, the matrices relate this document to a complete set of test procedures for system level tests (Hardware-Software Integration Tests). In order to reduce the overall size, the complete matrix consists of two levels (top and bottom).

On the top level, the Software Requirements are divided into functional groups of requirements (e.g., real time requirements, systems mechanization requirements, self test parts, etc.). The functional test groups are defined by the review described in 2.1.1.

On the bottom level, each functional test group is divided into its separate testable items as defined by the review described in 2.1.1. Each item is then identified by a test number which includes a reference to the test group and the software requirement paragraph and subparagraph.

Furthermore, all software modules participating in the performance of the required function are attributed to each item in the matrix. Thus, the matrix relates a complete partition of the systems functions to test reference numbers and the associated software modules.

Additional information can be included for the Software Version, Test Version, and Test Status of each matrix element. The matrices are stored in a suitable form on the SDSS for easy automatic processing and updating.

2.1.4 CROSS REFERENCES

From the above described matrices, various Cross Reference Tables can be generated automatically.

These tables relate:

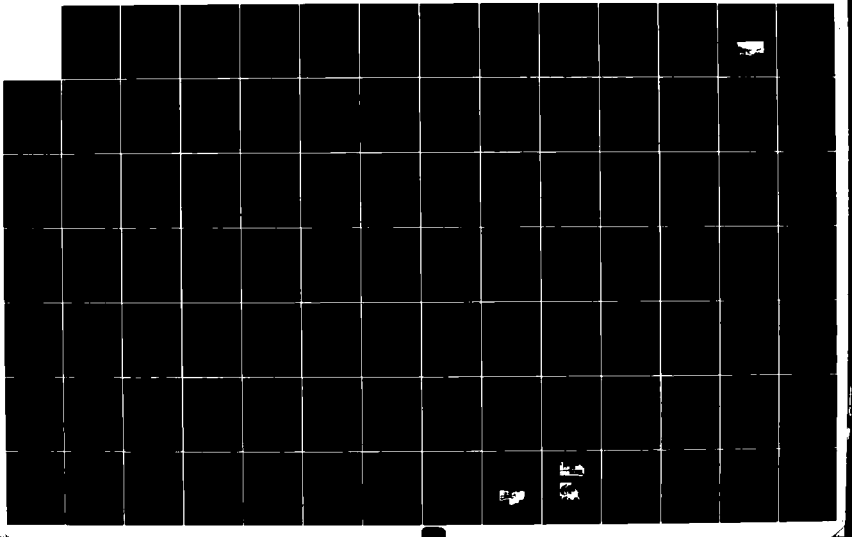
- modules to their individual tests
- modules to Software Requirements
- Software Requirements paragraphs, subparagraphs, statement to their individual tests.

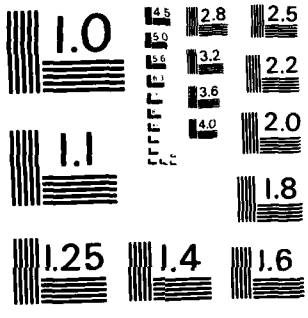
This tool is important for controlling the tests after a software modification.

For each change in the Software Requirements, the Module Design, or the Program Code, a complete list of necessary regression tests is generated by the system. Furthermore, the various tables yield for Quality Assurance or Certification Authorities a direct and clear reference from requirements to tests.

AD-A127 131 SOFTWARE FOR AVIONICS(U) ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT NEUILLY-SUR-SEINE (FRANCE) JAN 83 AGARD-CP-330 4/5

UNCLASSIFIED F/G 9/2 NL





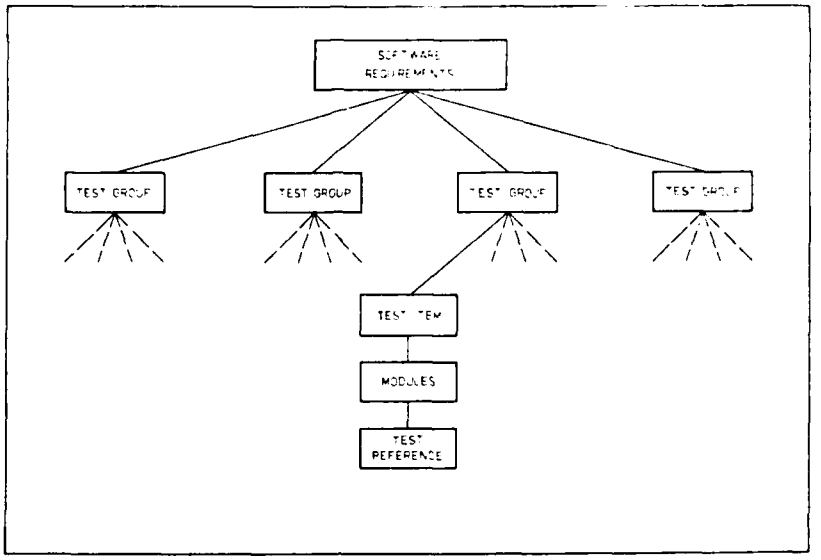


Figure 3: Test Evaluation Matrix

2.1.5 STANDARD TEST PROCEDURES

A standardized Test Procedure form has been developed for Hardware/Software Integration Tests. This form ensures the completeness of the test definition and eases the access to the Test Procedures for updates or modifications.

2.1.6 UPDATE OF TEST PROCEDURES

All elements of the Test Procedure which depend directly on a specific software version can be updated automatically on the SDSS by releasing a new software version. This tool ensures the completeness and correctness of the update and decreases considerably the time needed for the definition of the regression tests.

2.1.7 AUTOMATIC COMPLETENESS CHECK

This tool ensures and documents that each item of the Software Requirements is covered by a Test Design and a Test Procedure or that each relevant Test Procedure has been modified and updated for regression tests.

The above described tools and methods guarantee that the process of software verification is controlled and documented in a complete and consistent way. The tools ease maintenance and facilitate for customers and certification authorities the insight into the organization of the process.

3. METHODS FOR VERIFICATION

The different parts of software verification, as shown in Figure 1, are described in general in applicable papers for software verification and validation. Flight tests are not included in the verification process, although they are necessary for validation of the avionic system. They are not subject for discussion under the topic of this paper.

3.1 DESIGN VERIFICATION

Two major verification activities are performed during the design phase of the software:

A) The Preliminary Design Review ensures that the software requirements are complete, correct, and consistent with higher-level systems requirements (e.g., Product Specification).

B) The Critical Design Review ensures on the higher level that the Top Design is complete, correct, and consistent with the Software Requirements. On the lower level, the consistency and completeness of the Detailed Design with the Top Design is checked.

Both reviews are performed by a group of systems designers, software designers, and software test engineers and follow a formal, defined procedure. This procedure ensures the completeness of the review and documents the results in a standardized form. Additionally, extensive simulations of all mechanization algorithms and control logic are performed in parallel using high order language programs on a host computer to verify the basic systems design.

3.2 PROGRAM VERIFICATION

For all program verification procedures, the tester issues a Discrepancy Report to the Software Control Board in case an error is found.

That Board defines the necessary change actions and the regression test activities following the general procedure depicted in Figure 4.

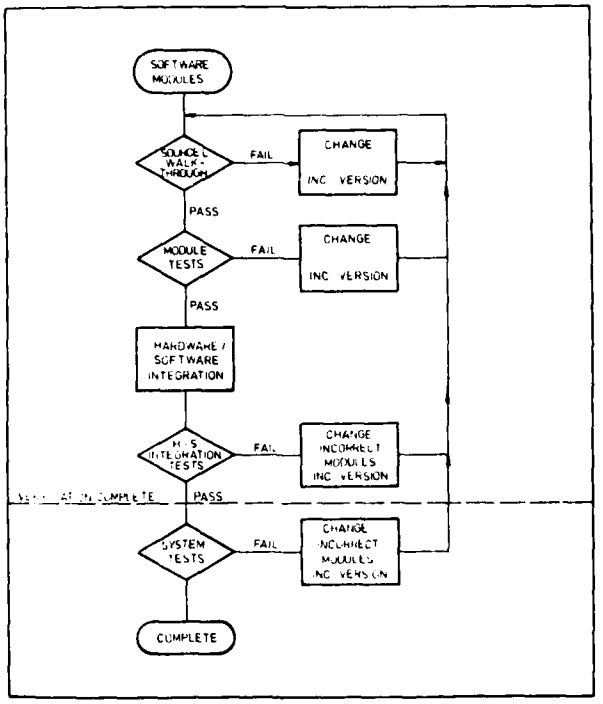


Figure 4: Action Flow for Tests and Resulting Changes

3.2.1 STATIC CODE ANALYSIS

The first process of program verification using the actual code is the static code analysis. This action is started when one of the programmers is satisfied with any program part (module). In this project, the team of analyzers was independent of the programmer team.

The objective of the code analysis is to detect all errors unintentionally implemented during the coding process. The nature of the errors expected to be found in a real-time application implemented in assembler language differ from problems in ordinary applications in high order language, i.e., as described by Howden.

A comprehensive checklist was developed to analyze the code against the Software Design Manual for correctness. Additional tests were implemented to test the safety and maintainability of the code against LITEF coding rules. Test subjects are the individual software modules with an average size of 150 statements of assembler code.

The completeness and the results of the analysis are traced and documented using the tool of the status matrix processing implemented in the SDSS.

3.2.2 DYNAMIC MODULE TESTS

The main objective of the dynamic module tests is to verify that the tested module is capable of exactly performing the logical and arithmetic functions defined by the designers. Compared with the static code analysis, the dynamic module tests are more independent from the generated code. While the code analysis verifies how the module is performing its task, the dynamic testing mainly verifies that the module is performing the task. In this manner, the dynamic module test is almost totally independent from the program language used for code implementation; however, the requirements to drive the hardware environment (integrated sensors) of an avionic system have a strong impact on the test case and test data selection for the relevant modules.

Module tests in our definition may include actual or simulated modules at levels in the calling hierarchy different than the module under test. If additional actual modules are used for the test, they must be tested in advance. The allowed configurations are shown in Figure 5.

The test data sets are designed manually using the requirements of the Software Design Manual and included in the test driver programs. In most cases, the test drivers compare the program response with expected results. These expected values are calculated on a host computer using high order language implementation of the test subject to increase the level of diversification. Tools have been developed to unify the construction of the test drivers.

The definition of the tests, including test case selection, is traced using the Module Test Evaluation Matrix on the SDSS.

The results of the tests are traced and documented in the same manner as for the static analysis.

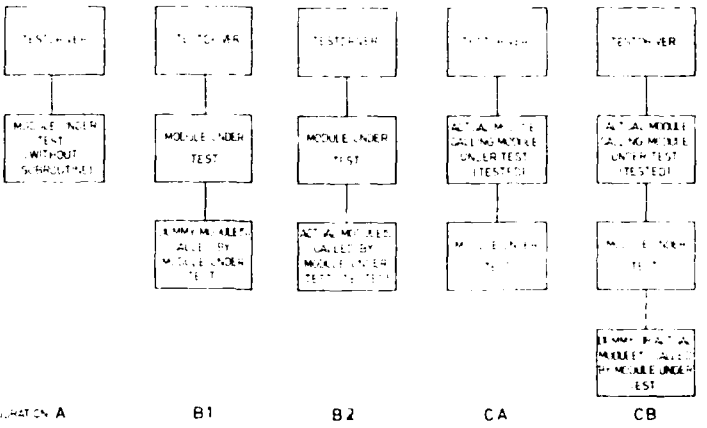


Figure 5: Possible Module Test Configurations
Test drivers include test data and expected results. The actual module in Configuration B2 has to be tested in a configuration other than C.

3.2.3 HARDWARE/SOFTWARE INTEGRATION TESTS

Hardware/Software Integration Tests verify the software performance at the system level. The complete program is implemented in PROM's in the real hardware environment. No restriction, modification or extension of the code is permitted for these tests.

The tests verify in particular

- the hardware-software interface
- the real-time behavior of the system
- test of module groups
- the basic systems mechanization
- self-test capabilities

The test equipment consists of various stimuli generators for all possible external inputs, recording facilities for normal outputs, and a real-time emulator and logic analyzer to trace the program flow in real time under operational conditions on code level (no "black box" testing). Hardware failures can be induced to check the self-test part of the software.

The test design is based on the Test Evaluation Matrix. To enhance the readability of the test procedures and to make certification and maintenance easier, the design of the tests is done in two steps:

A) The Top Level Design, which gives an overall description of each individual test with the expected results;

B) The detailed test procedure, which specifies unambiguously the complete set-up of the test equipment and all actions to be performed.

As mentioned before, the Test Design and Test Procedures are implemented on the SDSS for automatic update and documentation. For each individual test step defined in the Test Evaluation Matrix, one Top Level Test Design and one or several Test Procedures exist. The Test Procedures are identified by the Test Reference Number specified in the Test Evaluation Matrix.

4. RESULTS OF THE VERIFICATION PROCESS

As described in the previous paragraphs, the three different levels of program verification will normally detect errors on different software levels:

The static code analysis generally detects:

pure programming errors, missing or incorrect comments, violation of coding rules, etc.

The dynamic module tests detect:

errors in the bottom design of a module as incorrect dynamic behavior (overflow, underflow), errors in the implemented logic, inaccuracies in the arithmetic, etc.

The system level tests detect:

errors in the top level design of the software, the real-time behavior, and the hardware interfacing.

The results of the individual tests have been traced during the entirety of the verification process and are plotted below (Figure 6). The shape of the distribution is influenced by several facts:

A) The verification process started when the program and the documentation were already in their nearly-final states.

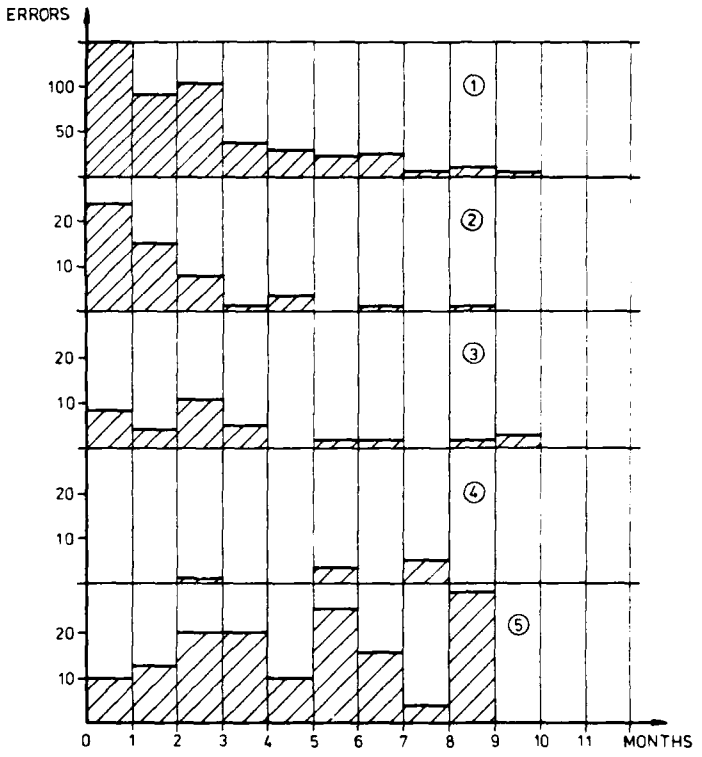
B) Several new software versions released during the verification process due to changes of the system's requirements (see bottom curve).

C) For each version, the tests were performed in the order:
   - static code analysis
   - module tests
   - hardware-software integration tests

The errors or inconsistencies in the documentation (Software Requirements, Software Design Manuals, etc.), module headers, comments, etc. take the majority of all findings of the verification process.

From the errors in the program, the main part is detected during the code walk-through procedures, followed by module tests and hardware-software integration tests.

The steadily decreasing error rate, even in the presence of numerous software changes, indicates the effectivity of the applied methods.



5. CONCLUSIONS

After completion of the verification process and thorough company and customer flight tests, the LTR-81 AHRS has been certified by government authorities for a A300 AIRBUS installation.

The systems have been in normal continuous installation for more than three months.

As a measure of the quality of the adopted design and verification methods, one can consider the number of program errors detected during operation after completion of the verification process. With a total estimated time of software operation of about 2,500 hours after verification, no program error has been detected. Thus, the successful application of the design and verification methods to a medium size assembler program has been demonstrated.

Furthermore, the special methods and tools developed for the verification control have proved to be time- and cost saving for the test design and maintenance. The implementation of the complete documentation on a computer system eases updates and enables rapid delivery of new versions.

After a critical review of the complete LITEF software verification process, the primary customer strongly recommended a similar approach to the manufacturer of an inertial navigation system, currently under development.

The developed tools will be expanded in future to cover larger parts of the design and in particular, the design verification processes.

Furthermore, test driver and test data sets for module tests will be extracted automatically from the design specifications.

REFERENCES

Aerospatiale	"Quality Demonstration of Airborne Computer Software for Certification," Aerospatiale Draft, Nr. 451 285/81, May 22, 1981.
ARINC	"Attitude and Heading Reference System," ARINC Characteristic 705, Annapolis, USA, Version 3, April 22, 1981.
BDLI	"Annehmbare Methoden fur Entwicklung und Nachweisführung von Software," Bund der deutschen Luftfahrtindustrie, Ausgabe 1.0, November 22, 1979.
EUROCAE	"Recommendations on Software Practice and Documentation for Airborne Systems," Final Draft by the European Organization for Civil Aviation Electronics (EUROCAE), Paris, October 1980.
M. Fujii	"Independent Verification of Highly Reliable Programs," COMPSAC 77, pp. 38-44.
W. Howden	"A Survey of Static Analysis Methods in Software Testing and Validation Techniques," IEEE Computer Society Press, 2nd Edition by E. Miller and W. Howden, 1981.
E.F. Miller and W.E. Howden	"Software Testing and Validation Techniques," IEEE Catalog No. EHO 180-0, Library of Contress No. 81-81431, 1981.
G.J. Myers	"Software Testing Principles and Practice," John Wiley & Sons, New York, 1976.
RTCA	"Software Considerations in Airborne Systems and Equipment Certification," Radio Technical Commission for Aeronautics, DO 178, prepared by SC-145.
UNIX	UNIX is a trademark of Bell Laboratories.

Progress in Verification of Microprograms

Stephen D. Crocker
Director, Information Sciences Research Office
The Aerospace Corporation
P.O. Box 92957
Los Angeles, California 90009
USA

Development of a usable, practical program verification system has been a goal for a number of years. Although systems are not yet available for general use, substantial work has been done by several groups and relatively powerful systems are coming into existence. We describe our own work on the development of a microcode verification system and give a picture of how such a system might be used in the near future.

Microprogram Verification

A microprogram is a program written for the interior or lower level of a two level machine. Although microprograms are just like other programs in many respects, there are certain distinctions peculiar to microprograms.

Microprograms usually implement the instruction set of an upper or target machine. The specification of the target machine is generally subject to precise specification. In contrast, formulating a precise specification for other classes of programs, e.g. interpretation of radar images, may be difficult and uncertain.

The architecture of the host machine is usually more complex than the architecture or programming language at higher levels. A large number of operations may be in progress at the same time, and some of these operations may take more than one cycle to complete. Characterization of the behavior of each microinstruction may be difficult. In this respect, microprograms tend be more complicated than other classes of programs.

Efficiency is a primary concern in microprograms because the microcode is at the heart of the execution of every program. Speedups in the microcode result directly in speedups of the overall system. This concern with efficiency leads to intricate programs which are hard to debug.

Because microprograms are at the heart of a system, assurance of their correctness is worth quite a bit. In this respect, microprograms tend to merit greater effort to assure correctness than other classes of programs.

Since correctly operating microprograms are a requirement for military applications and since microprograms are often very complex, application of formal program verification techniques to microprogram correctness was recognized as potentially quite valuable. Our group began work on this problem in 1977 at the University of Southern California Information Sciences Institute, and we continued this work when we moved to The Aerospace Corporation last year. The chief result of this work has been the development of the State Delta Verification System (SDVS).

Although the SDVS is not quite complete, it is reasonably close to its final form and has been used to verify several pieces of microprograms. A large scale test is scheduled for fiscal year 1983 and 1984. In operation, the verification system will be used in the following manner.

1. A formal description of the host machine is prepared. This description is written in ISPS [1] and captures all of the information needed to understand how any microinstruction will be interpreted on the actual machine.

2. A formal description of the target machine is also prepared. It specifies how the target machine is supposed to operate. It is written in ISPS.

3. The correspondence between the state information in the target machine and state information in the host machine is worked out by the microprogram designer. This correspondence covers such things as which register in the host machine will hold the program counter for the target machine.

4. The microprogrammer then writes the code and annotates the code with his reasons for believing that each segment of the microcode implements particular aspects of the target machine. These annotations are an important part of the input to the verification system.

5. As the microprogram is developed, the microprogrammer uses conventional tools such as an assembler and simulator to check out the code, but he also submits the code to the verification system. The verification system checks that the microprogram implements all aspects of target machine. To carry out this check, the verification system uses the description of the host machine and the correspondence between states to compute the effect of every sequence of microinstructions. The annotations in the microprogram provide guidance to the verification system to see what case analysis and what induction steps to carry out. Discrepancies between the simulated behavior and the required behavior are flagged for the microprogrammer to consider.

State Deltas

In order to build a verification system, there must exist a clear theoretical basis for reasoning about the behavior of programs. We have chosen to use a formulation of machine behavior based on "state deltas" [2]. In the state delta formulation, machines are viewed as sequential engines that step through a large number of states. At each state, the state description vector is potentially very large, and is best thought of in terms of current values of access functions. For example, in the course of interpreting a single microinstruction, a typical machine might step through a few dozen tiny steps. The state space consists of the main memory, the control store, the registers, and the internal control sequence status. Each step causes a change to only a small part of the total state space.

Since we expect to work with a large number of state transitions and we expect each transition to affect only a small fraction of the total state space, we focus our attention on changes to the state space over intervals of state transitions. Formally, a state delta is a formula that relates the differences between three states, known as "now", "precondition time" and "postcondition time." A state delta formula, written schematically as $_{E}[P \rightsquigarrow_{M} Q]$, has four parts, an environment list (E), a precondition formula (P), a modification list (M) and a postcondition formula (Q). The meaning of a state delta is

> If between now (the present state of the machine) and some future time, the precondition becomes true before any of the places listed in the environment list are modified, then there will come a yet later time at which the postcondition is true and (at most) only the places listed in the modification list will have changed.

The precondition and postcondition are logical unquantified formulas using the usual predicate connectives (and, or, not, etc.) and the usual arithmetic (+, -, *, etc.) and logical (bitstring-and, bitstring-or, field selection, array element selection, bitstring concatenation, etc.) operators over terms. References to values in the precondition state are made using a prefix "." operator. In the postcondition, references to the new value of a place are made using a prefix " # " operator.

We translate the ISPS machine descriptions mentioned above into state deltas. For example, the ISPS fragment

```
A ← MEM[PC]
```

translates (approximately) into the state delta

$$[\rightsquigarrow_{A}\ \#A=.MEM[.PC]\ ]$$

except that in the actual translation there is a representation of where this action fits into the sequence of actions. *These translations are then used inside the verification system as the formal basis for the proof.* (In the actual implementation, the translation is carried out incrementally and is interwoven with the actual proof process, but the effect is the same as if the translation had been carried out previously.)

The primary advantage of state deltas is that attention is focussed on the changes that take place during segments of computation. This permits great economy of expression in the characterization of the behavior of a machine.

The State Delta Verification System

As described above, the role of the verification system is to accept descriptions of the host and target machines, a copy of the microprogram, a mapping between the state space at the host level and state space as viewed at the target level, and the annotations by the programmer. Having accepted these inputs, the system then checks whether every required aspect of the target machine is indeed implemented by the microprogram as it would execute on the host machine.

Our verification differs from many others in that *it does not try to intuit the proof* without aid from the microprogram designer. Instead, we ask only that the verification system follow the rationale supplied by the designer. *In principle, this reduces the key element of the verification system from a theorem prover to a mere proof* checker. Although there is a large theoretical difference between a theorem prover and a proof checker, practical considerations tend to blur the distinction. Theorem provers are necessarily incomplete. This means that there are always theorems that are true but beyond the power of the theorem prover to prove. In practice, this means that theorem proving programs have to be interactive and accept hints or other help from the user. This help serves the same role as the formal proof supplied to a proof checker, although it is often assumed that far less help is required for a theorem prover than for a proof checker. At the same time, proof checking programs are often augmented with

algorithms to prove various classes of lemmas automatically, thereby reducing the length of the proof needed from the user. The result is the clean theoretical distinction between a verification system that uses a theorem prover and verification system that uses a proof checker may not be so clean in practice. Nonetheless, the emphasis remains a bit different, and we have specifically chosen the proof checker style of system design.

The verification system is implemented with the following major components.

Proof Language Interpreter
: The annotations supplied by the microprogram designer are interpreted by this component to guide the proof process. The key elements of the proof language are directions to symbolically simulate until specific state is reached and decisions to initiate subproofs, case analysis proofs, and induction proofs.

ISPS Translator
: As mentioned above, the descriptions of the host and target machine are written in ISPS and translated into state deltas within the verification system.

Simplifier
: A large fraction of the verification work consists of normalization of expressions and other completely straightforward arithmetic and logical simplification. The simplifier consists of a number of cooperating decision procedures that see all of the consequences in specific areas. For example, there is a congruence closure algorithm that determines the truth of any formula that is related to other formulas only through substitution of equal terms. There is also a component that determines whether a conjunction of linear inequalities is satisfiable. The simplifier is based primarily on the work of [3], but we have extended to work with bitstring arithmetic, set partitions, bags, sequences and multiplication and division of integers. For some of these domains, only partial decision procedures are possible, and we try to characterize for the user exactly what the simplifier is capable of proving.

Symbolic Simulator
: This component maintains state descriptions, applies state deltas and steps through time from one state to another.

The SDVS is written entirely in Interlisp and currently runs on both a Digital Equipment Corporation VAX 11/780 and a Xerox 1100 ("Dolphin") Lisp computer.

Experience

Although the system is not quite complete, we do have some experience with earlier versions of the system and can give some idea of how expensive it will be to use verification tools in the future.

Formal descriptions of the host and target machines have to be written. Our experience is that it takes between 6 and 12 months to write a full and accurate description of a medium sized computer. A large fraction of the time is spent on the details, since the documentation is usually incomplete in various ways. As part of this process, we have found it essential to have easy access to the designers of the machine in order to get answers to a myriad of questions. In our most recent experience, we have enjoyed access to the machine designers via the Arpanet, and this has been ideal. Responses have been very timely and the use of the system allows us to capture all of the interactions almost automatically.

The resulting formal descriptions range between 20 and 50 pages, depending on the complexity of the machine. We have adopted the policy of writing these as perspicuously as possible and using the formal descriptions for documentation of the machines and automatic development of emulators for the machines as well as for input to the verification system. These added uses of the formal description do not materially affect the time or cost to write the description, but they do materially enhance the utility of the result.

A natural result of the description phase of a verification project is that some errors are noticed and corrected. Our experience is that the majority of the errors detected at this time are simply discrepancies between how the machine works and how it is documented. The useful effect is that the documentation is brought up to date. However, detection of mistakes in machine design or microprogram design may be detected during this process.

Errors detected as the result of formalizing the machine descriptions are sometimes the source of a small philosophical controversy. Depending upon one's point of view, these may or may not "count" for credit as "errors detected through formal verification."

Turning to the actual proof, an interesting question is how extensive must the annotations be? A potential hazard is that the annotations might have to be several times as long as the microprogram itself. If this were so, there would be strong resistance to use of a verification system. Fortunately, our experience is that the annotations are reasonably short. For example, it often suffices to direct the system to simulate execution of the microcode until the goal of the subproof is reached. The simulation may span several microinstructions, yet only the single command "*" is needed to direct the system. In general, our experience confirms Wegbreit's findings [4].

The speed of the verification system is always of concern. In the present form, we estimate that it is now capable of simulating approximately one thousand microinstruction per hour. This is slow, but satisfactory for our current work. If the system were to be turned into a production system, there are a number of refinements in the implementation that would be appropriate. These refinements include rewriting the system in an implementation language and precompilation of the semantics of each microinstruction. Other refinements will be discovered as the testing and use proceed. Taking all of the refinements together, a complete rewrite of the system is likely to yield about two orders of magnitude improvement. Execution at that speed would bring it into the range of ordinary compilers and assemblers and permit its use within a production programming environment.

References

[1] Barbacci, Mario R., Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek.
The ISPS Computer Description Language.
CMU-CS-79-137, Carnegie-Mellon University, Computer Science Department, August, 1979.

[2] Crocker, Stephen D.
State Deltas: A Formalism for Representing Segments of Computation.
PhD thesis, University of California, Los Angeles, 1977.

[3] Nelson, Greg, and Derek C. Oppen.
Simplification by cooperating decision procedures.
ACM Transactions on Programming Languages and Systems 1(2), October, 1979.

[4] Wegbreit, Ben.
Constructive methods in program verification.
IEEE Transactions on Software Engineering SE-3(3):193-209, May, 1977.

VALIDATION OF SOFTWARE FOR INTEGRATION OF MISSILES WITH AIRCRAFT SYSTEMS

J. R. McManis

Naval Weapons Center
China Lake, CA, USA

SUMMARY

The integration of computerized missiles with the U.S. Navy attack aircraft has resulted in increased complexity of requirements for validation testing of the software for the aircraft's embedded computer systems. Also the cost and availability of these weapon rounds prohibits the widespread utilization of the actual weapon in live firings as a means to support system software validation. Yet it is precisely because of these same factors that the integration of these weapons with the attack aircraft system must be validated to the highest degree possible. In the course of the validation it must be proven that the whole system performs to the functional specifications and that, just as importantly, the software does not cause undesired behavior under normal or abnormal conditions. A methodology for validation, with emphasis on the aircraft system software, is described for areas critical to assuring that the system does meet its requirements.

1. INTRODUCTION

The possible adverse effect of anomalies in avionics software can be significantly greater than that in other software. For example, it is conceivable that a weapon could be inadvertently launched, or the aircraft catastrophically lost either causing consequent loss of life. In contrast, an error in business or financial data processing may seem catastrophic, but is actually only an irritation or inconvenience to the users of that software.

This paper addresses critical aspects involved in the validation of avionics software developed to integrate missiles with attack aircraft systems. The situation being covered here is where the missile and the aircraft both have embedded computer systems, and that they have evolved to their current state in separate and totally independent development efforts, such as the A-6E and the Harpoon missile shown in Figure 1.1. Also, to narrow the scope of this paper, the essential existence of verified system and software requirements is understood. The system and software requirements must also be sufficiently precise to provide the basis for design as well as subsequent verification and validation.

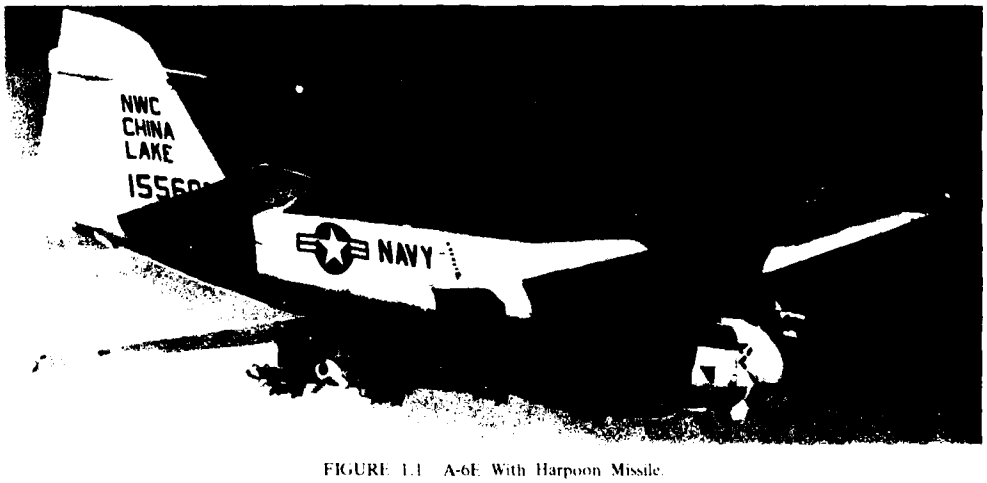


FIGURE 1.1 A-6E With Harpoon Missile.

2. GENERAL INFORMATION

2.1 Software Validation

The goals of software validation are to determine whether all software and system performance (specified through functional, interface, and test requirements) are being satisfactorily fulfilled, and that the software does not cause unexpected behavior under unforeseen circumstances. To achieve this, software's contribution to performance must be evaluated in a realistic operating environment where hardware, environmental, and personnel effects are in the loop when required (REIFER, D. J., 1978). It must also be understood that validation is not testing of requirements, but rather testing against requirements.

Software development actually results in parallel software developments as shown in Figure 2.1. After the requirements definition for the system and its software, and before the system software validation testing can commence, parallel efforts to modify both test facilities and software tools for data recording and analysis are required.

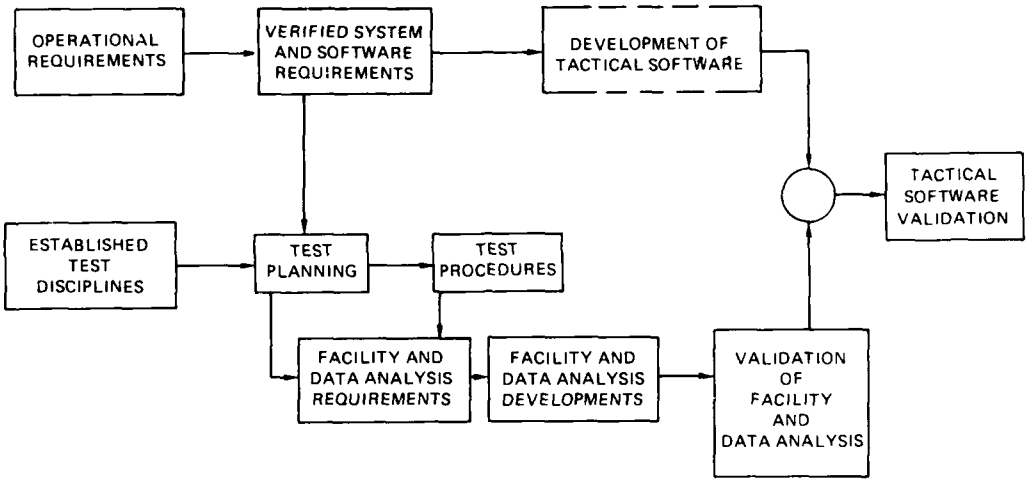


FIGURE 2.1 Parallel Developments Leading to Validation

2.2 Embedded Computer Systems

An "embedded computer system" is a computer system that is physically incorporated in a larger system whose primary function is not computation. An example can be envisioned as given in Figure 2.2. Aircraft and missiles are examples of these larger systems individually, or when they have been integrated into a single system. It is also important to remember this relationship during the process of validating the embedded software, and to be aware that the requirements of the larger system are paramount in the design of the real-time software (GLASS, R. L., . . ., 1980).

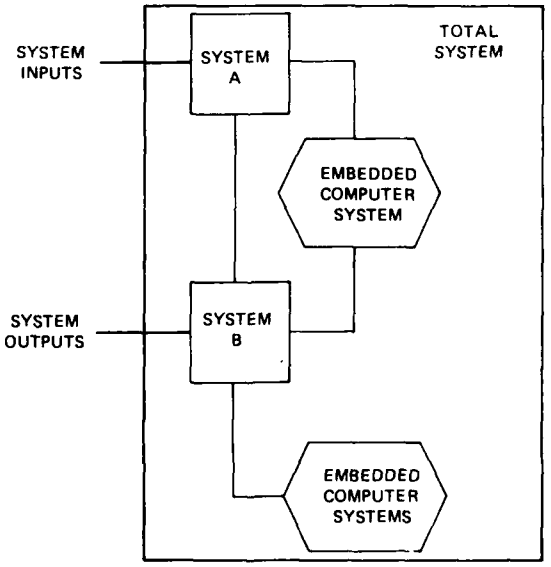


FIGURE 2.2 Embedded Computer System.

Severe restrictions are imposed upon the embedded computer systems in aircraft or missiles with respect to weight, size, power, environment, and accessible information. These restrictions, along with the need to have software that is essentially a black box to the aircrew (the actual users of the software), restrict accessibility to the software operating details. The difficulties and expense of retrofitting completed aircraft, compared with the relative ease of making changes in software, has resulted in an ever increasing proportion of system capability being assigned to software resulting in greater and more complex software requirements.

Also the ever increasing demands for expanded software/system requirements, along with the seemingly inherent shortage of time and memory for embedded real-time systems, have prevented the inclusion of those test and debug capabilities that are normally present in modern software systems.

3 SPECIFIC TOPICS

The topics discussed here have been separated into five major areas that comprise validation: requirements, design, test environment, test disciplines, and test planning.

3.1 System Software Requirements

This paper will not attempt to elaborate on the topic of requirements other than stating that they must exist in a verified state. However, their importance is too great not to be mentioned here. The lack of verified requirements for any software project definitely prohibits the achievement of thorough validation testing for completeness and correctness of that software. The completeness of the performance specification also determines the extent to which testing will provide assurance against unexpected events, in addition to testing the stated requirements as shown in Figure 3.1.

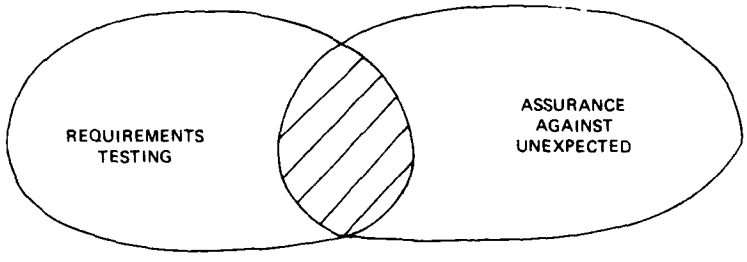


FIGURE 3.1 Magnitude of Overlap Determined by Completeness of Performance Specification.

3.2 Software Design

Software design for the aircraft computer in an aircraft missile integration project usually involves a relatively small amount (10 to 25%) of new design requirements and a lesser amount of modifications of the baseline (RANG, E. R. ..., 1979). Even this amount however is *sufficient to prohibit* rigorous proof of all possible operations of the modified software, and this testing is in addition to that required to validate the baseline software. Thus the requirement for thorough validation indicates that the software design should include considerations for testing to be able to get assurance that the validation is reasonably comprehensive, and that the system will not do undesired things under unforeseen circumstances.

3.3 Test Environment

3.3.1 Facilities

To determine the requirements for a facility, one must first realize the purpose for the facility. In this case it is "To provide *a tool* to test Operational Flight Programs (OFP) in a manner such that the OFP will operate in all respects, as if it was being operated in the aircraft system." It should be noted that the facility is probably envisioned to be used throughout the OFP software development process for: requirements analysis, design, coding, debugging, integration, and validation. The facility is *not* a flight trainer, but a tool for testing OFPs.

To allow this testing to be meaningful, the design of the facility must have the following characteristics.

(a) The simulation must contain valid models of all systems that interface with the OFP, with integrated drivers for these models. While some models need only to represent the reactions of the systems to simulations from the drivers, others must operate in a closed loop fashion with both the driver and the OFP.

(b) Access to data (both OFP and models) for recording throughout the operation of the facility is *essential in testing* the accuracy and correctness of OFP operation.

(c) The ability to repeat runs in the facility is required for comparison of tests of different OFPs, as well as to obtain repeatable phenomenon.

In order to provide the ability for validating built-in tests and degraded operation, the simulation software (in addition to *providing an* accurate representation of the system and its environment) should provide the capability for introducing system fault, anomaly, and off-nominal conditions.

Validation facilities should include a full complement of *the computer subsystem equipment*, including the operator interface (keyboards, switch panels, etc.). Everything else should be simulated, but with provisions made to allow substitution of actual interfacing compatibility. This requires that the system and environment simulation operate in a computer external to the operational system computers enabling the operational software to retain its integrity (OSBORN, L. A., ..., 1977).

The following characteristics are only necessary for some forms of testing.

(a) Actual system hardware (even the aircraft computer itself) is required only for hardware-software interface testing and for troubleshooting on some problems.

(b) Operation of the facility in less than a fully automatic sense is not desirable for validation testing, but a person in the loop is required for development or troubleshooting tests, where the person-in-loop mode of operation provides greatly increased flexibility.

(c) Real-time operation of the facility only becomes necessary for use of actual hardware, and near real-time operation is all that is required for person-in-loop use.

3.3.2 Data Recording and Analysis

The critical areas requiring concentrated effort for data recording and analysis are (1) to process all data available, (2) to automate as much of the data processing work as possible, (3) to have the data analysis programs produce output in a form suitable for inclusion in reports, rather than spewing forth reams of paper. This process is diagrammed in Figure 3.3.2.

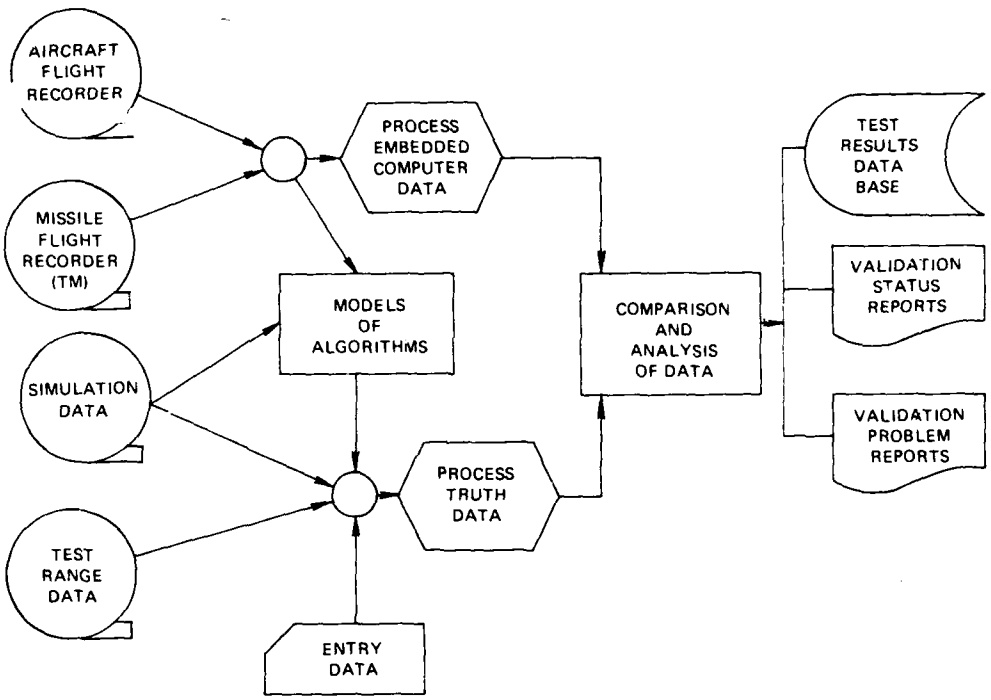


FIGURE 3.3.2 Process of Analysis of Test Data.

To completely satisfy the requirements for the first area requires the availability of some kind of truth data (such as from the simulation or range facility). This simulation or range data needs to be compared with the flight recorder data to determine the accuracy of the OFP formulations. The simulation data should also be compared against the flight profile plan to determine if the correct profile was flown.

The limitations on the accessibility of data from the embedded computers place a burden on the OFP software designer to make as much data available as possible, and to minimize redundancy in the data available. The burden on the developer of the processing and analysis tools, is to utilize all data available and to be able to test all specified functions.

The second area mentioned is critical if we are going to make validations both thorough and efficient. Manual scanning of data cannot possibly be as efficient and thorough as automated or computerized methods. Given both truth data (simulator or range) and flight recorder data, the differences can be compared with predetermined (constant or variable) tolerances to check for problems. Where tolerances are not met, validation problem reports could be issued. Where the comparisons come within failing a predetermined percentage of the tolerance, a notice could be issued to have an analyst check the data for possible problems. For some functions truth data will not be available from the simulator or range facilities. For these, software models of the algorithms could be driven by flight recorder data to generate the truth data for comparison. A weapon delivery scoring program is valuable in that it not only can determine accuracy for each weapon delivery, but also maintain a data file of the accuracy achieved with respect to the runs and modes selected. This data file can then be used to compute statistical data reports such as circular error probabilities for various modes and or weapons throughout the validation.

By computerizing the data checks, functional groups can be set up, e.g., to check Navigation (where all position and velocity data would be checked), to check out radar functions (where the radar ranging variables are checked out), etc. The test procedures could then have a check list to select the processing programs that need to be run to check out the functions of interest for that run.

The third area to be worked on for data processing is to minimize output of the programs and to present the results in formats designed to ease writing of the intermediate and final reports. The ouput of these processing programs would vary in format depending on the data being checked. It could be in the form of plots, tables of differences, or a list of problems and notices. This would improve the visibility as to status of a validation and keep the reporting aspect of validation work to a minimum of effort.

3.4 Test Disciplines

Throughout the course of the validation, there are several disciplines used that should be similar to those used for other validations. The more important disciplines to be applied include: test documentation (plans, procedures, and reports); configuration management; quality assurance; and error processing. Each of these disciplines should be firmly established in organizational policy before test planning occurs.

3.4.1 Validation Documentation

A complete set of validation documentation would include a plan, a set of procedures, and the reports on test results. A standarized validation plan that provides the details on the baseline functions of the OFP is a desirable starting point for the modifications to produce a validation for the integration project.

A complete set of test procedures can be written only if an adequate performance specification is available. Also the process of writing the procedures should attempt to be a direct translation from the requirements and test information in the performance specification. To ease this process, software tools could be developed that would allow the manual effort to consist of a menu selection style of extracting the test information from the performance specifications and designing procedures to test the specified requirements. This would greatly speed-up the process of writing procedures and would eliminate many errors commonly found in test procedures. The program to decode the procedure menu selection sheets could also produce software files that drive the simulation facility through preconceived maneuver profiles.

In addition, a data file could be set up that would contain a list of functions tested by each run (procedure). This (when used by an appropriate data management program) would allow computerized recording of the functions tested and those that have had problem reports written. This data management capability would allow summaries of progress to be obtained at any time in formats that show what has been tested, and what has yet to be done.

The menu selection forms for the procedures could also specify the types of data processing that should be conducted after the run. More on what is meant by "types of data processing" is given in paragraph 3.3.2.

The procedures produced for use in the simulation laboratory should be adaptable (if not immediately usable) for use in flight tests. Perhaps a short form could be produced for each complete procedure that would basically be a translation of the menu selection style of forms without the details required for facility operators who may not have the familiarity with the aircraft system and its operation. The short form could also produce the kneeboard cards necessary for notes and entries during the flight. This would provide the means for integrating the tests in the simulation laboratory with the tests in the aircraft.

3.4.2 Configuration Management

An automated software configuration management capability, available on a central operational software development facility with interactive terminals, has become necessary due to the increase in size and complexity involved in avionics software. This would include a computer program library for a central repository and control agency.

3.4.3 Quality Assurance

A good reporting scheme is critical to the management of a software development project. A good way to accomplish this is to maintain an independent quality assurance function that will provide visibility as to validation status to the project managers. This would include information on how many tests were attempted, how many were completed, how many were successful, partially successful or failed.

3.4.4 Error Processing

Effective procedures must be in place to handle the processing of errors detected during validation testing. These procedures must cover classification of the error as to impact, cause, effort required for correction, and retesting required. In essence, the procedure tells "what to do" when the validation testing is perturbed by discovery of errors.

3.5 Test Planning

The success of validation testing is determined at the outset of the development project. Only by alloting the validation effort its fair share of resources and time will it be possible to assure that the developed software achieves its validation goals. However, by letting a validation program evolve subject to the vagaries of development problems and slips in schedule gives very little confidence that the validation will be successful. The test objectives must be set and agreed upon by project management, the development group, the test group, and the user/customer. Test requirements should then be developed based on the objectives, goals, and the planned development approach. An effort to have in place, and acquire as necessary, appropriate tools, based on the requirements and the plans is required. An arsenal of tools and test methodology should be selected that will achieve the goals of validation. A validation plan that provides appropriate interfaces throughout the testing should be prepared that represents a well-thought-out test approach to enable comprehensive testing (HARTWICK, R. D., 1977).

4. CONCLUSIONS

The conclusions to be drawn here are primarily separated into the same areas as discussed under Specific Topics.

4.1 System Software Requirements

Though this paper did not cover the topic of requirements, other than stating that they must exist in a verified state, their importance cannot be emphasized enough. The lack of verified requirements for any software project definitely prohibits the achievement of thorough validation testing for completeness and correctness of that software. Also, any function stated to be a requirement must be amenable to being tested. If it cannot be tested as defined, either it is not a requirement, or it is defined improperly. The use of available requirements evaluation tools will greatly aid in achieving the correct and thorough requirements specification.

4.2 Software Design

The limitation on accessibility to data and the limited debug capabilities in embedded computers can be partially overcome be requiring the software design to utilize the access capabilities (such as flight recorders or laboratory breakout boxes) for providing the maximum data possible with a minimum of redundancy. Also, the data output should be selected to attempt to enhance automation of the data analysis programs that would greatly increase the efficiency of reviewing test data and allow more thorough testing with less effort.

4.3 Test Environment

The requirement that at least part of a validation be performed in a realistic operating environment demands the availability of a real-time simulation of the aircraft and missile environment and other system elements outside the operational computing subsystems. The more accurately the system environment and, in particular, the computing system interfaces with other system elements can be represented, the more confidence can be placed in the developed software.

4.4 Test Disciplines

The complexity of the system and software to be tested requires a formal and disciplined approach to validation. Each of these disciplines should establish organizational policy before test planning begins so that each project does not "reinvent the wheel".

4.5 Test Planning

As much effort should be devoted to the preparation of good test plans and procedures as was devoted to the generation of the performance and design requirements for the software. These plans and procedures are essential not only for thorough testing, but for troubleshooting problems that are detected during validation so as to determine the state of the software when the problem occurs and when it does not.

IMPLEMENTING HIGH QUALITY SOFTWARE

E J Dowling

Ferranti Computer Systems Ltd
Ty Coch Way, Cwmbran
Gwent, UK

SUMMARY

High quality software is essential in applications such as avionics. Each stage of development (specification, design, etc.) must be validated, but this paper concentrates on the implementation process. Various factors affecting implementation are discussed and some solutions are considered, in particular the range of tools that is available, and the way they can be applied. A set of tools that has been developed within FCSL, and the method they are designed to support are described. The traditional debug program is shown to be only one part of the whole tool set. Finally, the advantages and problems of Ada and its environment (APSE) are discussed.

INTRODUCTION

Ferranti Computer Systems Ltd is part of the Ferranti group of companies, which has a broad range of interests including avionics hardware and software. FCSL is a major British Ministry of Defence contractor and part of the work described has been supported by the Procurement Executive of the MoD. This paper discusses the problems involved in implementing high quality software for applications such as avionics, and some of the methods and tools developed within FCSL to help overcome these problems. No claim of originality is made for much of the contents, although the use of some of the tools seems to be far from universal. Rather, the aim is to consider the problems and some obvious (?) solutions.

To some extent, the definition of "high quality" software depends on the application to which it is to be put and possibly subjective judgement. However, the attributes of high quality software will normally include:

(a) meeting requirements on functionality and performance

(b) cheap to produce and maintain

(c) modular

(d) portable

(e) well structured

Many of these qualities depend on stages earlier than implementation, but the implementation process can have a dramatic effect on most of them.

The paper is concerned only with the final stages of producing a system - ie. its implementation. Of course, the earlier stages are just as important if the end result is to be a system that does what was originally intended. Numerous tools and methods have been developed to help ensure that the original requirements for the system are understood and fully specified in an unambiguous way. (DoI 1981a).

Once such a specification has been produced, similar tools and methods cover the design stage, aiming for a design that is complete, consistent and conforming to the agreed specification. A goal for the future is to have automatic code generation based on formal specifications and designs, but the demands of real-time systems such as in avionics would seem to make such an approach very questionable. However, a hybrid scheme based on hand-tuning the automatically generated code could be acceptable.

In general, these more rigorous methods for the earlier stages of software production appear to offer promise, but their use in complex,real-time systems, by "average" staff seems to be a thing of the future. In contrast, this paper is concerned with more practical, or heuristic, methods and tools for the implementation stage, assuming the design has already been produced and is consistent, complete, etc. The validity of such assumptions is not questioned here!

THE PROBLEMS

This section of the paper describes the problems associated with producing software for systems such as those in avionics. The overall aim of course is to ensure that the software meets its specification (which here is assumed to be correct, well-defined, etc). However, several more specific aims can be identified.

For the purposes of this discussion, the system will be assumed to consist of concurrent "activities" which interact dynamically, that is they are tasks/schedulable items/processes, etc. The activities are made up of "modules" which only interact "statically" (the definition of a module is left deliberately vague). The overall aim can then be thought of as ensuring that:

(a) each module meets its specification

(b) all modules in an activity fit together properly so that the activity meets its specification

(c) the activities fit together properly to form a system which meets its specification

The above steps represent a natural set of stages in the testing of a system. At each stage, there will typically be functional requirements for the software to meet, but also performance requirements such as speed, response time, size etc. Finding whether these requirements are met may be complicated by several things, for example:

(a) the software runs on a small machine with no "normal" peripherals (backing store, terminal, etc).

(b) the system includes peripherals that are not available yet (or at least not at the site where it is best for the majority of testing to take place).

(c) the processor itself, or some peripheral, is unreliable (because it itself is still under development), so that the hardware is being tested as much as the software is, and it is not always obvious where the fault lies. This sort of problem is clearly most common within hardware manufacturing organisations.

(d) the hardware configuration includes multi-processors.

These difficulties may be compounded by factors over which more control should be possible. Such factors include:

(a) the number of errors introduced during coding

(b) how easy the code is to understand (for debugging)

(c) how well the testing at each stage is carried out, ie. how much confidence there can be at any level that a new error discovered is at this level and not from a previous level (but did not get found when testing that level).

The source of these problems can be seen by considering the following example. Given an overall design, assumed to be "perfect", the software might be implemented in the following way:

(a) detailed designs are produced using flowcharts

(b) the flowcharts are coded into a low level language (assembler), in an unstructured way and without comments

(c) some module testing is done to make sure each is "roughly right", using a "peek and poke" low level debugger.

(d) real testing is left until the "roughly right" modules are combined into much larger units; again, a low level debugger is used.

Of course, at stage (d) all sorts of static (intra-activity) and dynamic (inter-activity) errors come to light and have to be isolated, using the low level tools available. Any problems due to lack of peripherals, faulty hardware, etc. mentioned above only add to the difficulties. In practice, this hypothetical worst case may well be improved by procedures such as:

(a) code reviews take place to ensure project standards on structure, comments, identifiers, etc. are observed

(b) each module is tested as fully as possible, perhaps "completely"

(c) each activity is tested statically before any dynamic interaction of activities is investigated

Such procedures raise the worst case to a level at which, no doubt, some systems are produced, but there are still a lot of problems due (to varying degrees) to:

(a) the use of flowcharts

(b) the use of a low level language

(c) human fallibility in the review processes, and the cost of such effort

(d) the lack of suitable tools for testing

(e) the lack of any indication of how well each module has in fact been tested

Again, all these problems are on top of those due to lack of peripherals, etc.

Thus two classes of problems can be identified : those such as lack of peripherals, etc. ("system problems") and those such as coding errors ("implementation problems"). Even if there are no system problems at all, the implementation problems may make the software expensive, yet unreliable. Similarly, system problems will give difficulty no matter how the implementation problems are solved. On the other hand, the two classes of problems are closely related; for example, if the memory size available is very restrictive, a low level language for implementation may be unavoidable, no matter how undesirable this may be.

The following section expands further on these implementation and system problems, and discusses some possible solutions to them. A wide range of solutions is covered, but the next section includes details of a practical subset used within FCSL.

SOME SOLUTIONS

Design Languages

Traditionally, flowcharts have been used for software documentation for both the design and maintenance phases. There is a strong body of feeling that because of their pictorial nature, they are the best way of representing the information they are intended to convey. However, recent experiments (Schneiderman 1982) have shown that of several alternatives available for maintenance documentation, flowcharts were the least successful for use in understanding existing code. (Having detailed descriptions of the data areas involved and the data transformations performed turned out best).

For use as design documentation, produced before the code and on which the code is based, flowcharts have some serious drawbacks. The basic problem is that it is generally not at all easy to tell if the flowchart is well structured; especially since by their nature they tend to spread over several pages. All too often, the choice can be between coding from the flowchart and unavoidably using a GOTO, or producing structured code that differs from the flowchart, thereby invalidating perhaps some of the quality assurance procedures (if the flowchart has been inspected and approved).

Documentation is only of real use if it is up to date and this will be achieved most easily and reliably if it can be done automatically. Here again flowcharts present difficulties since they are not in a directly machine-readable form and making modifications and additions requires some quite sophisticated graphics capabilities.

An alternative to flowcharts is a design language, which may take many forms. Its general purpose is to represent the logic of the software at a level higher than that of the code itself. The information conveyed by the design language (DL) may be purely concerned with control flow (ie. the sequence of operations to be carried out) or it may include data flow (ie. how specific data is modified by the software).

Programming languages such as Ada may be used as a DL and such a use allows information on data encapsulation etc. to be included, as well as automatic checks for consistency etc. to be made. On the other hand, if only control flow is considered, a much less formal DL is needed, only requiring the usual structures such as loops, case statements, conditional statements, etc. The more formal approach allows automatic checking to take place and possibly direct derivation of source code from the DL (by macro expansion, for example); a less formal approach makes the DL more flexible, allowing the designer greater power of expression without the need to worry about rigorous syntax rules if the text is to be understood (as opposed to processed) only by a human, not a machine.

Whatever exact form it takes, a good DL has the following advantages, corresponding to the weaknesses identified in flowcharts:

(a) it is machine readable

(b) it is compact

(c) its form makes it immediately obvious whether the design is well-structured or not (indeed, the DL need not contain a GOTO)

Of course, an unstructured DL design is worse than a structured flowchart and both can be used well or badly, but the gap between a flowchart and the final code is usually greater than that between the equivalent DL design and the code. It is this code that is the end-product of the whole activity, and an approved design in DL has a better chance of resulting in correct, well structured code than does a flowchart.

Programming Languages

The advantages of high level languages (HLLs) over low level ones (ease of writing, maintaining and understanding) are well known and need not be expanded upon here. Their disadvantages (resulting in relatively slow and large programs)are potentially serious in applications such as avionics. However, the degree of handicap involved depends greatly not only on the efficiency of the compiler, but also on the quality of the high level language code. Just as most low level languages have associated "tricks" to obtain optimum efficiency, so there are usually particular ways of using even a standard high level language to improve performance. Of course, the value of these techniques must be judged against any loss of clarity they involve (although this might be negligible, depending on the methods used; the "best" way of expressing some logic in an HLL is often the most efficient).

Where even the most efficient HLL-derived code is not fast/small enough, it will be necessary as a last resort to use low level (assembler) coding. However, even so, the HLL can be considered as the main implementation language with small parts produced in assembler. Where the HLL permits, the assembler may be embedded in the HLL source, but it can be segregated or hidden (in dedicated procedures/ subroutines, by macros, etc) so that the reader is presented with uninterrupted high level source. The parts of the code which would benefit most from use of assembler may be identified by use of a tuning tool as described below.

More recent HLLs (eg. PASCAL and Ada - more fully discussed below) offer even greater advantages over low level languages. Not only do they provide the usual control flow facilities (to allow structured programming, etc), they also contain strong data typing. Used properly, these facilities should result in many more errors being found at compile-time rather than during testing (or even later!). Undoubtedly, getting programs to compile will take longer, but the results are well worthwhile.

Code Auditors

The review process described above can be automated to some extent by the use of code auditors to scan the source and check adherance to project standards for naming conventions, presence and form of comments, etc. Of course, the helpfulness and correctness of such comments can only be assessed manually. Similarly, there may well be project standards that proscribe certain valid, but undesirable (eg. GOTOs) or dangerous (in some way), constructs in the programming language and a code auditor can be used to check for these. Standards for source layout may best be enforced by using a formatting program (pretty printer) to process the source (although there are arguments for and against such a tool).

If a sufficiently formal design language is used, similar tools can be used for both design and implementation source, but in both cases the value of these tools is necessarily limited (eg. to checking if a comment is present, but not whether the comment is useful). Nevertheless, they can make a significant contribution to the efficiency and effectiveness of the processes involved.

Static Test Methods And Tools

A variety of methods and tools exist (in various stages of development) for use in the static testing of programs; all are static in that the software is not executed, but rather the source is analysed in one way or another. This feature makes static methods largely machine-independent.

Complexity measures, etc. Obtaining a complexity measure for the code is closely related to code auditing, but since the risk of errors is directly proportional to the complexity of the code, the process is also useful as part of the testing process, if only to identify the most complex (and therefore most error-prone) parts of the software.

Various techniques have been developed, ranging in sophistication. Simple measures may be based on a count of the number of particular language features (loops, conditional statements, etc). Alternatively, the number of "knots" may be counted (ie. the number of times control paths through the code cross). A program adhering to the usual structured programming rules will contain few knots. If the source is analysed to obtain its representation as a directed graph (with "blocks" of code as nodes and control flow as arcs), various node reduction techniques can also be applied.

Formal Proof of Correctness. Given a specification of a program's intended behaviour in a suitable form it can be possible to prove mathematically that the code meets this specification. Such a proof is obviously very valuable to have. However, there is still the problem of whether the program when it runs will do what its source suggests (hardware or software (eg. compiler) errors and limitations may prevent this). Although some tools have been developed to help, obtaining the formal proof in the first place may well be more difficult (and so error-prone) than writing the program.

Data Flow Analysis. A relatively common error is anomalous data flow (usually reading from a variable before it has had a value assigned to it), although this is perhaps more likely to happen by accident in languages allowing implicit variable declarations (eg. FORTRAN). Data flow analysis aims to identify such errors, although it is usually better done by the compiler than a separate tool (a remark that applies to several other techniques).

A common problem with static methods is that of aliasing and variable equality. For example, consider the code

```
PROCEDURE (I,J);
A[I]:= 0;
K := A[J];
```

If within PROCEDURE, J is set equal to I, A[J] is defined and there is no error. However (in the absence of any similar code elsewhere), if J is not set up within PROCEDURE, the assignment to K is invalid. A data flow analyser must make some decision about whether to flag this code as erroneous (although in this case it could be argued that the code deserves to be questioned because of its obscurity). The form of procedure call in some languages (eg. Ada) helps in this area.

Symbolic Execution. Symbolic execution involves obtaining values of variables in terms of symbolic rather than numeric, input values. So after symbolic execution, each variable has a value that is an algebraic expression, usually involving predicates. As mentioned above, aliasing and variable equality can cause difficulties, and together with conditional statements, loops, etc., can result in very complex symbolic values being obtained, even with quite sophisticated tools. Such values are clearly difficult to check.

Symbolic execution can also be useful in the generation of test data (for dynamic testing). By calculating what conditions have to be met for various parts of the code to be reached, the space of input values can be partitioned into domains, each corresponding to a path through the code. Taking one set of values from each domain thus ensures all paths are followed. Infeasible paths are identified by the corresponding domain being null. Again, in practice serious problems may be caused by complex conditions, loops, aliasing, etc.

Path Counting. Counting the number of paths through a program gives a complexity measure as described above, but identifying the paths has another purpose : used in conjunction with data on which paths have been tested, the information can be used to obtain a test coverage measure (see below).

In its true sense, the path is not a very practical unit. For example, consider the code

```
27      loop while <condition 1>
28          <code>
29           if <condition 2> then
30               <consequence>
```

```
31        end if
32        <code>
33    end loop
```

There are two paths between lines 28 and 32, but suppose the logic is such that the loop body may be executed any number of times from 1 to 100 inclusive. The number of paths from line 27 to line 33 is then

$$\sum_{i=1}^{100} 2^i \quad (\sim 10^{30})$$

This sort of combinatorial explosion of paths makes them impractical for most uses and various more pragmatic approaches have been developed, relying on slightly different concepts. For example, instead of a path, an LCSAJ (linear code sequence and jump) may be used (Woodward 1980). Each LCSAJ is a triple

(start point, end point, destination)

such that all code between the start and end points is contiguous and is executed exactly once if the LCSAJ is executed. At the end point, a jump to the destination takes place. Thus, for example, the LCSAJs in the code above are

```
(27, 27, 34)        --  <condition 1> false
(27, 29, 32)        --  <condition 1> true,<condition 2> false
(27, 33, 27)        --  <condition 1> and<condition 2> true
(32, 33, 27)        --  <condition 1> true,<condition 2> false
```

- ie. 4 LCSAJs, but 10^{30} paths. Note that the paths can be generated by concatenating LCSAJs, or "pseudo-paths" of length N can be obtained by concatenating N LCSAJs.

Dynamic Test Methods And Tools

Dynamic test methods are characterised by the software under test being actually run (either on the machine for which it is aimed or on some host - see below). They are thus the most well-known and "natural" of methods, but are largely machine-dependent.

Debugger. A debugger is part of virtually every set of testing tools. Indeed, traditionally the debugger is the tool set, blurring the quite clear-cut distinction between testing and debugging discussed below. However, even in a "good" tool set, a debugger is a very valuable asset, to provide facilities such as:

(a) breakpoints

(b) "peek and poke"

(c) traces of variable values and flow of control

The user interface should be at a high level - ie. using source lanuage variables, etc.

Test Harness. Typically, a test harness provides a controlled environment to "slot in" the software being tested. It may provide such facilities as setting up initial variable values and comparing final values with expected results, error handling, I/O etc. The associated action of stub generation may be carried out by the test harness or a different tool, but together the facilities allow an incomplete (in some sense) piece of software to be tested in a well-controlled way.

Test Data Generator. Test data can be automatically generated in several ways. For example, if the user specifies the possible ranges of each variable's values, the tool may make a random selection from each range (although this will not of course test the programs behaviour with invalid inputs). This data is therefore primarily derived from the program's specification. Alternatively, the source of the program may be analysed (as discussed above) to automatically generate data to cause each "path" in the code to be tested. Generally, even if the input data is produced automatically, the user must still work out the corresponding expected output.

Assertions. At various points in the code, assertions about the expected value of a particular variable are possible and knowing whether the assertion is true or false can give useful information. As originally defined, assertions were a language feature of Ada, but they may also be implemented by, for example, inserting them as special comments, acted upon by a pre-processor to convert them to the necessary code.

Testing Measures. Of vital importance in a well-planned testing strategy is information on exactly what has been achieved at each stage of testing. Knowing that all tests have worked is of little value without knowing if this means 100% of the code has been tested, or only 10%. Several different measures have been proposed, but for each the basic process is:

(a) the software is analysed to identify paths etc. within it (see above)

(b) as the tests are run, execution histories are generated, giving information on the paths etc. followed. These are then processed in conjunction with the static analysis data.

Mutation. An alternative approach involves making minor changes to the code (eg. replacing [illegible] to deliberately produce incorrect mutants of the (presumably) correct original. The existing tests are then run on the mutant and should of course fail. If they do not, then either the mutant is equivalent to the original, or the test set needs enhancement. Mutants of the mutants can also be used, but serious problems due to the number of files to be handled can easily arise. Producing mutants in this way is a methodical form of the more general "debugging" technique of deliberately introducing errors.

A Software Validation Strategy

The two sub-sections above give a brief outline of some of the tools and methods available. The lists are by no means exhaustive and variations and hybrid forms are possible, but the position of the debugger as just one, albeit important, tool in the complete set is emphasised. However, a comprehensive tool set is only part of the answer; a disciplined method for using the tools is also needed.

So far in this paper, the term "testing" has been used rather loosely, as indeed it and "debugging" tend to be. However, the general process of software validation (making sure it does what it is supposed to) can be considered to consist of four distinct activities:

(a) testing - finding if the software is functionally correct (ie. does it do what is should, or are there errors?)

(b) debugging - once an error has been discovered by testing, finding exactly what and where the error is

(c) coverage analysis - ensuring the testing that has been done is thorough enough. The decision on what is "thorough enough" will depend on the application. For example, it might be considered unnecessarily expensive to test all paths in a simple utility, but essential to do this for an auto-loader.

(d) tuning - deciding if any performance criteria on size, speed etc. are met and making any necessary changes to meet these.

The tools and methods described above broadly correspond to one of these activities, although some can be considered as being useful in several stages. Examples are:

testing	-	test harness, symbolic execution
debugging	-	debugger, data flow analysis
coverage analysis	-	testing measures, mutation

No tuning tools are explicitly included in the lists, but obtaining the size of a piece of software is clearly very straightforward. Predicting its execution time (as opposed to measuring it) is fairly difficult if a real-time supervisor of some kind is involved, or the hardware includes cache memory, pipelining, etc. However, a broad indication can be obtained by analysing the corresponding machine code. In practice, tuning is probably best left until the later stages of development, when the execution histories used for testing measurement, or data from the supervisor, etc. can be used to identify the "hot spots" that would benefit most from re-coding, the example.

Using the tools discussed, a valuable strategy is described below. The essence of it is that the validation takes place in stages (in the traditional bottom-up or top-down way), with each stage being fully validated before the next is started. Thus the system is gradually built up from "building blocks", each of which has been thoroughly tested and so can be trusted. The source of any problem at a particular stage should then be obvious. Without such an approach, a failure at the end of the integration process could be caused by an undetected error internal to the very first unit tested. (In practice, if it is infeasible to test all paths in all units, it is at least very valuable to have information on which parts of each are untested).

To summarise, the strategy involves using full validated units at each stage, so that any error occurring is almost certain to be in the latest unit added or be an interface problem, but not something internal to previous units. For each unit, the strategy is

(a) Obtain an initial set of test data (manually or automatically)

(b) run the tests

(c) if necessary, perform debugging, modify source and re-run tests

(d) when all tests succeed, perform coverage analysis

(e) if necessary, extend test set, using information on untested paths, etc

(f) run new tests, correct any new errors found, perform coverage analysis again and repeat until coverage is sufficient

Note that an absolutely essential part of the whole strategy is the coverage analysis, allowing the user to know when each stage is complete. The strategy outlined extends equally well to the problem of multi-activity real-time systems. Each activity can be validated in isolation in the way described, resulting in a set of "trusted" units. These can then be incorporated into the final system including the supervisor, one at a time (if possible). Any problems are then almost certainly confined to activity interaction, rather than being some previously undiscovered internal error and should be relatively easy to isolate.

This strategy is an obvious one, but its success in practice almost certainly depends on having tools for coverage analysis. Without these, it is a laborious process for the programmer to check which paths his tests have executed, and the temptation to go on to the next stage too soon is great. Furthermore, there

is unlikely to be any evidence available for his supervisor to examine to decide if procedures are being following correctly.

The approach outlined above, if followed rigorously, should result in the great majority of errors in the software being found as soon as possible. However, no scheme based on tests derived from code can be fully effective since of course it cannot test what is not there. Thus if a whole area of code (eg. to cater for some particular circumstance) has been omitted, it will be possible for all paths in the existing code to be tested fully and perfectly, but for the software to be virtually useless. Similarly, it may be perfect for all valid inputs, but contain no code for out-of-range values. Path coverage techniques will not detect this. In addition, there is a wide spectrum of programs between the "perfect" and the "completely wrong" (Scowen 1982).

One solution to this is to use specification-driven rather than code-driven test data. To do this automatically requires a more formal specification than may usually be available, but in practice the test data the user thinks of himself will be specification-based, while that prompted by information on untested paths will be code-based. A set of tests initially derived by the user in the "traditional" way (ie. by trying to think of all the cases that need to be tested), and then augmented by those suggested by coverage analysis tools is likely to be the most successful.

As long as these deficiencies are recognised and having the required proportion of paths tested does not give a false sense of security, the strategy described should prove successful - certainly more so than relying on less firmly-based schemes. Testing all paths means the software may have been fully tested; not testing them means it certainly has not. Thus it is a necessary, if not sufficient, activity.

Host/Target Development

For avionic and other embedded systems, the target machine rarely has the program generation facilities needed and a host/target approach is required. The target may be completely independent of the host, or there may be a physical link between the two, enabling them to communicate. The host and target may have compatible machine codes, or more likely, be completely different machines.

Whatever host/target arrangement is used, the bulk of software validation can be carried out on the host (if, of course, it has suitable facilities). Errors in the software can be classified as machine-dependent or independent. The vast majority will usually be machine-independent (errors in logic, mismatching interfaces, etc.); only a minority will be machine-dependent (usually connected with problems in other target software, etc). Thus perhaps 90% or more of errors can be found on the host, but final testing must always take place on the target. If software that has been fully (as possible) validated on the host fails when it is run on the target, this points to a machine-dependent problem such as differing word lengths, etc.

Using just the host, machine-independent errors can be found by running the software on the host and validating it there (this requires a compiler capable of generating code for both machines). Alternatively, an emulator can be used. This usually imposes considerable time overheads, but can help detect some target-dependent errors (overflow, compiler errors, etc). Clearly, problems caused by dynamic interaction of tasks, etc are unlikely to be satisfacorily solved by using just the host.

If a linked target is used, validation can take place on the target, but under the control of the host. Typically, all pre- and post- processing takes place on the host, with testing and debugging tools running on the host, but communicating with the software under test running on the target.

If a linked target is not available, the difficulties increase significantly. Facilities may then be limited to obtaining a binary post mortem, or using hardware logic analysers. Some help may be given by host tools to analyse the post mortem dump (giving hifh level output) or provide lists to equate machine-level addresses with high level source variables, but doing as much as possible to validate the software on the host becomes even more important.

Solving "System Problems"

By definition, design languages, high level programming languages, a comprehensive tool set including coverage analysis facilities and a well-defined rigorous method of using them solve most of the "implementation problems" identified in the first section. They can also give significant help with "system problems", as indicated below:

(a) small target without peripherals - the only real answer here is to use a sufficiently powerful host linked to the target and then adopt a strategy as described above.

(b) "exotic" peripherals - a simulator for these must be used in their absence. Producing such a simulator may itself be a complex task, but the tools and techniques can be applied to the simulator just as to the "real" system. Alternatively, the test harness/stub generator may provide facilities that make the work required minimal. Once a simulator is available, it can be used to help fully test the rest of the system. When the real peripheral is finally included, any errors should be confined to those caused by discrepancies between the real and simulated actions, and should thus be relatively easy to track down, given the confidence in the rest of the software that a rigorous approach should have provided. Dynamic behaviour of the peripheral may be a source of difficulty, but again the user is in the strongest position if he has full tested the software with the simulator and has confidence in it, at least "statically".

(c) Unreliable hardware - without software in which the user has (well-founded, documented) confidence, the question of whether the hardware or the software is to blame is a difficult one. As well as the necessary tools, etc some reliable hardware is needed for software testing, possibly a host computer with a target emulator so that use of the target computer is delayed as much as possible. The problem of faulty peripherals and the initial use of a peripheral simulator is closely related

to that discussed above, although the cost-effectiveness of simulating an available peripheral must be judged in the light of the expected problems from it.

(d) multi-processor configurations - the first stage for such systems must be to fully test the code for each processor, perhaps treating others as peripherals and simulating them. If possible, the code for each processor can then be tested on the actual target hardware (if a host has been used for the first stage). Final testing of the complete system should then be primarily concerned with isolating dynamic, interaction problems. The system may be such that each processor has its own supervisor, or there may be one for the whole configuration. The monitoring facilities of the supervisor(s) - if any - can be used at this stage.

One of the difficulties common to a lot of "system problems" is that of dynamic interaction (of peripherals, processors, activities, etc). Testing on anything but the final system is unlikely to solve a lot of these problems (so using a host, with or without a target emulator, or a peripheral simulator is only a partial solution). The approach must still be to carry out as much "static" testing as possible so that only problems of interaction are likely to remain. When these problems appear, the requirement is for a tool whose use does not significantly affect the dynamic behaviour of the code (for obvious reasons). In addition, it is likely to be the final system that is being investigated, so the presence of a host-target link cannot be relied upon.

With the final system, the testing activity is essentially "using" the system (although not necessarily flying it!) so the need is not so much for testing aids as for debugging ones. Target hardware may include the facility to trap a write (or read) access to a particular data word and this may be of use in detecting when, for example, unexpected values appear in variables. A hardware logic analyser may be of use in the same way. Alternatively, a simply tool for setting breakpoints and "peeking and poking" might be used (but probably with a low-level interface because of the nature of the final system). Depending on the system, the supervisor may provide monitoring facilities which give adequate information (again, probably at a low level).

In the final system, the facilities available will almost certainly be significantly worse than at early stages (eg. when a host is available) and new problems are introduced (eg. at a breakpoint, what happens to other processors/activities - must/can they continue?). Thus it is highly undesirable to have to go back to find and correct an error totally "internal" to an activity, etc. because this was overlooked earlier. Once again, the value of a rigorous strategy is obvious.

Summary

This section has described some of the ways the problems (system and implementation) identified in the previous section can be tackled, without claiming to be in any way exhaustive. Many different approaches and methods are possible, but successful ones are likely to follow the same broad lines:

(a) the number of errors introduced in the first place is reduced by using a high level language where-ever possible.

(b) a well-defined validation strategy is employed, involving full testing of each stage before going on to the next, and supported by a comprehensive set of tools. In particular, coverage analysis tools are available.

(c) where possible/necessary a host/target approach is adopted, with the power of the host fully exploited.

The use of code auditors and a design language may also be included to help reduce the number of errors present.

The importance of a good tool set for testing etc. seems to be generally recognised today (although the Stoneman specification (DoD 1980b) is a step backwards in this area - see below), but coverage analysis is perhaps less widely accepted. Certainly, some practical alternative to "true" paths is needed as well as investment both in producing the tools and in their use, since the effort (money) spent in the early stages of testing (and possibly overall) is likely to be higher than in a "traditional" approach. However, given the recognised importance of testing as a proportion of the overall life-cycle activity, and the fact that the later an error is detected, the more expensive it is to correct, the value of being able to adopt a rigorous approach is obvious.

TOOLS AND METHODS WITHIN FERRANTI

This section describes some of the tools and methods that have been developed within FCSL for use with embedded systems. They are not the only tools used, but do form a set conforming to the ideas outlined above. Two quite different Ferranti machines are used as targets in host/target configurations, the F1600 (24-bit mini) and the ARGUS M700 (16-bit mini). For the F1600, a DEC VAX 11/780 is used as host, and for the M700, a code-compatible machine from the ARGUS 700 range. Future developments will probably include tools for a VAX/M700 system and possibly a VAX/F100 (16-bit micro) system. All targets are linked to their respective host.

All these targets are similar in that the primary programming language is CORAL (the ALGOL 60 - like standard British MoD high level language)(BSI 1980)with real-time systems using the MASCOT methodology and supervisor (MSA 1980). The following paragraphs describe the VAX/F1600 tool set and the ARGUS 700/M700 approach is broadly similar.

FCSL is not primarily a research organisation nor a software house, and the tools produced had to be of industrial standard (robust, etc), with documentation (user and technical) meeting exacting standards. In line with FCSL's status as an approved contractor under British MoD Defence Standard 05-21, rigorous quality assurance procedures were applied during all development stages and these also tended to increase

the effort needed. Within these and financial constraints, the aim was to produce a set of tools directed mainly at the "static" testing of MASCOT activities, relying on the MASCOT monitoring facilities to help with any interaction problems (enhancements of these to include tuning aids etc are planned). Thus the tools form a basic set which it is intended to extend in the future.

The tools produced are:

(a) Database Generator

(b) Unit Driver

(c) Debug Extension to Unit Driver

(d) FIXPAC Extension to Unit Driver

(e) CORAL Instrumenter

(f) Path Analyser

The relationship of these tools to each other and to other system software is shown in Figure 1

Database Generator

This tool takes data from the CORAL compiler and the linker and produces from them the database needed by the Unit Driver (see below). More than one compiler output and more than one linker output can be combined, so that the software under test (SUT) can be compiled in several units, and partial linking is also catered for.

The information from the compiler typically gives each variable's CORAL identifier, data such as its numbertype, dimensions (for arrays), etc., and its address (in intermediate form). The linker information is used to convert the intermediate form addresses to machine addresses (relocatable). (For the ARGUS version, the compiler-generated information is not available so the tool extracts this itself from the CORAL source of the SUT).

Unit Driver

The Unit Driver acts as a driver/test harness. It is kept entirely independent of the SUT (ie. not linked in with it) and allows the user to specify the entry point to the SUT, set up initial values and compare final values with expected ones (to an optional tolerance).

All reading/writing of values is done using CORAL identifiers and the tool makes validity checks on scoping, numbertypes, subscript ranges, etc. To help with integration testing, etc., the user may insert a breakpoint at the start and/or end of a procedure (and then examine values passed, set up values to be to be returned, etc).

Debug Extension To Unit Driver

The Debug Extension to the Unit Driver is an "optional extra" providing debugging facilities such as breakpoints (specified by CORAL source line number), examining and setting variable values, tracing successive values of a specified variable, "protecting" a variable (so that if its value ever changes the effect is as if a breakpoint had been hit), and tracing flow of control (in terms of source line numbers). Again, all variable references use CORAL identifiers, with validity checks made by the tool.

To maximise commonality and portability between the two distinct systems (VAX/F1600 and ARGUS 700/M700), and because of problems caused by trying to "graft on" these tools on top of well-established other software (compilers etc) at minimum cost, an implementation that relies on a modified version of the source for debugging has been adopted. However, in contrast, no modification at all is needed to run just the Driver. The required changes are made automatically by the Instrumenter (see below).

FIXPAC Extension To Unit Driver

FIXPAC is the F1600 assembler language and only the VAX/F1600 version of the Driver has this option. It is intended to provide a common user interface (command format, layout of results, etc) for testing at high and low levels. Thus if a system is primarily CORAL, but has sections of FIXPAC for efficiency reasons, only one tool need be used for all of it, rather than two distinct ones. The facilities offered are broadly the same as at the CORAL level, except that breakpoints may be inserted anywhere (this is the only debugging facility at the FIXPAC level since the Debug Extension is purely CORAL-based).

CORAL Instrumenter

As mentioned above, the Debug Extension relies on modifications to the SUT source (essentially, insertion of calls to the Debug Extension). The Path Analyser, described below, needs similar, but different, modifications and the Instrumenter has different modes to perform either or both of these actions. In both cases, the extra calls are added in such a way as to leave the functionality of the SUT unchanged (if necessary, introducing extra BEGINs and ENDs, etc). Of course, the dynamic behaviour of the SUT is changed by the extra calls, but given the intended use of instrumented code, this does not present a problem.

The Instrumenter produces listings of the source it processes (without modifications), giving line numbers and other information which form part of the user interface for the Debug Extension and Path Analyser. As an additional operation, in path analysis mode only, the Instrumenter analyses the source to identify "paths" etc and generate a static analysis data file for use by the Path Analyser.

Path Analyser

The Path Analyser takes as input the static analysis data from the Instrumenter and execution histories generated (by the calls inserted by the Instrumenter) by the SUT as it runs. It uses these to give information on:

(a) the code tested and untested

(b) the "sub-paths" tested and untested

(c) the proportion of "sub-paths" tested

(d) the comparisons made and the results of these comparisons

The basic path analysis unit used is the "sub-path module" (SPM) which is the well-known "basic block" (ie. code with one entry and one eixt point and no internal branches, loops etc.). A contiguous set of SPMs forms a "sub-path", which is essentially an LCSAJ without the information on the destination of the jump at the end of the linear code sequence. This concept has been chosen as in interim measure, based on cost and speed of implementation, etc., but it is hoped to extend the tools to use LCSAJs eventually.

The following example illustrates the way SPMs are used and their relationship with LCSAJs.

SPM No	line no	
	1	start
1	2	
	3	if then
2	4	
	5	else
3	6	
	7	end if
4	8	
	9	if then
5	10	
	11	if then
6	12	
	13	end if
7	14	
	15	else
8	16	
	17	end if
9	18	
	19	finish

LCSAJ	Sub-path
(1 3 5)	1
(1 4 7)	1 2
(5 9 15)	3 4
(5 11 13)	3 4 5
(5 14 17)	3 4 5 6 7
(7 9 15)	4
(7 11 13)	4 5
(7 14 17)	4 5 6 7
(13 14 17)	7
(15 19 -)	8 9
(17 19 -)	9

Sub-paths and LCSAJs represent a practical alternative to true paths for use in coverage measurement since they do not suffer from the combinatorial explosion associated with loops etc. On the other hand, they are "weaker" in that all sub-paths being tested may mean only a small proportion of the paths are. LCSAJs are better than sub-paths in that, if required, they can be concatenated to derive some, or all, paths (perhaps by means of a connectivity matrix indicating how LCSAJs are inter-connected).

The information given by the Path Analyser is at different levels, since clearly all code can be tested without all sub-paths being, and all sub-paths can be tested without all comparisons being fully tested. For example, in the code

```
if C1 or C2 then.....end if
```

suppose C2 had erroneously been coded to always fail (eg. "RANGE<0"). All sub-paths could be tested by having C1 first succeed and then fail, without ever detecting the error in C2. However, having all comparisons tested in both states does not ensure all sub-paths are tested, since clearly for code such as

```
if C1 then.....end if

if C2 then.....end if
```

a test giving both C1 and C2 true, and a second giving them both false, fully test the comparisons, but only half the sub-paths. The aim must therefore be to have all sub-paths and all comparisons fully tested.

Use of Tools

The tools are designed to support a strategy similar to the one outlined above, ie. based on fully testing each unit before going on to the next. They are also designed to allow activities for MASCOT systems to be fully tested "statically" before incorporating them into the real-time system. At that stage, the monitoring facilities of the MASCOT Kernel (supervisor) come into play. However, some of the debugging facilities, for example, may be used as well as, or instead of (in a non-MASCOT system), the Kernel monitoring.

The distinction between testing and debugging is reflected in the tools. Testing tells the user if the software works and quality assurance procedures require evidence of this. It may also be felt that testing software then re-compiling it is not acceptable.Thus testing is a "permanent" activity to be carried out on "production" code. The Unit Driver supports this approach.

In contrast, debugging is a transitory activity, of no interest at all to anyone other than the programmer responsible for it; all anyone else wants to know is whether the code now works, ie. they are concerned with testing. Once a bug has been found and fixed, as demonstrated by testing, there need be no record of the debugging activity (except perhaps for statistics gathering, etc). Since the tools are aimed primarily at detecting static, logical errors rather than dynamic ones, the fact that code is instrumented for debugging is irrelevant (the method of use described below takes care of errors introduced by instrumentation, if any, and of those removed by it, eg. by introducing new compiler optimisation boundaries).

The general outline of the method is given below, although several variations are possible

(a) the software under test (SUT) is coded in CORAL, based on designs produced using a semi-formal design language (with associated tools planned to generate flowcharts from the design language where contractual obligations make this necessary)

(b) an initial set of tests (in the form of Unit Driver commands to set up and compare values) is derived

(c) the SUT source is instrumented for debugging in anticipation that this will be needed

(d) the tests are run and debugging takes place

(e) the SUT is instrumented for path analysis (this could have been done at (c))

(f) the tests are re-run on this version of the SUT

(g) the Path Analyser is used to obtain information on the sub-paths etc tested/not tested

(h) if the required coverage level has not been attained, the information on parts untested is used to augment the test set

(i) the new tests are run, on the version allowing debugging if necessary

(j) when all the new tests work, the execution history associated with them is used together with that from the original tests (which need not be re-run if no significant changes to the SUT have been made) as input for a re-run of the Path Analyser

(k) the process of augmenting the test set continues until the required level of testing is achieved

(l) the full test set is run on the uninstrumented version of the SUT. If errors appear how the cause is almost certainly a system software problem.

When each unit has been tested like this it can assume a "trusted" position and be integrated in a top-down or bottom-up way. If full testing is not practical, or possible, at least the Path Analyser output is available to identify the parts untested and thus indicate the most likely source of any error. Of course, the cost of this level of testing in the early production phases is relatively high; the saving can be expected towards the end of production and during maintenance.

Host/Target Development

As stated, development is based on host/target linked systems. Currently, the split between machines for the VAX/F1600 system is:

Compiler	host
Linker	host
Database Generator	host
Unit Driver (plus extensions)	target
Instrumenter	host
Path Analyser	host

That is, all activities except testing and debugging take place on the host. The SUT executable image is loaded via the link; the Driver is available from target backing store but accepts commands from a host terminal or a command file loaded onto target backing store from the host. Similarly, Driver output may be direct to a host terminal, or to a target file, later transferred. Execution histories are built up in target files then transferred. (Arrangements for the ARGUS 700/M700 system need to be different since with the system used the target has no backing store).

Ada AND AN APSE

This final section discusses briefly the likely impact of Ada (DoD 1980a) and its environment, Ada Programming Support Environment (APSE). The title of the section reflects the fact that thinking of the APSE is wrong; not even the minimal APSE (MAPSE) is a single entity since at least four different ones are under development (CSC 1981, DoI 1981b, Intermetrics 1981, Olivetti 1982, Softech 1981, Texas, 1981). By definition, every APSE is infinitely extensible, so any of the tools described above could be included. Unfortunately, the Stoneman document specifies only a debugger for the MAPSE (although with an Ada-level user interface of course), which tends to perpetuate the rather woolly thinking on testing/debugging, especially since most MAPSE designs seem to closely follow Stoneman in this area.

The introduction of new compilers etc. provides good opportunities for planning ahead for testing tools. For example, the instrumentation, static analysis and extraction of variable information done by separate Ferranti tools as described above would in fact be much better done inside the compiler. Indeed, the complexity of Ada compared to CORAL, for example, may well mean this is the only practical way. Similarly, the trend towards a standard compiler front-end with a back-end for each target "bolted on" removes the need for instrumentation to be at the source level to achieve portability. Unfortunately, while no published designs seems to preclude such developments, there is little evidence that the potential has been exploited (no doubt for very good reasons of time and money at this stage). However, at least all compilers and linkers in an APSE should provide the necessary data for the MAPSE debugger and this may be of use for other tools.

Although Stoneman and the corresponding published MAPSE designs provide very limited tools as they stand (eg. only one (CSC 1981) even considers a coverage measurement tool which is the keystone of the strategies outlined above), there can be hope that the opportunities have not been completely lost. However, it will be up to APSE tool suppliers to fill the gaps left by Stoneman. More positively from the point of view of implementing high quality software, the Ada language itself and the parts of the MAPSE concerned with consistency checking do provide some help.

The strong typing of the language, if used properly, should result in there being a lot fewer errors for tools to deal with. Similarly, the data encapsulation facilities should help reduce problems caused by erroneous accessing of data (if it is encapsulated and made invisible, the compiler will fail illegal accesses). Thus data structuring should be made easier. The separate compilation features (eg. being able to use a package when its specification, but not its body, is available) support a step-by-step development strategy. Exceptions provide a well-defined error handling mechanism (although the presence and position of the exception handler is a factor that may affect the quality of the software). The MAPSE will ensure that if one compilation makes re-compilation of another unit necessary, this takes place (possibly automatically), and that inconsistent units cannot be linked to produce an erroneous executable image.

Unfortunately, most of these features depend on positive action by the programmer/designer to use them. Code auditors may be a possibility to ensure some adherance to project standards on such matters, but it is by no means obvious that this can be done effectively. A general problem will be that of educating users about not just the syntax and semantics of Ada but more importantly the ideas behind its structures - ie. not just how, but why. Many of the ideas may be unfamiliar to many programmers and so many initial programs will contain errors. The compiler will trap illegal programs, but problems arise with legal programs that do not do what the user expects. Similarly, more or less by trial and error, users may hit upon a small subset of Ada that they can get to work (or at least, most of the time it does), but which does not exploit the power that is there for them to use.

Many of the features of Ada are not in fact very new, even if they may be to many users. One that is, is tasking and special tools for this area may be needed. For example, a tool to provide information on the dynamic interaction of tasks at run-time, along the lines of MASCOT monitoring (eg. a record of each task's status, indicating when it was suspended, the rendezvous taking place, the entry points called, etc). Obviously, such a tool would be intimately connected with the run time system. (Of course, a real-time system written in Ada need not use tasking; the supervisor and activities could be produced in Ada just as they currently are in some other language).

The basic requirements for tools for the APSE are no different to those for current languages (except for some Ada-specific facilities to deal with unhandled exceptions etc. during testing), but its undoubted complexity makes it important that a full set of tools should be available (certainly more than the MAPSE's debugger). The same methods of building systems with proven units apply and the almost inevitable use of a host/target approach with a powerful host should remove most constraints of size etc. on the tools. On the other hand, the host/target scheme does suffer from the problems outlined above (since final testing must take place on the target) and there must be tools in the APSE to help in this area; they must not be confined to those designed for use on the host only.

To summarise, Ada on its own will not intrinsically lead to higher quality software since it may be misused). However, if its potential is fully exploited, great advances may be possible. Of at least equal importance will be an APSE with an adequate set of tools for software validation. Given these facilities, and a suitable method for using them, a major step forward should be achieved.

CONCLUSION

As discussed above, the two classes of problems that occur during the production of avionics (and other) software - "system" and "implementation" - can both be alleviated by the methodical use of a comprehensive tool set, ideally together with design languages and high level programming languages. A large variety of such tools, static and dynamic, are possible and those described that are used within Ferranti form one workable subset. A key feature is the information, available to the programmer and his supervisor, on what has or has not been tested.

The host/target approach is also an important one, especially for avionics software, and its relevance can only increase with the introduction of Ada and an APSE. These will not in themselves be a "philosopher's stone" to solve all problems, but if the APSE contains the right tools and the facilities of Ada are fully exploited they should together allow significant advances in the drive for high quality software.

REFERENCES

General

Most computing journals regularly carry papers on various aspects of software validation. A publication devoted to the subject is

"Testing Techniques Newsletter". Software Research Associates, San Francisco, USA.

Other periodicals occasionally have issues with a collection of papers on the topic, eg.

"IEEE Transactions On Software Engineering", Vol. SE-6, No.3, May 1980.

A wide range of techniques is discussed (and a comprehensive set of references given) in

"Infotech State Of The Art Report : Software Testing". 1979.
Infotech International Ltd. Maindenhead, UK

Myers. 1979. "The Art Of Software Testing". Wiley. New York.

Individual

BSI. 1980 "Computer Programming Language CORAL 66". BS5905. British Standards Institution, London.

CSC. 1981. Documents produced under Contract F30602-80-C-0292 for the USAF Ada Integrated Environment. Computer Sciences Corporation et al.

DoD. 1980a. "Reference Manual For The Ada Programming Language". US Dept. of Defense.

DoD. 1980b. "Requirements For Ada Programming Support Environments - Stoneman". US Dept. of Defense.

DoI. 1981a. "Ada-Based System Development Methodology. Study Report". UK Dept. of Industry.

DoI. 1981b. "United Kingdom Ada Study. Final Technical Report" UK Dept. of Industry.

Intermetrics. 1981. Documents produced under Contract F30602-80-C-0291 for the USAF Ada Integrated Environment. Intermetrics Inc. et al.

MSA. 1980. "The Official Handbook Of MASCOT". Mascot Suppliers Association. At UK Ministry of Defence, Malvern.

Olivetti. 1982. "Portable Ada Programming System. Global Design Report". Olivetti, Pisa, Italy.

Schneiderman, B. 1982. "Control Flow And Data Structure Documentation : Two Experiments". Comm. ACM, Vol. 25, No. 1 (January 1982).

Scowen, RS et al. 1982. "Seven Sorts Of Program". SIGPLAN Notices, Vol. 17, No. 3 (March, 1982).

SofTech. 1981. Documents produced under Contract DAAK80-80-C-0507 for the US Army Ada Language System, Softech Inc.

Texas 1981. Documents produced under Contract F30302-80-C-0293 for the USAF Ada Integrated Environment. Texas Instruments Inc.

Woodward, MR et al. 1980. "Experience With Path Analysis And Testing Of Programs". IEEE Transaction On Software Engineering, Vol. SE-6, No.3 (May 1980).

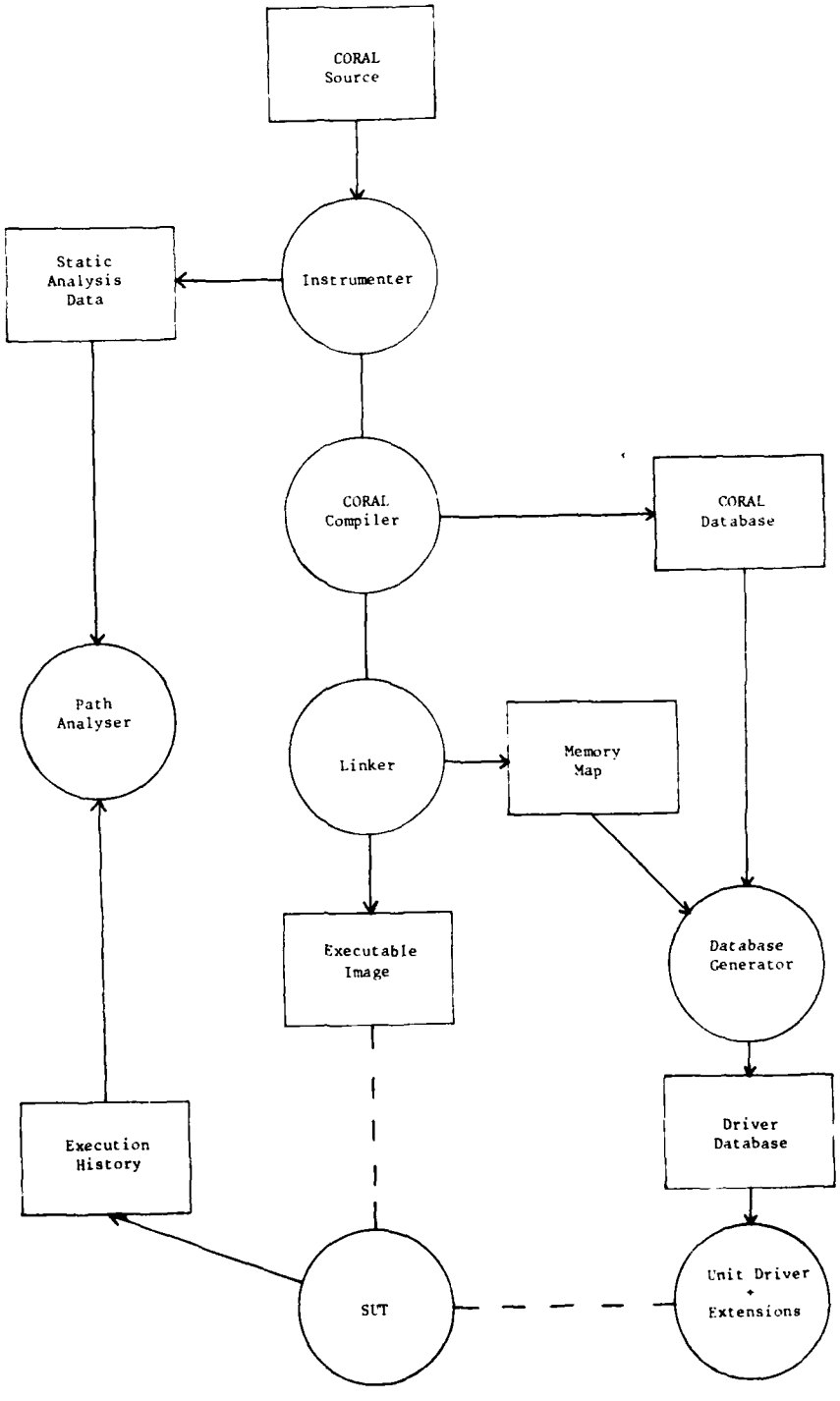

CORAL
Source
Instrumenter
Static
Analysis
Data
CORAL
Compiler
CORAL
Database
Path
Analyser
Linker
Memory
Map
Executable
Image
Database
Generator
Execution
History
Driver
Database
SUT
Unit Driver
+
Extensions

Figure 1

LA QUALITE DES LOGICIELS AVIONIQUES : SPECIFICATION ET EVALUATION

G. GERMAIN M. GALINIER
M. DELACROIX

INSTITUT DE GENIE LOGICIEL

46, rue de Provence

75009 PARIS

FRANCE

1. INTRODUCTION.

Dans la plupart des domaines industriels, l'évaluation systématique de la qualité est maintenant une tradition, des normes existent, des organismes de contrôle qualité interviennent. L'industrie du logiciel exhibe généralement dans ce domaine un retard certain, retard dû essentiellement aux facteurs suivants :

- la discipline est loin d'avoir atteint sa maturité
- la nécessité d'une évaluation de la qualité est récente
- le logiciel est un produit complexe
- le logiciel est un produit qui n'a pas d'apparence physique et l'équivalent du pied à coulisse ou de l'oscilloscope ne sont pas plus faciles à concevoir que les caractéristiques à contrôler.

Les premières notions de qualité sont donc liées au comportement du programme à l'exécution : performances et consommation de ressources machine, fiabilité, symptômes visibles et mesurables où le logiciel est considéré comme une boite noire ; ayant été intensivement étudiées elles ne seront pas évoquées dans cette communication. Ces évaluations ne sont que des constatations faites a posteriori sur un produit fini où toute intervention va coûter très cher (BOE82).

Le contrôle de la qualité non seulement du produit mais aussi du processus de développement via les documents engendrés est devenu une nécessité. Les inspections (FAG76) sont une illustration de cette approche globale.

Dans ce cas la qualité est liée au nombre, à la qualité et à la gravité des fautes détectées ; on voit apparaître une ébauche de classification des critères de qualité dans les phases essentielles du développement : conception, codage, test, ainsi qu'une métrique empirique permettant d'historiser l'expérience acquise sur un projet.

La notion de mesure au sens physique du terme n'est plus très loin (BAS80).

Le paramètre qualité isolé est défini par un modèle (abstraction du monde réel) qui doit être validé. A ce modèle est associé une métrique c'est-à-dire une évaluation quantitative du degré avec lequel le logiciel possède cette propriété, ce qui définit une échelle de mesure et la mesure elle-même associe un nombre à une unité de mesure. Aucun modèle définissant un paramètre de la qualité du logiciel n'a été totalement validé (comme la notion de température ou de résistance électrique par exemple) : les mesures acquises devront être interprétées, elles seront une aide mais jamais une indication absolue.

Dans ce cadre le logiciel avionique a un comportement très semblable à celui d'un autre logiciel temps réel mais avec des exigences de qualité supérieures surtout lorsqu'il s'agit de commande totalement numérique. Ce logiciel est généralement soumis à des tests de validation et de certification basés sur des standards d'assurance qualité. C'est là que les mesures de qualité réalisées seront particulièrement utiles (EUR82).

2. DEFINITION DE LA QUALITE LOGICIELLE

Celle qui suit, issue du glossaire IEEE (IEE82) la définit par l'ensemble des caractéristiques d'un produit logiciel relative à son aptitude à satisfaire des besoins donnés (tels qu'établis dans des spécifications écrites).

Cette définition lie la qualité à l'utilisateur par les besoins exprimés, mais ne précise pas les caractéristiques de cette qualité, caractéristiques qui doivent être modélisées pour être mesurables. C'est de 1973 que date la première étude générale (BOE78) sur les caractéristiques de la qualité du logiciel, sous contrat du National Bureau of Standards.

2.1. Caractéristiques de la qualité

Cette étude de TRW définit pour la première fois une approche globale de la qualité. Une arborescence de paramètres est définie sur trois niveaux et montre les relations logiques entre ces paramètres (Fig. 1)

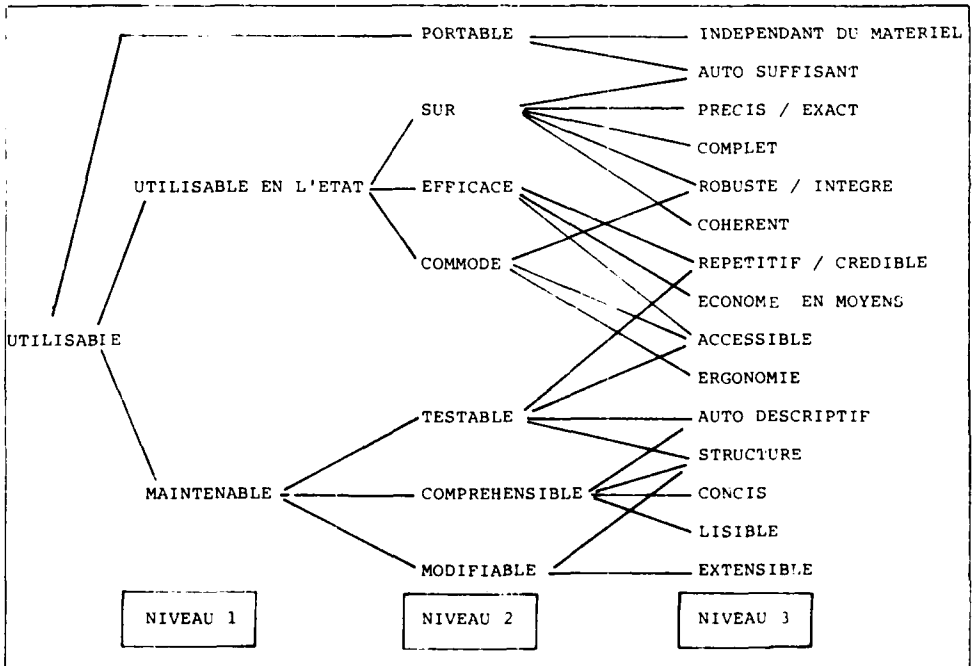


Fig. 1 Graphe de décomposition des paramètres de la qualité.

Chaque terme a reçu une définition précise , les flèches représentent une implication logique : un programme est maintenable s'il est aussi testable, compréhensible et modifiable ; au deuxième niveau le concept de compréhensibilité, par exemple, implique des programmes structurés, concis et lisibles. Pour chaque paramètre des métriques très pragmatiques sont définies ; ces métriques se traduisent d'ailleurs le plus souvent par des listes de contrôle, du même type que celles utilisées dans les inspections de Fagan (FAG76). (Fig. 2)

MAINTENABILITE

Spécifications des besoins et conception.

19 éléments contrôlables, dont par exemple :

La traçabilité entre les spécifications des besoins, la conception, le code et les cas de test est-elle définie ?

.
.
.

Un langage évolué est-il spécifié pour la réalisation ?

.
.
.

Les structures de données sont-elles faciles à modifier ?

.
.
.

Standards de programmation.

12 éléments contrôlables dont par exemple :

Tous les segments de codes sont-ils inférieurs à 200 instructions sources ?

.
.
.

Les noms de données et de procédures sont-ils significatifs ?

.
.
.

Les formats des messages d'erreur ou de diagnostic sont-ils standardisés ?

.
.
.

Produits (Code, documentation).

14 éléments contrôlables dont par exemple :

La documentation et les commentaires ont-ils un bon indice de lisibilité ?

Existe-t-il un dictionnaire des noms des variables et des références croisées pour les modules ?

Fig. 2 Exemple de liste de contrôle

Le point de vue adopté est très orienté vers l'acquisition de logiciel, cette approche peut donc largement être utilisée dans un processus de qualification ou de certification.

L'étude fait déjà apparaître les interactions positives et négatives des paramètres : l'amélioration de la robustesse, par exemple, a un impact positif sur la fiabilité et l'ergonomie, mais diminue l'efficacité. Elle n'explique pas le processus de corrélation entre les paramètres de la qualité et les métriques, et ne fournit pas une caractérisation précise des trois niveaux.

Elle est représentative de l'état opérationnel actuel de l'industrie du logiciel, et malgré ses lacunes, elle peut être appliquée avec profit dans tout processus de qualification ou de certification.

2.2. Points de vue

L'étude de Boehm avait implicitement tendance à définir de manière absolue les paramètres de la qualité. En fait, la réalité des applications amène à relativiser ces définitions par les deux caractérisations suivantes :
. d'une part, un produit logiciel évolue souvent après sa livraison ; les paramètres de la qualité exigés par l'évolution du produit sont différents de ceux exigés par son exploitation normale,
. d'autre part, le point de vue de la qualité qu' a par exemple le maître d'oeuvre qui fait l'acquisition d'un logiciel (sous-traité) est différent de celui du chef de projet qui en fait la réalisation et pourtant ces deux points de vue sont nécessairement liés.

Les deux caractérisations précédentes sont prises en compte par les travaux de Mac Call (MAC78) sur lesquels nous nous sommes basés. Les principes en sont les suivants : (Fig. 3)

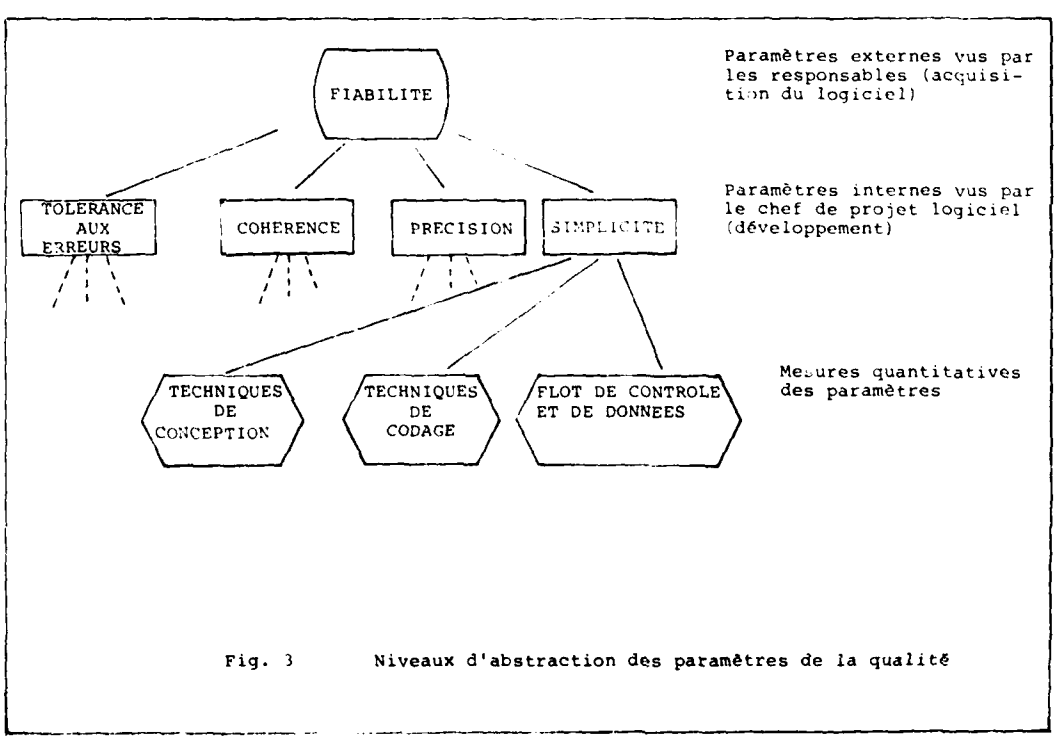


Fig. 3 Niveaux d'abstraction des paramètres de la qualité

2.2.1. Les paramètres de la qualité sont définis par trois niveaux d'abstraction.

. paramètres externes ou facteurs qui caractérisent la qualité du logiciel vue par celui qui en fait l'acquisition ou qui l'utilise : fiabilité, facilité d'utilisation, maintenabilité, ... Ces facteurs ne sont pas a priori spécifiques du logiciel. Ils permettent de caractériser le comportement externe de tout produit industriel.

. paramètres internes, ou critères, qui caractérisent la qualité du logiciel vue par celui qui réalise c'est-à-dire par les propriétés des documents et du code : cohérence, complétude, modularité, simplicité, ... Facteurs (externes) et critères (internes) sont liés par des relations du type : "est réalisé par". Par exemple, la fiabilité, paramètre externe, est réalisée par les paramètres internes suivants : tolérance aux erreurs, cohérence, précision, simplicité. Ces mêmes paramètres internes peuvent d'ailleurs intervenir sur d'autres paramètres externes.

. métriques. Elles permettent de fournir une mesure des critères. Cette mesure est réalisée sur les différents textes produits, pendant le développement du logiciel. Mac Call limite les mesures à deux types : binaires (présence ou absence d'un élément), normalisées (rapport nombre réel d'occurrences/nombre d'instructions). L'évaluation de la simplicité, par exemple, est basée sur la mesure de 25 éléments prélevés dans les documents de conception et dans le code : nombre (normalisé) de modules avec une entrée et une sortie, nombre de parcours logiques indépendants. Bien évidemment la saisie de ces informations doit être aussi automatique que possible.

L'approche est globale et prédictive c'est-à-dire qu'elle ne se limite pas simplement à constater la qualité d'un code puisque les évaluations s'appliquent sur les principaux documents engendrés lors du développement : spécification, conception,...(Fig.6)

2.2.2. Les paramètres de la qualité sont liés :

- au type de l'application : les paramètres qualité exigés pour un logiciel avion sont différents de ceux exigés pour un central téléphonique ou système de simulation.

- au type d'activité exercée :
  . exploitation du produit (fiabilité, conformité, efficacité pour un logiciel avionique par exemple)
  . révision ou maintenance du produit (maintenabilité, flexibilité, testabilité)
  . transfert du produit dans un autre environnement (portabilité, intégrabilité).

2.3. Les supports de l'évaluation

Pour mesurer il faut d'abord définir l'objet de la mesure. Trop souvent le seul objet mesurable est le code ; les évaluations qui en résultent sont nécessairement limitées et arrivent trop tard dans le cycle de vie puisque la qualité du produit est en grande partie déterminée lors des étapes précédentes. Insistons donc sur la nécessité d'appliquer aussi les mesures à la spécification des besoins et à la conception pour avoir des indications prédictives utilisables au cours du développement. (Fig. 6)

Signalons, aussi, que le contenu informationel de la mesure est fortement lié au degré de formalisme adopté, formalisme caractéristique des méthodes employées.

Tous les produits intermédiaires engendrés lors du développement peuvent être considérés, selon le point de vue adopté, soit comme de simples textes, soit comme des entités logiques, soit comme des descriptions de systèmes. Cela définira le type de mesure associé :

texte : si le texte est simplement narratif on pourra mesurer son indice de lisibilité •(IL) ; sur des textes à mots clés on pourra mesurer la complexité à la Halstead (ALF82) (DRE82). Ces mêmes textes constitueront par ailleurs, la base de mesures du type : nombre de goto ou taux de commentaires par exemple (FAG76) (MAC78)

entité logique : le texte a maintenant une structure logique qui décrit son comportement à l'exécution : le nombre de parcours logiques indépendants (MCA76), (ACF82) ou la complexité du graphe logique (SZE82), par exemple, sont mesurables.

système : le texte décrit un système décomposé en sous systèmes échangeant de l'information. L'entropie (MOH79) permet de mesurer l'"ordre" du système donc la qualité relative de sa conception (IGL82).

2.4. Remarques

Trop souvent la qualité du logiciel avionique n'a été vue qu'à travers le parametre fiabilité. Ce point de vue est beaucoup trop restrictif. La commande numérique des avions impose maintenant une appréhension globale.

Si, par exemple, les essais en vol demandent la modification d'un gain, celle-ci ne doit pas demander un mois pour être répercutée dans le logiciel, ce qui peut être le cas si les constantes du programme ne sont ni déclarées ni localisées.

L'utilisateur ou le client ne comprendrait pas qu'une modification aussi mineure entraîne des remaniements importants du logiciel et aboutisse à un coût démesuré.

Cela suppose que le facteur externe de flexibilité (ou adaptabilité ou maintenabilité perfective) soit pris en compte au moment du développement. L'évaluation (à l'aide des métriques) des critères internes liés à ce facteur (modularité, généralité, extensibilité, ...) peut permettre d'éviter ce type de désagrément.

• IL = 0.4(L+P) avec L nombre moyen de mots par phrase et P nombre moyen de mots de plus de 3 syllabes par 100 mots. (LWB77)

3. CARACTERISTIQUES DU LOGICIEL AVIONIQUE

3.1. Problèmes posés (GER79)

Nous nous restreindrons aux logiciels centraux ou logiciels système qui réalisent les fonctions globales et qui sont implantés dans le calculateur principal de l'avion ou de l'engin.

Ces logiciels centraux aéroportés s'apparentent aux logiciels de contrôle de processus en temps réel : intégration dans un environnement spécifique, fonctionnement en boucle fermée où le traitement dépend d'évènements extérieurs et doit être effectué dans un temps très court.

Ces logiciels présentent en outre les caractéristiques particulières suivantes :

- Forte complexité :

Bien que ces logiciels soient en général de taille moyenne (par rapport à d'autres domaines où le million d'instructions est courant) leur complexité est très élevée.
Cela tient à la complexité intrinsèque des fonctions qu'ils réalisent, à la diversité et à l'interdépendance de ces fonctions.
Cela tient également à de fortes contraintes temps réel : certaines boucles d'asservissement nécessitent des temps de réponse de quelques millisecondes (guidage et pilotage d'un engin par exemple).
Les problèmes de synchronisation entre les traitements internes au calculateur d'une part, et entre ces traitements et les échanges avec le système d'autre part, ajoutent également à la complexité.
Enfin, un contrôle permanent d'un grand nombre de paramètres relatifs à des organes essentiels est indispensable au cours même du vol ;

- Sûreté de fonctionnement :

Rappelons pour mémoire que les conséquences des erreurs dans un logiciel aéroporté peuvent être très graves.

- Evolutivité :

Les spécifications fonctionnelles du logiciel évoluent pendant tout le développement et jusqu'à la validation du système complet. De plus, plusieurs versions du logiciel d'un même avion peuvent être développées en parallèle ou successivement pour répondre à des missions différentes.

3.2. Paramètres de qualité associés

Les problèmes à résoudre sont les suivants :

1. Spécification et pondération des facteurs qualité pour appréhender globalement la qualité des logiciels décrits en 3.1.

2. Impact de ces facteurs, acquis lors du développement (spécification, conception, codage) sur le comportement ultérieur du système en qualification, exploitation, transfert.

La spécification de la qualité (Fig. 4) exige donc :
. d'identifier les caractéristiques essentielles du système
. d'appréhender les conséquences de la non-spécification d'un paramètre de qualité
. d'appréhender les relations entre les paramètres externes de la qualité.
Dans les logiciels avioniques tels que décrits en 3.1. les facteurs de qualité essentiels sont : (Fig. 5 - Fig. 6)

en exploitation : fiabilité
conformité
efficacité

en maintenance-révision : maintenabilité
flexibilité
testabilité

en transfert : actuellement le cas se pose peu, en effet, le transfert du logiciel d'un avion sur un autre n'est pas encore un processus courant.

La définition de ces paramètres externes estreportée en annexe.

Le paramètre externe efficacité a un effet négatif sur :

- la maintenabilité puisque l'optimisation du code et les techniques de codage spécifiques de l'efficacité (place mémoire et temps d'exécution) ont la réputation d'être en conflit avec les bonnes techniques de structuration et posent des problèmes de maintenance. Cela est un vieux débat, sans grande signification maintenant : les bons compilateurs génèrent un code qu'un programmeur aura du mal à optimiser, le supplément de place mémoire provoqué par la programmation structurée a été évalué à 10 % environ, sans pénalisation en vitesse d'exécution, le potentiel des matériels actuels et des architectures associées fournit une solution dans la plupart des cas critiques.

- la flexibilité puisque la généralité exigée accroît généralement l'overhead.

- la testabilité puisque la simplicité et la modularité exigées, sont en conflit avec les techniques traditionnelles de l'efficacité.

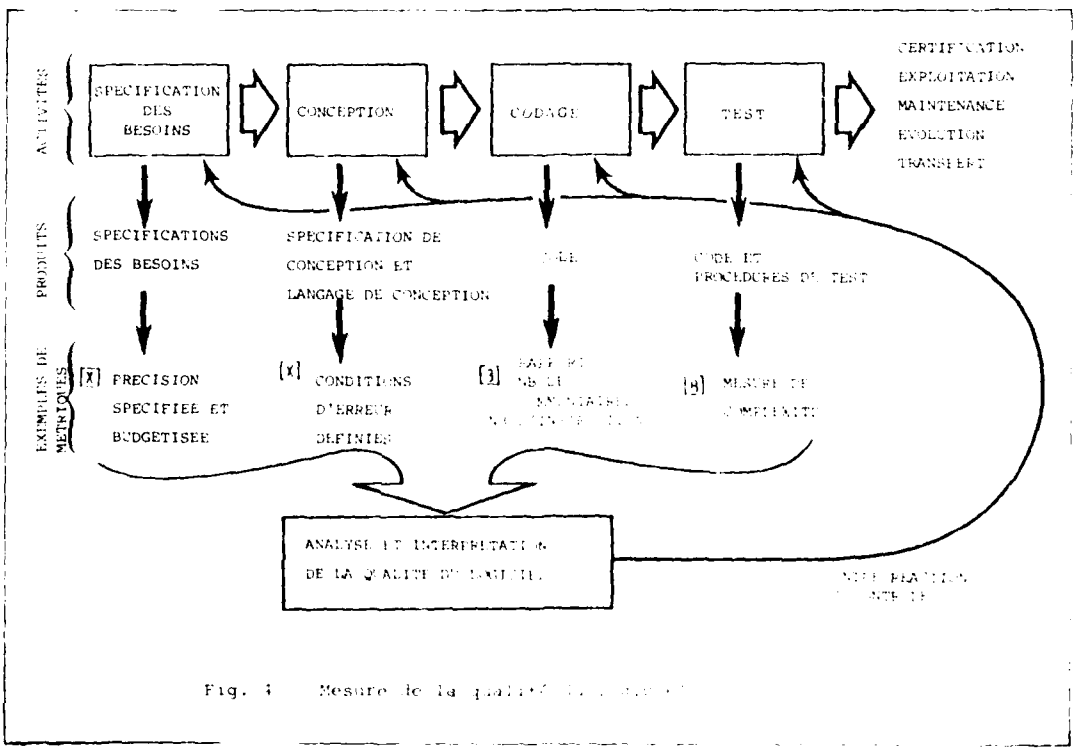


Fig. 4 Mesure de la qualité [illegible]

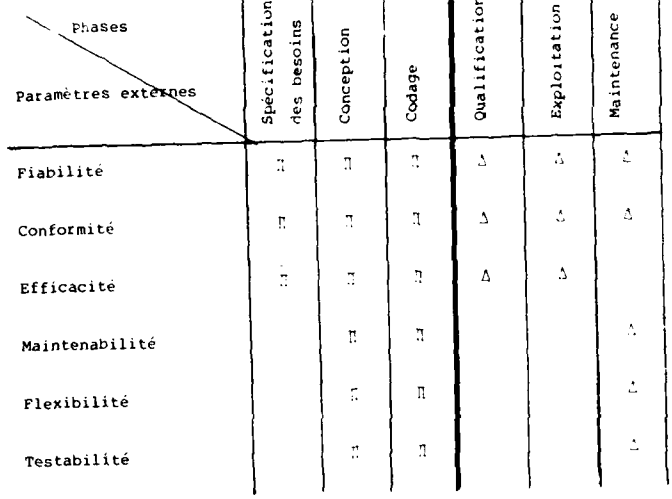
	exploitation			maintenance		
Paramètres externes / Paramètres internes	Fiabilité	Conformité	Efficacité	Maintenabilité	Flexibilité	Testabilité
Traçabilité		Δ				
Complétude		Δ				
Cohérence	Δ	Δ		Δ		
Précision	Δ		Π			
Tolérance aux erreurs	Δ		Π			
Simplicité	Δ			Δ		Δ
Modularité			Π	Δ	Δ	Δ
Généralité	Π	Π			Δ	
Extensibilité			Π		Δ	
Instrumentation			Π			Δ
Auto-description			Π	Δ	Δ	Δ
Efficacité mémoire			Δ			Π
Efficacité d'exécution			Δ			
Concision				Δ		

LEGENDE

Δ Le paramètre externe fiabilité implique les paramètres internes cohérence, précision, tolérance aux erreurs, simplicité.

Π Le paramètre interne modularité a un effet négatif sur

Fig. 5 Paramètres qualité d'un logiciel avionique



LEGENDE

□ où le paramètre est mesuré

Δ où les conséquences apparaissent

Fig. 6 Mesure des paramètres externes dans le cycle de vie d'un logiciel avionique

4. MESURE DE LA QUALITE DANS LE DEVELOPPEMENT D'UN LOGICIEL AVIONIQUE

Pour concrétiser l'évaluation globale de la qualité une procédure doit être mise en place, procédure comportant les actions suivantes :

1. Analyse des paramètres externes

1.1. Identification et pondération des paramètres externes
1.2. Analyse du rapport coût de la qualité / coût du cycle de vie associé et définition d'un compromis
1.3. Définition du compromis sur les paramètres externes
1.4. Définition des paramètres internes (facteurs)
1.5. Description détaillée des paramètres externes liés à l'application

2. Spécification des paramètres internes

2.1. Identification des paramètres internes critiques exigés
2.2. Définition d'un plan qualité pour obtenir ces paramètres

3. Spécification des estimations et des métriques

3.1. Spécification d'une estimation pour chaque paramètre externe
3.2. Identification des métriques spécifiques à appliquer

Cette procédure conduit à définir par exemple les tableaux suivants.

Paramètre externe	Estimation désirée par phase			
	Spécification	Conception	Code	Qualification
Fiabilité	0,8	0,8	0,9	0,99
Conformité	0,8	0,8	0,9	0,99
Efficacité	0,7	0,7	0,7	0,8

Paramètre interne	Métrique
traçabilité	références croisées
complétude	listes de contrôle
cohérence	cohérence procédurale cohérence des données listes de contrôle
tolérance aux erreurs	listes de contrôle des tolérances aux erreurs listes de contrôle sur les données en entrée recouvrement des pannes logicielles recouvrement des pannes matérielles conditions d'états sur les périphériques
simplicité	mesure de la structure de conception programmation structurée complexité du flot de contrôle et du flot des données complexité du code

L'application des métriques pendant le développement permettra d'analyser les mesures faibles, d'évaluer la variance des mesures, d'évaluer les mesures par rapport aux estimations et de déterminer les actions correctives à prendre.

La Fig. 7 est un exemple partiel des métriques que l'on peut appliquer lors du développement du logiciel. Il y a plusieurs manières de saisir ces métriques selon le degré d'automatisation du processus de développement du logiciel, selon le niveau de contrôle qualité opérationnel dans l'organisme, selon le niveau méthodologique. S'il n'existe ni méthodes, ni outil automatique, ni contrôle qualité, les mesures seront peu nombreuses et peu significatives. Le nombre de mesures possibles augmente avec le niveau méthodologique puisque toute méthode engendre une structure concrétisée dans le vocabulaire du texte résultant : méthode de spécification, méthode de conception,...

Manuellement, les revues de projet ou les inspections permettent de saisir la totalité des métriques nécessaires ; leur traitement ultérieur : normalisation, pondération permet d'obtenir une mesure quantitative, globale et objective de la qualité même si cette mesure doit être ensuite interprétée.

Cette saisie manuelle est fastidieuse, chère et se heurte souvent à des barrières psychologiques.

L'automatisation des métriques passe évidemment par l'utilisation de processeurs ; la plupart existent déjà et il suffit le plus souvent de les instrumenter. Prenons quelques exemples :

Spécification des besoins. Leur écriture avec un système de traitement de texte évolué permet de fournir : structure, références croisées, glossaire. Les systèmes d'aide à la spécification tels que DLAO (BCD82) pourront prendre en charge toutes les mesures de qualités possibles.

Conception. Les langages de conception structurés tels que Pseudocode (REI82) sont maintenant très répandus et les processeurs associés fournissent en sortie : table des matières, références croisées des variables et des modules, flot de contrôle, ...
Il est relativement facile d'instrumenter ces processeurs pour obtenir les métriques. Le système de programmation SSP (IGL80) prend automatiquement en compte ces mesures et de plus les associe au code engendré.

Codage et test. Il existe tout un arsenal de processeurs pour mesurer la qualité du code s'il est écrit en langage évolué (FORTRAN, COBOL, PASCAL). La plupart des outils d'aide au test : analyseurs statiques, analyseurs dynamiques, générateurs de tests, instrumenteurs fournissent les données associées aux métriques précédemment définies (exemple GRC81).
Les analyseurs de qualité multilangages comme, par exemple, le QUALIMETRE-C (QUA82, SZE82) fournissent toutes les mesures de complexité textuelles (HAL77) et structurelles (MAC76, MOH79) possibles.

Cette saisie automatique des métriques :
- assure l'objectivité et la fiabilité de la mesure
- évite les réticences psychologiques
- réduit les coûts de l'évaluation de la qualité
- systématise la mesure.

Nous avons là les premiers éléments de l'outillage. Les futurs environnements de programmation (AIG82) supporteront les plans d'assurance qualité par les outils de développement et par les outils de mesures qu'ils intègrent.

5. CONCLUSION

Maxwell disait "mesurer c'est connaître". La métrologie de la qualité du logiciel en est à ses débuts mais elle contribuera à faire évoluer le développement du logiciel d'une activité artisanale vers une activité réellement industrielle. Nous avons souligné le fait que les modèles actuels ne sont pas suffisamment validés pour que l'on puisse leur accorder une confiance absolue ; ils sont cependant des indicateurs suffisamment fidèles, même si l'interprétation humaine doit rester en éveil. Un taux de complexité supérieur à la norme n'est peut-être pas représentatif d'un module mal programmé mais indique peut-être que ce module gère un flot de contrôle important et qu'il doit retenir l'attention.

Le logiciel avionique, avec ses exigences de sûreté de fonctionnement, ses critères de certification est une cible privilégiée pour l'évaluation quantitative de la qualité

La mesure de la qualité
- conforte le plan qualité qui gère le développement du logiciel et permet d'enrichir l'historique des projets de données objectives
- fournit les éléments du contrôle qualité lors de l'acquisition d'un logiciel
- contribue à fournir les arguments de recette pour les organismes de qualification ou de certification.

METRIQUE	SPECIF. BESOINS	CONCEP-TION	CODE
PRECISION - EXACTITUDE			
1. Analyse des erreurs spécifiées fonction par fonction au moment de la spécification des besoins	DLAO		
2. Spécification des précisions requises sur chaque entrée-sortie, traitement ou constante	DLAO		
3. Adéquation de la bibliothèque mathématique du système		H	
4. Adéquation des techniques numériques utilisées		H	H
5. Sorties dans l'intervalle de tolérance à l'exécution			AQ
TOLERANCE AUX ERREURS			
16 éléments à contrôler			
SIMPLICITE (COMPLEXITE)			
S1 CONCEPTION			
1. Conception hiérarchisée, descendante		SSP	
2. Pas de fonctions redondantes		H	H
3. Autonomie des modules		SSP	
4. Exécution d'un module indépendante des précédentes exécutions		H	H/AQ
5. Chaque description de module inclut les entrées, les sorties, le traitement, etc...		SSP	
6. Chaque module a une seule entrée, une seule sortie		SSP	
7. Il n'existe pas de données globales		SSP	
S2 CODE STRUCTURE - PREPROCESSEUR			SSP/H
S3 COMPLEXITE textuelle et structurelle, des programmes et des modules		SSP	AQ
S4 SIMPLICITE DES TECHNIQUES DE CODAGE (MODULES)			
1. Flot de contrôle "du haut vers le bas"		SSP	AQ

LEGENDE : DLAO automatisable sur le système d'aide à la spécification DLAO
SSP " sur l'environnement de programmation SSP
AQ automatisable avec le Qualimètre
H manuel

Fig. 7 Exemple de métriques et outils associés

BIBLIOGRAPHIE

(AFC80) AFCIQ, F. DE NAZELLE (SNIAS), Président du groupe Assurance Qualité Logiciel, Pour un logiciel de qualité, Bulletin AFCIQ, Vol 15 N°3 Septembre 1980

(AIG82) AIGLE, Atelier Intégré de Génie Logiciel Expérimental, Rapport Final, Contrat Agence de l'Informatique Juin 1982

(ALF82) J-L ALBIN, R. FERREOL, Collecte et Analyse de Mesures de Logiciel AFCET 1er Colloque de Génie Logiciel Juin 1982

(BAS80) V.R. BASILI, Tutorial on Models and Metrics for Software Management and Engineering, IEEE Catalog N° EHO167-7 Octobre 1980

(BCD82) P. BARDIER, S. CHENUT-MARTIN, F. DOLADILLE, Définition et Conception de Logiciel Assisté par Ordinateur, AFCET, 1er Colloque de Génie Logiciel, Juin 1982

(BOE82) B.W. BOEHM, Les facteurs du coût du logiciel T.S.I., Vol 1 N°1, 1982

(BOE78) B.W. BOEHM and others, Characteristics of Software Quality, North Holland 1978

(DRE82) F. DEVRON, Une application de la mesure de complexité des programmes, AFCET, 1er Colloque de Génie Logiciel Juin 1982

(EUR82) Eurocae, Etude et Homologation du logiciel des systèmes et équipements de bord, RTCA DO-178/Eurocae ED-12, Mai 1982

(FAG76) M.E. FAGAN, Design and Code Inspections, IBM Systems Journal Vol 15, N°3, 1976

(GAL80) M. GALINIER and al, SSP Système Support de Programmation, Journées BIGRE Environnements, Supports et Systèmes de Programmation, Rennes, Juin 1980

(GER79) G. GERMAIN, Quelques problèmes liés au développement des logiciels aéroportés, *Journées Génie Logiciel*, IRIA, 1979

(HAL77) M. HALSTEAD, Elements of Software Science, North Holland 1977

(IEE82) IEEE, A glossary of Software Engineering Terminology, IEEE Project 729 July 1982

(IGL82) IGL, Assurance Qualité et Conception des Produits Logiciels, Rapport interne n° 27-QUA, IGL, Juin 1982

(LPW77) M. LIPOW, B.B. WHITE, B.W. BOEHM, Software Quality Assurance : An Acquisition Guide-Book TRW-55-77-07, November 1977

(MAC78) J.A. McCALL, An Introduction to Software Quality Metrics, Software Quality Management, Petrocelli Book, 1979

(MCA76) T.J. McCABE, A Complexity Measure, IEEE Software Engineering Vol 2 N°4, 1976

(MOH79) S.N. MOHANTY, Models and Measurements for Quality Assurance of Software, ACM Computing Surveys Vol 11, N°3, September 1979

(SZE82) J. SZENTES, Somika - An automated system for measuring software quality, AFCET 1er Colloque de Génie Logiciel, Juin 1982

-:-:-:-

ANNEXE

Définitions des paramètres externes (facteurs)

Conformité : Attribut général du logiciel, lequel doit, au fur et à mesure de sa production, demeurer conforme aux documents issus des étapes précédentes. Une fois achevé, le logiciel doit notamment être conforme aux spécifications, c'est à dire correspondre aux besoins exprimés, et sa fabrication doit être conforme aux exigences de qualité.

Efficacité : Aptitude d'un logiciel à se limiter à l'utilisation des ressources (mémoire, unité centrale, durée des exécutions) strictement nécessaires à l'accomplissement de sa fonction.

Fiabilité : Aptitude d'un logiciel à assurer une fonction imposée dans des conditions données, pendant une durée donnée.

Flexibilité : Attribut d'un logiciel qui désigne sa capacité à prendre en compte des situations non rigoureusement identiques à celles prévues et précisées dans l'exigence des besoins.

Maintenabilité : Facilité avec laquelle un logiciel peut être modifié dans le but soit d'en corriger des défauts, soit d'en étendre les possibilites. Ceci implique que le code soit compréhensible, modifiable et testable. A cette caractéristique se rattachent celles d'adaptabilité, de complexité, de compréhensibilité, de concision, d'extensibilité, de lisibilité, de reparabilité, de testabilité.

Testabilité : Facilité d'élaboration de jeux de données de test et de vérification du bon fonctionnement du programme lors de l'exécution de ces derniers. Cette caractéristique a trait à la fois aux aides au test et au diagnostic et à l'effort nécessaire au test.

DISSIMILAR SOFTWARE IN HIGH INTEGRITY APPLICATIONS IN FLIGHT CONTROLS

by

David J.Martin

Flight Controls Division

Marconi Avionics Ltd,

Rochester, England.

SUMMARY

Digital computing techniques and, in particular, microprocessors are now being used increasingly in secondary flight control systems such as spoilers and airbrakes, automatic trim, and slats and flaps control.

These systems, though relatively undemanding in computing power, nevertheless have high integrity requirements, particularly in fly-by-wire applications.

The requirements of high integrity systems are defined and examples of failure surviving systems given.

Having discussed the impact of digital processors on system design, various solutions are reviewed.

Multiplex similar redundant systems have been used widely in flight controls and some of their strengths and weaknesses are detailed. Software techniques applied to similar redundant systems are then briefly described.

A dissimilar redundant solution, using two different microprocessors, is discussed and the impact of this architecture on software procedures is then reported.

The paper concludes with a review of the strengths and weaknesses of the dissimilar architecture approach as seen to date.

1. INTRODUCTION

High integrity in the context of flight control systems means safety critical. With the advent of digital equipment providing various primary and secondary flight control functions via the software, it has been necessary to develop techniques for producing safe software. In general the methods used fall into two categories: those aimed at producing error free software (fault avoidance) and those aimed at providing software which can continue to operate after errors (fault tolerance). The fault avoidance approach has been successfully used by Marconi Avionics for the Panavia Tornado, Boeing YC-14 and British Aerospace Jaguar Fly-by-Wire automatic flight control systems. Currently being developed is a slat and flap control system for the Airbus Industrie A310 for which Marconi Avionics supplies the control computers. This computer uses a form of fault tolerant software in which two dissimilar programs are executed by dissimilar processors.

Fault tolerance can also be divided into two methods; recovery blocks and dissimilarity. In general, flight control systems do not lend themselves to the method of recovery blocks where parts of the program are re-run using alternate blocks of code after an error has been found. This requires time for the extra processing which is not usually available as the normal processing needs to be continuous. Recently however, a method for concurrently evaluating the alternate blocks has been suggested (Shepherd, J., 1982).

The dissimilar approach is used for the A310 slat and flap control computers and the reasons for this choice of method are described in this paper.

Initially the requirements of high integrity systems and a brief history of typical implementations are described. This is followed by an overview of the particular requirements of the slat and flap control system which led to the decision to produce dissimilar software. In this application dissimilarity means the generation of two different computer programs from a common specification, by separate programming teams using different programming languages, separate translators and host development facilities, with different target processors.

2. REQUIREMENTS OF HIGH INTEGRITY SYSTEMS

Modern aircraft encompass a large variety of sub-systems each having different integrity constraints. The passenger entertainment system will obviously have a lower integrity required of it than the flight control system. It does not perform a safety critical function and hence its loss can be tolerated. Similarly any one of the navigation equipments on board an aircraft could cease to function without giving rise to a hazard. This is due to the various dissimilar types of navigation systems available to the pilot, eg inertial navigation, ADF, VOR/DME, Omega, Decca, secondary radar (it is unlikely that all will be available in one aircraft) and the last-ditch techniques of astro-navigation, dead reckoning.

In contrast, flight control systems and stores management systems have a higher level of safety required of them. Flight controls are obviously safety critical and the loss of, say, the elevator controls during final approach will almost certainly give rise to a catastrophe. This has led to the development of stringent design techniques based upon earlier proven sound practices in order to achieve the integrity requirements. These requirements are typically defined in probabilistic terms. For instance the CAA has suggested that, for automatic landings, the overall flight control system must be so safe that it will not cause a fatal accident risk in the landing phase greater than 1×10^{-7} per landing. This risk is the sum of the constituent elements of the system including mechanics, hydraulics and electronics. A typical risk figure apportioned to the electronics (sensors/computing/actuation) is 1×10^{-9} per landing.

If the mean time between failures (MTBF) of the automatic flight control system is known or can be inferred from previous experience or from lengthy test flying of a developed system, then the probability of failure of the equipment during the critical landing stage can be calculated from the formula:

$$\text{Probability of failure} = \frac{\text{Critical time}}{\text{MTBF}}$$

The critical time is the time between an irrevocable decision to perform an automatic landing and the completion of the landing. Because the decision would only be taker if all the necessary equipment were functioning, only subsequent faults need be considered. For an automatic landing this decision need not be made until the aircraft passes the minimum break-off height, typically about 60ft, so that the critical time is about 15 seconds.

Therefore an MTBF of $\frac{15}{1 \times 10^{-7}} \times \frac{1}{3600} = 41500$ hours

minimum is required of the automatic landing system if all failures are assumed to be significant. Such a figure is clearly impossible to achieve with any single system of automatic equipment so that at least one alternative means of control and guidance must be available if a landing is to be completed following a failure of the primary system.

Hence, the automatic flight control system must have some level of in-built redundancy. There are two pieces of data required in order for the flight controls to remain functional after a failure. These are firstly, the knowledge that there is a disagreement between redundant elements and second, the element that has failed. This allows the faulty channel to be isolated. Consequently a triplex, triplicated, duplex monitored or duplicate monitored configuration is necessary. Any such system will survive a single fault and continue operating. At least two independent faults must occur during the 15 second landing period before the system ceases to function. If the probability of a failure in a single lane in the system is p then the probability of two lanes failing is $3p^2$ for a triplex scheme. Hence $3p^2 = 10^{-7}$ gives an acceptable MTBF of

$$\frac{15}{1.83 \times 10^{-4}} \times \frac{1}{3600} = 22.75 \text{ hours}$$

This is easily achievable and the safety requirement is not the determining criteria. A system with 22.75 hours MTBF is not acceptable to an airline operator and a much higher MTBF is required.

The automatic flight control during autoland represents a system requiring fail-operative capability. A slat and flap system in contrast requires fail-safe capability. That is, it is only necessary to identify that there has been a failure in order to shut-down the system operation and lock the surfaces. However, the requirement that the probability of inadvertent deployment of slats or flaps should be less than 1×10^{-9} per hour means that the reliability requirements are similar to the autoland case and the same exacting design disciplines need to be applied.

3. IMPACT OF DIGITAL COMPUTERS ON SYSTEM DESIGN

High integrity systems, such as are required for automatic operation of primary and secondary flight controls, have existed for at least two decades and a wealth of experience has been accrued about them by the aircraft industry. Proven techniques for achieving safe operation of flight control equipment have been developed. These techniques allow the systems to be more easily analysed by making their structure highly visible. With the advent of digital computers the race towards ever increasing functional capability has intensified, but in addition the problems that need to be solved have changed.

Many of the design practices that have evolved for use with analogue implementations are readily transferred to digital systems. However, two areas stand out as requiring new techniques:

a) The general purpose nature of a digital computer necessitates a far deeper understanding of the hardware failure modes and effects. For instance, failure of the central processor can affect all functions that the computer is intended to perform. In contrast an analogue system is readily analysed as there is dedicated circuitry for each function.

b) Software is a new entity. Embodied in it are the functional requirements of the system. Executed by a general purpose processor it has potential access to any part of the flight control equipment. However, with no decades of experience to fall back on, there are only now the beginnings of a unified approach to the design, construction, analysis and testing of software.

To be truly fail-operational or fail-safe, a system must be free from common mode failures and dormant failures must not affect system performance during or after first failure.

While many of the inherent analogue problems related to nuisance disconnect/failure transient/dormant failure trade-offs are minimised in a digital mechanisation, the extent to which the above aims can be achieved is still the essence of good system design.

Common mode failure stems from four main causes:

i) External environment of the system

ii) Inter-channel interference

iii) Common design or manufacturing errors in the hardware

iv) Common design or programming errors in the software (specific to digital systems)

In the next section the methods used for avoiding common mode errors in both the hardware and the software are detailed.

4. SOLUTIONS TO REDUNDANCY FOR DIGITAL FLIGHT CONTROLS

4.1 Multiplex Similar Redundancy

An early digital autopilot/flight director system (AFDS) having fail operational capability is that of the Panavia Tornado. Two digital processors operate asynchronously with respect to each other and they also operate on data from different sensors (which are closely matched, however). Therefore with the fairly high control law gains adopted, including the use of integral control, it is necessary to provide some sort of inter-channel synchronisation of the digitally computed signals in order to avoid divergence of the computer outputs which would result in disconnects. This inter-channel synchronisation takes the form of a mixture of intermediate signal consolidation (achieved by analogue cross-feeds between the computers) and low authority synchronisation of integrator states. These are shown schematically in Figure 4.1.

Single failure operational capability was also exhibited by the electronic flight control system (EFCS) for the Boeing YC-14 short take-off and landing (STOL) transport aircraft (Corney J, 1980). This aircraft featured upper surface blown (USB) flaps which used the Coanda effect to increase lift by blowing the jet engine efflux over the upper surface of the wing. This configuration results in a high degree of coupling between aircraft axes and since the USB flaps are fully fly-by-wire, ie there are no mechanical input linkages, the EFCS had to be failure operational in order to meet the system reliability requirements during STOL operation.

A triplex configuration was chosen (Figure 4.2) to satisfy the failure operational requirements.

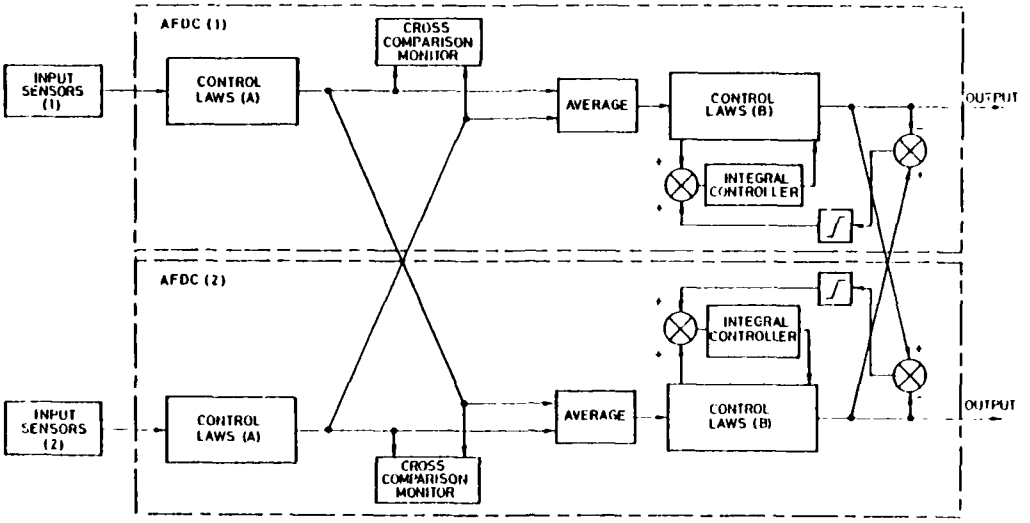


Figure 4.1 AFDS Inter-lane Consolidation and Synchronisation

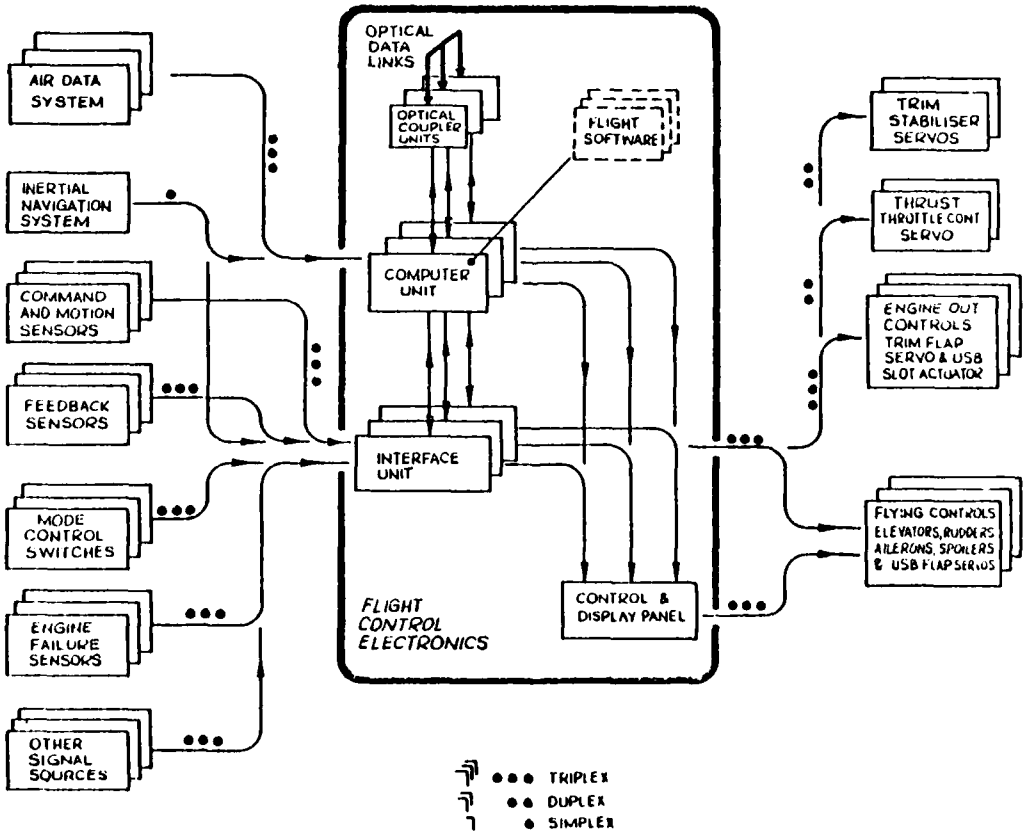


Figure 4.2 YC-14 EFCS Configuration

Each computing lane contained a central processor, program and data stores and input and output interface cards. The EFCS makes use of sets of triplex input sensors whose outputs are consolidated as in Figure 4.3.

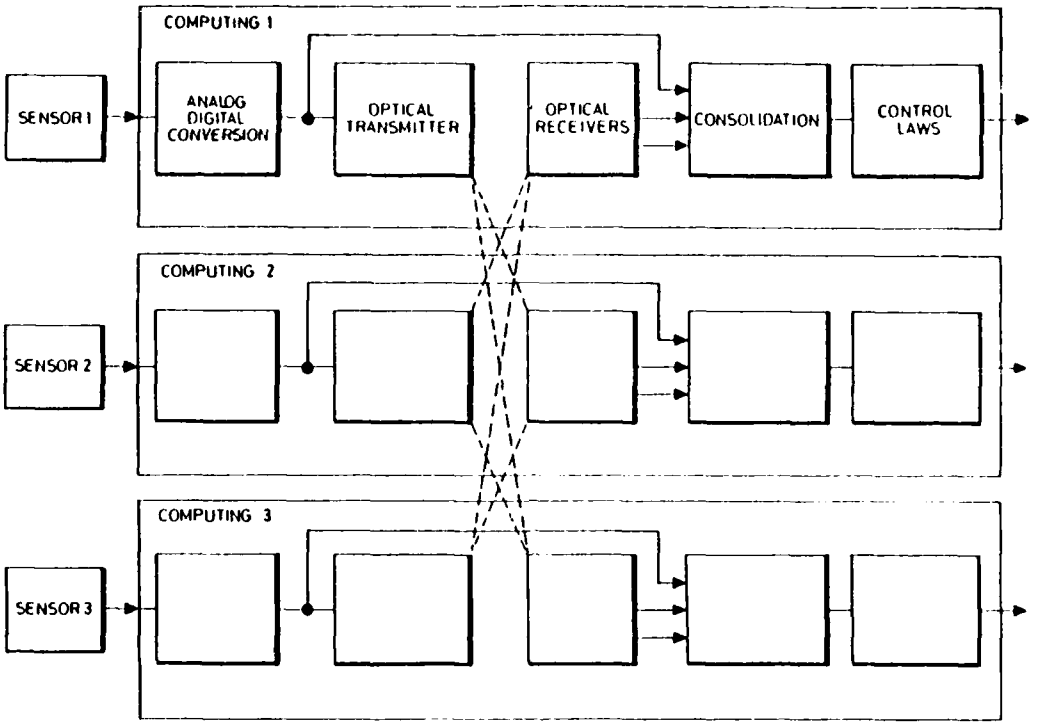


Figure 4.3 Triplex Input Sensor Data Consolidation

Each sensor output is transmitted in serial digital form along optical fibres to each of the other two channels. Thus each computer has available input data not only on its own sensors but also on the sensors associated with each of the other two channels. Identical algorithms in each computer consolidate the input sensor data; this consolidation enables the sensor inputs to each channel to be equalised and it also enables faulty sensor inputs to be detected and isolated. The software in each channel is identical and the program cycles of the three computers are synchronised with each other in order to remove apparent sensor errors due to sampling time differences, and to ensure that the computers are all working on identical input data. On the EFCS (unlike the Tornado AFDS) it is therefore unnecessary to provide means of cross feeding integrator states in order to prevent divergence of computer outputs.

Optical data transmission was adopted for inter-channel communication to maintain the highest possible integrity by removing the possibility of many types of electrical malfunction in one channel being transmitted to one or both of the other channels which could cause loss of the system. This form of transmission is also not subject to electro-magnetic interference and eliminates any chance of electrical earth loops.

Building on experience gained during the Tornado AFDS and YC-14 EFCS developments, the British Aerospace Jaguar fly-by-wire integrated flight control system features a quadruplex architecture. This is required in order to achieve two failure survival which is necessary because there is no mechanical reversion or back-up to the electrical flight controls. In addition the aircraft will eventually fly with a negative static stability margin.

The four processors are a refined version of the purpose built processors used on the YC-14. They are synchronised to each other and the software in each is identical.

Input sensor data for each channel is transmitted via optically isolated serial digital links to the other computers where it is consolidated using a quadruplex version of the arrangement shown in Figure 4.3.

4.2 Software Development for Similar Redundancy

Whereas hardware is subject to both design errors and to component failures, software cannot fail and is subject only to design errors. Thus the software is only dependent on the particular states of its input data over a period of time. Design errors in hardware are reduced to acceptable levels by bench, rig and flight testing. In general procedures for hardware design are well established and hence well understood by both designers and certification authorities. The same cannot be said about software.

In a high integrity system it is clear that the hardware and software need to be produced to the same safe standards.

Gradually, with the consecutive programmes of Tornado, YC-14 and Jaguar FBW, a method for the development of safe software has been refined (Figure 4.4). Experience has been built upon as the safe operation requirements have become more stringent with each new aircraft.

This proven approach to high integrity flight software design, production and testing is directed towards reducing the risk of occurrence of common design errors in flight software to acceptable proportions from the point of view of flight safety.

Procedures have been developed for implementing flight resident software, as specified by a software requirements document, which provide close configuration control and good visibility of the design, implementation and test stages.

A modular top-down structure for the software has been developed, which gives good visibility throughout the development phases. Each design and test function is well documented and described by means of control documents for each aspect of the software development. Design reviews are used at various stages in the software implementation process in order to maintain a continuous and close check on the integrity of the software design. Following design approval at the preliminary design review by the specialist groups associated with the project, a configuration control system is enforced to permit only authorised changes by means of a formal change request procedure.

Strict rules for methods of structuring, designing, scaling, coding, assembling and testing the flight resident software modules have also been developed. Equally strict configuration control rules ensure that these 'production' rules are implemented. Adherence to these rules throughout the software development process is an essential prerequisite for safe software in which the probability of design errors is rendered extremely remote.

These design and control procedures together with the modular structure, simple coding rules, use of a macro expand facility, and the visibility provided by the control document structure all aid the task of analysing the software design for errors. All aspects of the software structure and execution in the computer are analysed and tested for possible errors. This verification of the software is a continuous process covering software requirements, module design and coding.

4.2.1 Design Methodology

Following the production of a software requirements document which details the functions to be performed by the computer program, these functions will be split into modules. Module interfacing documentation is defined at this stage such as hierarchy diagrams (Figure 4.5), input and output variable definitions and scaling information.

The modules which may be called once or many times during a programme iteration are grouped into frames to meet the need for different functions to be executed at different iteration rates to meet the combined requirements of system performance and computing time limitations.

On completion of the software tasks in each frame, the processors enter a 'dynamic stop' or 'pre-master reset' (PMR) mode. Every frame period, a master reset (MR) signal is generated which returns control to the executive module which then calls the next frame. This process is repeated indefinitely; the frames being called cyclically while the processors are running.

This technique enables the program required for a system to be broken down into easily managed sections, each section defining a function or a process. Additionally, a programming change to any one module will not cause changes in other modules, provided the rules of communication between modules are not contravened. Also, the addition or deletion of modules can be achieved without greatly affecting the system program.

4.2.2 Software Testing

Flight software is progressively tested at a number of levels in order to confirm its compliance with the system requirements. During most of its development, the software is tested on a software simulation of the flight computer running on a host computer.

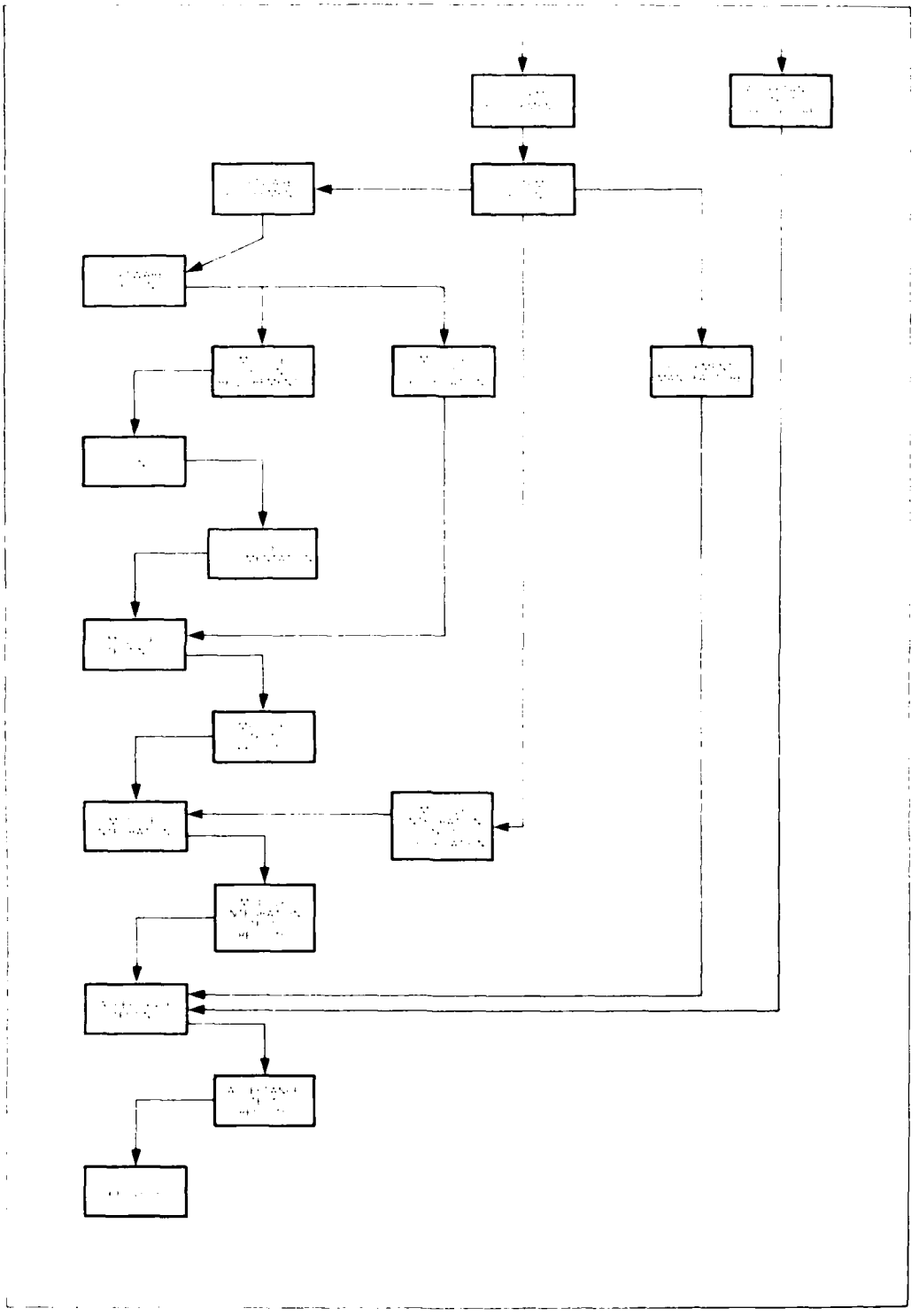


Figure 4.4 Software Development

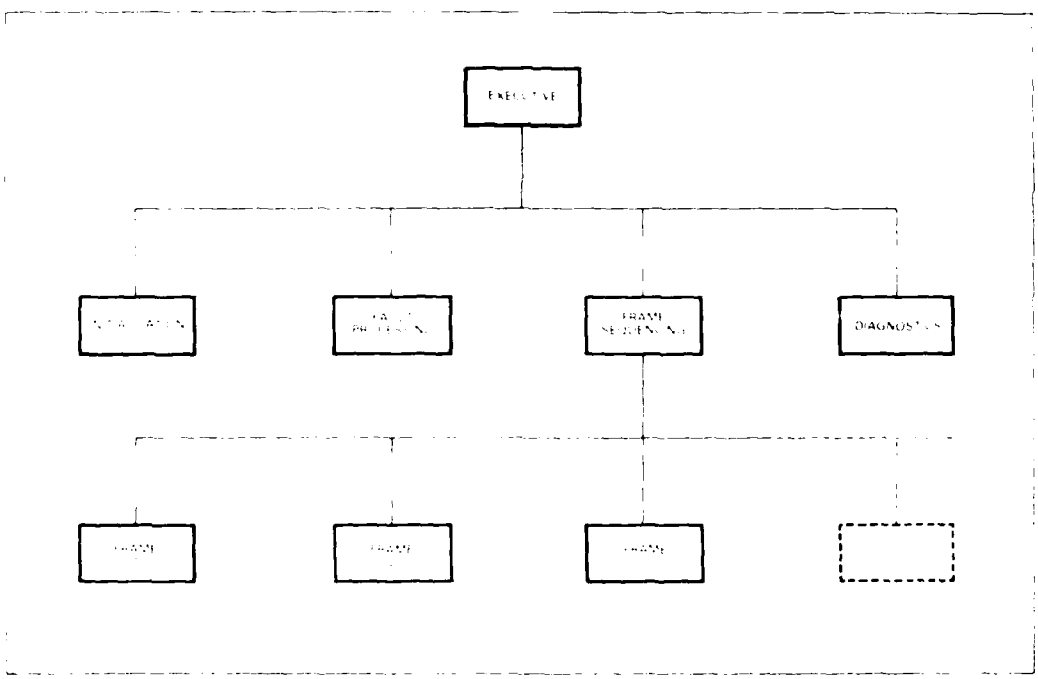


Figure 4.5 Software Structure

Module level testing is concerned with demonstrating compliance with the appropriate module design specification. All combinations of *maximum*, *minimum* and selected intermediate values are tested as well as certain combinations of logic inputs. The tests ensure activation of all decision paths in the module as well as checking for overflow conditions, subroutine calls, parameter passing and the correctness and continuity of the arithmetic.

The programmer who designs and codes a module is precluded from producing the test specifications and performing the tests on that module.

Frame level testing is concerned with demonstrating compliance with the system requirements and specifications. For example, a complete control law axis may be processed by one frame. Tests are made for intermodule compatibility, data/program module compatibility, control flow, data flow, end-to-end resolution, dynamic tests of filters, monitors and limiters.

The testing procedure will be the same as that for the modules and will use the same general purpose test harness. Testing at this stage will be concerned with the detection of program errors and system performance.

Testing of the fully assembled program will be concerned with ensuring that the program functions correctly, as a system. A test specification is produced which checks the software package against the customer's requirements.

Following completion of the total program tests, the flight software is integrated into separately verified flight hardware, although certain critical hardware-related software modules are integrated ahead of this time.

Initial integration tests are made involving end-to-end static tests, the repetition of the total program tests, start up tests and mode sequence tests. This is followed by closed loop testing with a hybrid computer containing an aerodynamic simulation. The complete system is subjected to automated test sequences for all modes of operation and flight conditions prior to any flight testing.

4.3 Multiplex Dissimilar Redundancy

In order to satisfy the computing throughput requirements for previous digital flight control systems, a purpose built processor has been developed. This currently uses a bit-slice implementation to give typically 800,000 instructions per second based on a typical control system instruction mix.

For secondary flight control systems such as spoilers/airbrakes, automatic trim, slats and flaps, and even some primary functions such as yaw damping, the throughput requirements are much less onerous. Computer units for these applications are capable of being implemented using any of a number of 8 and 16 bit microprocessors.

These systems, though relatively undemanding in computing power, nevertheless have high integrity requirements.

An example, currently at the flight trials stage, is the slat and flap control system for the Airbus Industrie A310 commercial aircraft.

Failure survival constraints are imposed upon the electronic control system since there are no mechanical links from the pilot's controls to the surfaces, ie it is a fly-by-wire system. The safety constraints are defined by probabilities for various occurrences, including:

1) Inadvertent deployment of the slats or flaps must have a probability of less than 10^{-9} per flight hour.

2) Slats or flaps no longer operating and no warning given to the pilot, must have a probability of less than 10^{-9} per flight hour.

3) Slats not operable must have a probability less than 10^{-5} per flight hour.

These constraints relate to the entire slat and flap operating systems and include the electronic, mechanical and hydraulic components. Hence the requirement for safe operation of the SFCC has to be better than the above figures.

In considering candidate system architectures to achieve the stated requirements, one of the major considerations was the comparative simplicity of the task in relation to other flight control tasks. Operation of the slats and flaps as performed by the software, is mainly a sequence of logical expressions rather than arithmetic expressions and complex filters as would be found in typical autopilots and autostabilisers. This results in throughput and instruction set requirements that can be readily achieved by commercially available microprocessors. The use of microprocessors allows a significant cost saving when compared to such a system using a purpose built processor. However, the disadvantage of microprocessors in high integrity applications is that their internal workings are not visible. Thus the failure mechanisms of such processors cannot be predicted.

Another major factor in the choice of an architecture is the views of the certification authorities. Lack of experience in the industry and the difficulty in assessing the integrity of software has led to recommendations from the authorities that consideration be given to

a) the use of monitoring, limiting or other provisions which are independent of the digital computation, to reduce the effect of failure within it.

b) the use of dissimilar elements in critical portions of the equipments, particularly where analysis may be difficult or inconclusive (e.g. the processor).

The third major factor is the failure survival requirements. In a control system such as on the Jaguar fly-by-wire aircraft, it is not enough just to identify that there has been a failure. It is also necessary to continue to work correctly after the failure. Hence the necessity to identify where the failure is and to isolate or absorb it. However for the A310 slats and flap system it is sufficient to know that there has been a failure. On recognition of the failure, there are brakes in the wings which are operated to freeze the surfaces in their current position. Although this may mean a slatless or flapless landing or, indeed, an immediate return to the airport from which the aircraft has just been taking off, it does not prevent the continued safe flight and landing of the aircraft. This necessity for fail-safe capability is reflected in the safety requirements as outlined above, item 3 (slats not operable: probability $<10^{-5}$ per hour) being more an availability than a safety constraint.

It was concluded, therefore, that two different microprocessors should be used in parallel to perform the slat and flap operating task. This makes the probability of a common internal failure or design error within the microprocessors extremely unlikely. It also ensures dissimilarity of the software at the code level.

Although this was felt to increase the level of confidence that a common coding error would be minimal, it does not resolve the problem of common software structure or algorithmic design errors. Hence it was decided to perform two complete software development tasks, one for each microprocessor, with only the system requirements as produced by the customer being common to the two.

4.3.1 System Description

A schematic diagram of the computer architecture is shown in Figure 4.6. Two computers are required in order to meet the availability requirements.

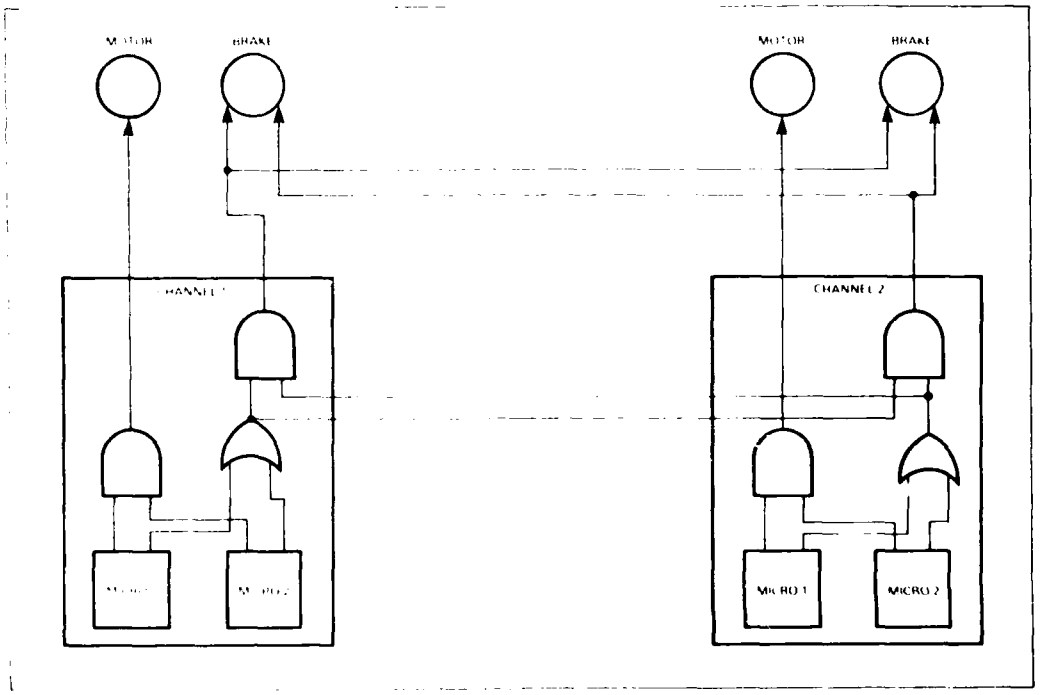


Figure 4.6 Computer Architecture Schematic

Internally each of the computers contains two different microprocessors. A computer will only drive its hydraulic motor if both microprocessors agree, otherwise the motor output is locked leaving the other computer to drive the system.

If a computer fails, this will be detected by the different outputs from microprocessors 1 and 2 and the computer will then isolate itself.

If there is an uncommanded movement of the output e.g. a torque shaft breaks, then both computers will detect the incorrect deployment and will operate the wing brakes. This freezes the flaps/slats in their current position.

4.4 Software Development for Dissimilar Redundancy

A separate software requirements document (SRD) is produced for each lane. Having produced the SRD, the software development procedure then follows the normal path, in each lane, of top-down analysis to produce a modular structure, and to design and generate the code for each module.

At this stage, instead of embarking on module testing, the approach that has been taken is to assemble the software for each lane and then to perform hardware/software integration testing. It is felt that, since the lanes must agree to provide the computing function, they each provide for the other the most stringent test environment. This test philosophy is amplified in section 4.4.2.

To avoid the possibility of design errors being introduced by a common assembly fault, two different host computer facilities are used to assemble code for the two processors.

The choice of microprocessors from two different suppliers further reduces the risk of the associated assembler packages having common errors.

Each microprocessor memory is loaded from the associated development system in a dedicated format. These formats, which are different, are read by the PROM programmer which is independently checked for correct function.

The two software development activities are kept completely separate, the two programs eventually being proved by integration with the hardware.

A brief summary of the software development process is shown in Figure 4.7.

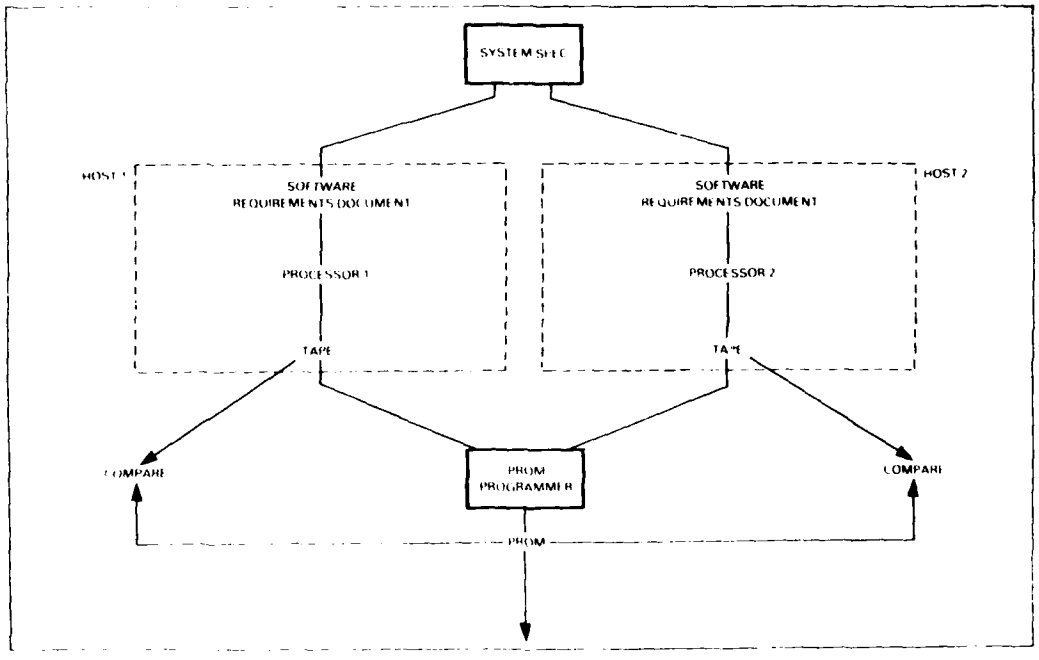


Figure 4.7 Dissimilar Software Development

4.4.1 Design Methodology

One disadvantage of trying to verify software by testing is that it is extremely difficult to prove that the software is free from design errors. In a similar redundant system the probability of an undetected software design error adversely affecting the system safety must be compatible with the integrity objective.

For the Airbus Industrie A310 SFCC, the software has been prepared twice, using two different software development facilities. The flight resident software suites thus produced are executed by different, asynchronously operated microprocessors. The outputs of the two microprocessors are continuously compared and any difference greater than a defined threshold causes the system to disconnect after a preset time delay, with all output drives being removed.

A design error in one of the two dissimilar lanes can never produce a hazardous output from the system.

Design errors in both dissimilar lanes could only produce a hazardous output if their resultant effect was identical and occurred within the pre-set time delay of the cross-lane comparison.

The benefits of dissimilarity are clear. The precautions that have been taken to ensure that dissimilarity is maintained are listed below.

- Software for each of the two processors is prepared by two different groups.
- The processors are different for the two lanes, are made by different manufacturers and have different instruction sets.
- The two suites of software are prepared and assembled, using two different software facilities - also made by different manufacturers.
- The processors obey instructions which have different object code and are located in differently mapped program stores.

- The data used by the processors are located in data stores which are also differently mapped.

- The processors have separate clocks and operate asynchronously so that there is no requirement for frame synchronisation to link them together.

- The program store for each processor is loaded using a PROM tape, the format of which is different for the two lanes, since each is generated by a different support system.

- Discrete words exchanged cross-lane for monitoring have their bit pattern deliberately shifted. This avoids the possibility of a common error when comparing the two lane outputs in each lane.

With these precautions it is unlikely that the same software design error can occur in both lanes to produce the same hazardous and undetected output simultaneously from both lanes.

4.4.2 *Software Verification*

In a similar redundant system, where software is common to all lanes, a single software error could cause loss of the total system. A software verification procedure has, therefore, been developed which reduces the software risk to acceptable proportions.

Several of the techniques thus developed incur no significant cost penalty and can be applied directly to the dissimilar approach.

With other techniques, such as module testing, the extent to which they are applied depends on the complexity of the module function and the consequences of not detecting errors. These techniques have to be critically reviewed and applied as necessary to achieve the required level of integrity.

Testing

The duplication of software is considered to provide sufficient integrity to meet the safety requirements of a system in which availability is not a predominant requirement.

Nevertheless, it was felt that, to provide the required availability, certain module tests needed to be performed. The modules are divided into three categories:

a) critical modules which will be subjected to full testing e.g. those that drive system outputs or monitor for asymmetrical deployment or runaways.

b) modules that will require supplementary testing because the software/hardware integration tests do not accurately check certain thresholds.

c) modules requiring no testing other than that performed during hardware/software integration.

5. CONCLUSIONS

A major task in any flight control system development is the proving that the design achieves the required integrity level. The conventional approach has been to duplicate the hardware and software. This paper has described an alternative approach of using dissimilar design and implementation. In the following paragraphs these two approaches are compared and contrasted in order to isolate the essential parameters required to decide which approach to adopt for a particular application.

The production of high integrity software involves four distinct phases:-

a) Accurate, unambiguous definition of requirements.

b) Structuring and preparation in a clearly visible manner to reduce the probability of design errors during implementation to an acceptable level and to assist in the identification and elimination of those errors that occur.

c) Testing to prove that the functions defined in the specification are realised and that no unforeseen and unwanted functions exist.

d) Configuration control to prevent the introduction of design errors into proven software when making changes.

In preparing software for the A310 SFCC described in this paper, proven procedures for items (a), (b) and (d) that have evolved over a period of more than a decade have been used.

With regard to testing, item (c) above, it is difficult to write more than a few words of software and guarantee that it is entirely free of errors. It is equally difficult to apply any measure of integrity to the software so produced. At best, one can only test extensively and state that the software is correct 'beyond all reasonable doubt'.

For a secondary flight control system, availability may be a less stringent requirement than safety. In this case the need for extensive testing can be avoided by adopting a dissimilar software and hardware approach in which the outputs of two dissimilar microprocessors are compared. Commands which agree are executed, commands which disagree for more than a predetermined time cause the complete channel to disconnect. Asynchronous operation and dissimilar hardware means that the probability of any fault condition, whether due to failure or design, causing the same erroneous output, simultaneously in both microprocessors, is negligible.

Provided that the software is tested sufficiently to ensure that availability is acceptable it is believed that this approach inherently gives higher confidence that faults will be detected than the more conventional extensive testing at module level which is essential for a similar redundant configuration.

However, for an application where a fail operative system is necessary, three dissimilar software suites may be required. This would incur a large software development cost overhead in comparison to an extensively tested single program for a similar redundant system. In addition there would be three sets of comparisons to make on dissimilar results in order to detect the presence of an error and isolate its source. This could lead to a different threshold for each comparison in order to reduce the occurrence of nuisance disconnects.

In short, the integrity of a system is dependent on the error probability and the availability is dependent on the nuisance disconnect probability. A dissimilar system will inherently have a lower probability of failing to perform correctly due to a single design error than a similar redundant system. Conversely, the similar redundant system which uses consolidated sensor data and synchronised processing, will inherently be less prone to nuisance disconnects than a dissimilar system in which the processing is asynchronous and sensor data is not necessarily consolidated.

In a similar redundant system, time is spent reducing the probability of there being residual design errors when the software is in service, by performing extensive testing. For the A310 dissimilar SFCC it has been found during hardware/software integration that most errors are due to there being more than one interpretation of the system requirements and to the difference in computed outputs from each lane causing nuisance disconnects. Hence more time has to be spent in reducing the occurrence of such disconnects than in a similar redundant system.

Therefore, the choice of a similar or dissimilar approach needs to be weighed in the light of the relative integrity and availability requirements. A system having high integrity but low availability requirements is better served by dissimilar software, the inherent integrity being at the expense of the higher rate of nuisance disconnects. Conversely, a system having high integrity and high availability requirements may be better served by a similar software architecture since the probability of a nuisance disconnect must be reduced to the same level as the probability of a design error.

REFERENCES

Corney, J., 1980, "The Development of Multiple Redundant Flight Control Systems for High Integrity Applications", The Aeronautical Journal, Royal Aeronautical Society, October.

Shepherd, J., 1982 "A Method of Designing Fault Tolerant Software". Certification of Avionic Systems Symposium, Royal Aeronautical Society, 27 April.

THE COST OF SOFTWARE FAULT TOLERANCE

Gerard E. Migneault
NASA Langley Research Center
Hampton, Virginia

SUMMARY

This paper proposes the use of software fault tolerance techniques as a means of controlling the cost of software in avionics as well as as a means of addressing the issue of system unreliability due to faults in software.

Observations are first made about the problem of escalating budgets for software and about the nature of some of the causes of the increased costs, the nature of possible actions and methods proposed--and not proposed--for addressing the problem. An experiment in the measurement of software "reliability" is briefly mentioned in order to support the construction of a simple model relating the cost of a software module to the effect upon the reliability of systems containing the module.

Attention is then paid to schemes for using dissimilar redundancy in software to obtain a degree of tolerance to software faults in systems which must achieve high levels of reliability. Another simple model is developed--expressing the relationship of a "fault tolerance" scheme to system reliability. The model serves to discuss and question the customarily expected benefit, an increase in system reliability, to be obtained from fault tolerance schemes.

Finally, the simple models are combined to develop a system level view of the relationships among cost, redundancy and reliability. The view suggests the strategy, unconventional in the software world, of deliberately choosing to develop less reliable, dissimilarly redundant software modules in order to lower total software costs and increase the credibility of the estimates of their "reliability."

INTRODUCTION

The assertion that the costs related to software have become significant, even dominant, factors in budgets for the acquisition and use of digital systems is widely accepted. Consequently, more attention is being devoted to understanding and developing methods for forecasting and controlling or reducing the costs. No adequate complex of methods appears to have yet come into use, however, and the growth in total cost continues--seemingly unconstrained when compared to the decreasing costs of associated hardware.

In general, the techniques available or proposed for abating the costs of software have had two common characteristics of particular interest. They have been conventional in the sense of being variants of quite general notions which are commonly, perhaps uncritically, believed to have beneficial effects upon costs. Additionally, they have been nonspecific. That is, with the exception of the customary prerogative of management to control the level and duration of utilization of resources, the techniques have provided no means by which arbitrary, but specific, amounts of costs could be exchanged for equally specific amounts of alternative consequences. For example, the concept of a "chief programmer team" is a particular application of the notion that the structure of an organization affects the quality of its product, an extrapolation of the aphorism that the structure of a system is a determinant of its behavior. Consequently, the utility of the technique is not questioned in principle and a priori, although the relation between marginal benefits and costs is nebulous and fractional application of the technique is clearly not an option.

Software related costs have grown for a number of reasons. The most readily obvious factor is undoubtedly inflation in the general economy. While an increase in costs due to inflation does not represent a real increase in the use of resources, it does indicate that the mix of resources utilized has become less optimal. This suggests that in order to counter the effect of inflation a successful cost reduction or control scheme should implicitly, if not explicitly, address the redistribution and replacement of costlier resources with less costly. Thus schemes which are intended to provide greater visibility and control of the existing development procedures are not likely to be very successful. The concept of a "chief programmer team" appears to be in this category, as are schemes to increasingly formalize documentation requirements and change configuration control procedures.

Perhaps a more significant cause of increased costs is the simple increase in the total amount of software required as digital systems with embedded software replace older technology. Not only do the costs increase in proportion to the increased amount of software, but, as the discipline of economics teaches, in the absence of equally rapid technological progress, more and thus less efficient resources must be used. In the case of software, a people-intensive activity, this means a lowering of the average level of capability of the personnel, technical and managerial, in both the development and maintenance phases of the software life cycle. This suggests that schemes which would be successful in countering this cause of increases in cost must lessen the need for people in the software life cycle. This can be accomplished either by eliminating activities in the software life cycle or by increasing the productivity of the people involved in the activities. If such schemes require an excessive investment of capital or the introduction of costlier resources in other areas of the software activity, they may not be successful as cost savers. The introduction of "programmer workbenches" and "higher order languages" are such productivity improvement schemes. They require considerable investment of capital and continued use of the more knowledgeable personnel. Moreover, they are selective schemes in that they affect the implementation stages of program development more than testing and maintenance stages. Considered as a cost control scheme "correctness proving" would appear to be in the category of schemes which eliminate an activity. That is, if a program were absolutely correct, then there would be no need for its maintenance. Presently, however, the technique requires more expensive resources, in terms of personnel and computer time, than it releases. Also, there is currently no agreement that the technique ever will generate the desired "perfect" software.

Increased costs of software have also been caused by more demanding requirements for digital system performance, requirements which have been achieved by means of ever more sophisticated software. To some extent these costs are the result of the very success of digital systems in providing computational capabilities which were not previously available. The increased complexity of the software requires either more knowledgeable, and thus more expensive, personnel in the development and maintenance phases or causes an increase in maintenance activity. This suggests that to counter the effect of demands for increased performance and sophistication, schemes should be sought which reduce the complexity inherent in software. This seems to be the goal of "structured programming." However, it seems to require additional rather than less training of personnel.

The notion advanced in this paper is that in an avionics context, and possibly in other contexts in which there is an appropriately demanding requirement for reliability and maintainability, techniques of software fault tolerance utilizing redundant modules of software can be used to control costs. They do the "good" things previously cited. The level of redundancy, a parameter usually considered only for design purposes, becomes available to management as a parameter for controlling costs. But more important, from the point of view of inhibiting acceptance, the notion is unconventional--in the world of software. It is unconventional because software fault tolerance techniques have been developed to enhance reliability and are considered to be more, not less, costly and therefore less, not more, desirable. Indeed, the suggestion for this paper arose as a reaction to a statement which expressed a consensus in a "working meeting" and was unchallenged in light of its apparent logic. The statement was that, for the purpose of defining requirements, the use of a quantitative measure of the "reliability" of software should be shunned since it would necessitate the use of software fault tolerance and redundancy techniques which would, in turn, increase costs. Hence because of the less conventional nature of the technique proposed and the need for its justification, this paper appears somewhat polemic.

MODELING COST

In order to express a relation between the cost of generating software and its reliability requirement, we borrow from a recent experimental study of software reliability (Nagel, 1982). Data from the study support the assertion that

> after k faults have been corrected in a program, the probability of error during each succeeding execution of the program can be approximated by the constant $e^{-(a+bk)}$, where the parameters a and b depend upon the program and the statistical distribution of the input data and can be estimated from data obtained during a controlled process of uncovering the faults.

The terms "fault" and "error" used in the preceding statement are not synonomous. A program is understood to be simply an embodiment of an abstract relation between variables which is usually defined by a specification, implied by a requirement, etc. Thus a program, the embodiment, has structure which is not part of the abstract relation. The program can be created with a structure which, for some inputs, generates outputs which are not those implied by the abstract relation. "Fault" refers to such a flawed structure; it can be remedied, presumably when its presence is signaled by the occurrence of an error. No statement is made here about the process by which faults are generated. "Error" refers to output data which, while consistent with the structure of the software which generated them, differ from the values implied by the original abstract relation. During operation, it is the error which propogates through a system; it is the error which can be detected and can signal the presence of a fault. Whereas faults have "always" existed, errors "occur" and thus correspond to events which have rates of occurrence. An execution of a program refers to the generation of an output data set in response to an input data set. The time period of an execution is assumed to be small compared with the use period, the many executions, of a program. While it might be possible in the future to estimate the parameters a and b from descriptive information about the program at the completion of a standardized acceptance test, for the present discussion it is sufficient that they can be estimated by a controlled process of repetitive trials beginning after a standardized acceptance test.

One conclusion to be drawn from the referenced study is that a software module in a system can be considered to be a component having a constant error rate during the time it is in operation (which is assumed to be a fraction of the elapsed time of system operation). Thus, conventional notions of reliability can be discussed if errors from software modules are considered to be causes of digital system failures. Of course, not all software errors would likely result in digital system failure; what will constitute a failure will depend upon the application. Therefore, equating one with the other is a conservative assumption--which will be considered again below. With this assumption, computations of mean time to system failure due to software module error have some meaning. Conversely, a reliability budget for the various components of a digital system can assign a maximum allowable error (failure) rate to a software module.

Thus, assuming that a meaningful input data stream can be obtained or generated and assuming the successful accomplishment of the process of uncovering and removing k faults from a software module, and the estimation of the parameters a and b in the process, the error (failure) rate of a module during its subsequent operation can be expressed as

$$\lambda_k = 3600\ m\ e^{-(a+bk)}$$

where m is the number of executions per second required by the application. The expected amount of time taken during the controlled process to discover and remove the k faults, that is, to "debug" a module to a criterion λ, can be expressed as

$$MTTD_\lambda = r\sum_{i=0}^{k-1} \frac{1}{\lambda_i}$$

$$= \frac{m\,r}{c(1-e^b)} \left(\frac{1}{\lambda} - \frac{e^a}{3600m}\right)$$

where 1/c is the time per execution, r denotes a number of repetitions required during the "debugging" process in order to gather the data from which a and b are estimated. Note that the term "debugging" is here used for the controlled process of testing an "accepted" software module to a required λ level.

Additionally, in the assertion above there is an implied ordering of faults in term of their contribution to error rates of software modules. This means that each fault will require an increasingly long time to be uncovered. If a module is not reliable at the end of "acceptance" testing, data for estimating the parameters a and b will be relatively easily accumulated and a long "debug" phase will be forecast. If a module is relatively reliable at the end of "acceptance" testing, then it will take a correspondingly longer time to accumulate the data for estimating the parameters. In either case, ensuring that the probability of subsequent errors appearing during operation of the software module will be arbitrarily small will be lengthy activity.

On the assumption that a conventional development procedure consisting of a requirements development phase, a program design phase, a program coding phase, and a standard functional and "acceptance" testing phase can be selected, and that "debugging" to criterion λ proceeds from the point of "acceptance", figure I depicts the cost profile for "developing" a software module to a λ critierion.

Let $\$_0$, represented by the area under the large hump in figure 1, be the cumulative cost of developing the module through the standard "acceptance" testing point as discussed in various software cost models appearing in the literature, for example (Putnam, 1978). It might be well to note that available software cost models do not appear to be very accurate forecasters and must be calibrated for each software development environment (Thibodeau, 1981). Here, as will be seen later, it is sufficient to note that two modules developed (to the point of acceptance testing) from the same functional requirement would be expected to have similar total costs--as forecast by the available models.

Assuming the "debugging" activity to be a constant rate activity, let $\dot{\$}$ be the cost per unit time of "debugging" the software module further to its specified criterion λ. $\dot{\$}$ is assumed to include the continuing cost of generating the input data sets (stream). The cost of "debugging" is simply represented as

$$\dot{\$} \times MTTD_\lambda$$

Recalling the expression for $MTTD_\lambda$ above, we express the expected cost of a software module as a function of its λ criterion as

$$\$_\lambda = \$_o + \dot{\$}\frac{m\,r}{c(1-e^b)} \left(\frac{1}{\lambda} - \frac{e^a}{3600m}\right)$$

The values of $\$_0$ and $\dot{\$}$ will, of course, depend upon the complexity of the software module and the size of the staffs required to develop and "debug" it, and upon the particular software development environments. In figure 2 the ratio

$$\frac{\$_\lambda}{\$_o}$$

is plotted versus λ for various values of the parameter

$$\frac{\dot{\$}\,m\,r}{\$_o c(1-e^b)}$$

A glance at the figure suffices to indicate that any significant reliability requirement implies a considerable increase in development costs over software with unstated reliability. Consequently, in light of the demanding reliability requirements associated with avionics, it is unrealistic not to expect an increase in software costs if system reliability is to be achieved by extending the reliability of conventional software modules. The task is to minimize the increase.

Of course, if the λ criterion truly reflected the reliability required of the software module and if the criterion were achieved, then an occasional occurrence of a system failure due to a malfunction of the software module after operational deployment of the system would not occasion any maintenance activities. By definition the occasional failure would be acceptable. Indeed maintenance action would be suspect unless it included a repetition of the "debugging" process described. In this case, maintenance costs would be negligible, consisting principally of an accounting system to verify that the occasional errors (failures) were no more occasional than forecast. In this sense, the extent to which maintenance activities are in response to component failures is a gauge of the extent to which reliability requirements are not truly being imposed on today's systems--either by oversight or by deliberate decisions made to exchange the costs of obtaining the desired reliability for costs at a later time, the maintenance costs. Such decisions should not, of course, be within the purview of the digital system developers alone, and certainly not of the software developers, since they must (should) consider the costs resulting from the unavailability of systems--a consideration to be left to the users of the system.

FAULT TOLERANCE

The concepts of fault tolerance and redundancy are hardly new in engineering. An automobile's spare tire is a mundane witness to this fact. Nor is the idea of building more reliable systems from less reliable components by means of redundancy and passive fault tolerance new in electronics and computers (Moore, 1956). The cost benefits in terms of reduced maintenance and outages of hardware systems with internal redundancy have also been addressed (Moreira de Souza 1981). What is novel is the notion that fault tolerance schemes can be devised to prevent system failure (or unavailability) due to design flaws (Anderson, 1981). Software faults are just such design flaws, and software fault tolerance schemes have been proposed in the past decade. The Recovery Block scheme and N-version programming are perhaps the two most widely known schemes.

Consider one "stage" of an N-version programming scheme variant as represented in figure 3. For convenience, the stage consists of an odd number, N, of dissimilar versions, P_i, of a program module which each receive input data from one of N dissimilar voter modules, Y_j. Each voter module performs a majority vote on the set of inputs, Y_j, which it receives from each of the dissimilar modules of a prior stage. The outputs, X_j, of the N program modules, P_i, in turn provide inputs to the voters of one or more subsequent stages. Thus the majority value of the set of outputs, X_j, defines the output, X, of the stage. It will be correct if a majority of the X_j are correct, and erroneous otherwise. In a similar fashion, the majority value of the Y_j defines the input, Y, to the stage. In all likelihood, even if all are correct the Y_j will differ in some small amount due to the dissimilarity of the modules generating them. What this means is that the voters will contain some complex logic to account for such legitimate variations. In effect the probability of errors in the execution of these modules cannot be dismissed as insignificant.

Consistent with the previous discussion, with each program module and voter module there is associated a probability that an execution of the module will produce an erroneous output, and the probabilities can be determined, and indeed made equal, by means of the "debugging" process. Then the probability of error, q, in an execution of a program-voter pair is simply

$$q = q_p + q_v - q_p q_v$$

where q_p and q_v represent the program and voter execution error probabilities. In the context of its application, a program-voter pair will appear to have an error (failure) rate (per hour)

1) $$\lambda = 3600\ m\ q$$

The question of the independence of execution errors in "independently" developed and tested software modules is a troublesome matter. On the one hand, the study of the "reliability" of software has not progressed beyond primitive models of individual software modules. On the other hand, studies of fault tolerance and redundancy have usually been focused upon the mechanisms of the schemes, reflecting in part a less than unanimous and enthusiastic belief in the credibility of current software "reliability" assessment methodology in the computer science community. There are exceptions, of course - for example, a study of the feasibility of the application of the recovery block scheme in an avionics application (Hecht, 1978). We shall return to the question later, but here assume that errors in module-voter pairs occur independently of errors in other module-voter pairs. With that assumption, the probability of an erroneous stage execution output can be expressed as

2) $$P_{bx} = (1-q)^N \sum_{i=\frac{N+1}{2}}^{N} \binom{N}{i} \left(\frac{q}{1-q}\right)^i$$

and a relationship between the stage error (failure) rate

3) $$\lambda_s = 3600\ m\ P_{bx}$$

and the component program-voter error rate, λ, established. The relation is plotted in figure 4 for various values of the parameters m and N. Because of the questionable assumption of independence of errors the plotted curves represent bounds on what is achievable.

Two "fault-tolerant" computer systems, SIFT and FTMP, have been developed under NASA contract and reported in the literature (Goldberg, 1981)(Hopkins, 1978). The design goal of the systems was to achieve, at some reasonable cost, systems of very high reliability for avionics applications. Neither of the systems utilized software fault tolerance schemes. However, as can be seen from figure 5, the architecture of the SIFT computer lends itself admirably to the N-version scheme described above and can be used to describe how the scheme would actually be implanted into hardware. Simply, each program-voter module pair would reside in a separate processor. With more processors available than required by an N-version stage, in the presence of a hardware failure, the hardware reconfiguration algorithm of the SIFT computer can assign another processor to be in the stage, thus maintaining the stage redundancy at a constant N.

COST VERSUS REDUNDANCY

The relations developed above may be combined to express the cost of software fault tolerance in terms of its level of redundancy, the reliability which it is intended to provide (actually the error (failure) rate which it is not to exceed) and the cost parameters. Recalling the expression for the expected cost, S_λ, of a software module as a function of its lambda criterion, we represent the cost of an N-version stage as

$$2\ N\ S_\lambda$$

reasoning that the N program modules will have the same expected cost as a result of having been "developed" to the same requirement. This is not inconsistent with the accuracy of the current software cost estimation models, as was noted previously. The factor 2 is included to account, hopefully conservatively, for the N versions of the voter modules which, as was noted, will contain some amount of complexity. This cost is compared to the cost of a single module having the error (failure) rate λ_S, the error (failure) rate desired of the stage, by forming the ratio

$$\frac{2 N \$_\lambda}{\$_{\lambda_S}}$$

in which λ_S and λ are related by the equations 1), 2), 3). In the special case N=1, there are no voter modules and the ratio is unity. In figure 6 this ratio is plotted against λ_S for various values of N, and the conglomerate parameter

$$\frac{\$_m r}{\$_o c(1-e^b)}$$

Note that each point along one curve, determined by a fixed set of the parameters above corresponds to a different value of λ --satisfying the relations 1), 2), 3). Hence, given values for λ_S, m and the conglomerate parameter, corresponding to an application requirement and development environment, one can determine the values of λ and N which minimize the ratio.

Note also that in some cases the optimal policy is to use surprisingly unreliable program-voter pairs. When such is the case, several changes in policy suggest themselves. First, it becomes feasible to accumulate more error (failure) history data on modules than is customary, thereby providing greater confidence in the estimates of their (un)reliability. Secondly, it becomes economically feasible to subject the software modules to real-use testing since observable results will occur quickly, thus providing a better understanding of the relation of module errors to system failures and a rationale for relaxing the conservative assumption equating errors to failures, if appropriate, and addressing criticisms about the incompleteness of testing based on solely simulated input data streams.

While the problem of independence of errors remains, it is tempered by the almost certain knowledge that only a fraction of a module's errors will be correlated with those of other modules. How significant or insignificant the fraction is is an appropriate subject for study and experiment in light of the cost benefits available from software fault tolerance. It is further tempered by the additional, almost certain knowledge that only a fraction of software execution errors will cause system failure. Again, study and experimentation is warranted--especially in light of the propensity of humans to believe that they know more than they do when dealing with subjective probabilities (Lichtenstein, 1981).

CONCLUSION

The thesis of this paper, simply put, is that decisions about the use of redundancy in software fault tolerance should be made with the understanding that they provide a cost minimization as well as reliability enhancement potential, and the rudiments of a technique have been presented.

REFERENCES

Anderson, Thomas , and Lee, Peter A., 1981, Fault Tolerance, Principles and Practices, Prentice-Hall International, London.

Goldberg, Jack, 17-19 November 1981, "The SIFT Computer and its Development" in 4th AIAA/IEEE Digital Avionics Systems Conference: Collection of Technical Papers, AIAA, New York, pp.285-289.

Hecht, Herbert, February 1978, Fault-Tolerant Software Study, NASA Contractor Report #145298, The Aerospace Corporation, Los Angeles, California.

Hopkins, A. L., Smith, T. B. and Lala, J. H., October 1978, "FTMP - A Highly Reliable Fault Tolerant Multiprocessor for Aircraft" in Proceedings of the IEEE, Vol. 66, No. 10, pp. 1221-1239.

Lichtenstien, Sarah Fischoff, Baruch, and Phillips, Lawrence D., June 1981, Calibration of Probabilities: The State of the Art to 1980, Contractor Report PTR-1092-81-6 prepared for the Office of Naval Research Contract #N000l4-80-C-0150, Decision Research Division of Perceptronics, Inc., Eugene, Oregon.

Moore, E. F., and Shannon, C. E., 1956, "Reliable Circuits Using Less Reliable Relays" in Journal of the Franklin Institute, Vol. 262, Part I, pp. 191-208, Part II, pp. 281-297.

Moreira de Souza, J. and Landrault, C., April 1981, "Benefit Analysis of Concurrent Redundancy Techniques" in IEEE Transaction on Reliability, Vol. R-30, No. 1, pp. 67-70.

Nagel, Phyllis M., and Scrivan, James A., February 1982 Repetitive Run Experimentation and Modeling, NASA Contractor Report #165836, Boeing Computer Services Company, Seattle, Washington.

Putnam Lawrence H., July 1978, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem" in IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, pp. 345-361.

Thibodeau, Robert, April 1981, An Evaluation of Software Cost Estimating Models, Contractor Report#1-940 prepared for RADC Contract F306-79-C-0244, General Research Corporation, Huntsville, Alabama.

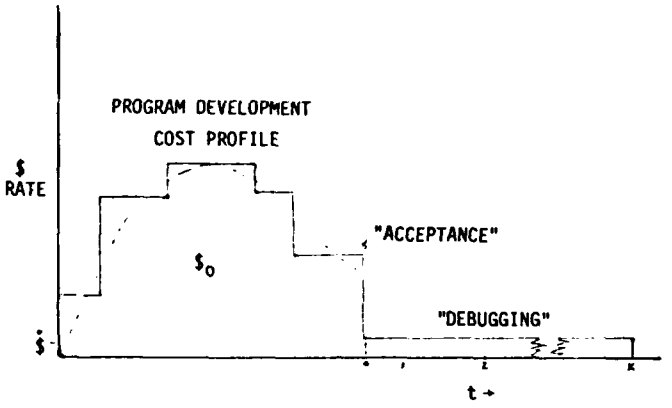


Figure 1

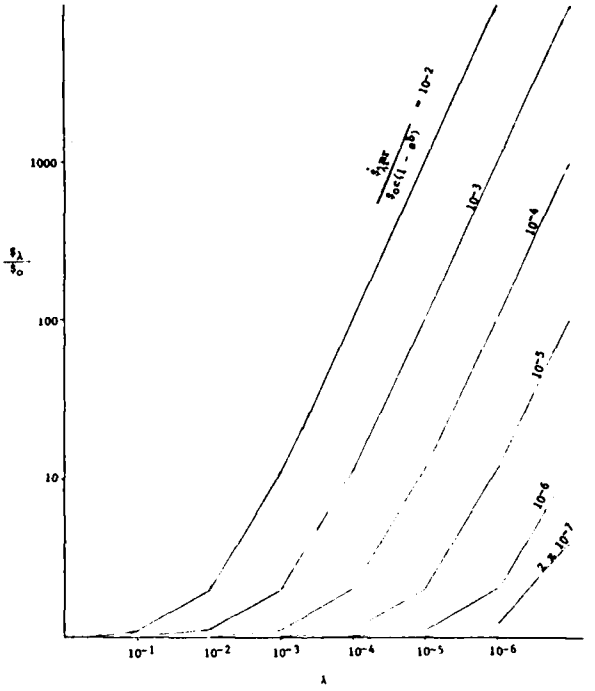


Figure 2

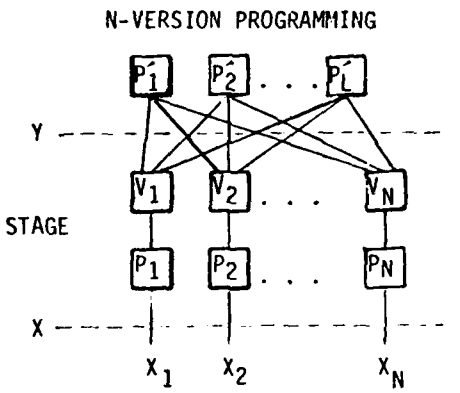


Figure 3

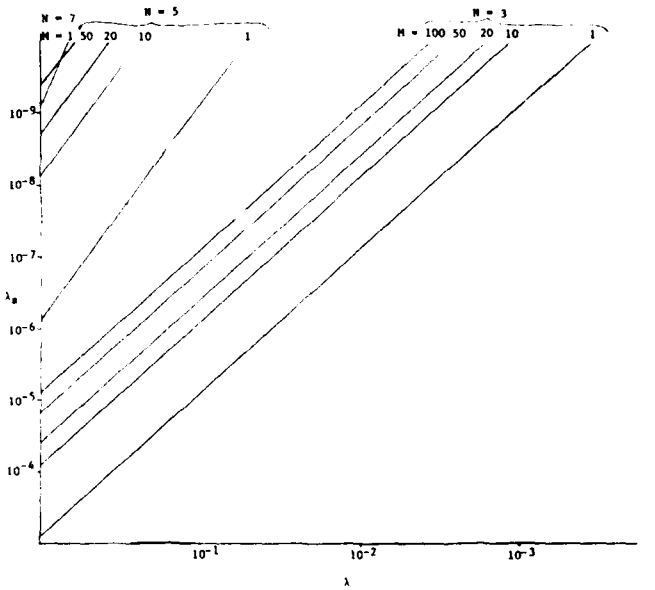
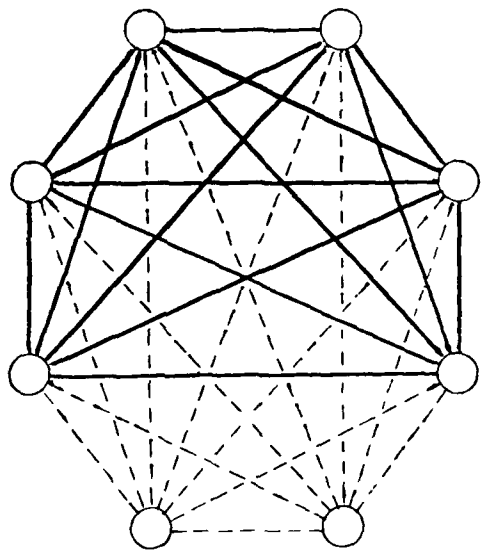


Figure 4



6 PROCESSOR SIFT CONFIGURATION
(EXPANDABLE TO 8)

Figure 5

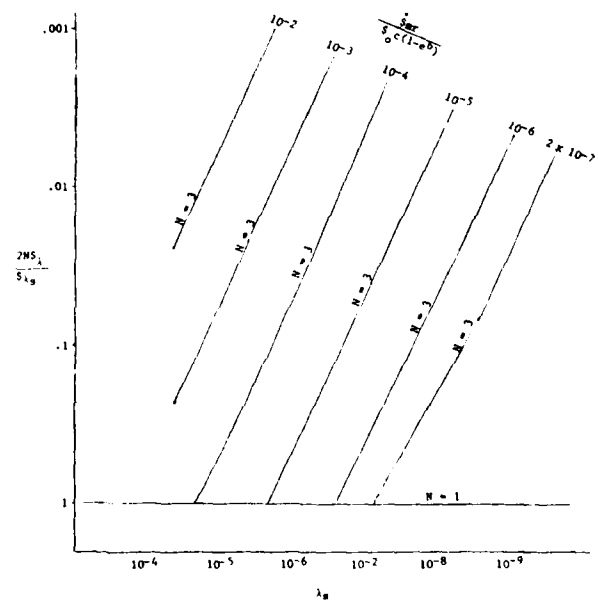


Figure 6

DISCUSSION FOR AVIONICS PANEL FALL 1982 MEETING ON
SOFTWARE FOR AVIONICS
Session 4 : SOFTWARE VERIFICATION AND VALIDATION - Chmn R. O. Mitchell (US)

Paper Nr. 27 - AN EIGHT POINT TESTING STRATEGY FOR REAL-TIME SOFTWARE
Presented by - R. E. Wilson
Speaker - E. J. Dowling
Comment - You have described a comprehensive strategy for testing; can you say something about any standard tools you have to support this strategy?

Response - In general no "standard" tools have been found to meet our requirements, but we have begun to develop some in house. Bureau "subsystem" testing environments are already available and in use. A general purpose "stimulation" software set to provide realistic input to test rigs is also available within the division.

Paper Nr. 27 - AN EIGHT POINT TESTING STRATEGY FOR REAL-TIME SOFTWARE
Presented by - R. E. Wilson
Speaker - A. Cameron
Comment - Repeated stress is laid upon the need for documented results of tests, which "should be carried out by external agencies" - e.g.: senior programmer who is not on the project, customer representative, etc. Please indicate who prepares the lists of tests to be carried out and how sufficiently rigorous testing is assured?

Response - The system designer begins the preparation of a testing plan in parallel with his initial design and these two items, design and test plan, are reviewed in parallel. To assure that the testing is carried out a number of mechanisms are used including check lists and sample testing by quality control staff.

Paper Nr. 27 - AN EIGHT POINT TESTING STRATEGY FOR REAL-TIME SOFTWARE
Presented by - Dr. R. E. Wilson
Speaker - D. J. Dekker
Comment - Does it not mean a lot of work to test the higher level modules already at point 1, using, the full input and output tests? It seems more easy to postpone the testing of the higher level modules to point 4.

Response - We do not spend very much time in coding the lower level stubs at point 1 testing. In general you will not be happy with code whose tests have been postponed.

Paper Nr. 28 - TORNADO FLIGHT CONTROL SOFTWARE VALIDATION: METHODOLOGY AND TOOLS
Presented by - Dr. Ing. R. Pelissero
Speaker - Dr. W. J. Cullyer
Comment - In the simulation computer, did you use the non-linear equations predicted by theory, by the aerodynamics or did you use the data from the prototype aircraft?

Response - Both. At first we used theoretical data. As results were obtained on the prototype aircraft we progressively modified these data

Paper Nr. 28 - TORNADO FLIGHT CONTROL SOFTWARE VALIDATION: METHODOLOGY AND TOOLS
Presented by - Dr. Ing. R. Pelissero
Speaker - H. G. Tjoa
Comment - Software verification and validation to what extent? When is it bug free? Is the responsibility by the maker of software or the customer?

Response - Test procedure is to be agreed upon and specified. When software complies with the test procedure it is considered to be validated/verified. The responsibility is carried by both software developer and "customer".

Paper Nr. 28 - TORNADO FLIGHT CONTROL SOFTWARE VALIDATION: METHODOLOGY AND TOOLS
Presented by - Dr. Ing. R. Pelissero
Speaker - C. D. Jack
Comment - On the engine part of the 1160 package, what sort of bandwidth do you aim for?

Response - I am not able to answer that at the moment, we use data we have from our engine department. We try to use recent data.

Paper Nr. 29 - APPLICATIONS OF NETWORK LOGIC MODELING AND ANALYSIS TO SYSTEM VALIDATION AND VERIFICATION
presented by - G. Sundberg
No Questions

Paper Nr. 30 - LANGAGE DE TEST DU LOGICIEL ET OUTILS ASSOCIEES (SOFTWARE TEST LANGUAGE AND RELATED TOOLS)
Presented by - Ing. P. Taillibert
Speaker - L/Cdr. J. F. Kramer
Comment - How much of the test data comes from software monitoring from within the tested machine and how much is captured by test probes external to the tested machine?

Response - All of the data is captured external to the running machine.

Paper Nr. 31 - SOFTWARE VERIFICATION OF A CIVIL AVIONIC AHR SYSTEM
Presented by - Dr. M. Kleinschmidt
Speaker - R. Malcolm
Comment - Did you say 2500 programmer hours for the whole project?
For the static code analysis, how many modules did you go through? Did you go through every single one? About how long did you spend on each?

Response - No, it was 2500 operational hours since the verification process.
Certainly, we went through every single module, about 130 of these. I would estimate we spent one to two days on each.

Paper Nr. 31 - SOFTWARE VERIFICATION OF A CIVIL AVIONIC AHR SYSTEM
Presented by - Dr. M. Kleinschmidt
Speaker - J. D. M. Grofnendijk
Comment - Is the approach stated coherent with the recommended RTCA DO 178 Guideline?

Response - The approach is based on and is coherent with RTCA DO-178 recommendations. However, for practiced implementation, these recommendations have been expanded in several parts:
- Application of a formal control mechanism for the design and verification phases as described in the present paper using manual and automated tools.
- Establishing of formal procedures and automated tools for the individual verification task to ensure completeness and uniformity of the tests done by individual members of the test groups.
- Application of even stricter rules in some verification tasks, e.g. in HW SW integration tests (code unchanged in PROMs, testing down to code level etc.) or module group tests (tests are not performed on host system, but in real time on the actual computer).

Paper Nr. 31 - SOFTWARE VERIFICATION OF A CIVIL AVIONIC AHR SYSTEM
Presented by - Dr. M. Kleinschmidt
Speaker - D. Weiss
Comment - Which kinds of tests are effective at finding which kinds of errors? i.e. Do you have distribution of error data showing types of errors according to the method used to find the errors?

Response - The answer to this question is in fact outlined in paragraph 4 of the paper. As we have established an efficient change control system, the complete history of the verification process for all individual tests and error types is available.
In this paper only the major error classes are plotted in Figure 5.

Paper Nr. 32 - PROGRESS IN VERIFICATION OF MICROPROGRAMS
Presented by - Dr. S. D. Crocker
Speaker - D. Weiss
Comment - We have found it to take 3-4 months of one person's time to write a PDP 11 45 machine description in ISP. This description was sufficient for compilations on an ISP compiler and subsequent interpretation by a program that simulates machine architecture.

Response - Although that is reasonably consistent with our data, I'd ask what is the level of detail and how accurate is your description. The PDP-11 is better documented and more widely known than the machines we usually deal with, but I wouldn't be surprised to find that it would take some additional time to track down every detail.

Paper Nr. 32 - PROGRESS IN VERIFICATION OF MICROPROGRAMS
Presented by - Dr. S. D. Crocker
Speaker - Dr. W. J. Cullyer
Comment - 1. In a state Delta, can the Environment, E, change between time t1 and t2, for example when encountering a declaration?
2. How often do you have to resort to the use of inductive proofs when steering your theorem prover?

Response - 1. Yes, the places listed in the environment list may change between time t1 and t2, but this situation is not usually related to declarations. Declarations would usually be translated as static relations among places and state deltas that contain only a post condition, e.g. a declaration that the range of X is 0 to 10 (inclusive) could be represented as approximately $0 \leq X \leq 10$.
2. On the average, one would use one induction proof command for each loop and for each recessively defined data structure, the user might choose to treat some loops and data structures differently. For example, the user can direct the proof system to step through a loop until it is finished; this will be useful once in a while.

Paper Nr. 33 - VALIDATION OF SOFTWARE FOR MISSILE TO AIRCRAFT INTEGRATION
Presented by - R. E. Westbrook
Speaker - N. Haigh
Comment - You mentioned that one function of testing is to ensure that software doesn't cause unexpected behavior under unusual circumstances. This can arise from numerical instability in the implementation of mathematical formulas. Would you comment on the stage, validation/verification, at which such numerical analysis can be effectively performed, and how it may be done?

Response - Numerical algorithms and the implementation of those algorithms in the avionics computer should be tested during the analysis phase and also during the module development phase. It is at these times that the basic accuracy and stability of algorithms is shown. Also,

at these times potential conditions leading to instability must be determined and reflected in the validation test plans and test procedures. This probably means that tests for occurrence of unusual conditions may be performed once the algorithm is integrated with the rest of the software. Correction of a problem, if found, will depend on the circumstances.

Paper Nr. 34 - IMPLEMENTING HIGH QUALITY SOFTWARE
Presented by - E. Dowling
No Questions

Paper Nr. 35 - LA QUALITE DES LOGICIELS AVIONIQUES - SPECIFICATION ET EVALUATION
Presented by - M. Delacroix
No Questions

Paper Nr. 36 - DISSIMILAR SOFTWARE IN HIGH INTEGRITY APPLICATIONS IN FLIGHT CONTROLS
Presented by - Dr. D. J. Martin
Speaker - K. A. Helps
Comment - Although you have described several precautions which you take to reduce common mode errors and failures, if such an error does occur does not this undermine the concept of using dissimilar redundancy to replace some normal validation procedures in systems where extremely high integrity is required? Surely the way to apply dissimilar redundancy is to apply it in addition to normal validation techniques or in systems where extremely high integrity is not required, since the risk of common mode errors or failures are not quantifiable with confidence.

Response - I think the essence of the question could be rephrased as follows for a similar redundant system:
Although several precautions are taken to reduce common mode errors and failures, if such an error does occur this undermines the concept of using similar redundancy. The risk of common mode errors remaining in an extensively tested suite of software cannot be quantified with confidence.

Paper Nr. 36 - DISSIMILAR SOFTWARE IN HIGH INTEGRITY APPLICATIONS IN FLIGHT CONTROLS
Presented by - Dr. D. J. Martin
Speaker - Dr. M. Kleinschmidt
Comment - Common mode errors cannot be excluded only by dissimilarity in some cases. How do you prevent the S/W from those errors to achieve the claimed error rate of 10^{-9}

Response - Common mode errors cannot be excluded by extensive testing of a single suite of software. They can only be reduced. At present we can't measure the reliability of the eventual software, we can only increase our confidence in the software by the procedures and techniques we apply. These are discussed and agreed with our customers and the certification authorities until an agreed approach has been decided upon. We believe, that with the dissimilar approach we are starting from a position of strength in that the architecture provides inherent integrity, whereas with one suite of software you know there are errors and the testing is necessary to reduce the probability of these common mode errors to less than 10^{-9}.

Paper Nr. 37 - THE COST OF SOFTWARE FAULT TOLERANCE
Presented by - G. E. Migneault

No Questions

THE MANAGEMENT OF A LARGE REAL-TIME MILITARY AVIONICS PROJECT

by P.J. Carrington, R.M. Gisbey and K.F.J. Manning (Marconi Avionics, Rochester)

1. Introduction

The AQS 901 is an airborne submarine detection system installed in the Royal Australian Air Force Orion and the RAF Nimrod Long-Range Maritime Patrol Aircraft. To counter the modern submarine threat, the development of sensor and processing systems to detect and locate the enemy submarine has a high priority. Expendable, sensitive underwater listening devices, called sonobuoys, pick up the faint but characteristic submarine sounds. These sonobuoy signals are transmitted on an RF link to the aicraft where real-time analysis is performed by the AQS 901 Sonics Processor to extract the wanted signal from the noise, to present the data to the operator in the most easily assimilated form, and to provide a wide range of user options for display manipulation and data combination.

The AQS 901 system consists of 22 units of special-purpose hardware and 150K of CORAL software. The project started in 1973, the first flight trials took place in 1977, and the system went into service in 1980. The software is now in maintainance and has thus been through all phases of the software lifecycle.

2. Project Background

In any major project, the circumstances - technical, financial and administrative - play a major part in shaping the project, and any conclusions drawn from the experience of the project must be interpreted in the light of these circumstances. I will therefore first briefly summarise the main factors that have influenced the way the AQS 901 project was tackled.

The customer wanted a major advance in his Anti Submarine Warfare (ASW) capability. To this end a number of sophisticated new sonobuoys were being developed and a new processor, the AQS 901, was required to process them. A range of advanced processing and display techniques were to be implemented in the AQS 901, particularly in the use of CRT displays. The development was funded on a "cost-plus" basis and administered directly by the UK Ministry of Defence. The principal system design criterion was to maximise performance, in order to effectively counter the submarine threat, with secondary criteria to minimise unit production cost and the volume of the processor to meet the constrained installation space on the aircraft. It is interesting to note the relative costs of hardware vs software and development vs production.

Table 1 Relative Costs of AQS 901 Hardware and Software

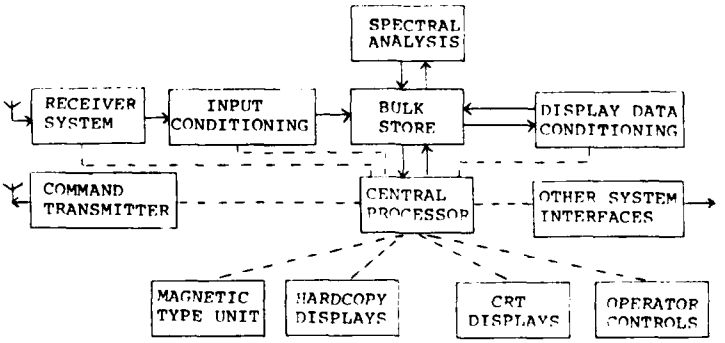
	HARDWARE	SOFTWARE	TOTAL
DEVELOPMENT	27	7	34
PRODUCTION	66	0	66
TOTAL	93	7	100%

Total Number of Production Systems is 107

Because of the impact on cost and weight of additional hardware and the relatively low cost of adding software, there was a natural tendency to solve problems by implementing solution in software rather than hardware. Also, because of the extent of the development and production tasks and the need for an early in-service date, development of hardware, processing algorithms and software progressed in parallel, rather than serially.

In addition to these general project circumstances, there were a number of specific factors which the software management had to contend with. The system, designed in 1974, has a centralised processor architecture with 4 interrupt levels and 16 channels of asynchronous I/O and DMA.

Figure 1



The wide range of tasks that the CPU has to carry out, from millisecond response times for dedicated hardware control, to hundreds of milliseconds for operator interaction and up to tens of seconds for complicated floating-point arithmetic algorithms, made the design of efficient software difficult, and compounded the software testing problems. Also, many of the dedicated hardware units had to operate at very high loadings to achieve the necessary system performance, significantly complicating the software scheduling task. On examination of the system specification and user needs, in particular in the operator interface area, the requirements evolved during project development into a much more extensive and complex software design than originally envisaged either by the contractor or the customer. In fact, due to these and other factors, the software program, initially estimated at 16K, grew to a final size of 150K words of CORAL!

The testing of the software and its integration with the hardware was also not without its challenges. The statistical nature of the data (signals in random noise) and the operator inputs (selection of options at random times) made tests time-consuming to carry out and pass/fail criteria difficult to define. The wide range of options made the testing of all processing combinations impossible, and the real-time nature of the system, with its high loadings, did not allow sophisticated debugging aids. In addition, the Software Team had also to contend with a new language - this was the first major avionics project written in CORAL, a new CORAL Compiler for the GEC 920 Computer, and a new host and operating system.

The majority of these factors became evident as the software development progressed and it should be said that the techniques employed to cope with them "grew up" with the project and were not part of any predefined approach to software development.

3. Software Statistics

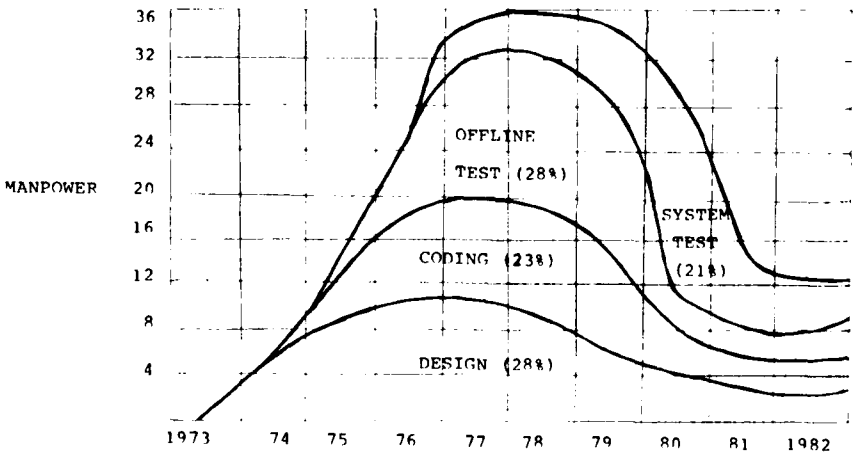
Before describing those ingredients of the software management which most contributed to the overall success of the AQS 901 project, it is worth reporting some of the software development statistics. Table 2 summarises the code production rate.

Table 2 AQS 901 Coral Code Production Rate

	PRODUCTION ONLY (DELIVERED TO CUSTOMER)	PROTOTYPE AND PRODUCTION
TOTAL CODE (K WORDS)	150	260
TOTAL MANPOWER (MAN YEARS)	220	220
CODE PRODUCTION RATE (WORDS PER HOUR)	0.4	0.7

The overall rate of 0.4 words per hour includes all software design, programmer and system testing effort and is comparable to that achieved on other real-time projects. The lifecycle diagram shows the usual split for this type of system into approximately 50% design and code, 50% test.

Figure 2 AQS 901 Software Effort



The loading of most of the special purpose hardware was very high, exceeding 90% for the FFT processor, and program, data and display stores.

4. Software Development : The Lessons Learned

4.1 Software Design

- Formalising the design at the right time is important.

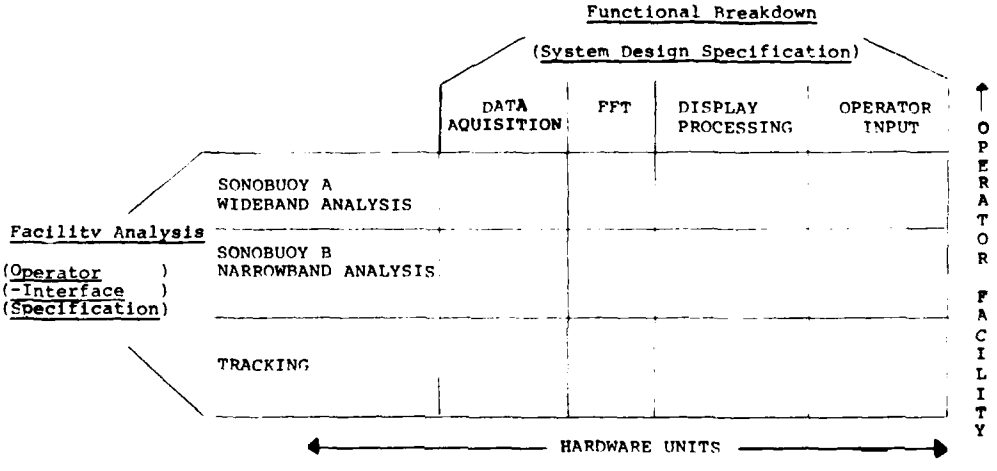
  In the early stages of system design, creativity in devising the best solution to the customer's needs is aided by flexibility and informality in system design documentation. However, at some point, the decision must be made to change to a formal design statement under change control procedures. If this decision is taken too late, additional requirements and changes can too easily be accepted without a proper evaluation of the impact on system loadings, software complexity, cost and timescales. Estimation and control of system loadings is essential. Simulation of software to establish processor loadings was found to be unproductive.
- Your system design specification is the bridge between the customer's requirements and the detailed design.

  The most valuable system design statement proved to be the System Design Specification (SDS) which is a high-level contractor-generated document stating the overall system, software and hardware design and responding to the customer's Requirement Specification. The SDS acted as a focus for all design groups within the project team, as well as bringing out differences in interpretation of the Requirement Specification between customer and contractor. It is amazing how often the same words can have different meanings for different people! The SDS must be produced as early as possible and updated regularly.

- The SDS is often hardware - or function - oriented : you also need a facility description document.

  In a combined hardware-software system, it is natural to base the architecture and modularisation of the software on the hardware units, e.g. an FFT segment, a CRT display segment. Whilst this approach is beneficial in making maximum use of common functions and in efficient scheduling, it does not lend itself to the analysis of user facilities and it can be difficult for the customer to be satisfied that the system does provide the facilities he needs operationally. The production of a facility-based document, such as an Operator-Interface Specification, at the same time as the System Design Specification, is essential if misunderstandings of the system's technical capability are to be avoided. Figure 3 demonstrates the two opposing methods of breaking down the system design.

Figure 3 System Design Documentation

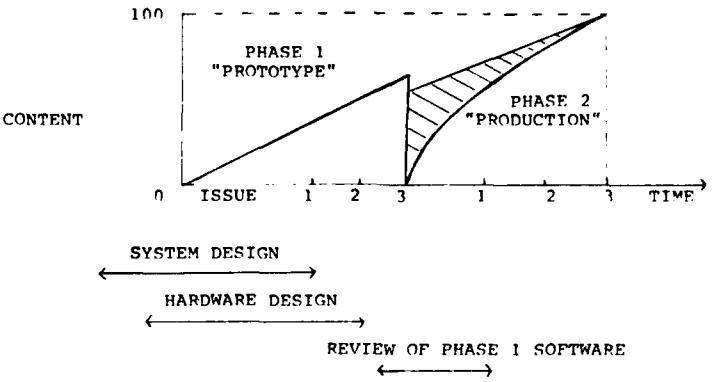


4.2 Software Production

- Two-phase software production finds the problems early but avoids compromising the software structure.

  The production of AQS 901 software evolved into a two-phase process, as in Figure 4, which proved most successful.

Figure 4



The first phase (or "prototype") software was written to the Requirement Specification and to be run on representative hardware. However, it did not cover all facilities in full depth. And it was recognised that, because software design and production started early, there might be a need in the second phase to redesign all or part of the software for higher efficiency or performance when greater familiarity with the hardware and processing methods was acquired. The advantages of this approach are that the early practical experience of the hardware and algorithms rapidly identifies the major software problems, whilst ensuring a high degree of carry-over from the prototype to the production software by aiming the prototype at the full specification and the target hardware.

- Multiple issues provide early performance feedback. Depress customer expectations of early issues.

  Within each phase, a number of issues, or software packages (say three), are provided for formal testing and flight trials. The customer's expectations of the early packages must be depressed - it is better to provide a few well-tested facilities than a lot of incorrect ones.

4.3 Software Testing

- The handover point between the Software and System Test teams, and the software capability, should be well defined.

  In the testing of the software and integration with the hardware, it is essential that the point of handover of software from the Software Team to the System Test Team is well-defined. Formal Test Design and Reception Test meetings provide the forum, in the AQS 901 project, for the originator of the software module to explain the design, processing techniques and degree of offline testing carried out, and for the System Tester to assure himself that the software module is adequately designed and tested for integration on the hardware. An intermediate test phase, where the Software Team check the basic "load and run" capabilities of a new package on the target hardware before handover, can save System Test Team effort.

- Early flight trials are valuable for feedback on performance and ergonomics of facilities.

  The AQS 901 system was subject to flight trials and Performance and Integration Testing from the earliest issues. Whilst these made the software development plans and schedules very tight and difficult to adhere to, in hindsight they were beneficial to the project in identifying operational and performance problems at an early stage, before release of software to the RAF; these undoubtedly would have been overlooked by less formal testing methods.

- Formal Integration Testing and Performance Demonstration prior to issue ensure quality.

  Before an issue is released to the RAF, eight days of formal Performance and Integration Tests, witnessed by customer representatives, are carried out.

4.4 Software Maintenance

- Changes in personalities, authorities and procedures at the handover to maintenance should be planned for.

  The maintenance phase is generally characterised by a change in personalities involved in the project. To avoid confusion and clashes, it is advisable to clarify and define authorities, procedures, lines of communications and the aims of the maintenance phase at as early a stage as possible.

- A project-wide "Shopping List" for enhancements and fault correction makes for easier control and planning.

  The maintenance phase has benefited from the drawing up of an Operational and Performance Problems List which is a single project-wide document containing all known software faults and required enhancements and which is used by the customer to set priorities and draw up software development plans. In fact, the majority of AQS 901 maintenance involves the addition of new facilities.

- Patching, suitably controlled, is a good method of rapid fault correction.

  On a practical aspect of maintenance, the use of patching (modification of the program after compilation and linking) has been found to be a better way of achieving a fast response to software problems, particularly during trials, than recompilation.

4.5 Software Management

- Small Programming Teams (4-6), with a high level of design authority, work well.

  The Software Team who carried out the design, production and testing of the AQS 901 Software numbered at most thirty at the peak of the development. The team structure is along the lines of the "Chief Programmer Team" approach with units of 3-5 programmers led by an experienced Software Designer/ Programmer. Each team has a great deal of design authority for the module or part of the program for which it is responsible.

- Software Team members generally - have a computer science degree
  - start with little previous experience
  - are trained "On-the-Job"

  Virtually all programmers are University graduates in Computer Science or a scientific subject. Most come to the Software Team with little or no experience of real-time or CORAL programming, and expertise is built up by "on-the-job" training.

- The motivation of teams and individuals is essential.

  One of the key factors in the success of the project has been the degree of motivation and commitment to the project of all levels of staff. Recruitment is directed towards candidates demonstrating initiative and enthusiasm, and this has paid off in the development of a Software Team capable of, and motivated to, tackling the difficult technical problems the AQS 901 project has imposed, within the tight timescales required by the customer.

- A good and equal relationship between contractor, customer's financial agency and customer's technical agency facilitates the resolution of conflicts.

The mutual understanding developed between the contractor, customer's financial and technical authorities proved valuable in the resolution of conflicts of timescales, cost and performance, and in the assessment of alternative courses of action and the risks involved.

- Pert planning by a committed and experienced planner gives good visibility of progress.

  PERT planning has been used successfully. The most useful plans have 250-500 activities, with a detailed breakdown to 2-3 week tasks over the ensuing 12 months, and updated every 4-6 weeks.

5. Conclusion

The AQS 901 project has been a success. The system is now well-proven in operational service and well-regarded by its users. The capability and performance the AQS 901 provides is significantly greater than that originally envisaged by either customer or contractor. This is largely due to the ability to extend system performance through software. The complexity of the software development task increases dramatically with parallel software, system and hardware design, but, by careful and practical management of dedicated teams, the task is achievable.

6. Future Developments

The AQS 901 project started in the early seventies : since then hardware has become significantly smaller and more capable, and software development tools and techniques are starting to appear. I will finish by noting some of the areas which our experience on the AQS 901 project suggests are the most important for the next generation of avionic mission systems.

- Multiprocessor systems make software more modular.

  Distributed processing, made possible by microprocessor and dynamic store developments, is the key to software modularisation, with each processor capability matched to the task it has to perform.

- Smaller and faster processors and store can relieve the need for high loadings - but will this opportunity be taken?

  With the reductions in hardware size and cost, the need for high loadings on processors and store dimish, making more structured and standardised, though less efficient, software designs possible. However, as these advances give the opportunity for the contractor to offer, and the customer to demand, more capability in smaller aircraft, the problems of software may not reduce as rapidly as might be envisaged.

- Better and standardised software development tools will improve software quality and production rates.

  Better software development tools allow the software engineer to spend more time on design and testing and less on paperwork. Standard tools mean that he does not have to learn to cope with new tools as the same time as meeting the demands of a new project. On the other hand, these tools must not impose excessive penalties or constraints on the target system, in execution time or code volume, as these may well be unacceptable in a weight-constrained avionics system.

- Design statement and validation techniques are essential for the future complexity of systems.

The automation of the design statement and validation processes is essential if the scale of system and software complexity now being envisaged is to be achieved. There is, however, a long way to go in developing tools with the necessary flexibility, and it is unlikely that there is a single answer applicable to all types of software development project.

- Testing of real-time systems may be aided by automated statistical sampling methods.

  There can be no real-time system, of any significant size, for which every possible route through the program and every combination of parameters can be exercised and checked. Exhaustive testing is impossible in practice. Statistical sampling techniques are applicable to software and may be automated. These may provide the key to the testability/quality problem, particularly for very complex but not flight-critical software-hardware systems.

F/A-18 SOFTWARE DEVELOPMENT - A CASE STUDY

T. V. McTigue
Branch Chief
McDonnell Aircraft Company
McDonnell Douglas Corporation
Post Office Box 516
St. Louis, Missouri U.S.A. 63166

ABSTRACT

This paper presents a description of the successful Avionics software development for the U.S. Navy/McDonnell Douglas F/A-18 Hornet Fighter/Attack Weapon System. The Avionics Computer Subsystem consists of two central Mission Computers and a number of distributed processors embedded in various sensor and display subsystems. This distributed processing system is interconnected by and communicates over a MIL STANDARD 1553A serial 1 MHz command/response multiplex network. The avionics software architecture is discussed and the rationale is presented for the partitioning of the software tasks between the central Mission Computers and the distributed processors embedded in the sensor subsystems. The salient features of the software of the Mission Computers and the distributed processors are also discussed. Finally, the design of the Operational Flight Program (OFP) for the central Mission Computers is described, including a discussion of the development process and support facilities which were used for the software integration and validation.

1. INTRODUCTION

The purpose of the F/A-18 Hornet Weapon System is to deliver air-to-air and air-to-ground weapons on targets that must be detected, identified, tracked, and destroyed by the pilot using sophisticated sensors and weapons. In the course of an F/A-18 Hornet mission, millions of split-second computations and decisions must be made within the aircraft. The pilot, in addition to flying the aircraft, must constantly monitor the instruments and interpret the readings to ensure that the weapon system can accomplish its purpose. One-man operability was a prime goal in the design of the F/A-18 Hornet. Every decision and task that could be safely removed from the pilot was incorporated in a highly integrated computational subsystem. The operations within the subsystem are still at the pilot's command, but he is able to perform his primary tasks with confidence based on reliable, real-time operation of his computational subsystem. This subsystem consists of two mission computers and a number of distributed computers in various sensor and display equipments. The Operational Flight Program (OFP) for the Mission Computer was developed by McDonnell Douglas Corporation, St. Louis, Missouri, and was flight-tested and qualified by McDonnell Douglas and Navy pilots at the Naval Air Test Center, Patuxent River, Maryland. The first U.S. Navy squadron was activated at the Naval Air Station in Lemoore, California, in February 1981, and the first production aircraft was delivered in September 1981. The F/A-18 has also been selected by Canada and Australia and is under serious consideration by a number of other U.S. Allies.

2. THE F/A-18 SOFTWARE DEVELOPMENT

A popular misconception is that the software development effort begins when it is time to do the coding. On the F/A-18 the software effort began with the system design and mathematical analysis of the system equations. The software engineers participated in the definition of the system design as well as in the software implementation of that design. They had to consider not only the definition of the system design but also the impact of that design on the limited airborne computer resources of memory and execution time. In addition, the software engineers "rendered" the equations to a form suitable to the architecture of the airborne computing system without changing their performance or accuracy.

The software design methodology used on the F/A-18 included the following:

- o Independent Organizational Structure
- o Rigorous Software Control
- o Sensor/Mission-Oriented Software Partitioning
- o Functional Software Module Partitioning
- o Top-Down Design
- o Structured Software Design
- o Proven Design Practices
- o Thorough Software Testing

Each of these will be discussed in subsequent paragraphs.

2.1 Independent Organizational Structure

The structure and partitioning of the software organization was considered as important as the structure and partitioning of the software itself. Figure (1) shows the organization of the Software Engineering Group. This group was part of a Project Organization tasked with the day-to-day activities of design, development, test, delivery, and support of the product. The Project was staffed with personnel from various Functional organizations. Thus each individual had two independent reporting lines, Project and Functional. In addition, each Functional organization provided independent technical reviews and recommendations to the Project. The personnel assigned to the Project were moved from their Functional work area and relocated together in a Project work area. The Project/Functional organization partitioning provided built-in "checks and balances" at all levels of the organization.

The Software Engineering Group consisted of four Project subgroups and one non-project group that reported directly to a Functional organization. This structure provided organizational independence even within the Software Engineering Group.

The Software Documentation and Control Group was responsible for the management and control of the software documentation as well as for the configuration control of the software itself. This group was responsible for establishing and maintaining the development library, maintaining backup copies of the evolving program, compiling frozen module versions, generating updated versions of the OFP, and assigning and maintaining records of module and program identification unique for each version. Records maintained by this group provided the specific configuration identification of each version of the program compiled for development testing.

The Support Software Group was responsible for the design, development, test, and documentation of those programs used to support the MC OFP development. This support software included programs such as the compiler/assembler, database catalog, avionics simulation models, and automatic flowcharting programs.

Software design and development was the responsibility of the OFP Development Group. Even within this design group there were built-in "checks and balances". A programming team was assigned to each functional module. This team consisted of a Module Engineer and a Module Programmer. The Module Engineer was tasked with the software design whereas the Module Programmer was tasked with the module coding. Together they were responsible for all aspects of the software design, development, test and documentation of their module from software inception to installation. The Module Engineer and the Module Programmer were physically located next to each other to shorten the communications paths as much as possible. This arrangement was based on the premise that the number of errors in a program is directly proportional to the distance between the software designer and the software programmer.

A fourth group called the OFP Test and Integration Group provided independent verification and validation of the complete OFP. This provided organizational separation between the software design teams and the software test team. As an independent group, they were responsible for the final verification and validation of the OFP. It was their responsibility to prepare the software test plan, define the required test facilities, and prepare and run the test procedures for integration, test, and acceptance of the software.

One final "check and balance" was provided by the independence of the Software Design Integrity Group from the Software Engineering Group on the Project. This independence occurred due to the fact that the Software Design Integrity Group reported to the Functional engineering organization and not to the Project organization. The Software Design Integrity Group was responsible for the preparation of the Avionics Software Guide and for audits of the Project software groups' adherence to these software standards.

2.2 Rigorous Software Control

On the F/A-18 software control encompassed control of all aspects of the software effort, namely:

- Control of the software requirements
- Control of the software design
- Control of the software testing
- Control of the software configuration.

Just as the software development on the F/A-18 did not begin with the coding, neither did the software control. Control of the software began before there was any software. It began with control of the requirements imposed on the software by the system designers. From years of training, all engineers possess inordinate amounts of "technical greed", the desire to do a perfect job, to wring the last bit of performance out of a design whether or not it is required or cost effective. Control of software requirements was effected by performing trade studies to show whether a software requirement was cost effective, whether there was an alternate approach, what impact each approach had on the on-board computer resources, and whether the task needed to be done at all.

The next segment of software control involved control of the software design. Design control really means control of the designers of the software. Management techniques were employed which clearly established areas of responsibility, defined programming standards, established programming traceability, and required programming audits. The programming standards required the use of simple and straightforward coding which was easy to understand and maintain and which had the appearance that the entire program was coded by one programmer. Detailed programming "good design" practices were established and programmer compliance audited by engineering personnel outside the design team.

The next phase of software control addressed software testing. This is often the most neglected and least controlled segment of the overall software effort. On the F/A-18 this was not the case. Software testing began on the module level where every path in each module was tested. To confirm that every path was exercised, a special program called PATHFIND was used that checked that all paths in the airborne code had been exercised. After module testing was complete, the next phase was complete program testing. This was performed on the airborne computer operating in an input signal environment that simulated actual flight conditions to the airborne computer. In each phase of testing, a test procedure change log was maintained so that as changes were incorporated into the airborne program corresponding changes were made to the test procedures. Testing deserves and needs as much control as the other segments of the software effort since it comprises about 50% of the overall software development effort.

Finally the F/A-18 software configuration was controlled. This is probably the best known aspect of software control although historically not necessarily the best implemented. On the F/A-18 software configuration control did not wait until the software was delivered. It began with control of the design by means of control of the flow charts and eventually the code itself. No changes were permitted to a flow chart or the code without a written and approved Computer Program Change Request (CPCR). Once the computer program was released for flight test it was assigned a part number just like hardware. Installation in the aircraft and subsequent change of the program was controlled by changes to the part number (see Figure (2)).

2.3 Sensor/Mission-Oriented Software Partitioning

The F/A-18 airborne computational requirements were classified into two major categories (Figure (3)):

- o Sensor-oriented computations
- o Mission-oriented computations

Sensor-oriented computations were defined to be those independent computations, such as sensor coordinate transformations, platform management, and signal processing, which were peculiar to a particular sensor or display. Mission-oriented computations, such as weapons launch calculations, were defined to be those computations directly related to performing the mission and dependent on the integration of information from several avionics subsystems. Table I shows typical examples of the two categories of computations.

TABLE I - COMPUTATIONAL CATEGORIES

Sensor-Oriented	Mission-Oriented
o Air Data Calculations	o Air-to-Air steering and launch zones for gun and missiles
o Radar Signal Processing	o Air-to-Ground steering and release for bombs, rockets, gun, and missiles
o Inertial Platform Management	o Selection of best available data from various sensors
o Display Symbol Generation	o Integrated display management

The mission-oriented tasks were allocated to two central Mission Computers (MC) and the sensor-oriented tasks were assigned to embedded processors in each of the sensor subsystems. (See Figure (4)). This relieved the central computers of those tasks which could be more effectively performed and managed in distributed and independent sensor processors. This approach offered functional modularity of the sensors, whereas system integration was provided by the Mission Computers. Hence, improved sensors and displays can be added later to the Avionics System, and present ones can be changed, with minimum impact on other equipment. Likewise, if the armament is altered for new or modified weapons as the mission of the aircraft is enlarged, such changes can be accommodated primarily through changes to the Mission Computer and Stores Management Set software.

2.3.1 Sensor-Oriented Processing

On-board the F/A-18 Hornet there are four major subsystem-embedded reprogrammable computers and a number of smaller subsystems with embedded microprocessors with Read-Only Memories (ROM). Table II summarizes the computer hardware for the major subsystems with reprogrammable computers. Table III presents the computer hardware information for the subsystems with ROM computers.

TABLE II - SENSOR-ORIENTED REPROGRAMMABLE COMPUTERS

COMPUTER	CPU	SPEED	MEMORY
Inertial Nav Computer	2901	238 KOPS	16K Core
Radar Data Processor	2901	700 KOPS	250K Disk/16K RAM
Radar Signal Processor	54S181	7100 KOPS	250K Disk/48K RAM
Stores Management Processor	8080	200 KOPS	32K Core

TABLE III - SENSOR-ORIENTED ROM COMPUTERS

SUBSYSTEM	CPU	MEMORY
Air Data Computer	2901	5K ROM
Comm System Controller	8080	16K ROM
Flight Control Computer (4)	MCP-701A	44K ROM
Forward-Looking Infrared	9900	32K ROM
Laser Spot Tracker	2901	12K ROM
Maintenance Monitor Panel	8080	1K ROM
Maintenance Signal Data Recorder Set	8080	14K ROM
Multipurpose Display (2)	2901	5K ROM

Each sensor computer performs only those computations necessary to perform its well-defined task. This includes all computations required to translate some measured physical parameter, such as air pressure, into useful information for the pilot, such as altitude, airspeed, and Mach number. Once the information is computed, it is sent to the Mission Computer over the Avionics Multiplex (MUX) bus. There it is used with information from other sensors to perform the mission-oriented computations as well as for display to the pilot. Figure (5) lists the major sensor computers and their allocated software computational tasks.

2.3.2 Mission-Oriented Processing

The Mission Computer Subsystem consists of two identical computers built by Control Data Corporation (CDC). They are the new U.S. Navy Standard Airborne Computers designated the AN/AYK-14. The rationale for two Mission Computers was the same as for two engines. When they both are operational, they provide increased weapon system performance. When one is not operational, the other provides enough performance for self-defense and safe return. Although the hardware of the two computers is identical, their computer programs are different and are dedicated to specific processing tasks. The AN/AYK-14 is a high-speed, general purpose digital computer specifically designed to meet the real-time requirements of airborne weapon systems. The computer uses four AMD 2901 four-bit slice Large Scale Integrated (LSI) circuits to implement the 16-bit Central Processing Unit (CPU). The CPU is microprogrammed by means of ROM firmware to emulate the instruction set of the U.S. Navy Standard Shipboard Computer designated the AN/UYK-20. By emulating the AN/UYK-20 instruction set, the AN/AYK-14 can use the same CMS-2M Higher Order Langauge (HOL) support software originally designed for the AN/UYK-20. The AN/AYK-14 consists of ten plug-in modules and a single plug-in modular power supply contained in one Weapon Replaceable Assembly (WRA) weighing about 42 pounds and occupying 0.625 cubic feet. Each computer contains 65,536 (16-bit) words of 7/13 mil (inside/outside diameter) core memory for a total of more than a million individual cores per computer. The memory in each Mission Computer can be doubled from 64K to 128K within the present equipment envelope simply by replacing the two present 32K memory modules with two recently-developed 64K modules.

Each of the two Mission Computers is dedicated to specific processing tasks by means of its stored program. One computer is assigned the Navigation (NAV) And Support processing tasks and associated display management. The other computer is assigned the Air-to-Air and Air-to-Ground Weapon Delivery processing tasks and associated display management. The stored program in each computer has a small backup software module for selected functions of the other computer. These backup modules are executed only in the event the primary computer for these functions should fail. The funtional software modules in each computer are shown in Figure (6).

2.4 Functional Software Module Partitioning

A modular partitioning of the software was used which partitioned each computer program into software modules of manageable size based on a functional grouping of computational tasks. The rationale for modular software was analogous to that for modular hardware. First, it permitted each module to be independently developed, debugged, and tested in parallel with the other modules. Second, it allowed changes to occur within a module without causing changes to be made in other modules, as long as the external modular interface remained the same. Analogous to the modularity and controlled interfaces in the hardware, new programming modules could be added and old ones deleted without impacting the whole program as long as the module interfacing rules were followed. Documentation and undertanding of the total computer program was simplified, since each module could be described and learned as a separate entity.

2.4.1 Executive Module

The executive program module imposes order and structure on the entire F/A-18 operational flight program. All functional program modules are processed under executive control, which sequences them in an appropriate flow and calls them at a rate consistent with their requirements.

Six major tasks are performed by the executive module. First, it initializes the MC after start-up or after restart from a power interruption. Second, it schedules the order and rate of execution of each functional module. Third, it schedules the order and rate of input/output operations for the OFP. Fourth, it controls the servicing of all interrupts, external and internal. Fifth, it manages intercomputer communication between the Navigation MC and the Weapon Delivery MC. Sixth, it uses the scheduling and input/output management functions to ensure proper sequencing of the other modules.

2.4.2 Air-to-Air Module

The air-to-air module performs the following functions:

1) initializes the radar air-to-air search pattern based on the weapon selected
2) computes aiming reticle for director or disturbed gun mode
3) computes aiming reticle for director or manual rocket mode
4) computes maximum and minimum launch ranges and steering cues for missiles
5) computes other aircraft and target parameters for display.

2.4.3 Air-to-Ground Module

The air-to-ground module performs the following functions:

1) performs visual and sensor-aided designations of ground targets
2) automatically positions sensors
3) calculates ballistic release times
4) calculates steering cues for weapon release and reattack
5) calculates launch envelope data for air-to-ground missiles
6) issues release pulses for correct weapon delivery and weapons intervals
7) manages strike camera (SCAM) for damage assessment.

2.4.4 Navigation Module

The navigation module performs the following functions:

1) selects/calculates best available aircraft attitude, position, and rate data
2) calculates steering to prestored waypoints
3) performs velocity and position updates
4) performs target marking
5) calculates range, bearing, heading, and steering error to selected waypoint and TACAN station.

2.4.5 Data Link Module

The data link module decodes and processes messages received from a shipboard, airborne, or ground-based terminal. The messages contain information used in the following functions:

1) waypoint insertion
2) display of data for vectoring to airborne targets and rendezvous points
3) display of automatic carrier landing data
4) processing of couple requests to the flight control computers
5) processing of test messages
6) processing of radar target data and aircraft data to be transmitted in the reply messages.

2.4.6 Tactical Controls and Displays Module

The tactical controls and displays module manages the following functions:

1) Radar Control Panel/Display
2) Forward-Looking Infrared (FLIR) Control Panel/Display
3) Laser Spot Tracker (LST)/Strike Camera Control Panel/Display
4) Air-to-Ground Guided Weapons Control Panels/Display
5) Stores Management Control Panel/Display

2.4.7 Support Controls and Displays Module

The support controls and displays module manages the following functions:

1) Cautions/Advisories Display
2) Built-In Test (BIT) Display
3) Test Pattern Display
4) Engine Display
5) Checklist Display

2.4.8 Navigation Controls and Displays Module

The navigation controls and displays module manages the following functions:

1) Horizontal Situation Display Control Panel/Display
2) Attitude Director Indicator Display
3) Data Link Display
4) Up-Front Control Panel Data Entry/Readout
5) Moving Map Film Strip

2.4.9 Head-Up Module

The head-up display (HUD) module manages the HUD Graphics program. Symbology controlled by the HUD module includes aircraft flight data, data link cues, navigation cues, radar status, armament status, air-to-air weapon delivery cues, and air-to-ground weapon delivery cues.

2.4.10 Inflight Engine Condition Monitor Module

The inflight engine condition monitor module monitors various engine and associated aircraft paters to provide engine health information to the pilot and maintenance personnel. Cautions, advisories, and real time engine parameters are displayed in the cockpit. Life usage indices and other engine maintenance information are transmitted to the Maintenance Signal Data Recorder (MSDR).

2.4.11 Inflight Monitoring and Recording Module

The inflight monitoring and recording module monitors and processes various aircraft sensor outputs for control and display of pilot cautions and advisories and transmits avionic and non-avionic equipment failures to the MSDR Control is provided for the data recorder and provision is made for the recording of tactical data in air-to-air and air-to-ground modes. Also, aircraft fatigue levels are monitored and recorded during flight.

2.4.12 Avionics Built-Test Module

The avionics built-in test module provides the control by which an operator can run total system or individual tests on each of the interfacing subsystems. It also evaluates data received by the MC from each of the interfacing subsystems as to their operational status. This data is correlated by subsystem and current status and is displayed in the cockpit. In addition, the data is converted into predefined codes each representative of a specific failure of an individual subsystem for transmission to the MSDR.

2.4.13 Mission Computer Self-Test Module

The mission computer self-test module performs the following functions:

1) immediately after computer turn-on, tests those functions which, when tested, interfere with normal computer operation

2) periodically tests selected functions of the computer which, when tested, do not interfere with normal computer operation as well as performing an end-to-end check of the capability of the MC to communicate with each peripheral

3) maintains error information for later maintenance action; and

4) latches WRA fault indicator and sets WRA status signal as required.

2.4.14 Mission Computer Backup Modules

A backup module is resident in each computer. Each backup module performs essential software functions of the other mission computer when a failure occurs in that computer.

2.4.15 Mathematical Subroutines Module

The mathematical subroutines module supports other program modules by providing common mathematical routines such as trigonometric, logarithmic, and matrix operations.

2.5 Top-Down Design

A Top-down design approach was used for each module of the MC software. The resulting design was a hierarchy where each top-level program called several second-level subprograms, each of these calling a number of third-level subprograms, and so on (usually to about 4 or 5 sublevels) until a subprogram was reached that did not call any other subprograms. This top-down design approach resulted in many small subprograms (subroutines), each one being limited to a specific processing task and being able to be represented on one flowchart. The top-down design approach for a typical module is shown in Figure (7).

2.6 Structured Software Design

Recent advances in Structured Program design were instituted on the F/A-18 and tailored to the airborne software. Work is in progress on development of ADA, a Pascal-like structured programming language for airborne software. Although such a language was not yet available for the F/A-18, many of the benefits of a structured programming language were obtained by employing a structured flowchart design technique. Programs coded from these structured flowcharts were considerably easier to understand, test, and subsequently modify as changes were incorporated. Only five structured programming constructs were permitted to be used:

- SEQUENCE
- IF . . . THEN . . . ELSE
- DO WHILE
- REPEAT UNTIL
- CASE

2.7 Proven Design Practices

The proven design practices used on the F/A-18 are shown in Figures (8) through (11). The four major development phases were:

- o System Design and Verification
- o Software Coding and Integration
- o Hardware/Software Integration
- o System Integration and Test

The development process began with the design of the complete Avionics system, which was defined by a series of mode logic diagrams, equations, interface control documents, and display diagrams. From these the program flow diagrams were prepared and a software walk-through was held with a peer group of system and software engineers. The design was updated to include any changes resulting from the walk-through. After this, a software "Breadboard" was implemented in a higher-order language for system validation by pilots in a cockpit with a realistic operational environment simulation.

Once the system design was approved by the pilots, the software coding and integration began. The software was developed in a building-block approach. First, individual software modules were coded and tested. When this was completed, the build-up of the Operational Flight Program (OFP) was begun one module at a time until a complete OFP was assembled.

The third phase then integrated the software with the airborne hardware in which it was to run. The computer and its OFP were treated as a single entity with all inputs simulated and all outputs displayed on cockpit equipment.

In the final phase, the Avionics system was built up, just like the software, one element at a time until the complete avionics system was integrated. Finally, the Avionics system was installed in an aircraft and flight tested.

2.8 Thorough Software Testing

The F/A-18 software methodology was based on testing the flight program "before", "during", and "after" the actual coding of the program. Initially, the systems analysis and design approach was validated using an IBM 370 computer facility to produce a FORTRAN "breadboard" version of the proposed OFP. Subsequently, that design, and its implementation in the language of the flight computer, were validated for both man/machine and actual hardware compatibility. Figure (12) illustrates the steps involved.

- o Step 1 consisted of creating FORTRAN models of selected equations, algorithms and mode control. These models provided the analytical validation of the equations and algorithms to be used in the OFP.
- o In Step 2, a FORTRAN model of the baseline design was used at the flight simulation laboratory to evaluate the important control and display interface with the pilot and to test the mechanization proposed for the weapon system. This step provided vital confirmation of design adequacy at an early stage, and allowed alternate approaches to be examined.
- o Step 3 used a bit-by-bit functional simulator (emulator) of the MC hardware to verify coding, test software interfaces, and evaluate timing relationships in advance of first computer delivery. First, individual modules were tested. When this was complete, the build-up of the OFP was begun one module at a time until the complete OFP was achieved.
- o Step 4 took place after arrival of the first Mission Computer. At this point, hardware/software inconsistencies were isolated and corrected, leading to preliminary confirmation of correct OFP software integration with the MC hardware.
- o As other equipment arrived, Step 5 was taken to test the mission computer and its software with each actual interfacing equipment. This step provided integration of the OFP/MC with the individual equipment followed by integration with groups of related equipment.
- o Step 6 then reintroduced the man-in-the-loop to verify the total man/machine system. This used the MCAIR flight simulation laboratory with the OFP running in the actual mission computers, in addition to flight hardware for controls and displays, to checkout the system about to be flown.
- o Step 7 was the final step in software development. Prior software testing had assured that ultimate flight testing would proceed unhampered by software problems, thus permitting efficient use of the flight tests to validate the system implementation and to improve and refine the avionics system.

3. F/A-18 SOFTWARE DEVELOPMENT FACILITIES

The F/A-18 Hornet integrated software development process, discussed above, made use of three separate facilities:

- o Software Development Facility
- o Software Test Facility
- o Cockpit Simulator Facility

3.1 Software Development Facility (SDF)

The Software Development Facility is a modest-size data processing facility. It uses an IBM System/370 commercial computer system and standard peripheral equipment, operating system, and language processors. This facility is used for all FORTRAN processing, database processing, and compilation/assembly of airborne MC programs.

Figure (13) is a photograph of the Software Development Facility showing the IBM S/370 mainframe and associated peripherals. The facility includes the following equipment:

- o (1) IBM 370/138 Computer (512K Memory)
- o (4) 100 megabyte disk drives
- o (2) magnetic tapes drives
- o (1) printer
- o (1) card reader
- o (5) CRT/Keyboard terminals

3.2 Software Test Facility (STF)

The Mission Computer Software Test Facility is a minicomputer-controlled, real-time simulation and test facility used to test the airborne Operational Flight Program (OFP) in the MC and to integrate the MC and its OFP with the other avionics with which they interface. The STF accomplishes this by simulating the inputs to the MC and sending them out over the Avionics MUX in response to the MC requests for data from various aircraft sensors. The MC processes these inputs as though it were flying in an aircraft and then issues output data to the simulated sensors and to the cockpit displays. In general, the input sensors are all modeled in software in the minicomputer whereas the CRT's used to display the MC outputs are the actual displays used in the cockpit. This provides a realistic input signal environment for the MC and a realistic display of MC outputs for test and evaluation by the engineers and programmers. Figure (14) is a photograph of the STF.

3.3 Cockpit Simulator Facility

The Cockpit Simulator Facility (Figure (15)) is a laboratory complex oriented primarily to manned, real-time flight simulation. It includes a CDC Cyber 175 computer, four crew stations, terrain maps, horizon and target displays and associated hardware. Each crew station includes complete flight controls and instruments and is located in a forty-foot fiberglass dome. Target and terrain imagery is projected on the dome and presented in the cockpit on software-driven displays or actual flight display equipment. Both visual and sensor (electro-optical, infrared, radar) imagery is supported. The facility is used for weapon system design, pilot training, tactics development, and effectiveness assessment.

4. SUMMARY

The F/A-18 computational subsystem is a distributed computer system interconnected by a MIL STANDARD 1553A multiplex system. The software is partitioned into Mission-oriented computations performed in two central Mission Computers and Sensor-oriented computations performed in distributed processors in the sensor and display equipment. The factors that contributed to its success are summarized in Figure (16). Many of these same factors make the software easily adaptable to changes and expansions in the F/A-18 mission requirements and ready to share a long and successful future with the F/A-18 aircraft.

References

1. Griffith, V.V., Keifer, L.F., Paxhia, E.C., et al., "Aircraft Avionics Trade-off Study (AATOS)," McDonnell Aircraft Co., St. Louis, Mo., ASD/XR 73-20 Final Report, Nov 1973.

2. Finke, H.G. and Rosenkoetter, E., "Aircraft Avionics from the Aircraft Manufacturer's Point of View," McDonnell Aircraft Co., St. Louis, Mo., MCAIR 73-023, Sept. 1973.

3. McTigue, T.V., "F-15 Computational Subsystem", AIAA Journal of Aircraft, Vol. 13, No. 12, Dec. 1976, pp. 945-947.

4. McTigue, T.V., "F/A-18 Tactical Airborne Computational Subsystem", NATO AGARD Avionics Panel Symposium on Tactical Airborne Distributed Computing and Networks, Roros, Norway 22-26 June 1981.

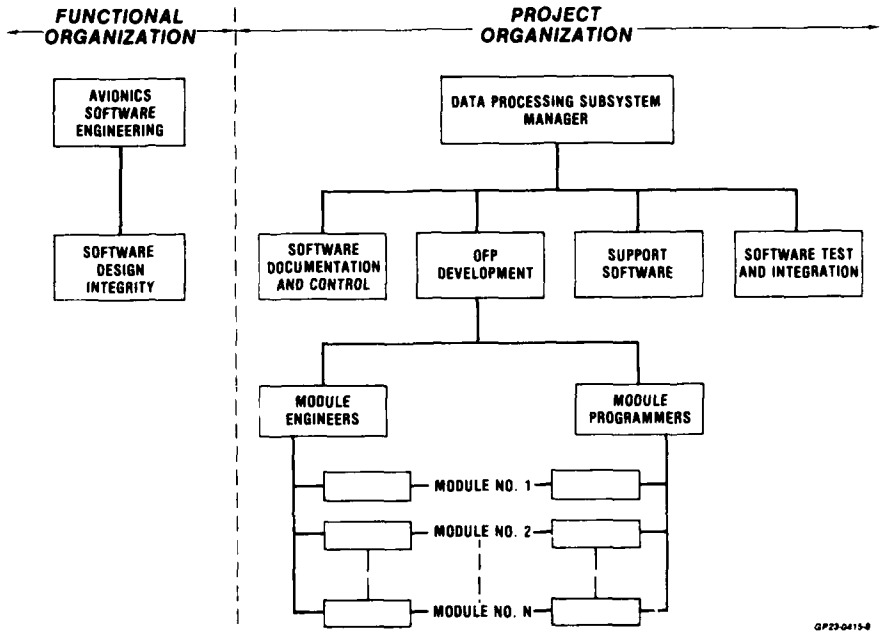


FIGURE 1
SOFTWARE ENGINEERING ORGANIZATION

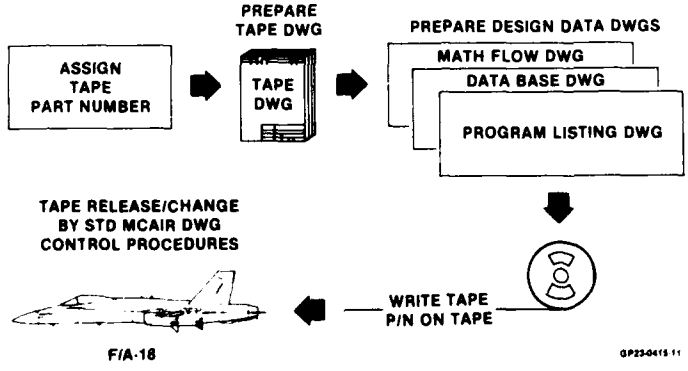


FIGURE 2
SOFTWARE CONFIGURATION CONTROL

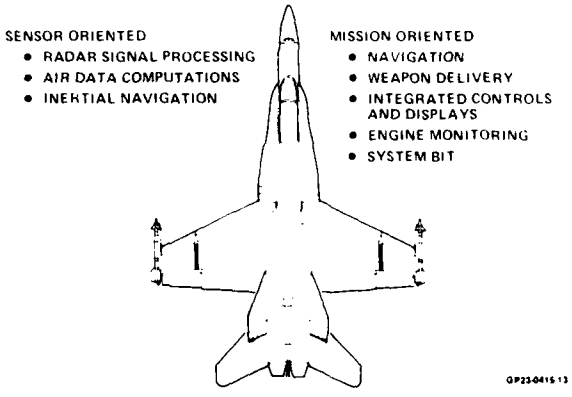


FIGURE 3
F/A-18A COMPUTATION REQUIREMENTS

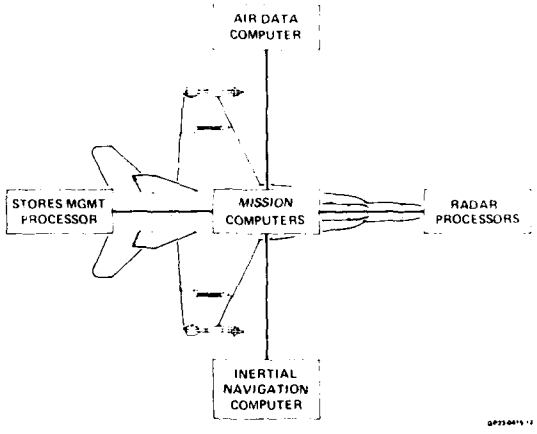


FIGURE 4
F/A-18A MISSION vs SENSOR PARTITIONING

AIR DATA COMPUTER	INS COMPUTER	STORES MANAGEMENT PROCESSOR	RADAR SIGNAL PROCESSOR	RADAR DATA PROCESSOR
• PRESSURES • AOA • SIDESLIP • ALTITUDE • AIRSPEED • MACH • TEMP • AIR DENSITY	• ALIGN/GB • ACCELERATIONS • VELOCITIES • PRESENT POSITION • ATTITUDE	• SPARROW INTERFACE • SIDEWINDER INTERFACE • GUN INTERFACE • BOMBS INTERFACE • HARM INTERFACE • WALLEYE INTERFACE • MAVERICK INTERFACE • RACK/VIDEO CONTROL • JETTISON • WEAPON INVENTORY	• RCVR GAINS • SIGNAL THRESHOLDS • RANGE GATING • PULSE COMPRESSION • AMPLITUDE WEIGHTING • RANGE RESOLUTION • TARGET DETECTION • TRACK S/N	• TARGET POSITION • TARGET VELOCITY • TARGET ACCELERATION • TARGET RANGE • VELOCITY ERRORS • DISPLAY DATA

GP23-0415-14

FIGURE 5
SENSOR ORIENTED SOFTWARE FUNCTIONS

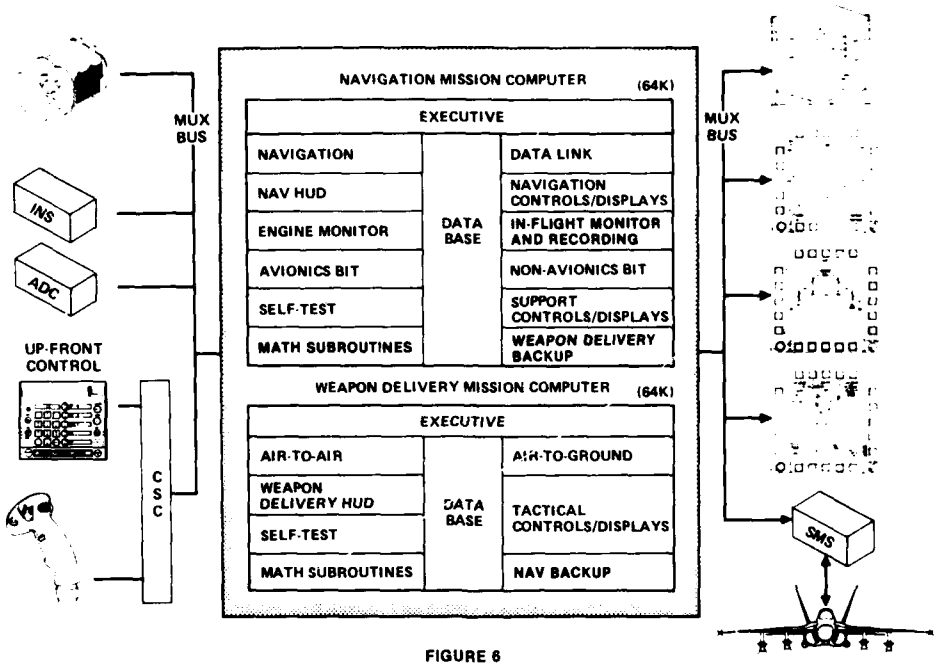


FIGURE 6
MISSION COMPUTER FUNCTIONAL SOFTWARE MODULES

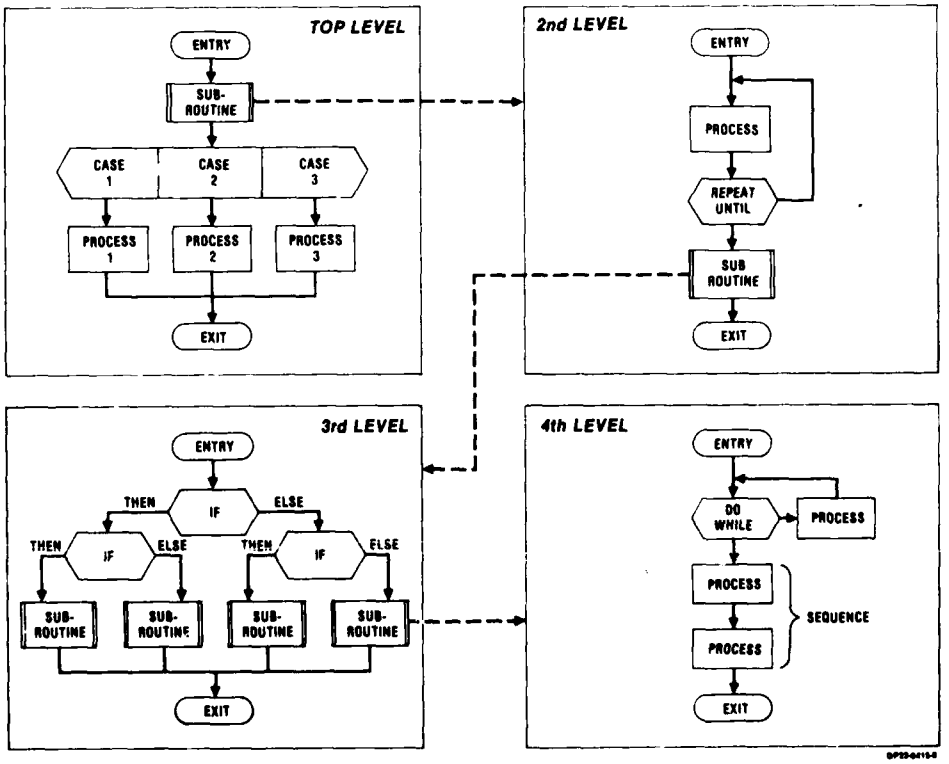


FIGURE 7
TOP-DOWN STRUCTURED SOFTWARE DESIGN

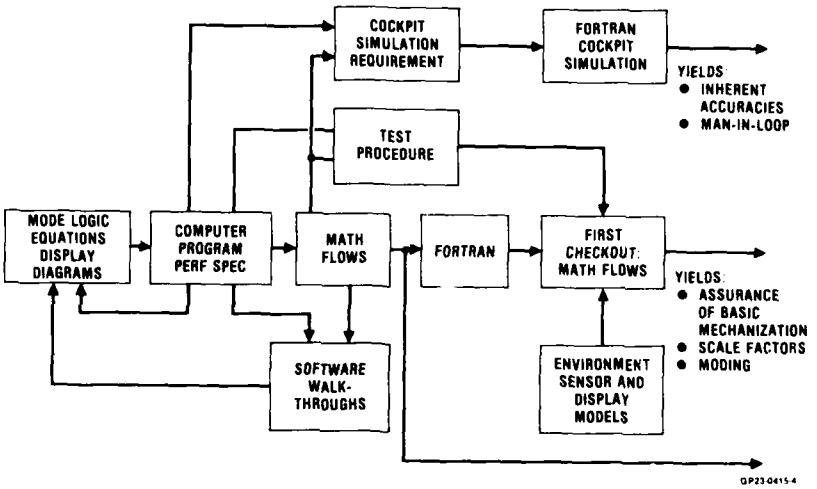


FIGURE 8
SOFTWARE DESIGN AND VERIFICATION

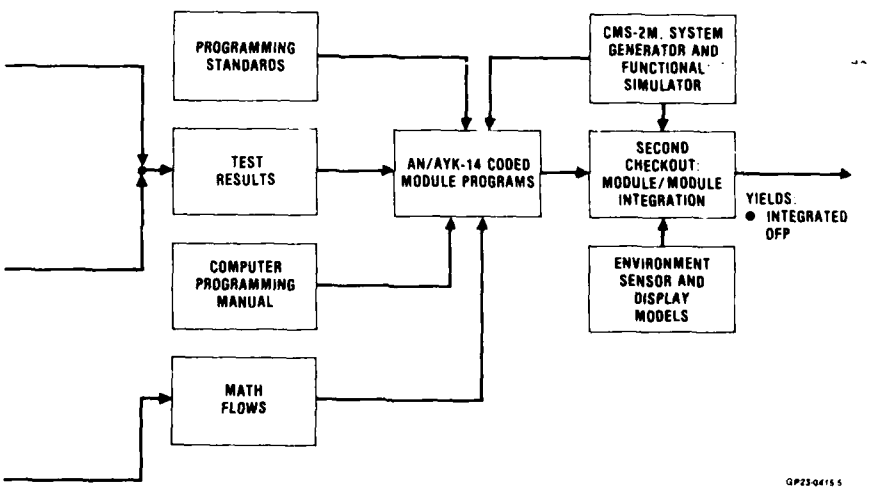


FIGURE 9
SOFTWARE MODULE CODING AND INTEGRATION

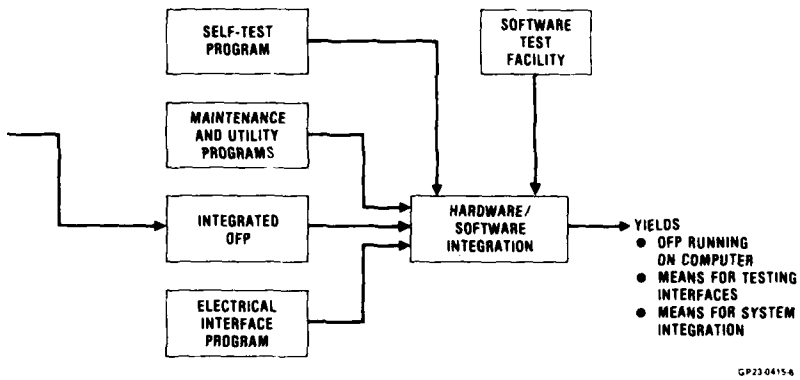


FIGURE 10
MISSION COMPUTER HARDWARE/SOFTWARE INTEGRATION

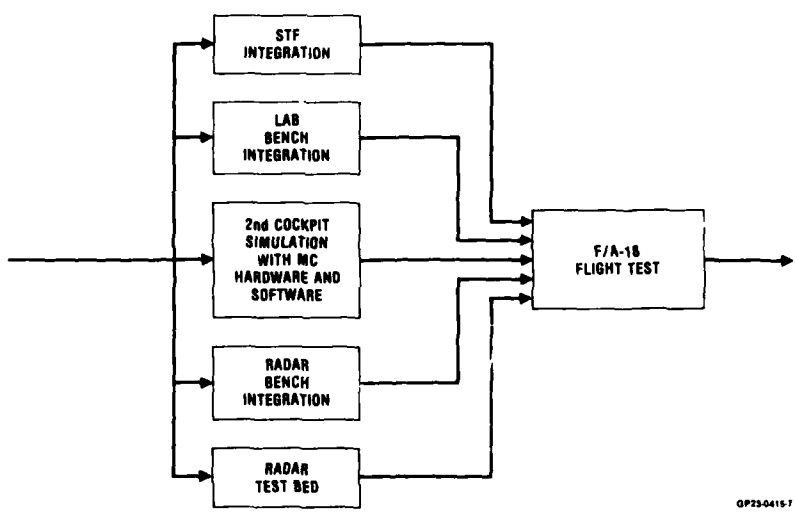


FIGURE 11
AVIONICS SYSTEM INTEGRATION AND TEST

MC = Mission computer
OFP = Operational flight program

FLIGHT TEST	7 FLIGHT TESTS USING MC AND OFP IN AIRCRAFT
COCKPIT SIM (OFP/MC)	6 MAN-IN-THE-LOOP TESTING USING ACTUAL MC AND OFP ON THE MCAIR COCKPIT SIMULATOR
BENCH INTEGRATION TESTS	5 MC/OFP TESTING AND INTEGRATION WITH INDIVIDUAL SUBSYSTEMS ON BENCH IN LABORATORY
MC/OFP INTEGRATION	4 BENCH CONFIGURATION HARDWARE/SOFTWARE INTEGRATION USING MC ON BENCH IN LABORATORY
FUNCTIONAL SIMULATOR	3 BIT-BY-BIT SIMULATION ON IBM 370 OF THE EXECUTION OF OFP CODED IN MC LANGUAGE
COCKPIT SIM (FORTRAN)	2 MAN-IN-THE-LOOP TESTING OF FORTRAN MODEL OF OFP USING CDC 6600 AND MCAIR COCKPIT SIMULATOR
FLOATING-POINT VALID (LAB STD)	1 VALIDATE EQUATIONS, ALGORITHMS, AND MODING ON IBM 370. FORTRAN MODEL PROVIDES LAB STANDARD FOR SUBSEQUENT TESTING

GP23-0415-10

FIGURE 12
F/A-18A MC SOFTWARE DEVELOPMENT LADDER

GP23-0415-18

FIGURE 13
F/A-18A SOFTWARE DEVELOPMENT FACILITY

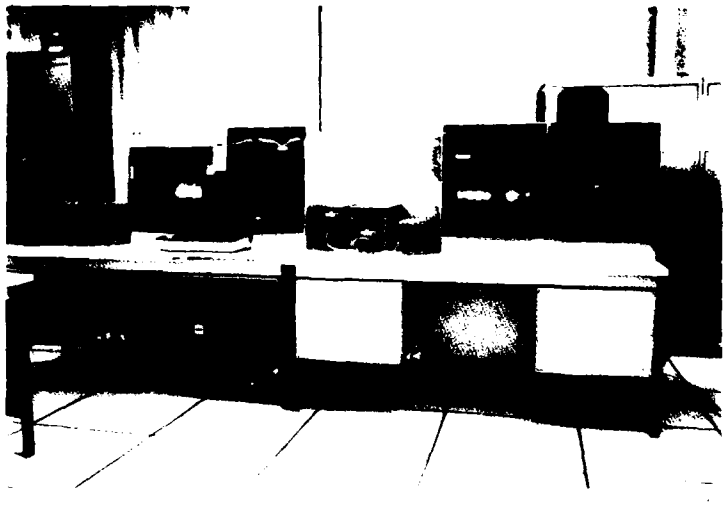


FIGURE 14
F/A-18A SOFTWARE TEST FACILITY INTEGRATION BENCH

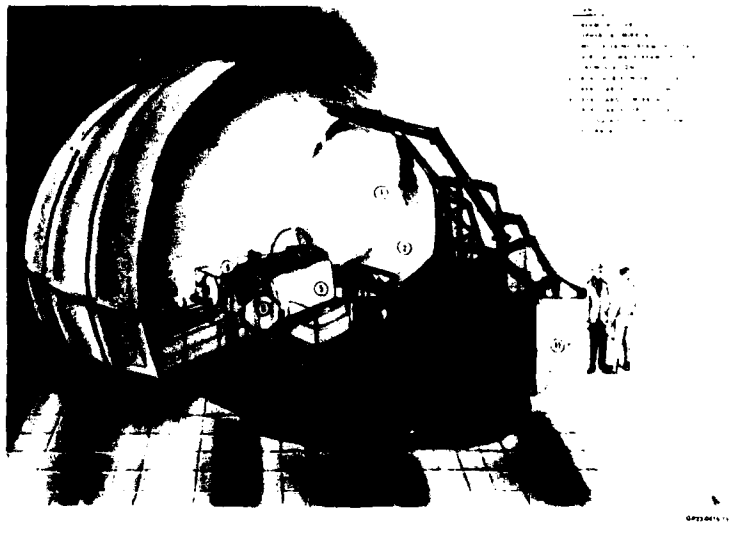


FIGURE 15
F/A-18A COCKPIT SIMULATOR FACILITY

- INDEPENDENT ORGANIZATIONAL STRUCTURE
- RIGOROUS SOFTWARE CONTROL
- SENSOR/MISSION-ORIENTED SOFTWARE PARTITIONING
- TOP-DOWN DESIGN
- STRUCTURAL SOFTWARE DESIGN
- PROVEN DESIGN PRACTICES
- SOFTWARE DESIGN AUDITS AGAINST PROGRAMMING STANDARDS
- SYSTEM DESIGN VALIDATED IN FORTRAN IN COCKPIT SIMULATOR (SOFTWARE "BREADBOARD")
- SOFTWARE TESTED IN STF WITH SIMULATED SENSORS AND ACTUAL DISPLAYS
- SOFTWARE TESTED IN COCKPIT SIMULATOR USING AIRBORNE COMPUTERS
- TOTAL WEAPON SYSTEM SOFTWARE RESPONSIBILITY PLACED WITH WEAPON SYSTEM CONTRACTOR

GP23.0415.28

FIGURE 16
FACTORS CONTRIBUTING TO F/A-18A SOFTWARE SUCCESS

A LIFE CYCLE MODEL FOR AVIONIC SYSTEMS

WissDir Dipl.-Ing. Helmut Schaaff

Bundesakademie für Wehrverwaltung und Wehrtechnik
Mannheim, Seckenheimer Landstr. 8-10

SUMMARY

A life cycle model for avionic systems has to put emphasis on the design activities and it has to differentiate three stages of design. Therefore this paper particularly emphasizes the design activities. It states that pure functional thinking is of special importance in the early phases and that this has to be strictly distinguished from technical thinking.

The model presented is an answer to the software problem and gives hints for the project management. It helps to urge early definition of user requirements and it forms the base for the application of proper tools for that purpose.

The model in its basic philosophy conforms with the regulations (DV-Richtlinie Band IV) of the German Minister of Defence (BMVg) and is refined according to the author's personal opinion.

1. INTRODUCTION

Today's avionic systems are computerized systems and therefore software has to be developed during the life cycle of such systems. This transfers the software problem in the area of avionics.

There exists a lot of experience in software development in the commercial world - mainly bad experiences - and it is worth while to learn from these, to draw consequences from them and to transfer them in a proper way into the field of avionic systems which has a series of specific requirements:

- high reliability even under hard environmental conditions
- short response time
- intensive man-machine-dialog.

These specific requirements influence the life cycle model.

The difference between the life cycle of a commercial information system and of an avionic system can briefly be summarized in the following two statements:

A commercial information system needs software design.
An avionic system needs integrated hard-software design, because the restrictions caused by the hardware have an enormous impact on the software and therefore have to be considered in detail.

It is generally accepted in the life cycle of software that there has to be more emphasis on design, but the term design is not precise enough. So, in the following chapter, I will present further detail.

2. DESIGN ACTIVITIES

2.1 THE THREE STAGES OF DESIGN

Considering the design activities in some more detail one has to realize that there are actually three stages of design corresponding to the differences in point of views which exist between

- the user of the system
- the system engineer

and

- the software engineer.

They all look at the same system but they see different things and they see them in a different way.

The user sees the man-machine system, the man-machine interaction and he has an understanding of the functions of the system in relation to its environment.

The system engineer sees the hardware and software structure and the interaction between hardware and several layers of software.

The software engineer sees the application programs, their algorithms and their data.

According to these different views, different stages of design might be named:

users point of view:	functional design
system engineer's point of view:	technical design
software engineer's point of view:	software design

2.2 THE SEPARATION AND INTEGRATION OF FUNCTIONAL AND TECHNICAL DESIGN

The separation:

The distinct separation between functional and technical design is the specific point in the life cycle model presented.

Why is this necessary?

The necessity results directly from the symptoms from which software developments generally suffer:

1 Too many design errors

These are mainly errors due to the fact that the functional interaction in the system has not been completely understood.

2 Bad user acceptance

The essential point for the user acceptance is the man-machine dialog, which very often does not take the real user into account.

3 High cost and late delivery

Because of a lack of knowledge about the user requirements the selected computer was often too small and too slow. Therefore the programmer had to save both processing time and storage space, but this, in the long run, could not be achieved by mere programming tricks. So, a more powerful computer became inevitable in the end, and the software had to be rewritten - a waste of time and money.

4 High maintenance cost

Because of a lack of knowledge about the user requirements nothing was known about the future of the system. In which direktion it had to be easily changeable and/or extendable was never considered. Therefore the changeability and extensibility was not incorporated into the system, and the implementation of changes became expensive because a lot of code had to be rewritten and retested. This is not the only reason for high maintenance cost but it contributes essentially to it.

All these experiences have one thing in common: The lack of knowledge about the functions of the system. The consequence for the life cycle model presented is the formal introduction of an activity called "functional design".

The acceptance of this activity brings project management into a position to urge the user to state his requirements as early as possible and to specify them as completely and as precisely as possible and thus to ascertain a required standard of quality. This helps the project management to prevent the technical design to be started without a qualitatively sufficient base which could only lead to a misdevelopment. You got something but the user expected something else.

Therefore the formal introduction of an activity "functional design" and its distinct separation from the technical design is required because it increases the influence of project management in a vital point.

Separation and integration:

Up to now I have stressed the separation of functional and technical design, but of course both deal with the same system, the functional design describing the requirements and the technical design describing the way they can be realized. Both need each other.

The relation is shown in figure 1.

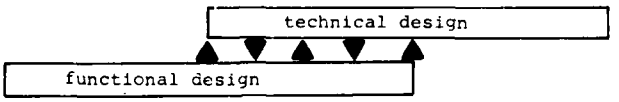


figure 1: relation between functional and technical design.

The functional design precedes because it has to produce preliminary results before the technical design can begin. From then both run in parallel and they are refined in mutual interaction. This interaction is vital to the project because the functional design gets the feedback that it is feasible and the technical design gets all the information it needs to define a cost-effective way of realization.

Therefore it is necessary for the project management to achieve both, the separation and the integration of functional and technical design. This is no contradiction - it is the consequence of the dualism of projects of that kind.

2.3 OBJECTIVES OF THE FUNCTIONAL DESIGN

The functional design, has to produce a solid base for further development activities and by this it helps to reduce the risk of development.

Therefore, the functional design has to describe the functions - in our case - of an avionic system including its operators (pilot, navigator etc.). It comprises the functions, their data and interaction between the functions. It describes which parts of the functions have to be performed automatically and which have to be operated by man, and by this it determines the man-machine dialog in its principal form.

It also states the requirements of response time, accuracies, quantities etc. These requirements are primarily those of the functions of the system. The requirements of the components of an airplane and its avionic system are derivated therefrom. To make it quite clear: The requirement that an airplane has to navigate with a certain accuracy is a requirement of the function "navigation". This belongs to the functional design. From this is derivated the technical design concerning the accuracy of a gyro-system. This requirement pertains to the technical design.

One might object that the user is not able to answer all the questions about the functional design. Of course it is difficult, but this difficulty must not lead to a postponing to later phases, it must lead to the application of proper tools. In my mind it is not acceptable that the user does not go into the details before he sees a prototype. He should be able to find the same results working with a simulation model which is cheaper to build and easier to change. This is finally connected with the question of the representation of the user in the project.

3. THE ACTIVITY CYCLE

In a life cycle of an avionic system there are a number of activities which form an activity cycle. The activities describe the kind of work, which has to be done. A complete activity cycle consists of the following activities:

analysis:	investigation of the predecessor system in order to learn about its functions and to discover the bottlenecks.
functional design:	describing the functions and data of the new avionic system and the requirements of these functions from the users point of view.
technical design:	An integrated hard-/software design which specifies how the avionic system has to be realized.

software design:	design of the application programs and the data base.
implementation:	coding, testing, and integration of the programs; gradual integration of the avionic system.
introduction:	bringing the system into the field.
utilization:	using the avionic system maintenance of hard- and software.

These activities generally do not follow each other in a strict sequence; very often they have to be done in parallel one preceding and the second following with a delay which is necessary because the first has to produce initial results. Only if two activities run in parallel interaction can take place and this goes especially for the acitivities "functional" and "technical design".

A complete activity cycle is shown in figure 2.

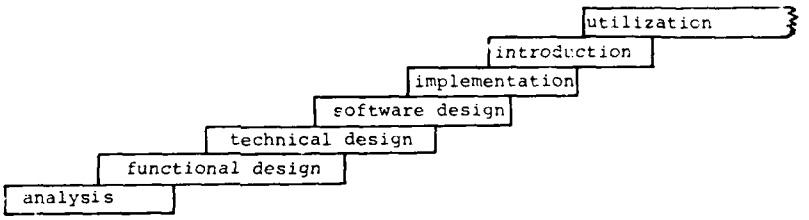


figure 2:
activity cycle

4. LIFE CYCLE MODEL

Up to now we have talked only about activities and for technically minded people these are the most significant partsbecause they define the kind of work which has to be done, but they do not form a complete life cycle model.

Actually they are only one of three aspects of a life cycle model.

Aspect 1 is the time which defines the sequence of the so called phases.

Aspect 2 is the kind of work which is defined by the activities.

Aspect 3 is the object of development, the prototype or the version.

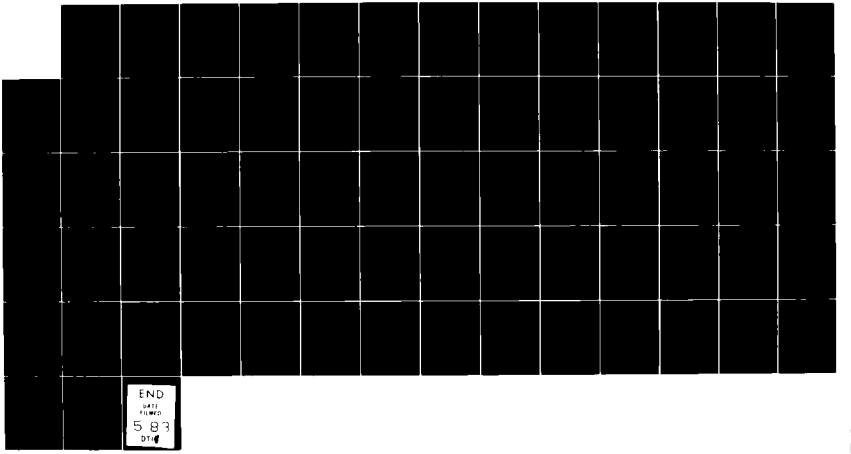
The aspects 1 and 2

Former life cycle models considered phases and activities as the same thing and by this they formed a strictly sequential model. The disadvantage of such models was that interaction between two activities or phases was generally ruled out; of course it happened nevertheless but it was considered as something irregular and project management had the tendency to prevent it. But, as we pointed out earlier, this interaction is extremely important and therefore this kind of models are not appropriate.

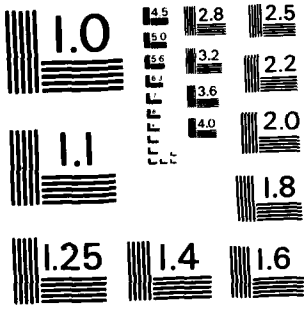
There is a strong necessity for phases and activities as different aspects. The aspect activity is described above in detail. The aspect phase is required especially from the point of view of the top management which has to approve the money. There have to be fixed milestones, which determine the objectives of contracts and which form a base for the financial schedule of the project. The aspect phase is determined by the life cycle of the weapon system while the activities are related to the avionic system. The ends of phases are marked by certain documents and by formal management decision. The relation between phases and activities is depict in figure 3. The names of the phases comply with the management regulations of the German Ministry of Defence.

AD-A127 131 SOFTWARE FOR AVIONICS(U) ADVISORY GROUP FOR AEROSPACE RESEARCH AND DEVELOPMENT NEUILLY-SUR-SEINE (FRANCE) JAN 83 AGARD-CP-330 95

UNCLASSIFIED F/G 9/2 NL



END
DATE FILMED
5 83
DTIC



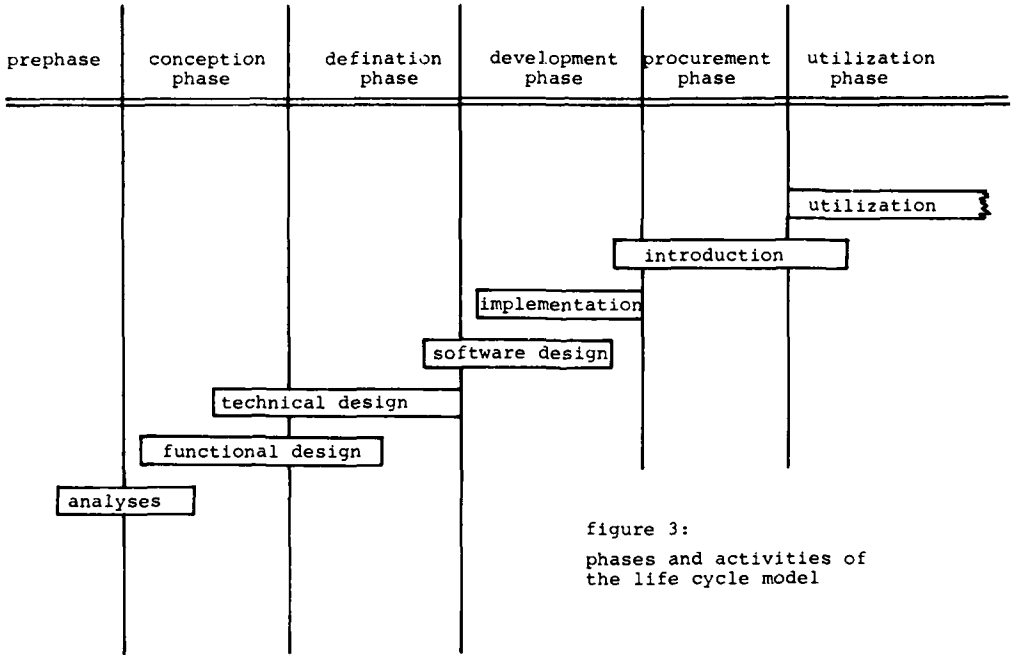


figure 3:

phases and activities of the life cycle model

Aspect 3: prototype or version

Up to now we have talked about a life cycle of a system with only one object. In reality there are several prototypes during development and a series of versions during utilization. A realistic life cycle model has to be able to handle this because it has to form the principle of order for the whole life time of the system.

We have to face the fact that within one phase different prototypes may be in different activities.

Performing a midlife conversion of an avionic system is nothing else but accomplishing a complete activity cycle in the utilization phase. This is necessary because the implementation of changes has always to begin with analysis and functional design because the changes have first to be integrated in their functional context. After this one has to decide which is the best way for technical implementation (what software and what hardware has to be changed). This is technical design.

The aspect "prototype or version" requires that we relate an activity cycle to each prototype and to each version within the life cycle. An avionic test rig for example is nothing else but a special kind of prototype, with its own activitiy cycle.

Figure 4 depicts the situation for an example of a complete life cycle model which shows an activity cycle for one prototype and three versions of the avionic system within the timeframe determinedby the phases of an airplane.

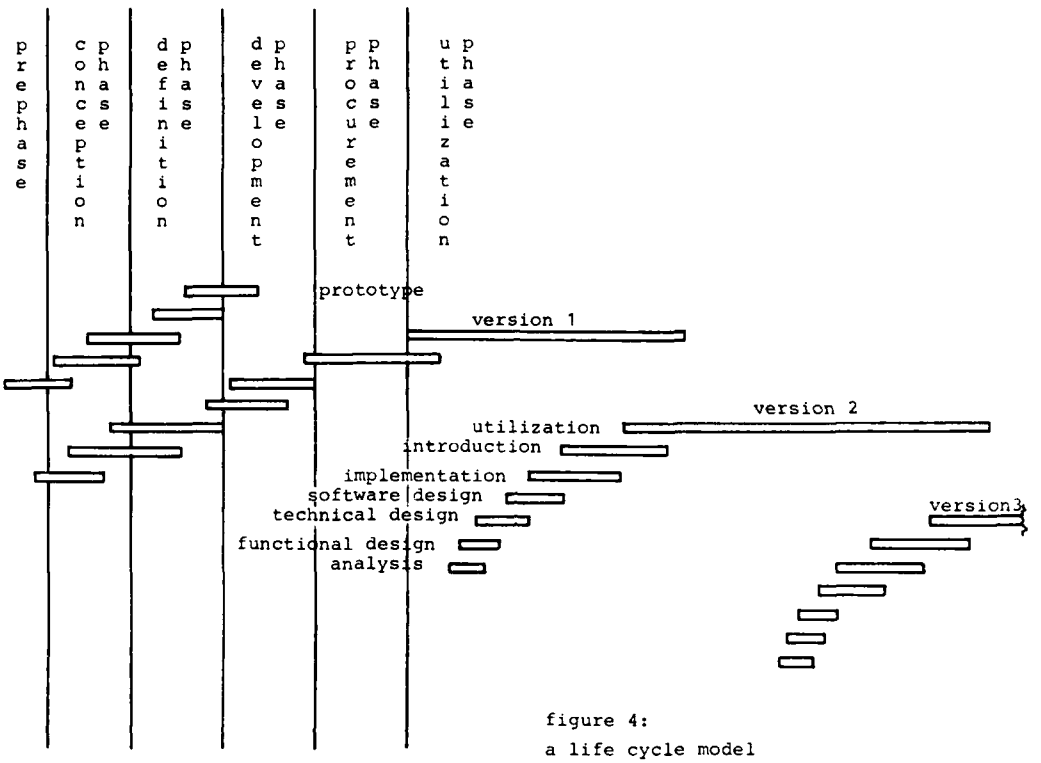


figure 4:
a life cycle model

5. THE LIFE CYCLE MODEL AS THE BASI FOR PROJECT MANAGEMENT

The life cycle model serves as the principle of order in the project:

1. It forms the base for project planning.
2. It separates the responsibilities in an natural way between user und data processing specialist. The user is in charge of the functional design because only he can say what he needs.
3. It reduces the risks in development because it cares for a better base for the technical design.
4. It improves the final acceptance by the user because it involves the user early and urges to look for an appropriate man-machine dialog.
5. It serves as a base for a documentation standard (a standard of content).
6. It forms the base for the systematic in the project documentation and all the paper work of the different versions. Thereby it improves transparency in the project.
7. It forms the base for the quality assurance measures (reviews etc.)
8. It defines areas for the application of software-technological tools.
9. It forms the base for a configuration management by defining the currently valid documents, which serve as the base for the current work.

6. CONCLUSION

The objective of the project management of an avionic system must be to bring forth the user requirements as completely, as correctly and as early as possible because this saves money and time. The life cycle model presented helps to achieve this especially by the introduction of the formal activity "functional design" and its distinct separation from the technical design.

The presented model is valid for avionic systems but not only for these. It is valid for military embedded computer systems in general.

REFERENCES

METZGER, Philip W., 1973 "MANAGING A PROGRAMMING PROJECT", Prentice-Hall

FLOYD, C., 1981, "A PROCESS-ORIENTED APPROACH TO SOFTWARE DEVELOPMENT" in Systems Architecture Proc. of the 6th European ACM regional Conf., Westbury House 1981

AVIONICS SOFTWARE SUPPORT COST MODEL

BY

DANIEL V. FERENS
U.S. AIR FORCE AVIONICS LABORATORY
AFWAL/AAAS-2
WRIGHT-PATTERSON AFB, OH 45433, U.S.A.

SUMMARY:

The United States Air Force (USAF) Avionics Laboratory is currently developing the Avionics Software Support Cost Model (ASSCM). This effort was contracted to SYSCON Corporation in September, 1980 and is scheduled for completion in September, 1982. The ASSCM will be based on historical software support cost data from Air Force Logistics Centers and is designed for use during the conceptual phase of a project. The current version of the ASSCM is limited to U.S. Air Force and Navy avionics software applications. However, the model is being developed in a modular format so that it can be expanded to include other software applications. These applications may include space systems, ground-based systems, and NATO software systems.

Successful completion of the ASSCM will represent a significant advancement in the area of life cycle cost analysis. The ASSCM will aid U.S. and NATO organizations in analyzing and, consequently, reducing software support costs.

I. INTRODUCTION:

The Concepts and Evaluation Group of the USAF Avionics Laboratory (AFWAL/AAAS-2) is responsible for analyzing life cycle costs of avionics systems during the conceptual, or early design phase of these systems. These costs consist mainly of the costs associated with acquisition and support of hardware and computer software. To analyze these costs, AFWAL/AAAS-2 develops or acquires cost models or methods which are suitable for use during the conceptual phase of an avionics program. The ASSCM was developed to satisfy the Avionics Laboratory's need for a model to analyze software support costs.

II. BACKGROUND:

1. The Problem: Software support costs comprise an increasingly significant portion of system life cycle costs. In fact, Dr Barry Boehm, a renown cost analyst of TRW Corporation, has stated that, by 1990, software support costs may account for as much as 60% of total system life cycle costs (Ref. 1). The Avionics Laboratory, however, has had no adequate model or method for analyzing software support costs during the conceptual or early design phase of an avionics software program. A study performed by the Avionics Laboratory in 1979 (Ref. 2) concluded that no models existing at that time were adequate for the Laboratory's analysis requirements. The Avionics Laboratory, therefore, decided to develop a model that would be useful during the conceptual, or early design phase of an avionics system. It was decided that the model should be based on historical data which reflects actual costs of supporting Air Force avionics software along with current Air Force policies and procedures for software support. The model development effort was divided into two phases. The objective of the first phase (Phase I) was to determine the feasibility of developing the model and to determine a roadmap for model development. The objective of the second phase (Phase II) was to develop a model using the roadmap determined during Phase I. The Avionics Laboratory decided that the Phase II effort would not be accomplished unless the Phase I effort was successful.

2. Phase I Results: A contract for the Phase I effort was awarded to Hughes Aircraft Corporation in April, 1979. Hughes was required to perform four tasks to determine model feasibility and define a roadmap for model development. Detailed results of the Phase I effort are compiled in the final technical report (Ref. 3) for the effort. However, a summary of results for each Phase I task is presented below.

The first task required Hughes to survey existing software support cost models for possible applicability to the Avionics Laboratory cost model effort. Hughes surveyed over 20 existing models, but concluded that no existing models were adequate.

The second task required Hughes to visit USAF Air Logistics Centers (ALCs) where most Air Force avionics software is currently supported. During these visits they were asked to determine current Air Force policies and procedures for supporting avionics software and to investigate the availability of historical cost data for the software model development. Hughes visited all five Air Force ALCs and discovered that Air Force avionics software is normally supported using a block change procedure. Instead of making continuous changes as requested, the ALCs, requested changes are grouped into a "block" and incorporated during a block change cycle lasting from six to twenty months, depending on the software being supported and organization performing the block change. It is possible for emergency changes to be performed outside of the block change cycle, but this is rarely done. Hughes also determined that there was a limited amount of historical cost data available for avionics software. The data available included block change cost data for software programs on aircraft such as the A-7D, F-111, FB-111A, and F-16. Although the amount of data was limited, Hughes concluded that development of a model was still feasible.

The third task required Hughes to determine several approaches to model development and to select the best approach. The approaches suggested by Hughes included an element estimate, or "bottoms-up" approach, an analogy, or "sideways" approach, and a cost-estimating relationship, or "top-down" approach. Hughes decided that, although the top-down approach would be most desirable for the model, the limited amount of data available could not easily support this approach. Much more data would be required to perform the statistical regression analysis necessary to develop cost-estimating relationships. Hughes

recommended, therefore, that a bottoms-up approach was best because it could be used with a limited amount of historical data. The approach included the use of default values in order that the resultant model would be useful during the conceptual phase.

The fourth and final task required Hughes to define a roadmap for model development. The roadmap was incorporated into the Statement of Work for the Phase II effort. The four basic tasks for the Phase II effort are as follows:

a. Data Collection: Collect additional data to insure that as much historical cost data as possible is incorporated into the cost model.

b. Model Development: Develop the model and related data base using historical data and code the model in FORTRAN programming language.

c. Model Validation: Validate the model using at least three programs for which historical data were collected, but which were not used in model development.

d. Model Installation: Install the model on computers at Wright-Patterson Air Force Base, OH for Air Force use. Also, train Air Force personnel in use of the model such that the model can be used without contractor assistance.

Hughes completed the Phase I contract in June, 1980. Based on the positive results of the Phase I study, the Avionics Laboratory decided to continue the model development effort into Phase II.

3. The Phase II Effort: A contract for the Phase II effort to develop the ASSCM was awarded to SYSCON Corporation in September, 1980. As of this writing, the first two tasks, listed under the Phase I roadmap, are nearly completed, and the entire ASSCM effort is expected to be completed by 30 September 1982. The completed and planned efforts for each of the four Phase II tasks are described below:

a. Task 1 - Data Collection: SYSCON collected historical cost data on over 15 avionics software programs at four ALCs. However, some of the historical data was incomplete, and SYSCON could only obtain complete data on fewer than 12 programs. This verified Hughes' conclusion that there would probably be insufficient data for developing a top-down model. Therefore, SYSCON decided to supplement the historical data collection with another model development technique. The techniques chosen was the Delphi technique, which involves a survey of experts. This technique had been used successfully in developing other cost models such as the RCA Corporation's PRICE-S software development cost model (Ref. 4). The Delphi technique also gives added flexibility to models such as ASSCM, as it can address issues for which historical data is not available.

For the historical data, SYSCON divided the programs into seven application types. These types are listed in Table 1. SYSCON chose at least one program from each type for a historical data base. The cost data and descriptive data from selected programs were used as a baseline from which that application type's cost would be computed. A sample descriptive data for a historical program used in the model baseline is shown in Table 2 under "Sample Historical Values." Also collected were cost data called "normalization factors" that are used to adjust baseline data into an initial cost estimate. This data is summarized in Table 3. The normalization process used in ASSCM is described in more detail in Section III of this paper.

TABLE 1: SOFTWARE APPLICATION TYPES

1. Fire Control Operational Flight Program (OFP)
2. Navigation/Weapon Delivery OFP
3. Navigation/Fire Control/Weapon Delivery OFP
4. Electronic Warfare (EW) Receiver
5. EW Jammer
6. EW Integrated System
7. Communication-Electronics (CE): Command and Control

The Delphi survey results were also input to the model baseline to be used as "Adjustment Factors." Knowledgeable personnel were asked to assess the relative manhour impact (increases or savings) resulting from changes in factors such as language used, size of program, complexity, etc. They were asked to assess the manhour impact in each of eight software support phases shown in Table 1, where sample data for two factors are presented. The Delphi survey also asked knowledgeable personnel to describe a "typical" software program in terms of the descriptors listed in Table 2. These values are used for the representative baselines and for input default values, which are discussed in more detail in Section III of this report. The "typical" software program in each category does not necessarily resemble the actual program used in the historical data base. Typical values for one program are listed under "Sample Default Values" in Table 2.

b. Task 2 - Model Development: As of this writing, the model is about 80% coded in interactive FORTRAN language. The details of the model are described in Section III of this paper. The development of ASSCM is scheduled to be completed in July, 1982.

c. Task 3 - Model Validation: After model development is completed, SYSCON will validate ASSCM using three programs for which historical data was obtained, but which were not used in the ASSCM data

base. ASSCM will be run using the historical data, and ASSCM cost outputs will be compared with historical costs. Any significant deviations will be investigated and may result in modification of some of the model equations. However, no major modifications are expected to be required.

TABLE 2: VARIABLES FOR WHICH THE USER PROVIDES VALUES

PARAMETERS	RANGE OF PLAUSIBLE VALUES	TYPICAL SAMPLE VALUE	SAMPLE HISTORICAL VALUE
1. APPLICATION TYPE (1-7)	Seven types available (see Table 1)	2	2
2. LINES OF CODE	4K - 500K	15K	16K
3. LANGUAGE (1 - 3)	Assembly, FORTRAN, Structured HOL	1	1
4. % MEMORY FILL	50% - 100%	90	100
5. % TIMING FILL	50% - 100%	90	95
6. DEVELOPMENT V&V RATING (1 - 3)	None, Done by Developer, Total IV&V	2	1
7. DESIGN RATING (1 - 4)	Poor, Fair, Good, Excellent	2	1
8. IMPLEMENTATION RATING (1 - 4)	Poor, Fair, Good, Excellent	2	1
9. INITIAL DOCUMENTATION RATING (1 - 4)	Poor, Fair, Good, Excellent	2	3
10. YEAR OF SUPPORT	1 - n	3	2
11. AIRCRAFT TYPE (1 - 4)	Cargo, Bomber, Fighter, Surveillence	3	3
12. NO. FIELDED SYSTEMS	1 - n	600	268
13. COMPLEXITY	1 - 5	3	4
14. RATE OF CHANGE	1 - 5	4	4
15. SKILL LEVEL MIX	1 - 5	3	4
16. CHANGE EFFICIENCY	10% - 100%	50%	50%
17. DIRECT SUPPORT EQUIPMENT	1 - n ($ Million)	7	6
18. EXPECTED SYSTEM LIFE	1 - n (Years)	20	20
19. BLOCK CHANGE LENGTH	1 - n (Months)	12	12
20. SUPPORT SOFTWARE MAINTENANCE (1,2)	yes, no	1	1
21. % OF WORK PERFORMED BY CONTRACTOR	0% - 100%	0	0
22. LOCATION OF CONTRACTORS (1,2)	on-site, off-site	0	0
23. REPRODUCTION MEDIUM (1 - 3)	Mylar Tape, PROM, Mag Tape	3	1

TABLE 3: NORMALIZATION FACTORS AND DEFAULT VALUES

FACTOR	DEFAULT VALUE
ORGANIC LABOR COST/MAN-MONTH BY GRADE	(VARIES)
CONTRACTOR LABOR COST/MAN-MONTH	$7,000
SUPERVISION RATIO	.13
ADMINISTRATIVE RATIO	.13
SUPERVISION COST/MAN-MONTH	$4,390
ADMINISTRATIVE COST/MAN-MONTH	$1,847
ADMINISTRATIVE COMPLEXITY FUNCTION	(VARIES)
INFLATION FACTORS	(VARIES)
TEST & EVALUATION RATIO	OFP: 3%, EW: 5%, CE: 3.5%
COST/HOUR/TEST AIRCRAFT TYPE	(VARIES)
COST/REPRODUCTION BY MEDIUM	(VARIES)
MEDIUM REPRODUCTION FACTOR	(VARIES)
SPACE/PERSON	Technical: 275 ft^2, Supervisory: 130 ft^2
BUILDING COST/SQUARE FOOT	$136
UTILITY COST/SQUARE FOOT	$1.20
FURNISHING COST/PERSON	$680
MATERIALS AND SUPPLIES COST/PERSON	$700
GENERAL COMPUTER COST/PERSON	$20,000
HARDWARE MAINTENANCE COST RATIO	10%

TABLE 4: EXAMPLES OF MODIFICATION FACTORS

PHASE	LANGUAGE		
	ASSEMBLY	FORTRAN	STRUCTURED HOL
Requirements Review	1.00	.94	.91
Design	1.00	.84	.78
Development	1.00	.67	.62
Integration	1.00	.83	.76
Test & Evaluation	1.00	.87	.82
Documentation	1.00	.82	.75
Repro/Installation	1.00	.95	.91
Support Software	1.00	1.00	1.00

TABLE 4. EXAMPLES OF MODIFICATION FACTORS (Continued)

PHASE	DEVELOPMENT VERIFICATION AND VALIDATION (V&V)		
	NONE	DONE BY DEVELOPER	TOTAL INDEPENDENT V&V COMPLETE
Requirements Review	1.41	1.00	.90
Design	1.56	1.00	.80
Development	1.65	1.00	.81
Integration	1.68	1.00	.78
Test & Evaluation	2.05	1.00	.67
Documentation	1.68	1.00	.83
Repro/Installation	1.18	1.00	.96
Support Software	1.00	1.00	1.00

NOTE: If year of support is 3 or more, then the modification value for development V&V rating is 1.00.

d. Task 4 - Model Installation and Training: It is planned that the model will be installed on computers at Wright-Patterson Air Force Base in September, 1982. At this time, a number of potential ASSCM users will be trained by SYSCON in a one-to-two day training course. The ASSCM will then become the property of the Avionics Laboratory who will maintain control of the model.

III. ASSCM DESCRIPTION:

The current description of ASSCM, which is described in detail in the ASSCM Software Design Specification (Ref. 5), is summarized in this section of the paper. The model is written in interactive FORTRAN for ease of use and ease of modification, and is designed for use especially during the conceptual, or early design phase of an avionic software program.

1. Model Inputs:

A listing of inputs to ASSCM is shown in Table 2. Although there are 23 input parameters for the model, the user is not required to specify every input. Instead, after specifying the application type, the user may merely allow the model to use any of the typical values for the category selected. The typical values probably are reasonable values for the type of software being analyzed. However, it is recommended that the user specify as many input parameters as possible so that his software program is described to the greatest extent possible. The user may also update normalization factors within the model before running it. These are described in more detail under "Model Processing" in this section of the paper.

2. Model Outputs:

The outputs of ASSCM consist of software support costs which are divided into four categories. These categories are: direct labor, indirect labor, direct support equipment, and indirect support equipment. Figure 1 shows a sample output cost listing, including the cost breakout in each of the cost categories.

The user has flexibility in determining what type of output he desires. He may merely ask for a listing of total costs for each year of software support, or a detailed summary of costs for each year as is shown in Figure 1. The user may also ask for cost derivation listings which show how costs were computed in each of the four major categories.

FIGURE 1. EXAMPLE OF ANNUAL COST SUMMARY

TOTAL ANNUAL COST				$892,503
TOTAL LABOR			$203,876	
DIRECT LABOR		$166,355		
REQUIREMENTS REVIEW	$ 11,220			
DESIGN	$ 18,050			
DEVELOPMENT	$ 18,050			
INTEGRATION	$ 18,050			
TEST AND EVALUATION	$ 45,370			
DOCUMENTATION	$ 38,540			
REPRODUCTION/INSTALLATION	$ 6,830			
SUPPORT SOFTWARE	$ 10,245			
INDIRECT LABOR		$ 37,521		
SUPERVISION	$ 26,779			
ADMINISTRATION	$ 10,742			
TOTAL SUPPORT EQUIPMENT			$688,627	
DIRECT		$498,183		
HARDWARE	$ 54,508			
SUPPORT SOFTWARE	$248,600			
TEST AIRCRAFT/TIMF	$188,000			
REPRODUCTION	$ 7,075			

FIGURE 1. EXAMPLE OF ANNUAL COST SUMMARY (Continued)

GENERAL		$190,444
FACILITY/UTILITIES	$ 10,880	
FURNISHINGS	$ 204	
MATERIALS/SUPPLIES	$ 4,200	
COMPUTERS/TERMINALS	$ 4,000	
MAINTENANCE	$171,160	

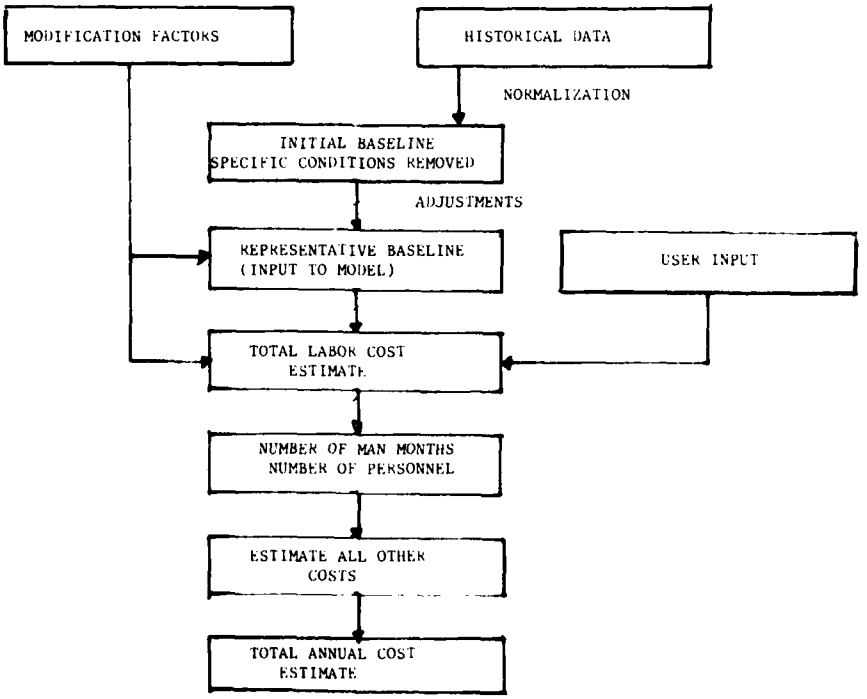
3. Model Processing:

This paragraph describes the overall methodology by which the model receives input data and computes software support costs. The overall algorithm is first described, then the modules which implement the algorithm are described in some detail.

a. Model Algorithm: Figure 2 is a block diagram of the basic ASSCM algorithm. The model uses historical data, modification factors (derived from Delphi surveys), and user inputs to compute software support costs. The basic algorithm process consists of twelve steps described as follows:

1) Receive user inputs
2) Select appropriate historical baseline
3) Derive normalization factors from default values and user modifications
4) Apply normalization factors to historical baseline to derive initial baseline
5) Apply adjustment factors to initial baseline to derive representative baseline
6) Apply characteristics input by user and modification factors to representative baseline to derive total direct labor cost estimate
7) Derive allocation factors from default values and user modifications
8) Apply allocation factors to total direct labor cost estimation to derive number of man-months and personnel
9) Derive all other cost elements from number of man-months, number of personnel, and default values
10) Sum all costs to determine total annual cost
11) Output results to user
12) Repeat process changing some data values is desired

FIGURE 2. ASSCM ALGORITHM



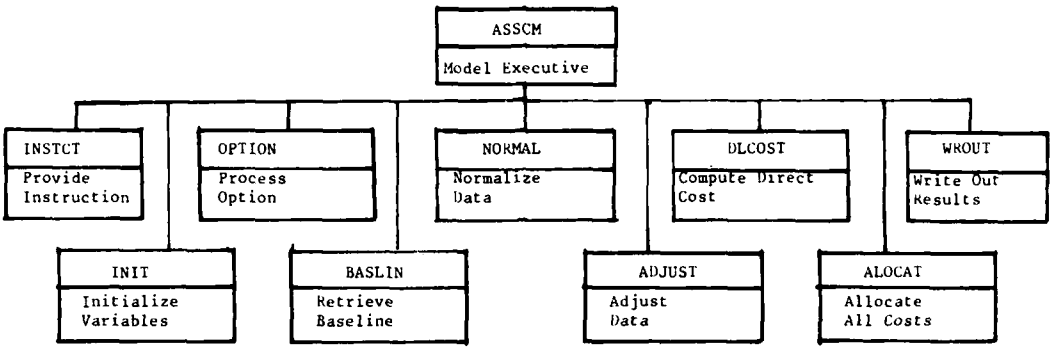
b. Module Functions: ASSCM software is developed in a modular format for ease of modification and understanding of the model. There are 10 modules in ASSCM which implement the model algorithm and aid the ASSCM user. The hierarchy of modules is shown in Figure 3, and each module's functions are described as follows:

1) ASSCM Executive: This module provides overall control of the ASSCM model. It invokes all other ASSCM modules as required.

2) Provide Instruction (INSTCT): This INSTCT module provides the user with detailed instructions for using the ASSCM.

3) Initialize Variables (INIT): The INIT module initializes all variables to their default values. These variables include normalization factors listed in Table 3, as well as the historical and typical values for input parameters as outlined in Table 2.

FIGURE 3. ASSCM MODULES



4) Process Options (OPTION): The OPTION module allows the ASSCM user to provide input data to the model, modify normalization factor default values, and choose desired output formats. These three subfunctions of OPTION are further described as follows:

a) For input data, the user may provide a value for all parameters listed in Table 2. If the user does not know a value for any parameter or chooses not to input a value for a parameter, the model will default that parameter to the typical value for the application type chosen. The user also may save his input values on a file that may be used later. In this way, the INIT module allows the user to analyze the support cost impact of changes in one or two parameters without having to create a new input file each time.

b) For normalization factors, INIT allows the user to view the current default values of these factors listed in Table 3. The user may then modify any of these values he chooses if he desires a different value than the ASSCM default value. For output format, the user may choose which format he desires to use, as described under the "Model Outputs" paragraph in this paper.

5) Retrieve Baseline (BASLIN): The BASLIN module retrieves and lists appropriate historical baseline data for direct labor based on the application type selected. Each data base contains numbers of man-months and skill categories of personnel required for each of the eight support phases during a block change. This data normally can not be modified by the user.

6) Normalize Data (NORMAL): The NORMAL module performs Step 4 of the model algorithm by deriving the initial baseline. The module uses historical data and the normalization factors defined in Table 3 to compute an initial baseline of annual costs. It accomplishes this by the following procedure:

a) Annual direct labor costs are determined by multiplying number of man-months required for a block change by cost per man-month, and adjusting for the length of the block change.

b) Indirect labor costs are then computed by applying supervisory and administration normalization factors to direct labor man-months.

c) Annual direct and general support equipment costs are then computed by applying appropriate normalization factors to the appropriate variables.

7) Adjust Data (ADJUST): The ADJUST module performs Step 5 of the model algorithm. It applies adjustment factors to the historical cost baseline derived from the NORMAL module to compute representative baseline costs. The representative baseline is based on typical values for each application type as derived from the Delphi survey. The ADJUST module first computes direct labor costs using equations based on modification factors derived from the Delphi surveys. These equations adjust costs from the historical baseline to the representative baseline. These are separate equations for each modification factor and for each of the eight phases of software support. Once direct labor costs are computed, all other costs are computed in a similar manner used in the NORMAL module.

8) Compute Direct Cost (DLCOST): The DLCOST module performs Step 6 of the ASSCM algorithm to derive the total direct labor cost of the software program specified by the user. This module is similar to the ADJUST module except that it adjusts the direct labor costs of the representative baseline to those of the user's software program. The same equations are used as in the ADJUST module. Only direct labor costs are computed by the DLCOST module.

9) Allocate All Costs (ALOCAT): The ALOCAT module performs Steps 7-9 of the ASSCM algorithm. The module first allocates total direct annual labor costs to each of the eight phases of software support by percentage, as is shown in Table 5, "Allocation Factors." The module then computes all other cost elements, defined in Figure 1. This module also sums all cost elements to obtain total annual costs.

10) Write Out Results (WROUT): The WROUT module retrieves cost data computed by the DLCOST and ALOCAT modules and outputs whatever data is requested by the user in the OPTION module.

TABLE 5. ALLOCATION FACTORS

Direct Labor

Phase	% of Total Direct Man-Months	
	Support Software	No Support Software
Requirements Review	9	11
Design	13	17
Development	16	21
Integration	10	13
T&E	16	21
Documentation	11	14
Repro/Installation	2	3
Support Software	23	0

IV. EXPANDED USE OF ASSCM:

Although the ASSCM is being developed primarily for use in analyzing support costs of USAF avionics software, it can be modified for use on other software, including NATO applications. One of the following sets of procedures will probably be followed depending on the degree of modification required.

1. Similar Software Applications: If the software is in one of the seven application types currently considered by ASSCM, and if policies and procedures used to support the software are similar to those used by the USAF, then ASSCM can be used with minimal modification. U.S. users can probably use ASSCM "as is," while other NATO users will need to change inflation tables and convert dollar outputs to their currency units. The annual inflation rates are normalization factors which can be changed as part of the normal data input procedures.

2. Similar Procedures, But Different Applications: If a user's application is significantly different that any of the seven application types listed in Table 1, it will be necessary to modify ASSCM internally before it can be used. Additional collection of historical data will be required on at least one program in the new category and the model input options must be adjusted to include this new category. It should not be necessary, however, to take a Delphi survey to update modification factors unless some parameters differ radically from those of the seven existing application types.

3. Different Procedures: If the procedures used to support software differ significantly from the block change procedures used by the USAF, ASSCM may be of limited use. It will probably be necessary to make major revisions to ASSCM or even to develop a new model to accommodate different procedures. In this case, a new model may be developed using the same methods used to develop ASSCM. These methods appear to work well when there is a limited amount of good historical data available.

V. CONCLUSIONS:

ASSCM promises to represent a significant milestone in the area of software life cycle cost analysis. The use of historical data insures that ASSCM reflects actual costs of software support, as well as the policies and procedures used by USAF ALCs to support software. The use of the Delphi survey results enables ASSCM to be useful for a wide variety of avionic software programs. The model has been designed to be easy to use, especially during the conceptual, or early design phase of a software program. Minimal input data is required. The model has also been developed in a modular format so that the model will be relatively easy to modify as new data becomes available or new application types are added. ASSCM can be useful to some degree for many U.S. and NATO software programs, especially on military avionics projects. The model may need some degree of modification, however, for applications significantly different from those for which ASSCM was developed.

REFERENCES:

1. Boehm, Barry W., "Software Engineering," TRW Publication 55-76-08, October, 1976.

2. Ferens, Daniel V. and Harris, Robert L, "Avionics Computer Software Operation and Support Cost Estimation," NAECON 1979 Conference Proceedings, May, 1979.

3. Hughes Aircraft Company, "Predictive Software Cost Model Study," Final Technical Report AFWAL-TR-80-1056, Volumes I and II.

4. RCA PRICE Systems, "RCA PRICE-S Reference Manual," Page II-17, December, 1980.

5. SYSCON Corp., "Avionics Software Support Cost Model: Software Design Specification," March, 1982.

A SOFTWARE-COST DATABASE FOR AEROSPACE SOFTWARE DEVELOPMENT

G.J. Dekker
National Aerospace Laboratory NLR
P.O. Box 90502
1006 BM AMSTERDAM

SUMMARY

Cost estimation of software development and control of the cost during the development are difficult due to the lack of applicable cost figures from previous projects, and consequently due to the lack of an accurate cost estimation and management method.

In order to improve this situation, a user-friendly method for the collection, storage and retrieval of software-cost data has been developed, with emphasis on aerospace software projects. Data is and will be collected regarding 47 well-defined cost factors, divided in 8 classes. It is felt that the clear definition of these cost factors will be of main importance for the applicability of the collected data.

When suitable data is available from completed projects, the impact of these factors on the software development cost can be estimated. This will lead to a more reliable cost estimation and cost management method.

The paper describes the cost estimation method that will be calibrated by means of the collected data, the implemented data collection and retrieval system, called a software-cost database, and the use of this system as management tool during running projects. For some projects, the cost database is already in use.

The described study has been performed under contract with the Netherlands Agency for Aerospace Programs (NIVR), contractno. 1870.

1. INTRODUCTION

The current trend in developing avionics and defence systems is a continuous move to implement functions in digital systems, especially in embedded computer systems. A result of this is that the development of the related software becomes more and more important for the cost of the development of the total system. This paper describes a systematic approach to control the cost of software development.

The management of software development cost concerns mainly two fields: cost estimation and cost control (Fig. 1). Cost estimation is done before a software project is started. Cost control takes place during project development. It includes monitoring the effort spent and taking corrective action when deviations of the planned effort do occur. A wrong cost estimate before a project will either result in an apparent cost overrun (when the estimate was too low) or in too expensive software (when the estimate was too high). Poor cost control during a project will also result in cost overrun, as causes of possible overruns are detected too late to make corrective actions effective.

In spite of the enormous investments in software nowadays, in most cases it is still impossible to give realistic estimates for the cost of software development, let alone to control this cost effectively. Two causes of this problem are identified (Posthuma de Boer, 1978):

- There is not enough quantitative and qualitative information available about the cost estimation process.
- There are virtually no interpretable cost figures available about previous projects.

This paper describes the results of a study to improve this situation. It describes the concepts of cost estimation and cost control, followed by an overview of available methods and databases. After that, the approach for the definition of a cost estimation method and the related cost database is given.
The paper concludes with the first practical results and the work planned for the coming years.

2. COST ESTIMATION AND CONTROL

The area of cost estimation can be divided in two related fields: Life-Cycle Costing (LCC) and Design-to-Cost (DTC).

Life-Cycle Costing involves the determination of the development cost from given, but most of the time vague, project specifications (see fig. 2). Usually this is done only in the early project stages; LCC can, however, also be used to determine the impact of major changes during a project and as a tool for the cost control process.

Design-to-Cost implies that the cost of a project is limited by a certain maximum and that the scope of the project should be adapted to this limit (Herd c.s. 1977). This adaptation can involve the removal of certain functions and/or design constraints from the initial specifications, possibly by making a trade-off between software and hardware cost. Normally, this adaptation is an iterative process, in which at every step functions are removed or constraints are adapted, until the estimated costs are below the maximum. At each step, the designer needs a quick way to obtain an accurate estimate and an indication of the cost impact of each function or design constraint.

Both Life-Cycle Costing and Design-to-Cost supply the estimator with a planning of expenses (see Fig.2). This planning is the basis for the cost control process. For software projects, the life cycle is divided into a number of project phases to allow for a better monitoring and control. Each phase ends with well defined products.

A useful cost control method allows that the total effort is divided between those phases. Furthermore, possiblities have to exist to update and refine the planning when the project proceeds and the project requirements or the development environment change during the project.

An overview of normally used phases and their products is presented in figure 3. Shortly they are:

- Conceptual phase. In this phase, the user requirements are defined.
- Definition phase. In this phase, the global technical solution is defined. The system is divided into sub-systems and their interfaces are defined. The overall dataflow is specified and the functions of each sub-system are identified. The first version of the user manual and the test plan are also produced.
- Design phase. In this phase, the system is designed to a sufficient level of detail to start the implementation. The products of this phase are the design description and the detailed test plan.
- Implementation phase. In this phase the system is coded and tested on unit level. The products of this phase are the code, the software documentation and the user manual.
- Qualification phase. The sub-systems are integrated and tested according to the test plans. The products of this phase are the approved code, system description, and test reports. After this phase the software is ready for operational use.

3. AVAILABLE METHODS AND DATA BASES

Roughly spoken the following methods are currently available for the estimation of software costs, (Fig. 4)

- The analogy method. The new project of which the cost have to be estimated is compared with previous similar projects. The cost of these projects and the differences in technical solution and development environment of those projects and the new project form the basis for the cost estimate.
- The parametric method. The characteristics of the new project are described with a number of figures such as size, complexity, experience of the developers etc. The cost of the project is obtained by feeding those figures into some magic formula. The latter is based on the cost data of previous projects.
- The decomposition method. The project is decomposed into a number of smaller units whose cost can be estimated more easily. Those cost are then summed to obtain a cost estimate for the total project.

All three methods are based on stored cost information about previously finished projects. With respect to this three problems exist (Dekker 1981):

- Cost data is scarcely available, among others due to the confidential aspect of cost figures.
- The use of ill-defined terms gives rise to unallowable differences in interpretation of the available figures. The use of various measures for a term like "software size" can give a variation in the measured results of a factor of 10 (Herd c.s. 1977).
- Especially for embedded systems it is difficult to separate software cost from the total system cost, let alone to know the allocation of software cost to the various life cycle phases.

It is therefore not surprising that none of the currently used cost estimating methods has been proven to be reliable (Mohanti, 1981). Recent publications stress the use of different methods which have to be tuned by experience until the results show no significant difference anymore. (See for instance Bauer, 1979, or Bjorklund, 1979.)

Another point which should be noted about the described estimation methods is that they are mainly developed for Life-Cycle Costing and do not support Design-to-Cost and Cost Control adequately.

4. THE SET-UP OF THE COST DATA BASE

The first step to solve the cost related management problems is the build up of a cost database containing well-defined data from previous projects. The set-up of this database should be such that it allows for the support of both cost estimation and cost control. Furthermore, the overhead for projects which have to provide data for the database should be minimal.

This leads to the following properties of the cost factors, whose data have to be gathered: (Fig. 5)

1. The cost factors have to be useful for a cost estimation and control method. This method has to be determined before the contents of the database is defined.
2. The cost factors have to be well-defined. This will prevent interpretation errors during collection and retrieval of the data.
3. The data has to be objectively measurable. For instance, a statement like "This software is very reliable" can mean almost anything, ranging from "I have heard no user complaints yet" to "The software has a MTBF of 10^9 hours".
4. The collection of cost factors has to be complete. If it appears that an important cost factor exists, whose data have not been gathered, the database can become meaningless. On the other hand, it is no problem if data has been gathered which appears lateron to have only negligible influence.
5. The cost factors have to be distinct. Relations and overlaps between the collected data will cause several problems:
   - o More data than necessary has to be gathered which will decrease the cooperation of project members during filling of the cost database.
   - o If there is a straightforward relation between two cost factors, one of the two factors will not be measured, but estimated from the other one. This will introduce erroneous data.

So for the set-up of a cost database for aerospace software development at NLR, first available methods have been studied and evaluated against the goals of the three cost management fields. This gave rise to the definition of the cost estimation and control method which is described in the next section. Furthermore, a list has been composed with about 100 cost factors which have been included in any estimation method. This list has been screened and only the measurable and non-overlapping factors have been left. (Note that a measurable factor is well-defined by its measure). After this data-collection and retrieval procedures have been defined and developed. The database has been implemented, using the IMF database management system

on the Cyber 170-855 computer at NLR.

The consideration that the database should be usable for monitoring and control software development has led to data-collecting and reporting procedures which supply the projectleader with up-to-date highly visible progress data, related to his planning. This also motivates projectleaders to supply the requested cost data.

5. PROPOSED COST ESTIMATION METHOD

Each of the cost management processes imposes its own requirements on a cost estimation and control method (Fig. 6).

Life-Cycle Costing is used already at the very beginning of a project. It is clear that at that time not many details are known about the project whose cost have to be estimated. This requires that the cost estimation method allows for global input. Of course, this will result in a rough cost estimate but that is acceptable in the early project phases.

When project development proceeds, more detailed cost factor information can be obtained. To obtain a more accurate cost estimate of the project using the cost estimation method, it is required that the method allows for detailed input of cost factors.

As already noted, Design-to-Cost implies that the scope of a new project is set according to a given budget. For that, the designer/estimator needs a way to assess the impact of design constraints and major functions. If this is known he can try to remove or adapt the most suitable function or design constraint in order to meet the given budget.

Cost control requires that the method gives cost data per project phase. Furthermore, if the requirements and/or environment of the project change, it is necessary that the planning is updated to incorporate those changes.

Given the above mentioned requirements, a cost estimation method has been developed which meets all of them (Dekker, 1981). Basically, the method is of the parametric type; analogy and decomposition have to be used, however, to estimate the size of the project. Each project is characterized by 8 cost factors, each of which is composed of a number of subfactors. see figure 7. Each cost factor can be computed from the related subfactors and the project-cost per phase will be estimated from the cost factors.

When a project is initiated, only the 8 cost factors have to be estimated, but, whenever possible, more detail can be entered by estimating single subfactors.

As most of the design constraints are simply represented by a single subfactor, the impact of them to the estimated cost can easily be assessed. The impact of major functions can be estimated via its impact on the size of the project. This impact is already known because of the use of decomposition to obtain the total size figure.

Changes during a project are reflected by changes in one or more subfactors. From the new set of subfactors it is easy to derive a new cost estimate and planning.

Consequently, the proposed combination of the three basic cost estimation methods supports all three cost management processes.

6. THE COST DATABASE

After the cost estimation method had been defined, it was rather straightforward to develop the cost database which shall support this method. At first, each of the subfactors has been defined and a measuring procedure has been devised for each of them. The definitions and measures can be found in an interim study report (Dekker, 1981). Furthermore the cost data collection method, and the related software has been developed.

The cost data is collected in a staged way. The first stage consists of a weekly form which has to be filled in by each project member. This form contains the cost related data, mainly the worked-hours per phase. Much effort has been put into the design of this form, because an ill-designed weekly form will demotivate people to fill it in.

These weekly forms are entered into the database, and by totalling this cost information the current status of the project can be derived weekly. Figure 8 shows a sample of the weekly report. (Note that only the later phases of the shown project have been monitored which explains the low percentages for the conceptual and definition phases). These weekly reports are a powerful cost control tool for the project manager, as he can detect deviations of his planning as early as possible.

The second stage of data collection consists of another type of forms. These have to be filled in by the projectleader after the completion of each project phase. They contain all subfactors which can be known after the project phase just finished. These are the data that will be used eventually to calibrate the proposed cost estimation method.

7. FUTURE DEVELOPMENTS

The use of the database for the calibration of the method is only possible if enough completed projects have been entered into it. It is therefore important that as much as possible data of software development projects in the aerospace area are entered into this database.

In the mean time the database will be used effectively by supplying reference points for the use of other cost estimation methods. For instance, the famous rule-of-thumb that a developer produces one to three statements per hour can receive a quantitative basis. Also the model of Putnam (Putnam, 1978) and the PRICE-S

model (Freiman, Park, 1979) will be evaluated using the database. In order to enrich the open literature on this point, the intermediate results will be regularly published.

8. CONCLUSIONS

The study described has resulted in a valuable cost control tool, which at the same time collects cost data from running projects. This data offers the possibility to produce a reliable cost estimation method and opens the way to better software project management.

9. REFERENCES

Bauer, T.H., 1979, "Software Cost Estimation Experience", Workshop on Quantitative Software Models for Reliability, Complexity and Cost: An Assessment of the State of the Art., IEEE-TH-67.

Bjorklund, H.A., 1979, "Application of PRICE-S to Embedded Avionics Systems", Workshop on Quantitative Software Models for Reliability, Complexity and Cost: An Assessment of the State of the Art., IEEE-TH-67.

Dekker, G.J., 1981, "Functional Requirements for a Software Cost Database", National Aerospace Laboratory NLR, NLR TR 81017 U.

Freiman, F.R. and Park, R.E., 1979, "The PRICE Software Cost Model", National Conference on Aerospace and Electronics, NAECON'79, IEEE.

Herd, J.H. et.al., 1977, "Software Cost Estimation Study: Study Results, Vol. 1. Doty Associated Inc., RADC-TR-77-220.

Mohanti, S.N., 1981, "Software Cost Estimation: Present and Future", Software-Practice and Experience, Vol. 11, pp. 103-121.

Posthuma de Boer, U., 1978, "Cost Estimation and Management Control of Software Development in Scientific Technical Projects", National Aerospace Laboratory NLR, NLR TR-78056 U.

Putnam, L.H., 1978, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem", IEEE Trans. on Softw. Eng., Vol. SE-4, Nr. 4, pp. 345-361.

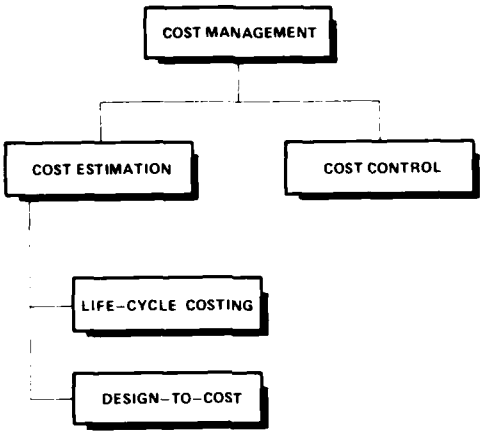


Fig. 1 Cost management fields

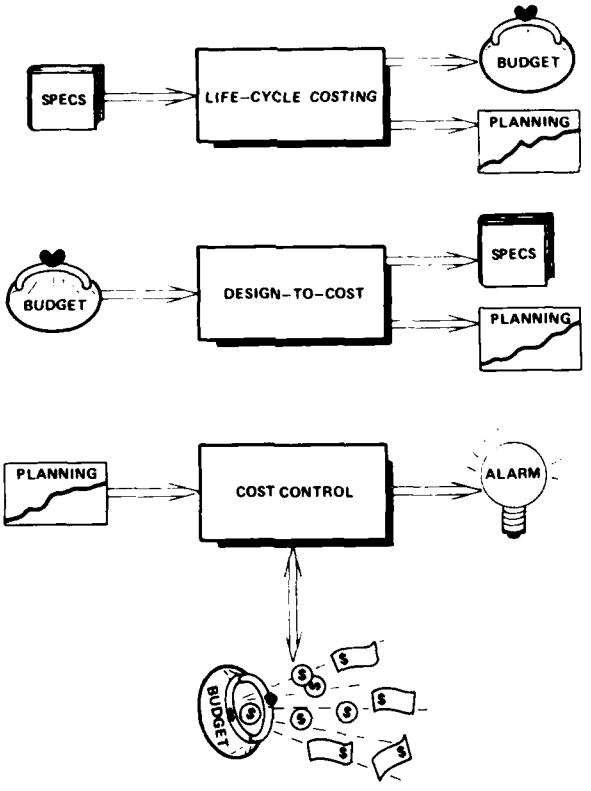


Fig. 2 Cost management processes

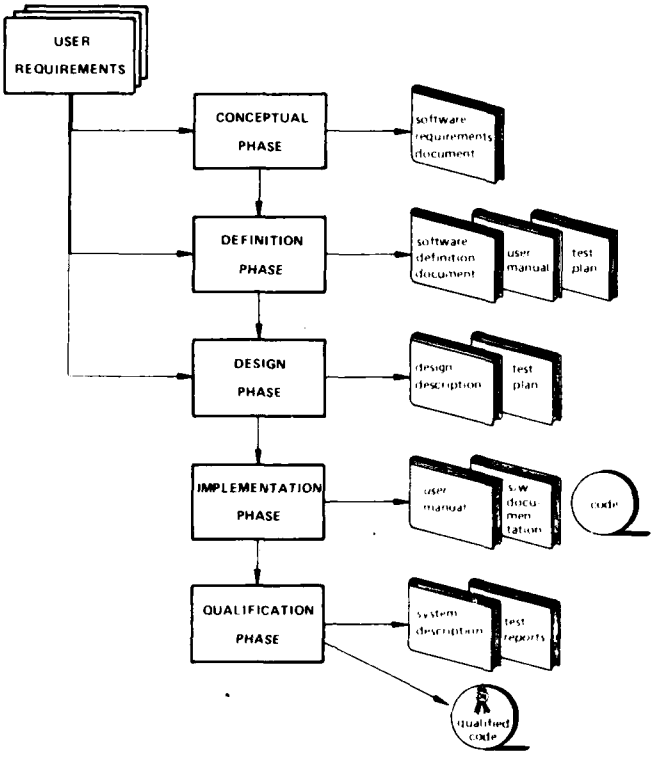


Fig. 3 Software development phases

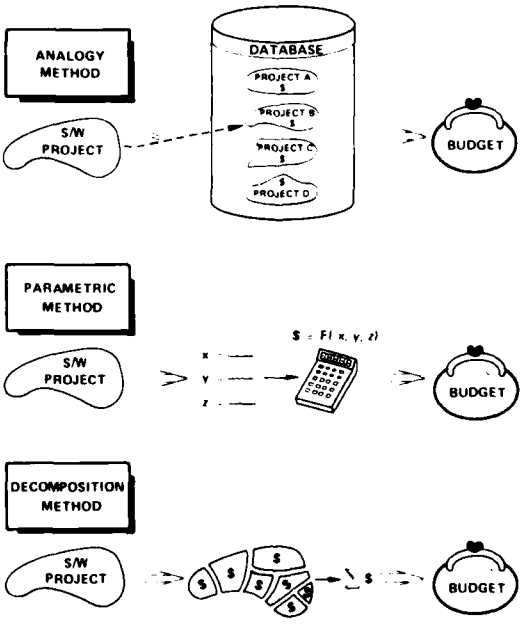


Fig. 4 Cost estimation methods

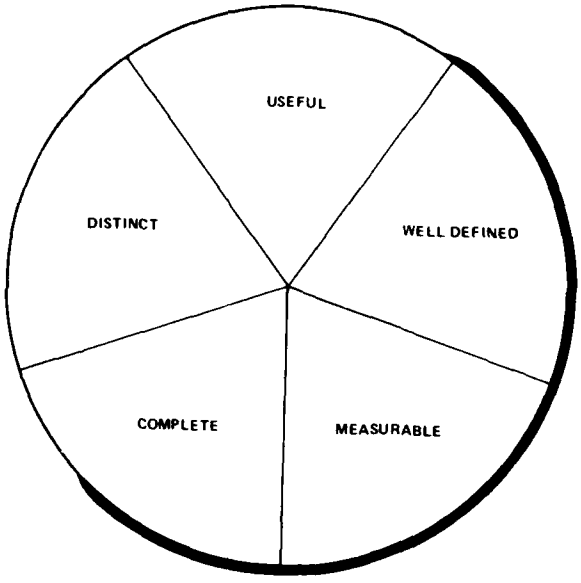


Fig. 5 Required cost factor properties

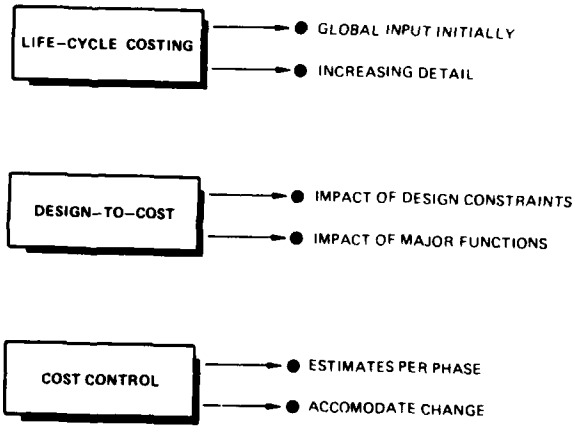


Fig. 6 Requirements for a cost estimation method

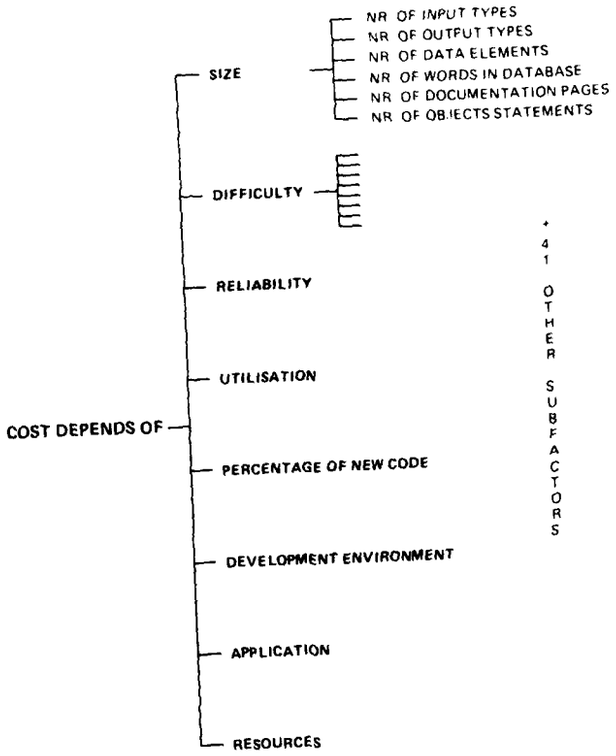


Fig. 7 Classification of cost factors

SUBPROJECT STATUS REPORT UPDATED TO YEAR: 82 WEEK: 8

PROJECT: KNMI GROUNDSTATION
SUBPROJECT: KOSMOSS APPLICATION SOFTWARE

PROJECTLEADER: INGEN-SCHENAU H A van
SUBPROJECTLEADER: POPTA R G van

2 PHASE OVERVIEW

COST DATA		CONCEPTUAL PHASE	DEFINITION PHASE	DESIGN PHASE	IMPLEMENTATION PHASE	QUALIFICATION PHASE
MANHOURS CAT I	ACTUAL	17 0	150 0	1003 6	848 5	332 9
	PLANNED	360.0	360 0	600 0	280 0	250 0
	%	4 72	41 67	167 27	303 04	133 16
MANHOURS CAT II	ACTUAL	41 5	71 5	340 2	1000 4	738 5
	PLANNED	200 0	720 0	1020 0	280.0	240 0
	%	20 75	9 93	33 35	357 29	307 71
MANHOURS CAT III	ACTUAL	0 0	0 0	0.0	0 0	0 0
	PLANNED	0 0	0 0	0 0	0 0	0 0
	%	0.00	0 00	0 00	0 00	0 00
TRAVEL COST	ACTUAL	261 0	309 0	0 0	180 0	346 0
	PLANNED	1000.0	1000 0	300 0	1000 0	1000 0
	%	26 10	30 90	0 00	18 00	34 60

Fig. 8 Sample status report

THE MILITARY USER VIEW OF SOFTWARE SUPPORT THROUGHOUT THE IN-SERVICE LIFE OF AVIONIC SYSTEMS

Wing Commander S.J. Barker RAF
Squadron Leader B. Hambling RAF

Ministry of Defence UK

SUMMARY

The view is presented that software-based military avionic systems should be considered as vehicles requiring continuous software development throughout their operating life. The reasons for software change are discussed and emphasis is given to an increasing need to adapt avionic systems to match a changing hostile environment with both speed and safety. The paper argues that more thought should be given to basic system design to facilitate both hardware and software replacement by greater modularity and reduced hardware/software dependence. Some examples of current system inadequacies are given. The need to reduce the cost of software development is emphasized and the type of software support environment as envisaged in the current ADA/APSE development is seen as a significant step towards this end.

1. INTRODUCTION

1.1 There are two distinct phases during the total life of a military weapon system separated by the point at which the system is accepted into service. Considering the entire life of a military system, the first phase is a relatively short one during which the system is designed, developed, tested and produced to meet a stated military requirement. The requirement against which it is designed will in some measure have been forward-looking and will have taken into some account the environment in which the weapon is to operate, not just on entry into service, but over the entire subsequent in-service life. During this period heavy reliance is placed upon the skill of highly specialised hardware and software design staff and is the time in which the main characteristics and constraints of the delivered product are settled. As the phase progresses the system becomes increasingly change-resistant by virtue of the design decisions that have already been taken and by virtue of the cost of implementation of such changes in extra finance, trade-offs or late delivery. The second phase is far longer, lasting as long as the system is economically maintainable and adaptable to the changing environment in which it operates. The weapon system will be extensively used, misused and often abused in a range of situations often with disregard for design limitations by a variety of people, many of whom are somewhat intolerant when a system does not adequately satisfy the need of the moment. This phase is characterised and driven by the desire for change.

1.2 The main proponents in each of these phases, that is the designer/contractor and the military user respectively, see the weapon system from different points of view. The designer looks forward to the day when he will be able to satisfy his customer's requirement and deliver a finished product. Although this product may exhibit some development potential, either through stated requirements, or through planned or fortuitous design, it is nevertheless an identifiable finished product. A customer, on the other hand, is interested not only in the system's ability to meet its specification on delivery, but also that he will be able to develop it easily, safely, economically and speedily as new operational requirements dictate. The haste with which military users find need to change a system often fills a contractor's design staff with horror, but the escalating capital cost of complex computer-based avionic systems makes it imperative that system performance is continually matched to the needs of the day over an increasing in-service life span. This can only be achieved with safety, with ease, within a tight budget and in a timely way, if systems are designed with more serious thought and emphasis given to the ongoing system development process from inception, through design and production, and into in-service life.

1.3 All that is said so far relates to military weapon systems in general. The term "weapon" is used rather loosely to include all those systems deliberately designed for direct engagement in warfare and consequently they embrace all ground and airborne real-time computer-based systems used in both offensive and defensive roles. Although many software and hardware considerations are equally applicable to all these systems, avionic systems are particularly sensitive to the traditional constraints of weight, space, flight safety and air worthiness. Therefore this paper discusses software issues that have wide-ranging applications but with special emphasis on those more critical in the avionic field.

2. THE NEED FOR CHANGE

2.1 Few people, if any, would disagree with the premise that some change to software will be needed within the lifetime of an avionic system. There could well be considerable difference of opinion, however, in predicting to what extent particular areas of software will be subject to change. It is therefore worthwhile to consider first of all, the factors that lead to change. These are fivefold: people, experience, interoperability, threat environment and technology.

(a) People

People are fallible. People are different. The people that foresee a system need, specify its requirement, advise on its development, test it, introduce it to service, and use it are all fallible and there will usually be many different people involved. Military systems are seldom bought tried-and-tested off the shelf. They are, more often than not, custom-built against a forward-looking design, using state-of-the-art technology, and consequently are subject to human fallibility both in stating the requirement and in system design.

(b) Experience

Although in principle weapon and mission tactics are foreseen when laying down a military requirement, much operating detail goes hand in hand with the detailed system design. In addition, operational usage of a system, and practical experience with it, will create the need to accommodate changes as particular operating capabilities change in their significance and importance.

(c) Interoperability

The lifetime of a modern avionic system will see the introduction of a range of interoperating systems. These will include new on-board weapons, radars, displays and data transmission systems that are part of, or communicate with, other computerised air and ground systems. The introduction of one of these systems may demand software changes in other systems to accommodate its introduction. Piecemeal implementation of software changes within an individual system may not therefore be adequate. The need to be a continuously effective weapon platform within a network of co-operating and interoperating military systems, may place very precise constraints on software development and implementation schedules.

(d) Threat Environment

Military systems are designed to operate within a hostile environment. As the threats which compose that hostile environment become more supported by, and dependent upon, computerised systems, so the possibility of dramatic change to the threat environment increases. Software changes to hostile computer systems, although developed over an extensive period, can be put into effect in a relatively short period of time. Consequently avionic systems will need the wherewithal to facilitate safe and rapid software development to cater for sudden and unforeseen changes in their hostile operating environments.

(e) Technology

The development of technology brings particular problems to high-cost, long-life systems. In the first instance, new technology often offers capabilities previously considered impracticable or costly. In the second place, with upwards of 20 years life expectancy in a system, advancing technology while not necessarily offering greater operational advantages, outdates equipment and makes it harder to support.

2.2 The above should adequately illustrate why the military user places such emphasis upon the ability to alter the software of avionic systems apparently with such indecent haste and often soon after delivery. It is not personal whim but rather pressing need that drives this requirement. The need will continue and will increase alongside the increase in computation and intelligence built into both friendly and hostile military systems. The fact that military systems need to continue to operate safely and interoperate correctly whilst permitting software changes to be implemented, imposes difficulties and constraints peculiar to their in-service life. Conversely, the increasing reliance on computer assistance in communicating interdependent military systems, brings a need for stability that in itself tends naturally to oppose changes. Whatever the difficulties, the vast investment in computer-based weapon systems will only show a worthwhile return if software is continually adapted to cater for the prevailing situation.

3. THE PRESENT POSITION

3.1 How equipped are we to fulfil this ongoing software development role? Not very well! It is not difficult to find a number of deficiencies both within the embedded software of avionic systems and within their software support environments. Within the UK the Jaguar development potential was limited from the outset by fully occupied core space and processor time: the Tornado GR1 computing power requirement was doubled even before initial development was completed; current analysis shows that the Harrier GR5 is likely to enter service with considerably less spare computing capacity than originally specified. In addition to the computing capacity restrictions apparent at entry into service, it is likely that some additional and desirable software features that arise during aircraft procurement phases are ruled out by virtue of lack of computing capacity to implement them. There are also other more subtle limits set to the maintainability of software. For example, "shoehorning" software into an inadequate hardware configuration may have particularly nasty effects, such as degradation of structure, modularity and order, and increased hardware/software dependence. In addition, monolithic or poorly modularised software causes unnecessary complications and ramifications when changes are subsequently introduced, while extensive use of relatively basic assembly languages to save core space increases the difficulty of implementing changes and increases the probability of error. That these features exist is the fault both of specification and design and it must be freely admitted that these reflect the slow evolution of software engineering awareness and practice that has occurred during the past ten years or so. In addition, the military user has made little or no attempt to measure or record the number and type of software changes proposed and achieved, the consumption of spare capacity with design change, or the operational penalties or opportunities lost for lack of software development potential and so is in a poor position to quantify and justify his cry for increased development potential.

3.2 Now although one could equally well point to examples in individual projects where there is some capacity for growth, when viewed across the field there is no consistent pattern of such capability. This is hardly surprising; requirements have not always been consistent, and those that have been consistent have been unable to withstand the conflicting pressures encountered during competitive bidding or subsequent system development. With custom-built military systems, it may not be possible either to specify the operational requirement to sufficient detail at the outset or to resist the pressures to amend that requirement during system development. Irrespective of the type of contract, when finance is short or budgets are in danger of being exceeded, those facilities that affect spare computing capacity and development potential, and which the customer cannot justify with any degree of certainty, are often sacrificed rather than incur the extra cost, the system re-engineering or the project delay. The military customer must therefore accept a share of the blame for inadequacies in the development potential of his systems; significant improvement can only be guaranteed when development potential is taken seriously enough to be insisted upon and paid for.

4. THE WAY AHEAD

Any change from the present position will rely upon a wider belief in the need to design into avionic systems from the outset, the ability to facilitate considerable ongoing system development, throughout their extensive in-service life. The user may well be forced into expending effort on finding out which areas of a system are most likely to change and in being more quantitative about those changes foreseen in the near-term. While the rate of change in the military world will make it difficult to accurately quantify long-term development, intelligent long-term estimation and accurate near-term prediction will assist in basic decisions concerning the amount of spare computing capacity required and its distribution within a system. The need to be able to adapt an avionic system effectively, economically, safely, and in a timely manner to match its changing operating environment must be recognised as a military requirement and funds allocated to those aspects of basic systems design that provide for it. The system designer for his part must make a conscious effort to eradicate those system features that inhibit future growth and so produce systems in which the desired level of flexibility can be achieved. Both the user and the designer must be realistic about the cost of flexibility and costings should examine overall life-cycle costs assuming a certain level of development activity throughout the avionic system's life. Cutting corners to provide the cheapest initial solution, or forfeiting spare capacity to meet the specified 'operational' requirements at no increase in initial cost are both expensive options in the long run and both seek vainly to avoid the harsh reality that much of the cost of developing a system is spent after delivery; short cuts taken before delivery often increase the 'cost per unit' of post-delivery development.

5. TOOLS FOR THE JOB

5.1 One must acknowledge the significance of the downward trend in hardware costs. The steady reduction in the cost of computing power has changed the balance of priorities in software design. Program efficiency is no longer the over-riding requirement for which much has had to be sacrificed in the past and which has largely perpetuated the use of low-level assembly languages. The use of Assemblers and the use of hardware-dependent and obscure programming tricks in preference to High Order Languages is no longer justifiable. If the use of a High Order Language or a structured approach to design inflates the requirement for core store and/or processor power then in most cases the effect is only a marginal increase in cost, provided that the requirements are established early enough. (Of course 'cost' here refers to initial or development cost - over the complete life cycle it is likely that the overall cost would be reduced rather than increased). Increments of computer power are cheap and may add little to space or weight requirements if they are specified before the hardware is manufactured and installed. After that any change in configuration will probably be expensive and may not even be possible. Having established that, it is not difficult to see the advantages of modularity both of hardware and software with the aim of minimising the dependence of any one system element on any other. This must include hardware/software dependencies and here two aspects are important: first of all one must minimise the extent to which software changes generate complementary hardware changes and vice versa; secondly one must reduce the extent to which software is system specific, so that at least some software will be re-usable and the application of an algorithm to a new system will not entail the unravelling of complex system-specific detail. As a consequence hardware and software modularity will be of increasing importance as will be the reduced dependence of hardware and software design upon each other.

5.2 In parallel with the downward trend in hardware costs, is the increasing cost of software development. This high cost is primarily due to the fact that it takes many skilled manhours first to produce the software and then to adequately test it in its operating or near-operating environment. When viewed against the military pressure for ongoing software development, it is essential that means are found to reduce both the overall effort and the elapsed time that it currently takes to introduce adequately tested software changes into operational avionic systems. Consequently any interest the military user has in the development and adoption of software languages, software tools or software practices should centre around the degree to which these things facilitate 'economic', 'speedy' and 'safe' software developments. The well-attested economic merits of standardisation with attendent benefits upon logistic support, training and manpower undoubtedly apply in the software field. However, software standardisation in itself, whether it be concerned with languages, tools or practices, is only one step. What the military user needs to know is how well any standards will enable software to be introduced not only economically but also with speed and in a well-proven manner, Within the UK, the adoption by the MOD of CORAL 66 as the standard preferred language has been a wise first move that is now beginning to yield identifiable benefits. Moreover, the experience of using such a standard language in a variety of systems with differing software support tools and practices has emphasized the importance of a standard high-quality software support environment. It is consequently this latter aspect of the current ADA/APSE development that has the greater potential benefit for the military user. Without doubt such an environment will be only a beginning. It is a step towards the day that has to come when we have software support tools that will enable system engineers to amend and optimise system performance and features without direct recourse to the perils of programming as it exists widely today.

6. CONCLUSION

6.1 This paper has a simple theme. The theme is that more importance needs to be attached to the design of military systems in order to satisfy the need for continuing software development throughout their in-service lives. The factors that drive this need are peculiar to military systems and some of them will be increasingly important as technology advances and automated military systems become more widespread. The ability to introduce proven software changes in timely co-ordination with developments in automated friendly systems and with speedy response to tactical developments in less friendly ones, demands improvements in both systems design and in software support environments. System design must facilitate the modular development, expansion and replacement of both hardware and software and must also reduce the dependence of hardware and software upon each other. A comprehensive software support environment is a 'must' in order to facilitate software development in an economic, safe and timely way. This calls for a greater belief in, and committment to, the need to simplify the means of performing software development in systems which themselves are increasing in conceptual ability and complexity. It cannot be effectively achieved without positive committment on the part of both the user and the system designer.

6.2 Only a passing reference has been made to High Order Language, software structure, software testing, software support tools and software practices. No mention at all has been made of Specification Languages, Design Languages or Configuration Control. This must not in any way be taken to underestimate the importance of these things. Indeed they are all essential features of the type of system design and software support that the paper advocates. Any assessment of their relative merits would not only be subjective but would moreover detract from the wider aim of establishing a raison d'etre for system design and software support within the framework of which each of these important factors has its individual and essential place.

DESIGN OF A SOFTWARE MAINTENANCE FACILITY FOR THE RAF

John Whalley and Timothy H. Scott-Wilson
British Aerospace PLC, Aircraft Group, Manchester Division,
Chester Road, Woodford, Bramhall, Stockport
Cheshire, England
SK7 1QR

SUMMARY

It has become common policy for the RAF to assume responsibility for the maintenance of software once an aircraft has been delivered to service. Considerable experience has already been gained with service software teams in support of aircraft such as Jaguar and Nimrod MR MK 1. In the case of Nimrod AEW MK 3 the RAF are to establish a software maintenance facility to support the software element of the Central Navigation System (CNS) and the Mission System Avionics (MSA.) This paper discusses the general requirements for software maintenance and describes the design of a software maintenance facility for the CNS. A brief mention is given of how such facilities may be improved in the future.

1. INTRODUCTION

On-board memory has shown significant increases over the last 15 years. For example the Navigation/Tactical System on Nimrod MR MK 1 contained 8K words of store whereas the replacement system on Nimrod MR MK 2 contains 128K words of store. Another trend has been the increasing proportion of total system cost borne by the software element of the system. Although the cost of hardware has been dropping in price, greatly increased program size and the lack of improvement in programming productivity have increased the ratio of software/hardware costs.

The term "life-cycle" is used to describe the period from conception to disposal of a system. The software life cycle can be in the order of 15 to 20 years and there is evidence (ref. 1) that the post -delivery phase of the life-cycle is responsible for 70-80% of total life-cycle costs. Because of this high post delivery cost and the difficulty shown by some design contractors in maintaining post-development software expertise it is sensible to use the approach adopted by the RAF and to use their own support facilities.

2. SOFTWARE RELIABILITY & MAINTAINABILITY

2.1 Software Maintenance Tasks

Software maintenance is required for three reasons:-

a) Changes to the software because of new operational requirements.

b) Changes to the software because of new engineering requirements.

c) Correction of software bugs i.e. inherent design errors in the software or programmer errors.

The Central Servicing Development Establishment (CSDE) of the RAF have recently issued a document (ref 2) which aims to provide guidelines for estimating software reliability and hence establishing software maintenance team requirements for Nimrod AEW Mk 3.

2.2 Software Reliability

Software does not wear out and components do not fail in the hardware sense. Software bugs occur because the program has been designed and coded in such a way that it fails to meet the system specification. They are logic errors rather than a function of physical life of a component.

Many techniques have been formulated for the measurement of software reliability (refs. 3-6 for example). Most of these are based around an exponential decay of the number of bugs in a program with increasing time. A step increase in the curve is likely to occur when a new version of a program is issued. The method suggested by CSDE (ref. 2) is based upon establishing the error discovery rate. The techniques suggested are based on the work of Shooman (ref. 6) and assume that the software error rate is proportional to the number of remaining errors although other researchers such as Littlewood (ref. 3) suggest that this may not be the case. The USAF evaluated three software error prediction models (ref. 7) over a period of four years. The results, however, proved to be inconclusive and most, if not all, of the software reliability measurement techniques suffer from the lack of empirical validation. Unfortunately a customer is unlikely to want to pay for the expensive logging and analysis of error data until the methods used to analyse the data are proved to be correct and meaningful. In the case of the CNS a limited amount of analysis is being performed but it will only be after the aircraft has been in service for some time that the validity of the analysis will be determined.

Software maintenance s facilitated by the use of a standard high-level language and a standard documentation system. The preferred standards for RAF contracts are CORAL 66 and AVP70 Spec 4 (ref. 8) respectively. The production and maintenance of documentation is in itself a mammoth task and the use of a computer based documentation system is highly recommended.

It is also necessary that a suitable fault reporting and configuration control system should be established.

BAe's experience has been that many of these elements have been developed piecemeal as a need has arisen. Efforts are now being made to consolidate all this data into a single computer database. The use of a database for project control is equally valid during both production and maintenance phases and this approach is now becoming well established (e.g. ref 9).

3. THE CENTRAL NAVIGATION SYSTEM

The Central Navigation System is based on a Marconi Avionics (Elliott) 920 ATC computer and is the heart of the overall integrated navigation system. BAe is responsible for "air vehicle" avionics including the navigation and autopilot systems. The CNS is positioned at the navigator's station (fig. 1) and consists of :-

920 ATC Computer (including 64K words of core store)
CNS Interface Unit
Navigation & Display Control Panel
Raster Scan Display Unit
Cassette Program Loading Unit

Two Ferranti FIN1012 inertial navigation platforms provide heading, position and attitude information and further navigation data is derived from a gyro magnetic compass system and an air data system. Position fixes can be obtained using TACAN, LORAN C or from position information supplied to the navigator from other sources. Navigation data is supplied to the Mission System Avionics (MSA) and steering patterns can be initiated by the CNS, the appropriate commands being transmitted to the Automatic Flight Control System (AFCS). The MSA supplies the CNS with data on targets of interest to the flight crew. The CNS can also input information to and interrogate a common database within the MSA. This database contains information of a more general nature such as airfield weather reports and can be written to by any of the operators.

The CNS Keyboard consists of three elements:-
Discrete Function and Status Indicators (DFSI)
Multi-function keyset (MFK)
Alphanumeric Keyboard (ANK)

The use of MFK has reduced the number of dedicated push buttons and indicators that would otherwise be required but has added to the complexity of the software. The unlabelled keys are used in conjunction with a series of legends or cues which appear at the bottom of the tabular display on the RSDU. The display is split into three main areas, basic navigation parameters on the top line, a selectable page of tabular data e.g. steering data) and an interactive display area containing MFK legends or a cue requesting input of alphanumeric data.

The computer is connected to the GEC display sub-system and the Marconi Avionics CNS Interface Unit by bi-directional serial data channels. The links with the Inertial Navigation Systems, TACAN and MSA are via uni-directional serial digital links of the type used on Tornado. The retention of older Nimrod MR MK 1 equipment has necessitated the inclusion of digital to analogue converters (DAC) and a single multiplexed analogue to digital converter (ADC)

3.1 CNS Software

There are two programs associated with the CNS:

The Operational Flight Program (OFP)
The Engineering Test Program (ETP)

Both programs are being developed at BAe Manchester and a flight trials version of the OFP has been operated successfully in the prototype aircraft for a period of about 2 years. The programs are written in CORAL 66 and documented to AVP70 and hence are modular in nature.

The OFP is split up into seven tasks :-

SUPERVISOR	-	Real-time control & interrupt handling.
DISPLAYS & DATA MANAGEMENT	-	Keyboard handling & formatting of tabular pages.
INPUT/OUTPUT	-	Control of data transfers and validity checking
BUILT-IN-TEST	-	Monitoring of external and internal functions
STEERING	-	Calculation of steering commands for the AFCS.
NAVIGATION	-	Calculation of position from raw sensor information and selection of appropriate navigation mode.
FIXING	-	Calculation of position correction factors from raw fixing data.

The ETP provides a full test and diagnostic capability for the CNS and also facilitates overall testing of the integrated navigation system. The ETP will be used to provide further information on a fault detected by the OFP built-in-test or by some other means. The RAF fault report form F720 will be suitably annotated with details of the software configuration and mode of operation to help determine whether or not a fault has been caused by the CNS software.

The sequence of software production is as follows:-

Code and test individual modules on the host computer (GEC 4080) using resident and cross-product software.

Integrate modules on a software development rig consisting of CNS equipment.

Integrate the software into the overall integrated navigation system on the systems integration rig.

Each of these steps involves a number of iterations. The systems integration rig consists of aircraft equipment and cabling supplemented by simulators where appropriate. It should be noted that the software production and maintenance facilties have virtually identical requirements and so a good starting point for the design of the maintenance facility is the existing production facility. Because the contractor is required to provide a technical support and validation service it is also important that the service and industrial software facilities are not too dissimilar.

4. DESIGN OPTIONS

Four options were offered to MOD:-

Option 1 - Hardware 'hot' rig with aircraft equipment based on BAe's systems integration rig.

Option 2 - Software simulation rig with a link between the simulation CNS computer and the CNS hardware.

Option 3 - Joint CNS/MSA rig driven by a general purpose simulation package - SIMBOX, based on a mainframe computer.

Option 4 - Microprocessor based sensor simulation.

These configurations were aimed at testing the software in order to determine the cause of software faults and to validate software changes. In addition it was also required that a facility for software generation should be available for editing and cross compiling programs for example. This would be a GEC 4080 computer and would act as a back-up to the main generation capability provided by the MSA software support facility.

The first two options were based around BAe's systems integration and software development rigs respectively. The systems integration rig consists of aircraft hardware laid out to correspond to the actual aircraft layout, dynamic inputs to the system being produced by a variety of hardware based simulations and an aircraft simulation computer. The software development rig consists of the CNS hardware which is either operated independently i.e. without external stimulus or via a low speed link with the GEC 4080 host computer.

Option 3 was suggested by the RAF because it utilises the simulation package SIMBOX. This software tool was developed at RSRE Malvern because of concern over the wasted effort in producing a multiplicity of simulations for different projects. It is aimed at simulating a large number of objects such as ships and aircraft in a defence environment and in the case of the MSA it would be used to provide simulated race track data.

The last option was in effect similar to option 2 except that the relatively high cost mini-computer was replaced by a series of low cost micro-computers. Each micro-computer would represent a system or group of systems.

4.1 Comparison of the Options

Option 1 carries the lowest risk because it is essentially the same as the systems integration rig at Woodford. However the rig contains a number of high cost items such as the hardware simulators. The computer used for the aircraft simulation is a Marconi-Avionics 920 ATC and can only provide a limited degree of operator interaction. In addition a GEC 4080 machine would be required as a back-up software generation facility. The advantage of using a hardware based rig is that it is likely to represent the aircraft system more faithfully than a software simulation which might suffer from inadequate modelling. However the modelling deficiency may be as a result of incorrect documentation of the hardware and in this case a hardware rig might not represent the true situation. In addition the hardware rig does not fully represent the aircraft system because it is necessary to use a series of hardware simulators. Increasingly these simulators are likely to be microprocessor based and hence contain software logic. A significant advantage of the hardware rig is that it enables visual checks of aircraft instruments to verify outputs from the computer.

Option 2 was based loosely on BAe's Software Development Rig. However the rig at Woodford only contained a low speed link between the GEC 4080 and CNS which was incapable of stimulating the CNSIFs and reproducing aircraft data rates. In order to produce the required data rate this option requires the production of a special interface unit between the two computers and additional software in the 4080 to drive the link. The systems external to the CNS are replaced by simulations within the 4080. This clearly raises the problem of testing one software package with further un-tested software. However, much of the simulation software had previously been validated in conjunction with aircraft/weapon system modelling for Nimrod and TSR 2.

Option 3 was also a software based solution but using a general purpose simulation package which could lend itself to supporting a number of target systems and hence form the basis of a multi-project software maintenance facility. This approach has been adopted by the US Navy Air Development Centre (ref. 9). However, the US facility required a substantial investment in software support tools over a number of years. Although SIMBOX is considered to be a sensible approach for war games and inter system types of simulation it is not suitable for dynamic systems simulation where the transient reponse might be of importance e.g. the interaction between the aircraft dynamics and the steering software.

Option 4 offers a modular microprocessor solution which given suitable standardisation could lead to its application to a number of projects. However, it carries a substantial development risk because of the use

of new hardware and software.

Clearly there are many other options that could have been offered. MOD finally decided on a combination of Options 1 and 2 the aim being to retain certain items of representative hardware whilst eliminating high cost items such as the hardware simulators whose total cost was greater by a factor of two than the cost of the simulation computer. The decision was also influenced by the necessity to provide a GEC 4080 computer for software generation and it was therefore more cost-effective to use the same computer to drive the software testing rig.

5. DESIGN OF THE SOFTWARE TESTING RIG

5.1 Hardware/Software Tradeoffs

With any design problem the implications of each decision must be considered in terms of a number of factors. These vary with the details of the tasks to be performed but include.

a) Cost
b) Availability within timescale
c) Technical Risk
d) Meeting the customer's requirement
e) Ability to adapt to future changes in the requirement.

In a completely hardware environment the answers to many of these points are often straightforward, but when a balance between hardware and software is involved, the benefits of each approach can be less obvious.

The requirement for the Software Maintenance Facility can be summarised as the need to provide an environment similar to that experienced on the aircraft in which realistic testing of the CNS programs can be undertaken. The diagram (fig. 2) shows the CNS as fitted in the aircraft together with the aircraft systems which supply data to it and receive data from it. Also shown is the most important part of any navigation system, the vehicle motion. It can be seen that a closed loop system is formed by the steering facility of the CNS, the AFCS affecting the aircraft position in such a way as to remove the error from the desired course.

Considering then how this environment can be translated into a ground based rig the first decision is how the aircraft motion is to be represented since in this case at least the real thing cannot be included. BAe have in the past used hardware simulators to generate the major signals required to drive development rigs. These however have not usually permitted any dynamic motions and have not been suitable for use in a closed loop situation. Whilst these facilities could be provided by a new hardware simulator, the alternative of software simulation was considered to offer several advantages. Firstly BAe have considerable experience of using computers to simulate aircraft motion in the fields of design (stability and control work) and modelling (development and analysis of an aircraft's operational capability such as the CNS steering facility). Secondly experience with the Nimrod Mk2 has shown that these modelling techniques adequately represented the aircraft and in some instances gave better results than a hardware rig. Finally aircraft models have been used with simulations of some of the other system that are part of the Nimrod AEW Mk 3 equipment which could be used by the Software Maintenance Facility.

The first of these are the Inertial Navigation Units which sense the rotation and acceleration of the aircraft to provide attitude and positional data to the CNS. Since there is no actual motion on a ground based rig a specially produced INS Simulator which has no gyros or accelerometers is available that can be used with an aircraft simulation such as that described above. These units however are extremely costly and are virtually built to order and thus have long lead times. The use of the software simulation was therefore chosen. In fact two independent simulations are to be used so that both INS systems are modelled.

The AFCS is the first system for which the aircraft hardware could be used without modification. It comprises several units making up the autopilot and the flight director system. The benefit of using the hardware was considered to be negligible since they do not contribute directly to the CNS software testing and a software simulation had already been developed.

The TACAN receiver has to be simulated by the computer if all the possible TACAN stations are to be represented. These number about 200 and also require the resolution of ambiguities where transmitters share channels. BAe did not have a software simulation of this system at the start of the project but the equations are not complicated being derived using spherical trigonometry with an allowance for the flattening of the earth.

The MSA is itself based upon a 4080 computer which performs many more functions than are required to stimulate the CNS. The Software Support Facility is required to represent the functions of the MSA that are available to the CNS to allow tesing of them without the need for the MSA to be present. However the ability to interface with an external MSA simulator is also required.

The Central Heading System (CHS) and the Air Data Computer (ADC) are both systems that, like the INS's rely upon aircraft motion and therefore were chosen to be represented by software simulations to avoid the need for special hardware systems.

The pilot's on-top button is used to provide an external fix by overflying a known position. Since this is of necessity a manual operation requiring visual reference, modification of the system was required. The method used was to respond to an on-top with the position of the aircraft simulation at the time of the request and was therefore implemented on the GEC 4080 computer.

The hardware systems that are retained are the Routine Dynamic Display (RDD) and the pilot's instruments both of which provide visual information to the operator that is useful during testing. The instruments are driven by data from both the software simulations and the CNS. The RDD is driven solely by the CNS and its

display consists of a moving light indicating the aircraft position on a chart.

In considering each system mentioned above the overiding influence upon whether the software simulation was acceptable was whether it imposed any limitation upon the use of the facility for testing the CNS. The use of software simulations has a useful side effect in that error conditions that may arise in equipment on the aircraft can be induced in the software without having to modify any hardware which allows the extensive BITE facility of the CNS to be exercised. There are of course servicing problems involved with the use of specially modified LRU's. The use of software also affords considerable flexibility both in the initial design and also for future modification to keep the rig up to date with changes on the aircraft which is to remain in service for up to 20 years.

It is worth mentioning that the cost of producing software is frequently underestimated as is the technial risk with a new program. Production of software does take time and it must be thoroughly tested to ensure correct performance, particularly with simulations when the software is emulating a real system. Major modifications later in the life of a system can be just as costly as the initial production if sufficeint provision is not made for easy comprehension of the software by the maintenance programmer.

5.2 Simulation Software

It has already been mentioned that the simulation software is drawn largely from BAe's previous experience. However few of the models were suitable without enhancement and also all required modification to fit into the real time enironment needed to stimulate the rig. The first change was to translate the models into CORAL 66 (many were originally in FORTRAN) which allowed the software to be put into a highly modular form and overcomes some of the problems associated with subsequent modifications. The Real Time control of the program was developed using the MASCOT approach but was implemented in the final system using the 4065 Nucleus which is basically a hardware implementation of a MASCOT-type kernel.

The simulation of the aircraft was developed to permit large variations of speed and height in order to allow simulation of all phases of flight. This was achieved using a combination of a small perturbation model (commonly used for aircraft simulations) for the lateral dynamics and pitch and a fuller representation of the dynamics for the in-plane velocities. The variation of aerodynamic derivatives with the major flight parameters height, speed and lift coefficient was included. The control of the flight was from the CNS or via the operator's VDU through the autopilot simulation to ensure stability. Thus manual flight is not simulated although this mode is of no significance to the CNS.

The IN simulation was developed during the Nimrod MR MK 2 project and includes a model of the errors of the system.

No errors are included in the CHS or ADC models since the CNS make no allowance for these. It was also decided that it would not be appropriate to include the effects of noise in the simulations.

5.3 Description of the rig

The diagram (fig.3) shows the final configuration proposed for the rig. It will be seen that all the systems being replaced by simulation have been taken into the 4085. It has also been necessary to introduce a special interface unit that converts the simulated data into the analogue and digital signals found on the aircraft and also presents the data generated by the CNS to the simulation software on the 4085. This unit is one of the side effects of the decision to use software simulations that affects both the cost and technical risk of the project. Its operation is also dependent upon a software interface that had to be written for the 4085. The other item that is shown but has not been mentioned up to now is the Operator's Monitor Panel (OMP) and associated peripherals. This unit provides the only means of monitoring the 920 ATC whilst it is running. It also permits the operator to examine and modify store locations either whilst the program is running or after having stopped it. Small changes to the program are patched in this way for testing and after proving would be added to the program cassette tape that can be read by the PLU. The Nimrod Software Team had suggested various improvements to the facilities of the OMP which involved the addition of a microprocessor to control it and this task was considered as a part of this project. However this would have involved high cost to the RAF and high risk to BAe so it was not undertaken.

5.4 Use of CORE to design the rig.

CORE (ref.11) stands for Controlled Requirements Expression and is a method for developing requirements and specifications for systems that was originally suggested by Systems Designers Ltd. and has been refined by BAe Warton. It is basically a formal notation and procedure for putting on paper the designer's thoughts which usually get forgotten subsequently. It also helps to identify the grey areas in the design and to locate faults in the thinking behind the conception of the new system. If one considers the cost of correcting an error in a system during its life cycle it is obvious that a little extra effort in defining the requirement at the start of a project can be very cost effective in terms of preventing expensive mistakes.

The CORE method expresses the requirement in a diagrammatic form supported by written notes. The start of the method is to define the viewpoints (fig. 4) from which the design is to be considered and to write down in tabular form all the actions performed and data involved with them. These are then drawn out as isolated threads. The next stage is to join the threads that depend upon each other into single threads that always occur as a sequence of actions. An operational view is then formed (fig. 5) which presents all the actions that occur during the lifecycle of the system. Typically a life cycle is between each start and stop of the system (i.e. switching power on to switching power off) and major events during the operation are identified. This process can then be repeated at a new level of detail but all the time feeding any corrections back to the previous level.

CORE will also indicate where the system can be broken into subsystems and for software systems, where the natural division into modules can be made. There has also been found to be a close parallel with the MASCOT activity-channel-pool notation (ref.12) which can be used to implement a CORE design in software.

5.5 Operator Interface and Control

The simulation is controlled by an operator who can monitor and control the progress via a VDU. A system of multifunction legends are used which allow the required function to be accessed via a selection tree of commands. The operator has to perform the initialisation of the simulation and also the mode of operation (i.e. automatic data generation or manual data entry). He performs tasks that would normally be under the control of the pilot, such as setting and changing height and speed or autopilot mode selection.

All the data passing between the 4085 and the CNS can be displayed and all data going to the CNS can be entered manually.

It is possible to induce many of the faults that are detected by the CNS BITE by making the data behave in the way necessary to fail the tests performed by the CNS. In some cases this can be achieved by setting a fixed value but in others certain rates of change must be exceeded. In all cases the condition must be able to persist for a variable length of time since the BITE detects spurious faults as well as regular and hard faults.

5.6 Use of the facility for other purposes

The Software Support Facility being based upon the GEC 4085 microcomputer can also be used to provide a bureau computer service although the loading imposed by the stimulation of the rig in real time precludes the simultaneous use in both roles. However it will be used to provide a reserve facility for software generation for both the MSA and CNS programs.

Because the facility can represent all the conditions encountered in flight it is useful for initial training of CNS operators and navigators. Full length flights could be performed including fixing, steering and BITE monitoring training and if the rig is used in conjunction with the MSA simulator, the whole crew could practise tactical exercises on the ground. This would improve the effectiveness of the aircraft during real sorties.

6. FUTURE IMPROVEMENTS IN SOFTWARE MAINTENANCE

As has already been stated software does not wear out in the sense that hardware does and software reliability problems are a function of poor design and inadequate requirement specification and the use of tools such as CORE will hopefully improve the situation. It is also recognised that good configuration control is also a major influence in the successful implementation of software systems.

Standardisation of host and target computers and the adoption of the standard digital bus MIL-STD-1553 should allow the production of a standard set of software development tools and the use of a general purpose software maintenance facilities.

Much software debugging is carried out using machine language and usually require the programmer to be at the console (as is the case with the 920 ATC OMP). The programmer has to mentally juggle with machine code instructions and hexadecimal, octal or binary to decimal conversion and spends a considerable amount of time formulating machine code patches to 'repair' the program. Recent microcomputer systems make this task easier by including a firmware package which allows such capabilities as the insertion of multiple breakpoints in a program, the retention of the execution results from the last 50 instructions save and the capability of referring to locations in say hexadecimal, decimal and binary. Although not yet very common another useful feature which should be utilised more in future is source or symbolic language debugging whereby the compiler or assembler symbol table is used by the computer to generate program references in source rather than in machine code. The technique of patching programs could also be eliminated by improving compilation speeds and providing a direct electrical link between host and target computers to provide rapid error correction and downloading from host to target. Again this practice is now becoming more common, particularly with microprocessors.

Although significant costs are associated with many of these improvements major savings can be made on the maintenance of future software systems provided that suitable hardware and software standards can be established so that the investment can be spread over a number of projects.

7 CONCLUSIONS

It has become common policy for the RAF to support the software of airborne computing system following delivery to service until the demise of the system, a period which can be responsible for 70-80% of life cycle costs. In order to establish the size of the maintenance team it is necessary to have some measurement of software reliability and the number of system requirement changes likely to affect the software configuration. The data available to date is largely empirical and further research is required to establish acceptable techniques.

Various options were discussed in order to satisfy the RAF requirement for a software maintenance facility for the Central Navigation System of Nimrod AEW Mk 3, the configuration finaly chosen for the software testing rig consisting of a mixture of aircraft hardware and computer based simulations of the system dynamics.

The effort required in maintenance could be reduced in future by the adoption of better software design techniques, the adoption of computer based management packages and improved software testing tools.

8. ACKNOWLEDGEMENTS

The authors wish to thank British Aerospace PLC Aircraft Group Manchester Division for their permission to publish this paper and the Ministry of Defence (Procurement Executive) for their support. The views expressed in this paper are those of the authors and should not be taken as representative of BAe or MoD opinion.

REFERENCES

1. BOHM, B "Software Engineer", IEEE Trans. Computers, col. C.25, pp 1226-1241, December 1976.

2. JONES, M.H. "In-Service Software Maintenance - Software Reliability assessment and Maintenance Team Manpower requirement assessment". CSDE/11146/5/2/RAD RAF Central Servicing Development Establishment, October 1978.

3. LITTLEWOOD, B "How to measure software reliability and how not to". IEEE Trans. Reliability, vol R-28 No. 2 pp 103-110 June 1979.

4. BOULTON PIP & KITTLER MAR "Estimating program reliability". The Computer Journal, Volum 22 Number 4 328-331.

5. MUSA D. "Validity of Execution-Time theory of software reliability" IEEE Trans. Reliability vol R-28, No. 3 pp 181-191, August 1979.

6. SHOOMAN, M "Structural models for software reliability prediction", Proc. 2nd International Conf. Software Engineering, San Fransisco, pp 266-280, October 1976.

7. SUKERT, A.M. "Empirical validation of three software error prediction models" IEEE Trans. Reliability, vol. R-28, No. 3, pp 199-205 August 1979.

8. MOD (PE) Air Technical Publications Branch. "Specification for Air Technical Publications covering software for operational real time computer based systems" AVP 70 Spec 4 2nd Issue, August 1976

9. STUEBING H. "A modern facility for software production and maintenance" AGARD-AG-258 - Guidance and Control Software, pp 3.1-3.14 May 1980.

10. DAVY, B "SIMBOX: A general purpose defence systems simulator". AGARD-CPP-268, pp14.1-14.8 January 1980.

11. WARD A.O. "An approach to the derivation and validation of requirements". AGARD-G-258 Guidance & Control Software pp1.1-1.23, 1980.

12. Mascot Suppliers Association. "The Official Handbook of MASCOT" June 1979.

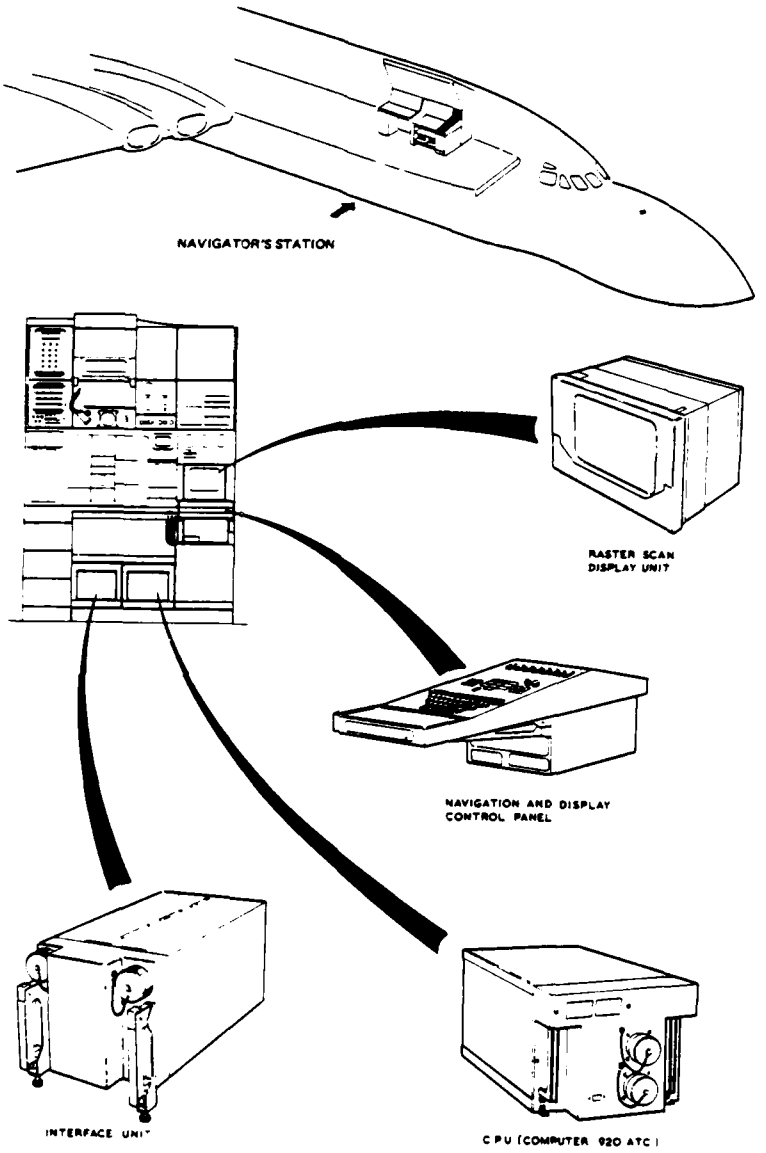


FIGURE 1
CENTRAL NAVIGATION SYSTEM

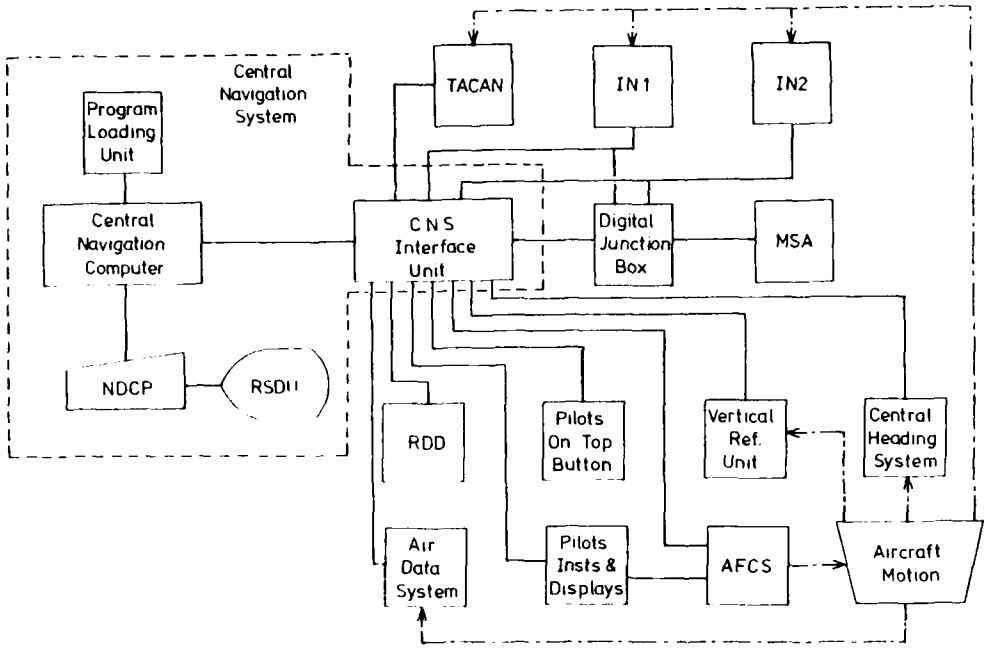


FIGURE 2
CNS AS FITTED TO AIRCRAFT

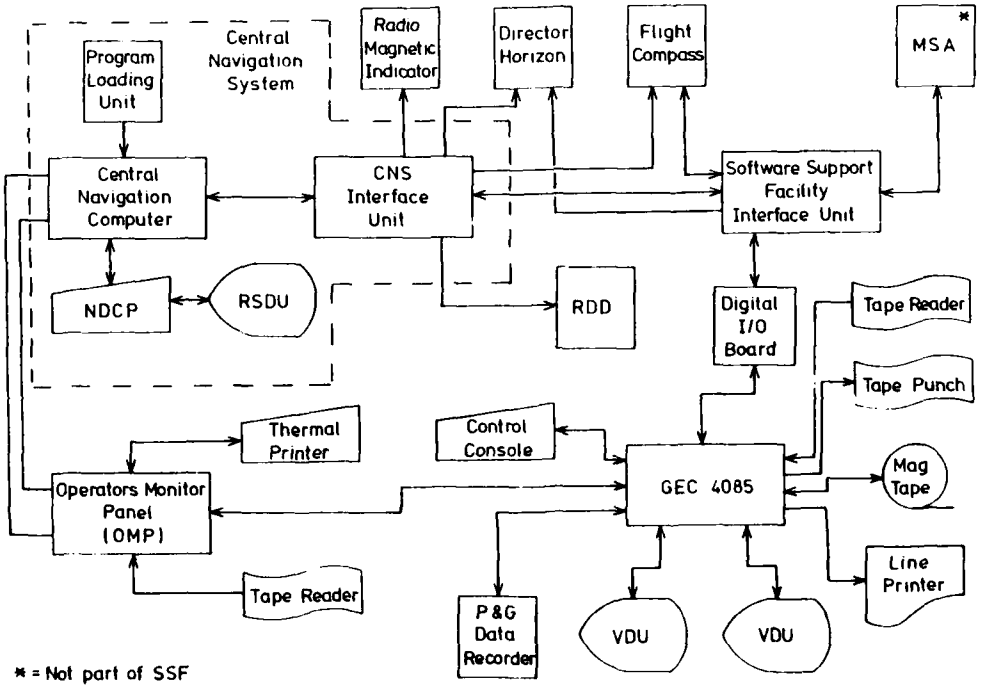


FIGURE 3
CNS SOFTWARE SUPPORT FACILITY

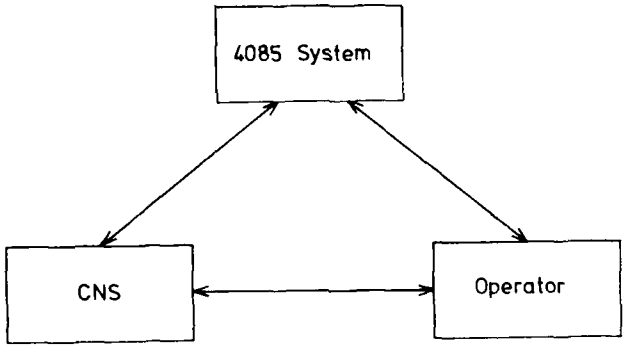


FIGURE 4
CNS SOFTWARE SUPPORT FACILITY VIEWPOINTS

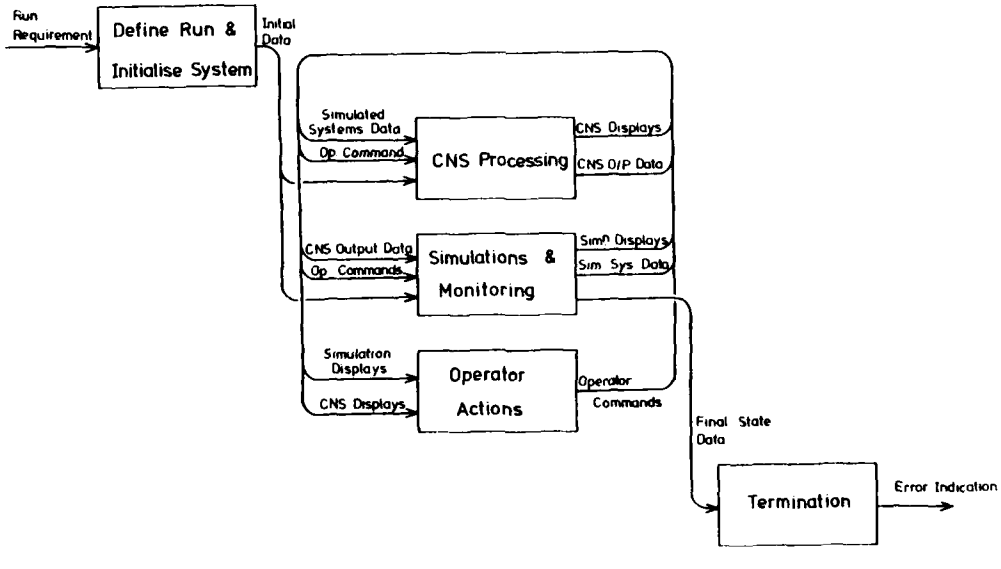


FIGURE 5
CNS SOFTWARE SUPPORT FACILITY SYSTEM OPERATIONAL DIAGRAM

A SOFTWARE ENGINEERING ENVIRONMENT (SEE) FOR WEAPON SYSTEM SOFTWARE

H.G. Stuebing
Staff Consultant
Software and Computer Directorate
U.S. Naval Air Development Center
Warminster, Pennsylvania 18974
United States of America

SUMMARY

A Software Engineering Environment (SEE) has been designed, developed, and used for the life-cycle support of weapon system software. This SEE consists of two types of facilities; software production and integration. The software production facility consists of a software system that runs on a commercial multicomputer configuration. The approach features increased management visibility of the software development process, increased programmer productivity through automation, reducing the cost-of-change during maintenance, and the use of automated regression testing to improve software quality.

These facilities have been used for seven years to develop and maintain weapon system software for several projects. This paper describes accomplishments, refinements to the code and test functions, and a general approach to extend the capabilities into the requirements and design phases. Techniques are described that simultaneously allow different methodologies, programming languages, and target computers to be implemented on the same host computer. Also discussed is the implementation of a SEE in a distributed computer network.

1. INTRODUCTION

"Software engineering" is concerned with developing software systems that satisfy the requirements of the user over the life of the system; a SEE assists the accomplishment of software engineering through sets of computer facilities, integrated software tools, and uniform engineering procedures. The term "weapon system software" inherently implies a concern with software for embedded computer systems and support over the entire life-cycle.

A generic view of the weapon system software life-cycle phases is shown in figure 1. This figure emphasizes the view that weapon system software is redeveloped several times during maintenance, the time after the initial version is delivered. The development process has overlapping phases, each with a measurable input and output. The phases overlap to show that there is an interaction between them. Within each phase, a set of activities is defined to systematically achieve the goals; the functions of management, quality assurance, and configuration management are included as activities in each phase. There are iterations horizontally, between activities of a particular phase, and vertically, between phases. During initial system development the work progresses through all phases. During maintenance the point of re-entry is determined by the scope of the intended change.

At the U.S. Naval Air Development Center (NADC), Warminster, Pennsylvania, facilities have been constructed to assist software engineering for weapon system software. Two types of facilities were built: software production and integration. The Integration Facilities were built for each project and consist of laboratory hot-mockups of the embedded computers with realistic simulation of external inputs. 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 Corporation (CDC) mainframes, a CYBER 175, CYBER 720, CYBER 760, and two CDC 6600's. This software production facility is called the Facility for Automated Software Production (FASP) and it is described in (STUEBING, H.G., 1980) and (FASP, 1979). The conceptual and architectural ideas of the FASP were strongly influenced by (BAUER, F.L., 1971) and (SOFTECH, INC., 1974). The FASP became operational in July 1975 and was the first integrated environment to be used for weapon system software and among the first integrated environments. The FASP supports the activities shown in figure 1 from Mission Requirements to Code and Test; however, only the Code and Test phase is supported by an integrated environment. In the earlier life-cycle activities the support is provided by loosely-coupled sets of tools, an important distinction discussed later.

2. ACCOMPLISHMENTS

In this section the accomplishments of the FASP are discussed; the reported period of operation is July 1975 through March 1982, nearly seven years. These accomplishments are given to set the context for the discussion that follows and to give encouragement to those who are contemplating establishing such facilities regardless of scale. In judging these accomplishments one must consider the methods that were used before the introduction of the FASP. Generally, before July 1975 the software development was done on the target computer itself. Sometimes the same target computer was used for development and integration. The state of support software and peripheral devices for these target computers was primitive compared to the state of commercial computers. Nevertheless, a large industrial-based work force had established a way of "doing business" with these facilities and produced large amounts of weapon system software. The software problems of those days are well documented.

2.1. Integrated Environment Hosted On Commercial Computers

The FASP experience has shown that the concept of an integrated environment hosted on large-scale commercial computers can be used as a true production facility. True production means that the availability and performance are adequate to produce weapon system software. The FASP was contractually specified as government-furnished equipment with guaranteed performance; the usage was almost entirely contractor personnel located at remote sites. A table of data is shown in figure 2. The data shows in summary form some of the key parameters measured with the

FASP. The amounts of software shown are larger than the software delivered to the fleet since some projects keep several versions active at any given time. Also, the data refers to on-line software and does not include the amount of archived software.

This approach used a one-time development of support software that not only eliminated such tasks from the contract but also used less in-house personnel than supporting separate facilities for each project.

2.2. Management Visibility And Control

The FASP provided the dual functions of an advanced programming system and a management information system; this allowed management visibility into the software development process at a detailed level. Data base controls allowed configuration management to be enforced from the beginning of projects, a welcome benefit. Also, the FASP was a natural way to have work standards uniformly enforced over a group of projects. The NADC was able to serve in various roles with regard to weapon system projects; these roles included System Prime, Validation and Verification, and Life-Cycle Support Activity. The FASP, with its remote terminals, could be used by contractors or other government laboratories regardless of geographical location.

An advantage of having software developed in the FASP is that there was no transition required when the project was transferred to the maintenance phase. Further, since the software was in a government operated facility with all management information, test data, and documentation in hand, the maintenance of the software could be competitively procured in a realistic way, a major change from the times of being captured to one vendor for the life-cycle.

When software was not developed in an integrated environment it was found to be poorly organized and impossible to recreate without the original developers. The development of interface control documents proved invaluable when such software had to be transferred to the FASP from another development facility. When the developed software did not follow the interface control documents, the effort to transfer the software was sometimes large. Transition efforts took a few days when the interface standards were followed and varied from one-half of a man-year to seven man-years when they were not.

2.3. Productivity

In (STUEBING, H.G., 1980) the productivity, measured in delivered source lines per man-month, was well over 400. This data was measured before significant interactive features were added to the FASP. The data was a two-fold increase over published industry data for real-time embedded computer software; the productivity data is believed to be greater with interactive features. The data before the FASP was sparse and was not consistently measured. There are local examples of turnaround time varying from one to several days with target computers being used for the development facilities. These times also do not include the courier travel time to and from the facility. With the FASP the turnaround time is measured as viewed from the remote terminal. The FASP speed improvement and the complete elimination of courier travel time reduced turnaround time by a factor of twenty.

With the FASP we have had examples of sharing large amounts of software between projects. The sharing is made easier because both projects are in the same facility and it is simply a matter of copying a data base. Further, the use of a common facility by a large group of people tends to result in better communication among the group as a natural byproduct.

2.4. Quality

The quality of the software produced by the FASP is significantly better than previously. Some of the reasons are: comprehensive unit testing with software emulators; enforcement of standards; better tools; and improved management visibility into the software development process.

Consider the following example. The FASP was being used to support a Verification and Validation effort on weapon system software that was developed by a contractor in a separate facility. An interface control document was in effect and the software was scheduled to be delivered as functional increments, each delivery adding to the previously delivered software and giving additional functional features. Figure 3 shows a plot of the delivered source lines (not counting comments). This software was quickly installed in the FASP and subjected to many unit tests, using the FASP in the regression testing mode with path coverage analysis. As new deliveries were received, additional tests were added and run with all the previous tests. Figure 4 shows the number of software errors that were recorded after each delivery; at the end of the effort 89% of all paths were tested. Figure 4 is significant for several reasons.

First, before the delivery to the FASP the contractor believed the software to be suitble for fleet use; however, the contractor had not used an integrated facility with regression testing or path coverage analysis to test the software.

Second, the errors were reported to the contractor while the development team was still in place and the resultant error data was a factor in the computation of the contract award fee. This resulted in the correction of many of the errors.

Third, all the error corrections were done before fleet delivery. Clearly, this was more cost effective than waiting until the errors were reported from the fleet.

2.5. Technology Transfer

The FASP has been used to support projects in several of the Naval System Commands for airborne, surface, and subsurface applications, a much broader scope of use than originally expected. This usage has all been on the central computer facilities at the NADC.

The entire FASP software system was successfully transferred to a major aerospace corporation. They plan to use it for all Navy software that they develop. Today, the full FASP system is only portable to other CDC CYBER computers.

A version of the FASP was rewritten using the UNIX operating system on the Digital Equipment Corporation VAX 11/780

computer system (UNIX is a trademark of Bell Laboratories). This version of the FASP supports several popular microprocessors; components that are appearing rapidly in weapon systems. It is planned to bring the UNIX version to the same level as the CYBER version, forming a product that will be easily portable to many other users.

3. "TO THE SEE"

3.1. The Software Problem

The hardware (the physical embedded computer resources of the weapon system) is generally considered less of a problem than weapon system software. The "software problem" has been covered extensively in the literature; however, different aspects of the problem have been emphasized over time.

In the early 1970's there was great concern over the quality of weapon system software. The performance, reliability, and user-friendliness were poor; most errors occurred during coding and remained undetected after testing and integration. This condition has significantly improved through better design methods and comprehensive testing; most errors are now traceable to erroneous requirements, not coding.

Today's paramount issue about software is productivity; that is, the achievement of a true economic increase in productivity over the life-cycle, (MORRISSEY, J.H., 1980) and (MUNSON, J.B., 1981). It is well known that software is a labor-intensive field and that the life-cycle costs are both high and rapidly increasing. For a given weapon system about 25% of the software life-cycle costs are for development and 75% for maintenance. Software productivity, in an economic sense, has only increased modestly when measured over the life-cycle. Most of the available labor is devoted to maintenance and the amount is rapidly rising because more and more systems are being deployed. The demand for labor with software skills has exceeded the supply, a trend expected to continue through the 1980's. As the balance of labor continues to shift to maintenance, less and less labor is available for development. Therefore, to reverse this trend in the future, it must be cheaper and faster not only to develop software but also to change it during maintenance.

A SEE consists of sets of computer facilities, integrated software tools, and procedures that support a weapon system over the life-cycle. A SEE serves as a unifying element to assist software engineering and forms a basis for attacking the software problems of quality and productivity. Testing remains the primary method for assuring software quality; a SEE can provide many automated aids to minimize the labor required for testing. Productivity is improved with a SEE not only by automating the testing of software but also by aiding all steps in the development process, including making it easier to reuse large amounts of software.

Each phase of the life-cycle employs different engineering methods. Within each phase there is usually a choice of several methods for each activity. The term "methodology" refers collectively to a selected set of engineering methods. Ideally, one methodology consisting of uniform methods would exist to cover the life-cycle. The transitions between phases would be smooth as well as the transitions between activities of a particular phase. An ideal SEE would be highly integrated, both horizontally and vertically. However, such an ideal state is some time in the future. Therefore, the issue of today is choosing a way to evolve toward the ideal SEE; it is a matter of implementing what is practical while continuing research into improved methods.

A key to success is to create a framework where new tools and techniques can be continually superimposed on existing work activities in a nondisruptive manner. Further, it is better to allow multimethodologies, to the extent possible, than to attempt to select the one "true" methodology. For example, in the Code and Test phase instead of building facilities that implemented only the Chief Programmer Team approach, it would be better to chose a way that allowed several methods, one being the Chief Programmer Team. The activities concerned with Code, Test, and Integration are better understood than the remaining phases; therefore, they offer a natural starting point. There is considerably more variability to the methods and techniques currently available in the requirements and design areas; thus, a loosely coupled collection of tools is more appropriate for those phases.

3.2. SEE Versus Programming Support Environment (PSE)

Several terms have appeared in the literature that are similar to "SEE." They are: programming environment, programming support environment, and software environment. These terms have been generally used to describe the Code and Test activities, although frequently mention is made to the requirements and design phases. Further, these terms have usually been restricted to the software concerns of a system and not the system as a whole. Therefore, the distinction is that the term "SEE" is more general and includes the above terms. A SEE refers to the support over the life-cycle including aspects other than purely software.

Of course, the term "environment" itself can be somewhat confusing in this context. This term is so general it is difficult to determine its limits in some texts. In this paper the term refers to the "work environment" for the phases of the weapon system life-cycle. The emphasis is on the facilities that support the work of each phase and the interface between the engineer and the computer. It is recognized that organizational and social factors are an important part of the work environment. These factors must be considered in the design of any computer-based support system but they are not the main points discussed in this paper. An important point is to recognize that the engineer's detailed view and use of the support facilities is different depending on the phase of the life-cycle. The needs are different between coding, unit testing, and integrating the weapon system software with the embedded computer. Likewise, the detailed view of the software is different between "development" and maintenance. The term "meta-environment" has been used to describe the aspects of environments that depend on the user's view of the system and the organizational and social setting (ELZER, P.F., 1979).

In industrial engineering there has been considerable work on facilities, both in concept and implementation. There are many valuable observations in this field that can be applied in part to a SEE. However, there are also some important limitations. For example, in industrial engineering the production facilities are oriented to rapid and automated replication of physical devices, devices that have been previously designed. Thus, there is an area called Computer-Aided Manufacturing (CAM) where the computer has been applied to the task of automating the production of physical devices. Separately, there is an area called Computer-Aided Design (CAD) where the computer has been applied to assisting the designer, making, for example, integrated circuit layouts. In the software field the work of coding and testing bears a similarity to production in the industrial engineering sense. This was the main theme in the develop-

ment of the FASP. Now as we attempt to extend these facilities into the requirements and design phases it is important to note that the analogy must be to "CAD" and not to "CAM." In these phases the facilities must support both the cognitive processes of the designer and the more clerical aspects of recording the results of the processes. Therefore, the theme of the SEE in the early phases is to assist the engineer during the cognitive processes and to automate the clerical aspects of recording information and generating documentation.

3.3. An Integrated System

The term "integrated system" is also frequently used in the literature. Now, the dictionary definition of "integrate" is clear: "to make whole or complete by adding or bringing together parts, to put or bring parts together into a whole; unify." Thus in creating an integrated system, the designer would do tradeoffs between the parts to achieve a unified whole. A SEE is referred to as an integrated system; it appears to the user as a unified whole that assists the accomplishment of software engineering. The users include both engineers and managers; the work activities vary over the life-cycle and the user interface and capabilities vary accordingly. Conceptually, the SEE may be considered as a single entity that presents to the user different views and capabilities according to the phase of the life-cycle. The implementation is most likely to be several computer-based facilities that have similar user interfaces but different specific capabilities depending on the phase of the life-cycle.

The two types of users, engineers and managers, means that their individual needs must be tradedoff such that the final system represents a unified whole. The software engineers need advanced programming capabilities for the Code and Test phase and the managers need relevant, consistent information, and a means to control the cost and schedule of the effort. The software engineers need compilers, linkers, system generators, and simulators to accomplish their work; these components are called "tools." The managers need a means of identifying the end-items that are to be built, a way of monitoring progress, and a way of insuring that the agreed-on procedures are being followed. The data base concept is a natural way of collecting such management information; it also provides a way of meeting many functions of the engineer.

Clearly, if an engineer were given just a tool set, the work could be completed. If the tools were compatible with one another, forming an integrated set, the work would be easier to accomplish. However, such a tool set usually relies on the host operating system and the engineer is thus free to create any set of files that is felt to be appropriate. If many engineers are working on the same project, the chances of them all retaining the same type of information in their files is small. In such a case the manager has no easy way to determine what end-items are being produced or what progress has been made. Of course, this hypothetical team could agree to follow a set of standards; but changes in personnel, deadlines, design changes, etc., would quickly destroy the good intentions. Also, the end-item productivity of this team is lower because experience has shown that a large part of their energy is diverted to handling the file system and writing support programs.

The computer is the natural and convenient place to integrate the needs of the engineer and manager and provide an integrated system to accomplish the work. A SEE is an integrated system in the above sense; it is much more than a tool set, it is a unified system that meets the needs of both engineers and managers.

4. CODE AND TEST

In this section the characteristics of an integrated system to support the Code and Test phase are briefly described (STUEBING, H.G., 1982) contains a more detailed description). The system that is described is not specifically the FASP, but one that has been generalized and refined based on seven years of operational experience with the FASP. During the seven-year period, three major evolutions took place along with extensive feedback from the users.

4.1. A Dual System

The system should provide the dual functions of an advanced programming system and a management information system. The needs of the manager must set the top-level framework of the system. This requires a selection of the engineering methods and procedures and deciding what information should be saved; it implies choosing a method or allowing only certain methods to be supported by the system. The methods must have a sound engineering basis and fit the organization's business methods. With the FASP it was a conscious decision to support several methods with the same system. The FASP facility is owned and operated as a government facility and used by weapon system contractors to develop and maintain software. Each contractor had different methods and procedures for doing business, yet each was able to effectively use the FASP. Some contractors have used the Chief Programmer Team approach, others a different team approach. All use some form of structured programming, although the details are different. Thus as a government facility, it was important to impose only reasonable constraints on the contractor and allow for the different ways of doing business.

An important concept in software engineering is incremental development. The idea is to first complete the software design and then to build the software in stages, or increments, such that each successive increment adds a new functional feature. This approach breaks the work into smaller pieces that are easier to manage. Thus, it is easier to judge progress on the total project and it has the additional benefit of allowing users to gain some early experience with the software system. Incremental development has proven to be valuable on large-scale software projects and should be supported by the SEE.

An important management need is the enforcement of configuration management principles. Configuration management principles consist of identification, control, status accounting, and the establishment of baselines. Decisions must be made regarding what software elements will be subject to configuration management. This involves deciding what is the smallest unit of software that will be configuration managed. Is it a "line" of code? Is it a "module?" Is it the basic compilation unit of the compiler? Is there more than one language to be supported and is the definition of a compilation unit the same for both? These decisions have a significant impact on the final system and must be made at the outset.

Along with structured programming came the idea of include segments. Include segments are fragments of code that are used in many modules without change. The programmer identifies these segments by name and places them

in the data base; in the source code of a module the segment is referenced by name. The system automatically locates the segment and "includes" it into the source stream before compilation. Since these segments are fragments of code, they cannot be separately compiled without errors; however, they are useful to the programmers. In the FASP during a typical month the data bases contained 48,000 modules and 28,000 include segments, showing the wide acceptance and use of include segments. Therefore, although include segments add to the configuration management burden, they are recommended for a SEE.Another related area is access control. Management level decisions are needed regarding what aspects of the system are subject to access control. In the FASP there are three dimensions of control; control over access to the software end products, control over software tools, and control over access to the computer system itself. In the latter case this implies cost control over the use of the computer system.

4.2 The Data Base

The data base is the most critical component of the SEE since it serves as the unifying element for all other components. The data base contains not only the weapon system software but also related technical and management information that contains the genesis and status of the total effort. Furthermore, the data base has a significant influence on the performance of the SEE, an important consideration in obtaining a true production environment.

Here the term data base refers to a fixed number of libraries that are encapsulated and managed as a whole rather than distinct parts. The software for a particular weapon system is contained in several "data bases." Each data base contains the following libraries:

- The source library, containing either the source code for modules or the source code for include segments, or both;
- The object library, containing the object code corresponding to the source 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; and
- The documentation library, containing all documentatiofn about the weapon system software.

The encapsulated data base is the basic unit that the SEE deals with. Several libraries are included because the relationship between those libraries must be strictly enforced. Thus, source and object code must have a one-to-one correspondence with no exceptions. A consequence of this relationship is that if compilation errors occur the data base (source and object libraries) will not be updated! Likewise, test data and test results are synchrcnized. Most important, at any time in the development schedule the management data is consistent with the rest of the data base; thus, managers always have access to accurate information.

An important feature of the SEE for large-scale projects is the automatic recompilation of dependent modules when certain software is modified; for example, if an Include segment is modified then all modules that use that segment will be automatically recompiled.

Commands are available that allow software to be shared between the data bases, allowing the total effort to be divided among several data bases and teams. Similarly, commands allow data bases to be divided into smaller ones or combined into larger ones. Other commands allow the data bases to be copied.

The integrity of the data base must be assured; therefore, during interaction with the SEE the system automatically creates a backup copy of the data base permitting instantaneous fall-back to the previous version. Additionally, commands allow archive copies to be made on magnetic tape for off-line storage. This level of protection is over and above that offered by the host operating sysem.

4.3 Procedures, Tools, Commands, And Processing

The use of the SEE involves sequences of tool and data base interactions. To simplify the use of the system a set of procedures is defined that are invoked by 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.

The software tools are programs that do certain functions for the software engineer. Examples are: editors, translators (compilers and assemblers), system generators, test analyzers, software emulators (target computer instruction level simulators), data extractors, report generators, and documentation aids. A tool or set of tools are automatically invoked as part of the execution of a procedure. Some tools are visible to the user, such as the editor, and require communication in a language unique to the tool. Other tools are invisible to the user, such as the librarian, and are automatically invoked when certain actions take place with the data base. In the latter case input and output data may be processed by other programs but all such actions are hidden from the user.

User commands cause procedures to be invoked. A command is a procedure name followed by parameter values. These values give the user flexibility in directing the procedure to accomplish the specific desired function. All commands are validated before being executed. The commands can be grouped into two categories, Immediate and Queued, depending on whether the data base is modified or not. In batch mode, there is no distinction and all validated commands are executed in the order received. In interactive mode an "Immediate" command is executed at once; all others are placed on a command queue. Once activated, the system executes the queued commands.

As an example of the power of commands, consider the FASP command Modify Software (MODSW). This command is used to create or modify software in the data base. In the FASP on the CDC CYBER computers, MODSW causes

315 job-contol-language commands to be executed; in the FASP on the VAX computer it causes 262 UNIX shell-script commands to be executed. These operating system level commands are all hidden from the user.

A general set of procedures has been developed and is described in (STUEBING, H.G.,1982). Figure 5 shows a list of these general procedures. The procedures are divided into functional groups and are described by process flow diagrams. These diagrams use a structured English description of the control flow for a procedure and a data flow diagram showing the process performed, the tools used, the data base contents used and produced, and other information required by the procedure. When a procedure is performed a certain amount of standard processing is done before and after the main processing for that procedure. The standard processing for each procedure is shown in figure 6 using structured English. Two process flow diagrams are shown in figures 7 and 8.

4.4 Testing

Testing remains the primary method for determining the quality of software. The SEE should support four distinct types of testing during the code and test phase. 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; and
- Trial testing that provides for testing proposed software changes without modifying the data base.

Progression testing is used during the development of new software or modifications to existing software. This form of debugging is frequently an intense creative process best performed interactively. It usually will involve interactive use of the software emulator to make experimental changes to initial conditions, data or instructions, and immediate rerunning of the test.

Regression testing is used once proper operation is achieved. It insures that the software does not deteriorate (regress) due to subsequent progression changes. Two forms of regression testing are used, explicit and automatic. In the explicit form the user specifies the tests that are to be performed. In the automatic form tests are automatically run whenever certain modules are modified. Test data, test results, and test directives are accumulated in the data base during the life of a module; also, an index is kept that relates tests to modules. A change to the module triggers the automatic running of all associated tests and a comparison of all results. The user is able to identify those portions of the test results that are important.

Automated test analysis is provided to check the quality of the tests themselves. In this form of testing a tool called Automated Test Analysis (ATA) scans the source code and inserts software probes at program decision points. This allows the decision-to-decision paths to be identified. When the instrumented code is run on the software emulator with the test input data, the system reports how many times each path was executed, flagging those not executed. Thus the percentage of total paths tested is available along with indications of "dead code" and code paths most frequently executed. This data allows the user to devise changes to existing tests or to develop more effective tests. The data on the most frequently executed paths is valuable when optimizing the speed of the program.

Trial testing consists of syntactic and semantic checks before the software is entered into the data base. This type of testing is used when changes are made to large existing bodies of software. In such cases there may arise uncertainty about the interactions between changes to the software and to tests. Also, uncertainties about the optimum changes that could be made may require that several different changes be tried before deciding on the best.

4.5. Interactive/Batch

In many ways the differences between interactive computer jobs and batch have disappeared; however, there are some fundamental differences that are important to the operation of a SEE. A batch job consists of a stream of user commands with all parameters and input data previously determined; an interactive job must have these items supplied on-line. Since the system is to be user-friendly, the interactive job must prompt the user for such items and provide some helpful information when incorrect data has been input. Thus, the SEE must distinguish between the two types of operation and provide some extra software for interactive usage.

In a SEE for weapon system software it is unlikely that all the tools will be interactive, especially the compilers. Therefore, it is a matter of judgement to determine what functions are best performed interactively; all functions should operate in the batch mode. There are three areas that should allow both interactive and batch operation: they are editing, debugging with the software emulator, and generating management reports.

With the editor there is a clear benefit to the user to be able to quickly inspect and change the software. Full screen editors appear to offer the best advantages. When debugging, particularly during progression testing, there is also a benefit to the user. Here errors tend to be discovered more frequently and once observed the remaining parts of the tests can be terminated, saving computer time. Management reports, especially the smaller ones, tend to be the most useful when they can be quickly and easily obtained by the manager whenever desired.

Just as interactive mode is best for progressive testing, batch mode is best for regression testing. Here the total running time increases as the project software grows, a case best left for overnight turnaround when computers are lightly loaded and costs are frequently reduced. For example, when the FASP has been used for maintenance of large bodies of software the ratio of interactive-to-batch commands is about 3 to 1 on the average; however, in times of intense regression testing the ratio becomes 1 to 2. If one considers tool invocation during the same period then the ratio of interactive-to-batch is about 1 to 3 on the average; during intense regression testing the ratio becomes 1 to 12.

4.6 Multilanguages and Multitarget Computers

The design of a SEE is greatly simplified if there is only one programming language to be supported for a single target computer. For weapon systems this is rarely the case. Figure 9 shows the matrix of languages and target computers in the FASP. The difficulties begin with the languages themselves. A SEE is dependent on the programming language, a point not generally understood. One problem is the definition of a module is different in all the languages. Also, there are different dependencies on the data base between the languages. For example, if one language has structured programming constructs with an include segment feature built into the compiler, and another language does not, then clearly there is a significant difference in the way that a SEE would support each language. In the latter case it may be desirable to provide the capabilities by way of preprocessors; however, the way the SEE supported each language would still be different.

Other problems arise owing to structural differences in the languages. For example, the versions of the CMS-2 language have an order dependency on the appearance of declarative statements and executable statements. The declarative statements are dispersed throughout the program in blocks followed by blocks of executable code with an order dependency on the referencing of data. In this night-marish state any change normally would mean that the entire program would have to be recompiled. However, in the FASP a special modular compilation feature has been added such that the system keeps track of the dependencies and only the appropriate blocks are recompiled. This example might appear extreme but it is characteristic of the difficulties that can arise.

With weapon system software there is likely to be several high-order languages and several assembly languages that must be supported. It is, perhaps, best to present the SEE to the user as an integrated string of tools that apply to a language and target computer. This would correspond to the vertical columns of figure 9. During the log-on process the user identifies the desired string, actions remain with that string because only those tools have meaning with one another. Internally, the system may use many common tools such as an editor or librarian; however, to the user the system appears as a unified set of tools.

The consequence of having strings of tools is that there are in effect several different SEE's. In the FASP there are four such systems concurrently operating in the host computers at any given time. From a maintenance standpoint, the FASP is maintained in a FORTRAN FASP. About 75% of the code is common across the four, the remaining unique to each language dependent environment. The language unique portions are maintained separately from the common portion and combined when a new version is desired.

4.7. Management

The success of a SEE depends on the degree that management is satisfied. Although a SEE brings many advanced tools to the user and makes the job of producing or maintaining software easier, it also constrains the user to work in a somewhat rigid framework, a point the user is sometimes quick to make. However, the benefits in productivity, improved quality, and stability over the life-cycle are great compared to any perceived loss of freedom by the user.

The degree to which management is satisfied depends to a great extent on the amount of involvement by management, the degree to which the system is understood by management, and how smoothly the SEE fits into the current methods of doing business. As with any management information system, the SEE requires that management become more involved with the operation at a deeper level than previously. However, once this commitment is made the gains are great.

The management view of the SEE is through the reports; therefore, it is desirable to generate clear, concise reports in terms that managers can understand. Reports that measure work progress and expenditures against planned profiles are interesting to management. For example, to report that the effort is on schedule and within funding regarding the number of modules, lines of code, storage size, and target computer execution time is obviously valuable to managers.

There is no general agreement across the industry on what precise software measures should be made; therefore, each organization must establish such measures and slowly refine them based on experience. It is important to allow a high degree of flexibility for SEE management reports.

Software complexity measures have been somewhat disappointing as absolute measures of software quality (PARISEAU, R.J., 1979). However, some are useful as relative measures and can be used for management control purposes. Generally, care must be taken in the selection of such complexity measures.

A promising area appears to be "earned values" reports and other related measures. These reports can be easily established in a SEE and have the benefit of being based on impersonal data directly from the software development or maintenance environment. Of course, considerable experience is needed to select the particular "value" that is earned; however, there is a reasonable expectation that this can be accomplished.

5. THE EXTENSION TO REQUIREMENTS AND DESIGN PHASES

It has been stated that the SEE should support a weapon system over the entire life-cycle as shown in figure 1. It is intentional that the term "system life-cycle" has been used rather than "software life-cycle." Today, software is so important that it must be taken into consideration at the system level. Here the term "requirements" is used is used somewhat loosely to cover the phases in figure 1 from Mission Requirements to Software Requirements; perhaps, the terms system requirements and system design are more accurate.

The requirements phase begins with high-level statements about the mission of the weapon system. During a subphase called concept formulation a set of requirements is evolved that begins to express the requirements in technical terms. The activities at this point are not highly structured. The system designers used high-level tools such as analytic simulations and the methods of Operations Research to do tradeoff studies and to verify the conceptual design.

Once the system requirements are expressed in technical terms the system architecture must be determined in

in detail. The critical issue is the allocation of the system functions to hardware or software implementation. The system designers need tools to assist the tradeoff analysis. This activity is probably the most difficult of the entire process since the final system's cost and performance are largely determined by these allocations. Once the allocations are made, any changes become not only increasingly difficult but also increasingly expensive as the system moves toward operational deployment.

When the hardware and software allocations are completed the system development splits into two paths; the resultant hardware and software efforts come together at system integration time. The software requirements should be expressed in a formal requirements language so automated tools can analyze them for completeness and consistency, the two major sources of errors. At the end of the phase the requirements should exist in a computer data base so all formal documentation can be automatically generated.

The software design process begins with the formal software requirements and results in a specific software design. This design is a specification for the code. During this process the designers synthesize a software system that satisfies the requirements. This involves considering several different designs and evaluating them according to performance, cost, and ease of change. A major output of the design process is information that will guide the unit and system level testing of the software.

There has been considerable work done in both the requirements and design areas; however, a uniform and consistent set of methods has yet to be developed that covers the entire proess. To date, somewhat singular efforts have been pursued that usually focus on one small step. For example, there are several software design methods that have been developed. The result is that the individual methods do not fit smoothly together, particularly at the boundaries between phases. There are two primary reasons for these problems.

First, the efforts to date have attempted to address the software issues, ignoring the distinction between software requirements and design and system requirements and design. Thus, system design remains a hardware oriented process and there is a considerably better interface between system design and hardware design.

Second, there has been a lack of thorough understanding of the requirements and design process. Recent work (LEFKOVITZ, D., 1982) suggests that if one forms a model of the work that identifies the cognitive processes that are used, then virtually all present methods have serious omissions. Further work utilizing this concept would seem to have good potential.

From a practical standpoint it is best to view the present state as a time of change. There are certainly tools and methods that can be profitably applied to the requirements and design phases; however, they do not fit well together and it is likely that new ideas and refinements will continue to emerge. It would be ideal to have methodological strings of tools to apply to the requirements and design phases; tools that assisted the engineers with the cognitive processes and the automated the recording of relevant information and generation of documentation.

A recommended approach to a SEE for requirements and design is to start with a highly integrated environment for Code and Test as previously described. Next, simplify figure 1 to reflect just the software concerns, as in figure 10. New weapon system developments would start at the top and progress through all the phases; each phase would have a support environment as shown in figure 11. It is understood that in the requirements and design phases that the tools and methods would be loosely coupled, although relationships between the phases can be determined and recorded in data bases for tracing purposes. The tools and methods are chosen off the shelf and then force fit together; several methodological strings should be implemented to gain experience with each. This approach is judged to be the most practical in the short term. As new, better integrated methods are developed they can be superimposed on this structure. The same approach can be used to extend the capabilities to the system design phases.

6. INTEGRATION FACILITIES

Integration facilities consist of a hot mockup of the weapon system computers with realistic simulation of external inputs. These facilities are used for hardware-software integration at the system level, evaluation of man-machine interfaces, and evaluation of hardware engineering change proposals. Typically, they form the hardware configuration baseline for the computer and associated subsystems. The simulation of realistic inputs allows the total system to be tested in a laboratory where sophisticated instrumentation can monitor the tests. This minimizes costly flight or shipboard testing.

Originally the integration facilities used special equipment or groups of minicomputers to simulate the external inputs; the capabilities of the test engineer were limited. Today, these facilities can take advantage of commercial computers to create an integrated test environment that both speeds the testing and takes it to greater depth. Modern integration facilities are full integrated environments and are electronically linked to the software development and maintenance computers for rapid loading of the mission software. Interactive capabilities allow symbolic debugging to be done on the target computer; extensive capabilities for storing test inputs and saving test outputs are now available.

Experience has shown that it is better to use separate computers to run the integration facilities than to attempt to use the host computer of the software production facility. This is because the Central Processor Unit (CPU) and Input/Output (I/O) utilization can be high in the integration facility computer during intense periods of real-time debugging. Further, the target computer and its subsystems frequently require extensive hardware checkout, particularly when new hardware is being developed.

7. SEE ARCHITECTURE

The goal of the SEE is to support the weapon system over the entire life-cycle. There are, of course, many ways to implement such facilities. The approach taken at the NADC was to coalesce the functions of figure 11 into two facilities as shown in figure 12. At the NADC there are large central facilities capable of supporting these activities and several integration facilities distributed throughout the Center. Therefore, one approach is to form clusters as shown in figure 13 and to interconnect the software production facilities by communications networks. Similarly, the software production facility could support just one integration facility with less capable host computers. It is important that the

production facilities be interconnected regardless of size because this communications capability will ultimately permit software sharing to take place between weapon systems projects.

Alternately, the functions of figure 11 could be allocated to separate host computers that are interconnected. The choice may be dictated by the scale of the available host computers, a judgement that may vary depending on the expected workload. However, a word of caution; the software tools of today do not efficiently use computer resources, thus, it is easy to underestimate the size of the host computers. Software emulators used to unit testing take a large amount of computer resources, for example.

An emerging factor in SEE architecture is the availability of microprocessors and the expectation that networking is close at hand. An excellent example of a workstation for a software engineer is (WIRTH,N.,1981). With this technology the problem becomes how to distribute the functions to retain the dual aspects of an advanced programming system and management information system. On the one hand powerful microprocessors appear to have the power for editing, compilation, document generation, etc., but will they be capable of efficiently executing software emulators of target computers? Further, how will configuration management be enforced and how will consistent management reports be generated in such a network? A large-scale computer may still be needed to collect the software for configuration management and other management reports, as well as for executing unit testing efficiently. These problems appear to be solvable and the direction toward microprocessors seems the way of the future.

8. REFERENCES

BAUER, F.L., 1971, "Software Engineering," Proceedings of the IFIPS Congress, pp I-267, I-274.

ELIZER, P. F., May 1979, "Some Observations Concerning Existing Software Environments," DORNIER Systems, GmbH, Postfach 1360, D-7990 Friedrichshafen, Germany, Defense Advanced Research Projects Agency.

"FASP Management Summary," April 1979, U.S. Naval Air Development Center.

"FASP Software Production and Maintenance Methodology," July 1979, U.S. Naval Air Development Center.

"FASP Handbook," December 1979, U.S. Naval Air Development Center.

LEFKOVITZ, D., 1982, "The Applicability of Software Development Methodologies to Naval Embedded Computer Systems," University of Pennsylvania, Contract N62269-81-C-0455.

MORRISSEY, J.H. and WU, L.S. Y., 1980, "Software Engineering...An Economic Perspective," Proceedings 4th International Conference on Software Engineering, pp 412-422, Munich, Germany.

MUNSON, J.B. and YEH, R.T., March 1981, Report by the IEEE Software Productivity Workshop, San Diego, California.

PARISEAU, R.J., 1979, "A Screening Criterion for Delivered Source in Military Software," Report No. NADC-79163-50, U.S. Naval Air Development Center.

SOFTECH, INC., March 1979, "Support Software Planning Study," Contract N62269-74-C-0269, U.S. Naval Air Development Center.

STUEBING, H.G., "A Modern Facility for Software Production and Maintenance," AGARDograph No. 258 Guidance and Control Software, May 1980, pp 3-1, 3-14, Military Electronics Defense Expo '80 Conference Proceedings, Wiesbaen, Germany, October 1980, pp 828-845, and Proceedings of the IEEE COMPSAC '80, October 1980, pp 407-418.

STUEBING, H.G., August 1982, "A Software Engineering Environment (SEE) for Weapon System Software - Functional Description for the Code and Test Phase," Report No. NADC-82183-50, U.S. Naval Air Development Center.

WIRTH, N., March 1981, "Lilith: A Personal Computer for the Software Engineer," Proceedings of the 5th International Conference on Software Engineering, pp 2-15, San Diego, California.

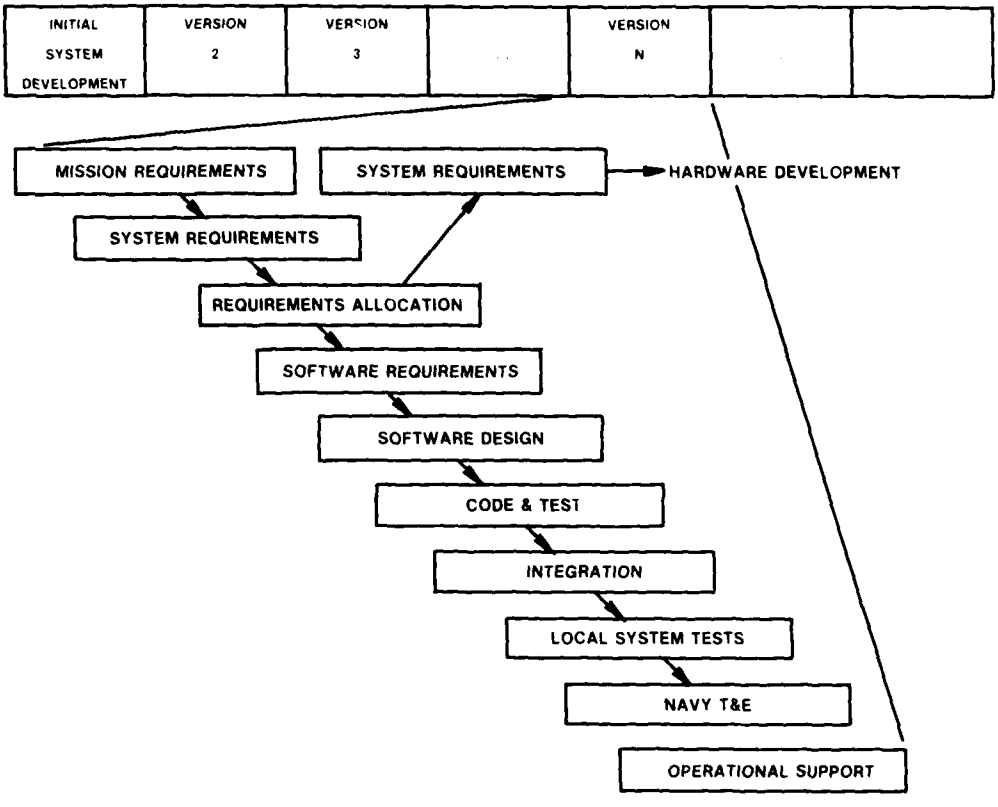


Figure 1. Generic System Development Process

FISCAL YEAR	PROJECTS	ACCOUNTS	JOBS	CPU (HOURS)	TAT (HOURS)	SOURCE LINES (MILLIONS)	OBJECT CODE (MILLIONS)
FY-76, 7T (JUL 75 - SEP 76)	3	10	57,686	197	0.53	1.6	1.6
FY-77 (OCT 76 - SEP 77)	6	20	104,652	907	2.0	2.0	2.0
FY-78 (OCT 77 - SEP 78)	13	78	110,368	1,035	2.2	2.9	2.9
FY-79 (OCT 78 - SEP 79)	28	236	105,032	1,344	0.65	3.7	3.7
FY-80 (OCT 79 - SEP 80)	35	299	118,960	1,449	0.66	7.2	8.2
FY-81 (OCT 80 - SEP 81)	41	438	135,265	1,855	0.93	12.8	12.9
FY-82 (OCT 81 - MAR 82)	47	492	67,445	1,153	0.88	14.8	16.3

Figure 2. Key Parameters Measured With The FASP

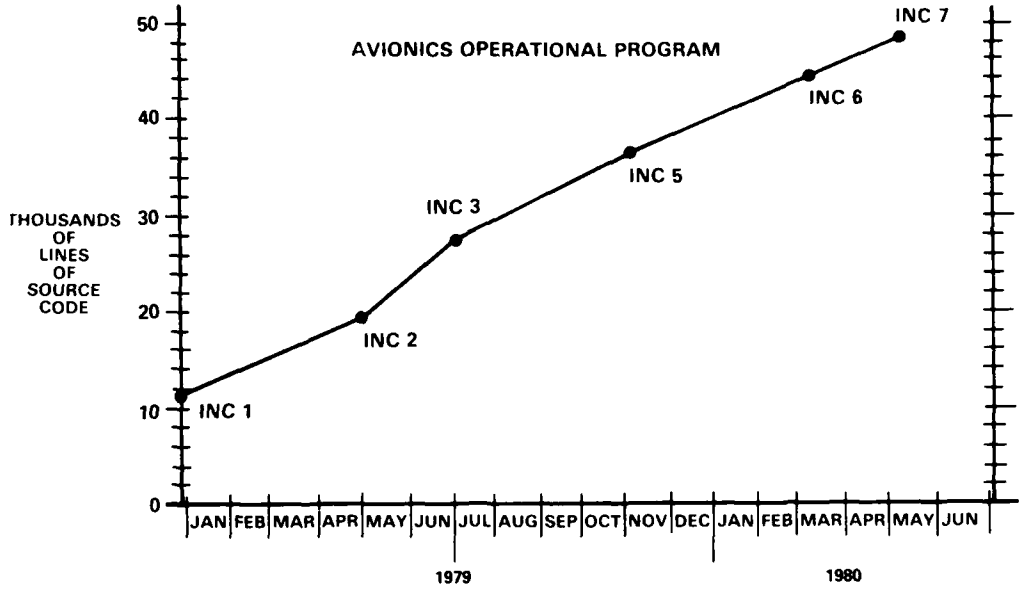


Figure 3. Delivered Source Lines

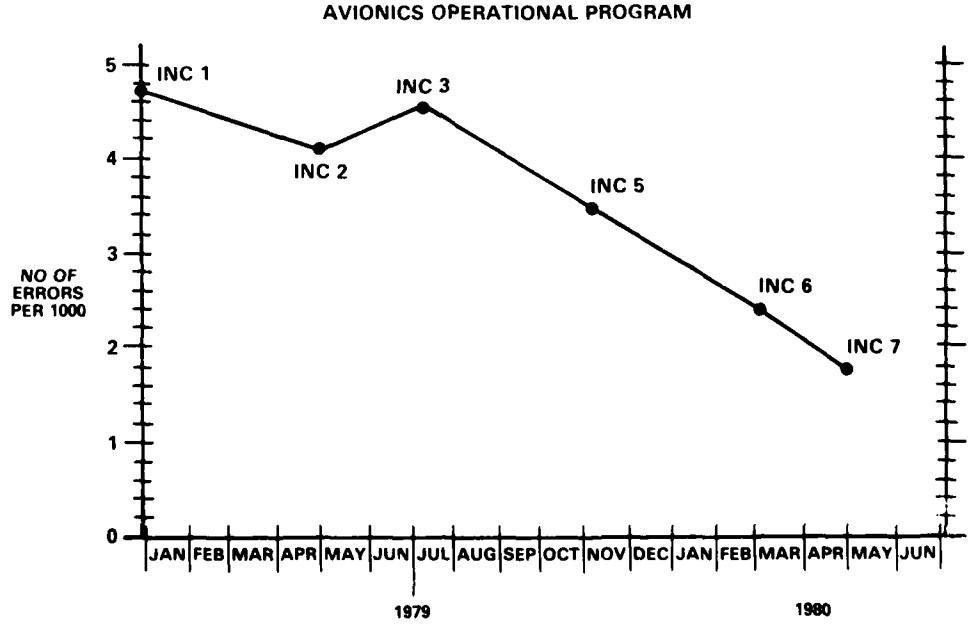


Figure 4. Number Of Software Errors Recorded

SOFTWARE DEVELOPMENT PROCEDURES
CREATE/COPY/SAVE/RESTORE A DATA BASE
DEVELOP SOFTWARE
INSTALL EXTERNAL SOFTWARE
SHARE/COPY SOFTWARE
CREATE LOAD IMAGES/TAPE
PRINT REPORTS
SOFTWARE TESTING PROCEDURES
ANALYZE SOURCE CODE
DEVELOP TESTS
INSTRUMENT CODE
EXECUTE TESTS
DEBUG TESTS
REGRESSION TEST
ANALYZE TEST RESULTS
USER ASSISTANCE PROCEDURES
LIST BULLETIN
LIST NEWS
LIST HELP
LIST USER MANUAL
CONTACT SPF PERSONNEL
SOFTWARE MANAGEMENT PROCEDURES
CONFIGURE A PROJECT
CONTROL ACCESS
PRINT PROGRESS REPORTS
IDENTIFY SOFTWARE CONFIGURATION
RELEASE SOFTWARE
TRACK STRs, SCPs, SEPs
CONFIGURATION STATUS ACCOUNTING
GENERAL PROCEDURES
DOCUMENT PRODUCTION
CONTEXT CONTROL
LOGGING ON/OFF
GLOBAL PARAMETER HANDLING
COMMAND QUEUE HANDLING
DATA CREATION
OUTPUT HANDLING

Figure 5. List Of General Procedures For Code And Test

```
Verify user access rights for this procedure
IF verification fails
THEN raise abort flag
  •  Notify user
ELSE
  •
  •     (unique process flow for procedure)
  •
ENDIF
IF abort flag is not raised
THEN save production data
  •  Make "saved" files permanent
ENDIF
Save job statistics
Make job statistics permanent
```

Figure 6. Standard Processing For Procedures

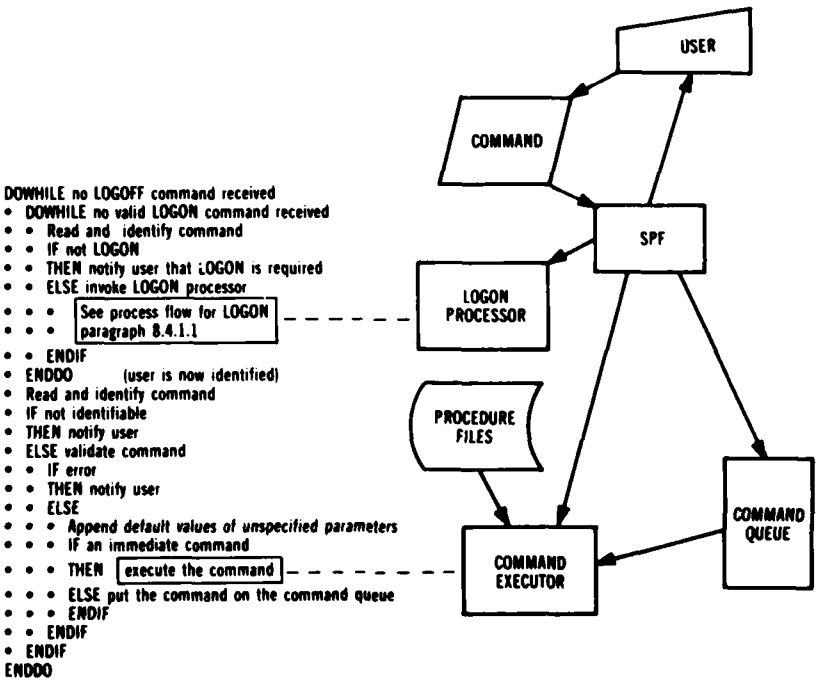


Figure 7. Command Processing Diagram

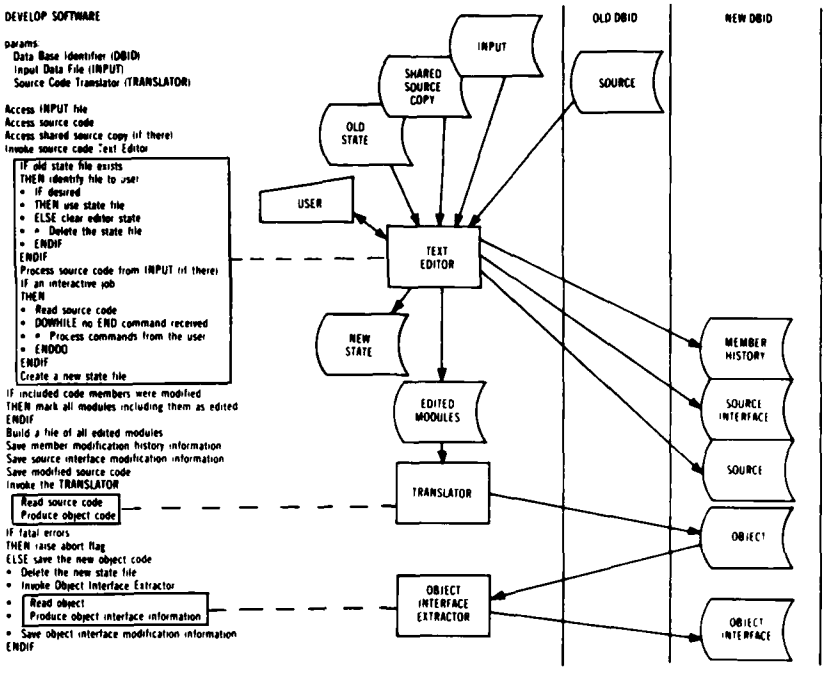


Figure 8. Process Flow Diagram

COMPUTER	AN/UYS-1	AN/AYK-14	AN/UYK-7	CYBER
COMPILER	SPL/I	CMS-2M	CMS-2Y	FORTRAN
ASSEMBLER	SPL	MACRO 20/14	ULTRA 32	COMPASS
SYSTEM GENERATOR	✓	✓	✓	✓
AUTOMATED TEST ANALYZER	✓	✓	✓	—
SOFTWARE EMULATOR	✓	✓	✓	—
MANAGEMENT REPORTS	✓	✓	✓	✓

✓ - PRESENT IN SYSTEM; — NOT PRESENT

Figure 9. Matrix Of Languages And Target Computers Of The FASP

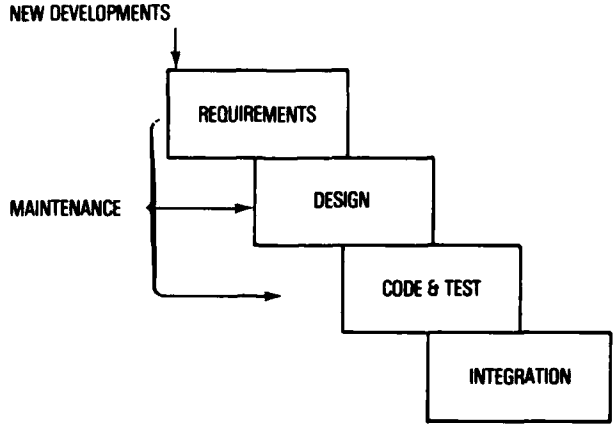


Figure 10. Simplified Development Process

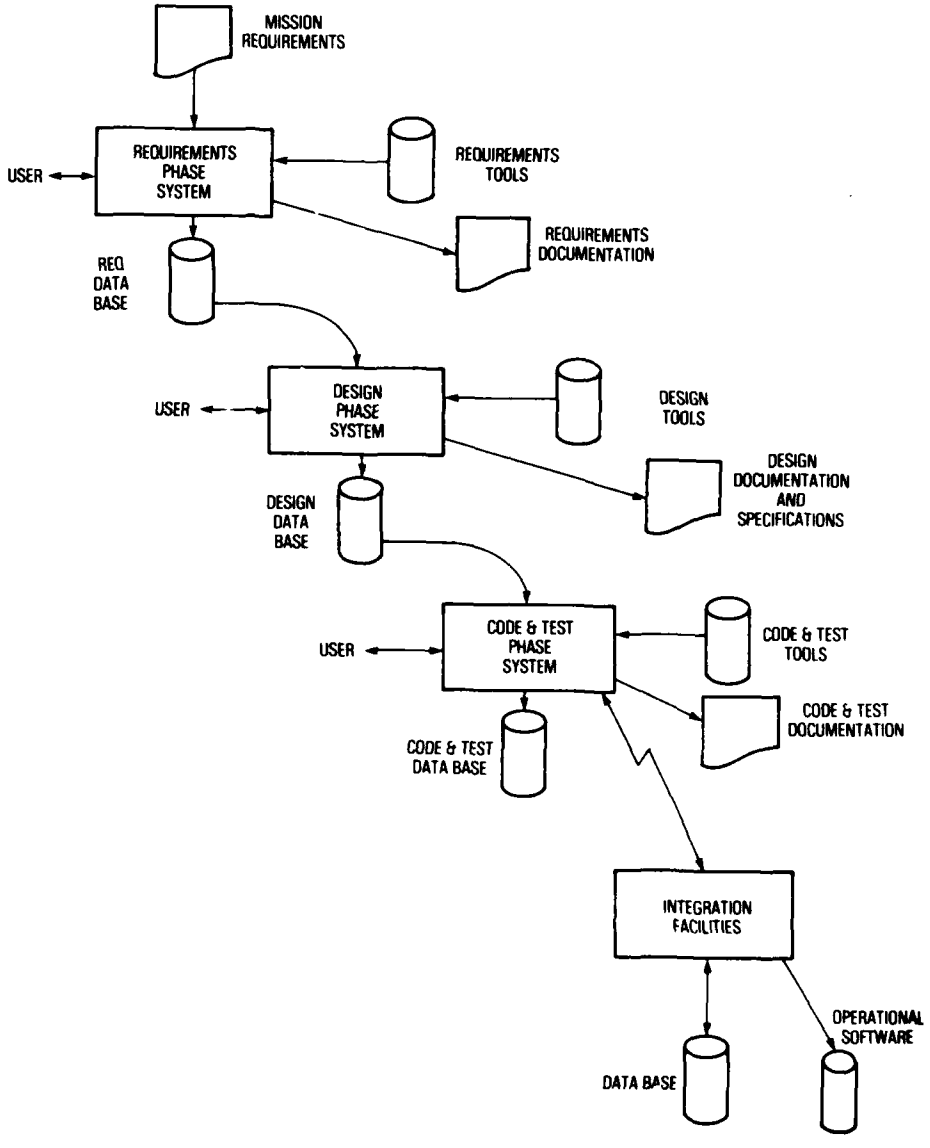


Figure 11. Separate Facilities For Each Phase

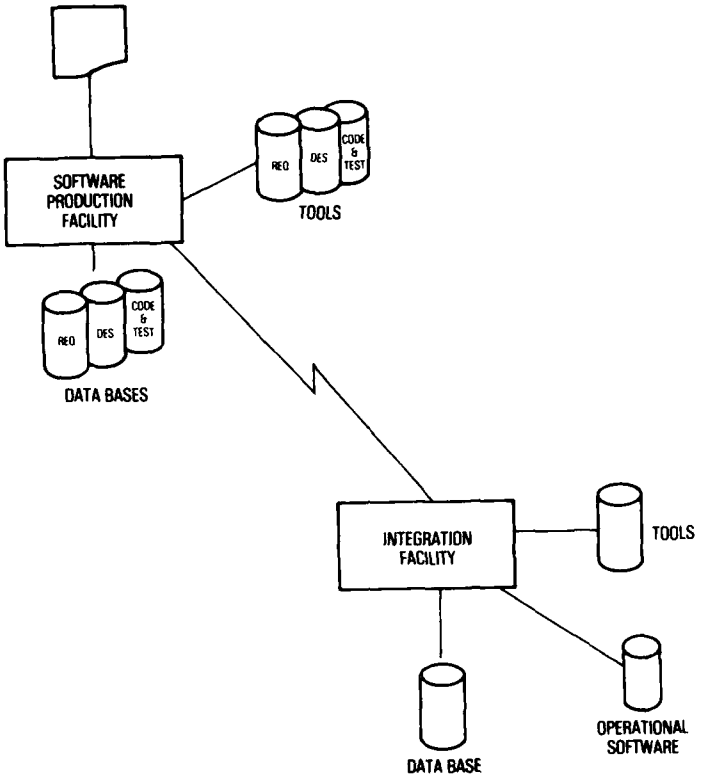


Figure 12. The NADC Two Facilities System

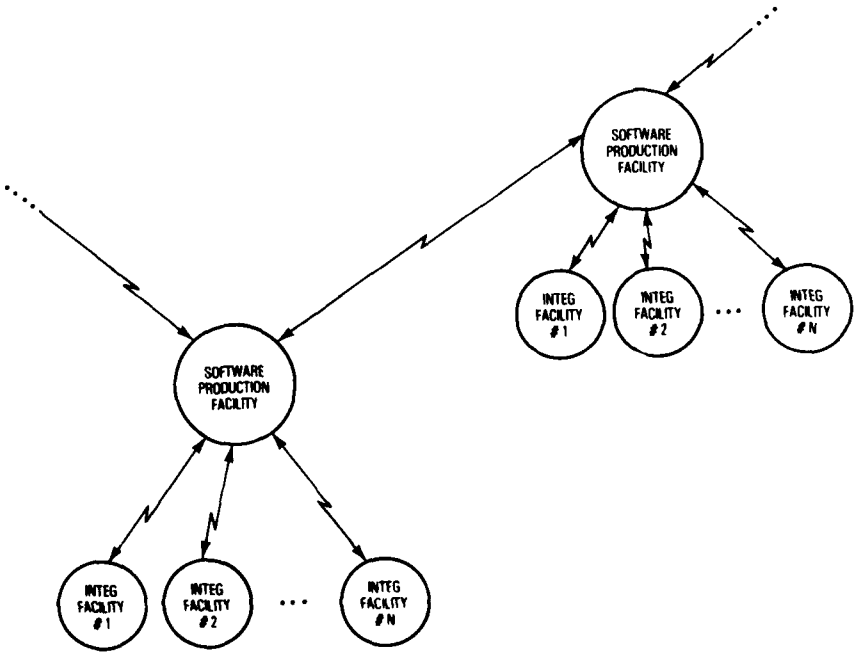


Figure 13. Interconnection Of Software Production Facilities By Communications Networks

ON AIRCRAFT TEST SOFTWARE FOR FIRST LINE MAINTENANCE

Dr.H.Klenk
MBB Messerschmitt-Boelkow-Blohm GmbH
Postfach 80 11 60
D-8000 Muenchen 80

SUMMARY

The avionics of an aircraft is subject to many hardware and software modifications. Therefore, any first line test software for the avionic system equipment must be very flexible to reduce software maintenance costs. In this paper a method is presented to provide suitable flexibility and easy maintenance for a test software package, even when written in a lower level language. The method essentially consists of a decisive separation of the test software package into an executive part and a descriptive part for the avionic hardware. Because the executive part is independant of the special avionic equipment, hardware modifications result only in a change of the descriptive part of the software. These changes are easy to handle and suited for future standardized methods using modern software development tools. The separation method has been used to design and implement our on aircraft test software with good results.

1. INTRODUCTION

The avionics of an aircraft is a complex system of sensors, interfaces and processors. For the developing companies this implies much work in the area of hardware integration; hardware/software integration; hardware and software modification as well as software maintenance. For the end user - the air crews and the maintenance personnel - the correct functioning of the whole system is crucial and error detection as well as maintenance becomes a large scale problem.

Another difficulty often encountered is the integration of new or modified avionic equipment. What both producers and end users therefore urgently need is a system that supports the following:

- hardware compatibility checks;
- hardware/Software integration checks;
- hardware configuration control;
- pre-flight checks;
- in-flight error recording;
- post-flight fault evaluation;
- maintenance assistance.

These tasks have been conferred upon a test software package which has been integrated within the Operational Flight Program (OFP) to allow in flight checks.

During the life cycle of the aircraft and thus the software, many modifications and extensions of the hardware have to be expected. This has an especially severe impact on the test software since:

- tests have to be modified because of hardware modification;
- tests have to be cancelled because of equipment cancellation;
- new tests have to be integrated because of new equipments;
- status and mode indications have to be modified and/or extended;

- the faults to be monitored during flight change according to the aircraft hardware configuration.

Therefore, test software must initially be designed in such a fashion that modification and extension can be carried out with a minimum of effort in redesign, coding, testing and aircraft release procedure. This is especially valid for a test package which has been written in assembler such as in our case. Therefore, this paper gives emphasis to this aspect. Software documentation is another critical point as is to be expected in a large package.

2. TEST SOFTWARE DESCRIPTION

The test software package has been designed for a classical star-like interconnection of avionic equipment and a central processor (see Fig. 1). Some equipment have their own dedicated processor for specific internal task execution. The principle design facets however, can be applied to many other system types (including a duplex bus system) with the only restriction being that all information and commands necessary to conduct a test have to be accessible to the processor performing it. Thus, in a multiprocessor system, special tests can be performed by special processors.

The central processor receives avionic data and status information from the equipment and commands from the control switches and panels. It also transmits data and status information to the equipment (including the displays).

When selected from normal OFF operation, the test software is capable of performing the following functions:

- receipt of commands from the system operator (using multi-function keyboard inputs) to perform specific tests;
- receipt and display of status and mode information of all relevant avionic equipment;
- upon request, display of avionic parameters and more detailed status information;
- upon request, display of faults and important parameters recorded during the last flight;
- upon request, start or repeat the test of a specific avionic equipment.

During normal OFF operation, only the fault recording task is active. This task monitors all equipment and records all faults which occur during flight, including certain important aircraft parameters upon the first occurance of the fault. Flight critical faults are not only recorded but also shown on the displays.

3. FLEXIBLE SOFTWARE DESIGN

The basic goal is to achieve an easy to modify (flexible) test software. This is accomplished by separating the package into an executive part that is as independent as possible from the avionic hardware environment and a descriptive part which defines the hardware but contains no executable code whatsoever. This separation is relatively simple for the display and recording parts, but is more complex for the equipment tests.

3.1 How to Separate

The first step in the separation procedure is to define which parts of the functions to be performed are common for all equipment. Normally, it is not possible to cover all functions in the common executive part, because several equipment may have specific requirements. Thus, the next step would be an introduction of all special functions into the executive part in such a way that they can be requested by the descriptive part for these specific

equipment.

The last step in the separation procedure - in my opinion the most critical one - has to be a look towards the future. Which requirements are probable to arise during the further life of the software ? After evaluation, all those functions which can reasonably be integrated in the design phase - both in the executive part and the structure of the hardware description part - should be covered. This step must be treated very thoroughly because the whole concept of separation loses many of its advantages if in practice every modification of equipment hardware causes a change in the executive part of the software. Nevertheless, one should be aware that hardware modifications may arise which cannot be covered by the current structure of the software. Therefore, a modular design of the executive part and an expandable descriptive part greatly facilitates the incorporation of new functional requirements.

3.2 Separation for the Display Task

As a simple example of separation, let us treat the display of status and data information for several avionic equipment. The executive part consists of:

- display, control and equipment selection;
- general format layout generator;
- data acquisition;
- transformation of data and display (the transformation may consist of either a special message for each status bit or of numerical conversion to display format and its unit).

Because all relevant equipment data for the display task are available in the memory of the central processor, we have no specific requirements for specific equipment. Furthermore, the display format allows both messages for status bits and numbers with units so that known possible future modifications are covered by this structure.

The descriptive part is structured in units or entries adapted to the executive part. In the case of the display task, a well suited unit is the description of one equipment. Thus, the whole descriptive part is a list composed of one entry for each equipment. The structure of the list can be a linked one, if only sequential stepping through is necessary, or a contiguous one, if direct access to each equipment is mandatory. An entry therefore may appear as shown in Figure 2.

The display task is adapted to modifications in the equipment hardware by simply changing the corresponding entry in the hardware description list with the executive part remaining completely unchanged. A new equipment is catered for by a new entry in the list, which describes the new equipment. Thus, the changes are limited to a very small portion of the software which allows quick error tracing and testing. Another advantage, which becomes even more important in the future when software development tools will be widespread, is that changes in the actual code of the list are suited to be performed automatically from higher levels of program description and design. This renders the change method even more safe.

3.3 Separation for the Testing Task

The task that actually runs the equipment tests requires much more effort in separation. The different equipment require different testing methods and so the testing must be broken down into small elementary test units in order to be able to find reasonably common functions. Because of the variety of equipment, there are many elementary functions (roughly 40 in our case).

The executive part therefore consists of an appropriate repetoire of functions and a test is performed by sequentially executing the required elementary functions in the order specified by the description part.

In a first step the common elementary functions have been established to be:

- etting parameters or bits to specific values;
- checking whether parameters are in range;
- checking status bits;

- issuing error messages if an error occured;
- performing jumps in the test list if specific conditions are met;
- checking if buttons have been pressed;
- checking positions of switches;
- displaying parameter values;
- issuing commands to equipment;
- introducing a time delay before the next action is started;
- waiting for input.

Again, the structure of the functions has been chosen in such a way that a future software development tool will allow automatic code production out of a higher level of specification, for example, from a quasi-code description. In the meantime, manual procedures have been established to integrate modifications without this tool, with as much safety as possible.

Future modifications and extensions of the hardware have been integrated insofar as they are known. To cover as many as possible unknown modifications, the elementary functions have been designed to be more flexible than momentarily necessary. In addition, universal functions not presently used have been included as well. Keeping in mind that even the most thorough analysis cannot cater for all future modifications, the test executive part has been designed in such a way that new elementary functions can be entered without significant alteration of the executive part. The only necessary alteration is a new entry in the list of available functions which points to the new executive part. Thus, a request of the new function by the description table will result in a check whether the new function is available in the function list and if so, control is transferred to it.

The descriptive part is structured hierarchially. The first level is subdivided into individual tests for each equipment to be tested. The next one is the action level which groups several elementary functions into one operator action. The bottom level are descriptors of the elementary functions to be performed. Thus, the descriptive part is a structured test list which tells the executive part which functions must be executed for specific tests. Since tests have to be selectable at random, the list is not a linked one. Its structure is given in Figure 3.

An example of the description of an elementary function is given in Figure 4.

Hardware modifications that influence the test execution are integrated in the test software by simply changing entries in the elementary function list, in the action list, or in the test list. A new test is introduced by adding one entry in the test list, with one list for the actions and one list of elementary functions for each action. Thus, the old test descriptors remain completely unchanged in the latter case and testing effort is decreased significantly.

4. STATUS

The test software package described above is currently in preparation for operational release. It has been tested on-aircraft on ground, where most of its facilities are available. Since the software is currently still in an early life cycle stage, little experience has been gathered to date with regard to actual maintainability. Nevertheless, during the final development stage, addition of a new test and some minor modifications to existing tests were already required. Their integration proved to be very easy both in design, implementation and testing since no new elementary function had to be included and only the descriptive part had to be changed. Especially error debugging proved to be quite simple, enormously reducing the cost of checkout on the avionic r[illegible]

5. CONCLUSIONS

Experience with former test software packages has shown that a flexible software design is mandatory in preventing exorbitant software maintenance costs. This is especially true for packages written in low level languages. As was shown in this paper, much flexibility may be gained by separation of the test package into an executive and a descriptive part. In the future, modifications can be implemented in a standardized way using modern development tools. Even the test phase will then be supported by these tools. This will result in an even more drastic reductions in development cost and test time than is available at the moment (i.e. at present changes are manually implemented).

One must on the other hand, be aware that not 100% of all future modifications will be covered by these standardized methods. This may be due to a possible lack of functions in the executive part. Thanks to the modular structure of the latter, adding new functions is simple and will not cause cost escalation. In total, we expect a large cost saving in future software maintenance due to the flexible design of our test software package.

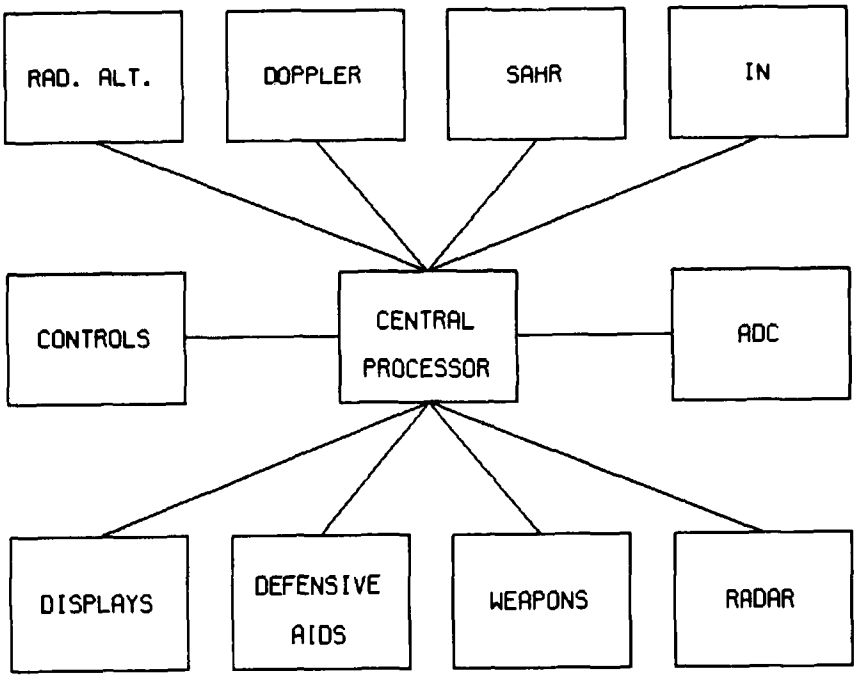


FIG.1 EXAMPLE OF A CLASSICAL AVIONICS SYSTEM

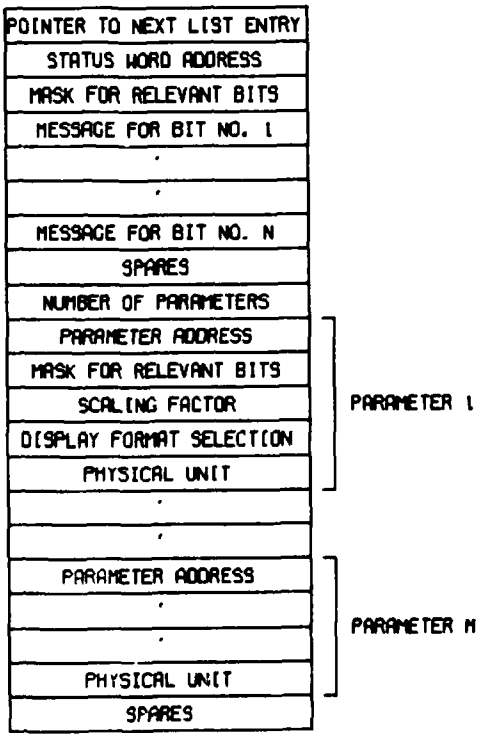


FIG.2 EXAMPLE OF A DISPLAY TASK LIST ENTRY

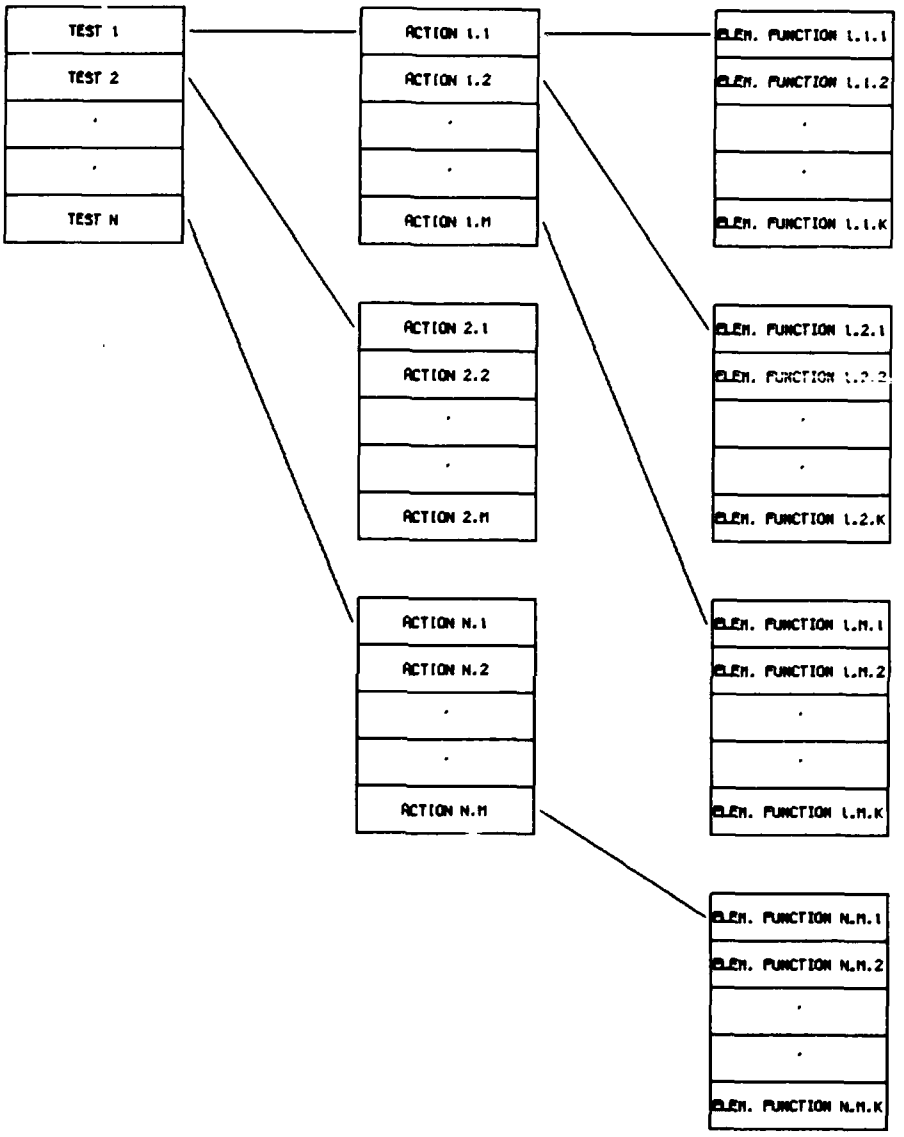


FIG.3 TEST DESCRIPTION LIST STRUCTURE

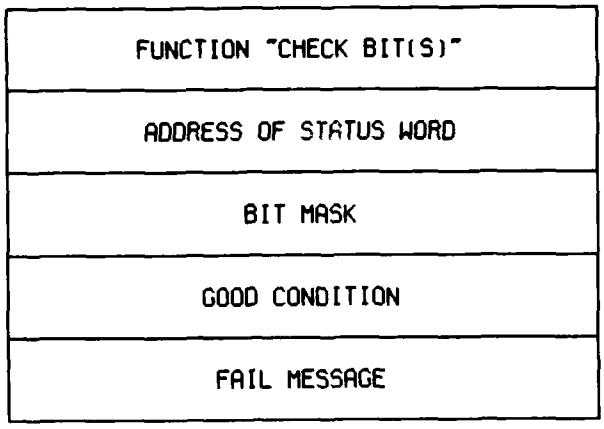


FIG.4 EXAMPLE OF AN ELEMENTARY FUNCTION LIST ENTRY

DISCUSSION FOR AVIONICS PANEL FALL 1982 MEETING ON
SOFTWARE FOR AVIONICS

Session 5 : SOFTWARE LIFE CYCLE CONSIDERATIONS - Chmn Dr. H. Hessel (GE)

Paper Nr. 38 - MANAGEMENT OF LARGE REAL-TIME MILITARY AVIONICS SOFTWARE PROGRAMS
Presented by - Dr. P. J. Carrington
Speaker - Wg. Cdr. Bonnor
Comment - 1. If MASCOT had been a required methodology for AQS 901 software, what impact would this have had on the project development.
2. If the RAF wish to make the best use of the major software investment in the AQS 901 and not incur a major software re-write at a future refit, what are the prospects of obtaining replacement processors which can use the current software with minimum change. How practical do you see it to be to buy or force a micro-processor to fit our software.

Response - 1. The use of MASCOT would have given a better structure to the software. However, this would have been at the expense of data throughput and processing facilities, which might affect operational capability.
2. Faster and smaller processors can be developed which are transparent to the software and hence avoid a major software rewrite. But, to make full use of the processor speed improvement, other areas of the system might need to be modified for faster store.

Paper Nr. 38 - MANAGEMENT OF LARGE REAL-TIME MILITARY AVIONICS SOFTWARE PROGRAMS
Presented by - Dr. P. J. Carrington
Speaker - W. Praedrich
Comment - Patching may be the better way to achieve a fast response on queries than recompilation. But increasing numbers of patches makes maintenance more and more difficult. What are you doing to overcome this problem?

Response - The number of patches is generally not the problem if sensible configuration control procedures are used. The problem occurs with patches which have major or wide-ranging system effects. These we normally only incorporate by recompilation, except for short-term trials modifications or for rapid response to essential operational requirements.

Paper Nr. 39 - F/A-18 AVIONICS SOFTWARE - A CASE STUDY
Presented by - T. V. McTigue
Speaker - L. Skorczewski
Comment - Since the F-18 OFP is now located in distributed processors connected by an asynchronous data-bus, was any attempt made to simulate this in the IBM/Fortran design prototyping exercise, and were any problems subsequently found on the system integration rig which were due to the hardware partitioning and asynchronous operation?

Response - The 1553 bus was not simulated in the IBM/Fortran exercise. However, we did simulate the 1553 bus traffic in the laboratory testing.

Paper Nr. 39 - F/A-18 AVIONICS SOFTWARE - A CASE STUDY
Presented by - T. V. McTigue
Speaker - N. Haigh
Comment - Did you find that adopting a top-down structure for the OFP added significantly to the storage requirements?

Response - No it did not add significantly to storage requirements, only about 10% to 12%.

Paper Nr. 39 - F/A-18 AVIONICS SOFTWARE - A CASE STUDY
Presented by - T. V. McTigue
Speaker - Dr. W. J. Cullyer
Comment - How did you propagate the use of the flow chart constructions, if...then...else and repeat... until and so on into CMS2?

Response - We could not directly propagate these standards into the CMS-2M language system (including an Assembler extension) used as the programming language because CMS-2M does not directly support these structured constructs. However, by structuring the flowcharts and then implementing these exactly in the available CMS-2M language constructs, we obtained many of the advantages inherent in a structured programming language.

Paper Nr. 39 - F/A-18 AVIONICS SOFTWARE - A CASE STUDY
Presented by - T. V. McTigue
Speaker - J. M. Janssen
Comment - 1. How much memory is currently left in the mission computers?
2. Were there timing problems? What is the duty cycle?
3. What are the growth possibilities? What software changes are required?
Response - 1. 11, 6K
2. No timing problems yet, in the future this is to be reconsidered.
3. Double the memory from 64K to 128K by replacing (2) existing memory cards with two new ones.

Paper Nr. 40 - A LIFE CYCLE MODEL FOR AVIONIC SYSTEMS
Presented by - H. Schaaff
No Questions

Paper Nr. 41 - AVIONICS SOFTWARE SUPPORT COST MODEL
Presented by - R. Shaw

No Questions

Paper Nr. 42 - A SOFTWARE-COST DATA BASE FOR AEROSPACE SOFTWARE DEVELOPMENT
Presented by - G. J. Dekker
Speaker - Dr. H. Hessel
Comment - How many projects have you in your data base currently and how many do you need?

Response - Currently we have one project completely monitored. Some 5 projects are available to be entered.
We need at least 30 to 40 projects to calibrate the formulae from the 8 cost factors to the development cost.

Paper Nr. 42 - A SOFTWARE-COST DATA BASE FOR AEROSPACE SOFTWARE DEVELOPMENT
Presented by - G. J. Dekker
Speaker - H. R. Simpson
Comment - Could you comment on the linearity of the relationship between cost estimates and the factors identified as contributing to cost?

Response - We did not give much thought yet about the exact form of the relationship between each cost factor and the cost. Much of the formulae relating cost factors to cost will presumably be linear. The relation between size and complexity and cost will presumably be exponential. This will however be shown by the cost data which we will gather.
Paper Nr. 43 - THE MILITARY USER VIEW OF SOFTWARE SUPPORT THROUGHOUT THE IN-SERVICE LIFE OF AVIONIC SYSTEMS
Presented by - Wg. Cdr S. Barker RAF
Speaker - S. Okman
Comment - Software reuseability can be achieved by many techniques. One technique that is possible today is to require of the Hardware Designer to support S/W developers on user equipment, i.e. Design the new computer to support older architecture as well as new ones. Example - AN/UYK-7 to AN/UYK-43 Project, as it is presently going on in the U.S.A.

Response - I agree with your observations and this is certainly one step towards achieving software reuseability. However it is also important to require the software designer to produce software that is independent of the target hardware characteristics and to provide him with the environment and the tools to do so.

Paper Nr. 44 - DESIGN OF A SOFTWARE MAINTENANCE FACILITY FOR THE RAF
Presented by - J. Whalley
Speaker - H. R. Simpson
Comment - Could you tell us please whether the RAF will be using your software validation, verification and test procedures when they make changes?

Response - This is still under discussion, however the RAF software team are currently based at Bristol Aerospace Manchester and are familiarizing themselves with the contractor's procedures which include a configuration control system based on a total data base system. There is therefore a high probability that the RAF will adopt British Aerospace's procedures.

Paper Nr. 45 - A SOFTWARE ENGINEERING ENVIRONMENT (SEE) FOR WEAPON SYSTEM SOFTWARE
Presented by - H. G. Stuebing
Speaker - T. F. Kensey
Comment - Reference figure 4. At what point were the errors detected within the software development programme?

Response - The errors were detected during unit module testing in the Code and Test phase. (See Figure 1)

Paper Nr. 46 - ON AIRCRAFT TEST SOFTWARE FOR 1ST LINE MAINTENANCE
Presented by - Dr rer nat H. Klenk
No Questions

APPENDIX

LIST OF ATTENDEES

AARKVISLA, S. Major	FO/LST, LTJINSP, Oslo MIL/HUSEBY, Oslo 1, No
ALLEN, K. Major	Directorate of Maritime Aviation 3-2CA, 101 Col By Drive, Ottawa, Ontario KIA OK2, Ca
ANDERSON, T. Dr	University of Newcastle Upon Tyne, Computing Laboratory, Claremont Tower, Claremont Rd, Newcastle Upon Tyne NE1 7RU, UK
APPLIN, P. Mr	Lucas Aerospace Ltd, 40 48 Chase Road, Willesden, London NE10 6PX, UK
ASTLEY, J. Capt.	Software Development Unit, Canadian Forces Base Greenwood, Greenwood, Nova Scotia BOP 1NO, Ca
BAARSPUL, M. Mr	Delft University of Technology, Dept. of Aerospace Engineering, Kluyverweg 1, 2629 HS Delft, Ne
BACOT, M.D.P. Mr	MASCOT, 17 rue Paul Dautier, 78 Velizy, Fr
BALL, W. Mr	Head, Tactical Software Eng. Div., Code 319, Naval Weapons Center, China Lake, CA 93555, US
BARBER, D. Mr	Room K 213, CAA House, 45 49 Kingway, London WC2B GTE, UK
BARKER, S. Wg Cdr	ACCS Team, NATO Hqs, 8 rue de Geneve, 1140 Bruxelles, Be
BERGER, P. Mr	Electronique Serge Dassault, 55 Quai Carnot, 92214 St. Cloud, Fr
BEVAN, F.H. Mr	Marconi Avionics Ltd, Airport Works, Rochester, Kent, UK
BOARDMAN, R.M. Mr	Marconi Avionics Limited, Airborne Software Division, Elstree Way, Borehamwood, Herts WD6 1RX, UK
BONNOR, Wg Cdr (RAF)	MOD (PE), A & AEE Boscombe Down, Salisbury, Wilts SP4 0JF, UK
BOSMAN, D. Prof. Ir.	Bldg. EF, Twente University, P.O.Box 217, 7500 Enschede, Ne
BOT, J. Mr	Fokker BV, Afd ISA/MS, Postbus 7600, 1117 ZJ Schiphol, Ne
BRACON, G. Ing.	Electronique Serge Dassault, 55 quai Carnot, 92214 St. Cloud, Fr
BRAULT, Y. Mr	Thomson CSF, 178 Blvd Gabriel Peri, 92240 Malakoff, Fr
BRUNORO, F. Dr	SIAI Marchetti, Via Independenza n. 2, Sesto Calende, (Varese), It
BRUYNS, H. Mr	Fokker B.V. ald CB/LW, Postbus 7600, 1117 ZV Schiphol, Ne
BUJAK, E. Mr	VFW/MBB-UT 2800 Bremen 1, Hunefeldstr 1 -5, Postfach 10 78 45, Ge
BURROWS, J. Dr	Lucas Aerospace Ltd, Maylands Ave., Hemel Hempstead, Herts HP2 4SP, UK
CALLAWAY, A.A. Dr	Flight Systems Dept., Royal Aircraft Establishment, Farnborough, Hants, UK
CAMERON, A. Mr	Smiths Industries Aerospace & Defence Systems Company, Evesham Rd, Bishops Cleeve, Cheltenham, Glos. GL52 4SF, UK
CARRINGTON, P.J. Dr	Maritime Aircraft Systems Division, Marconi Avionics, Airport Works, Rochester, Kent, UK
CAWTHORNE, G. Mr	Thorn EMI Electronics Ltd, Penleigh Works, Wookey Hole Rd, Wells, Somerset, UK
CHEVREUL, Mr	SAGEM, BP 51, 95612 Cergy Pontoise Cedex, Fr
CHINN, P. Mr	Airborne Software Division, Marconi Avionics Ltd, Borehamwood, Herts, UK
CHRISTENSEN, N. Dr	MBP, Semerteichstr. 47, D-4600 Dortmund 1, Ge
CINAR, U. Dr	SHAPE Technical Centre, P.O.Box 174, 2501 CD The Hague, Ne
CLARY, J.B. Mr	Research Triangle Institute, P.O. Box 12194, Corwallis Road Research Triangle Park, NC 27709, US
COLLARDEAU, C. Mme	Societe Crouzet, 25, rue Jules Vedrines, 26027 Valence Cedex, Fr
CORBISIER, F. L/Col.	Heidestraat 23, 2850 Keerbergen, Be
CORNELISSEN, P.J. Mr	Delft University of Technology, Dept. of Aerospace Engineering, Kluyverweg 1, 2629 HS Delft, Ne

COUDERT, F.J. Mr	SHAPE Technical Centre, P.O. Box 174, 2501 CD, The Hague, Ne
COULMY, P. Mr	Thomson CSF, 52 rue Guynemer, 92130 Issy-les Moulineaux, Fr
CROCKER, S.D. Dr	The Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009, US
CROVELLA, L. Dr	Aeritalia Gruppo Sistemi Avionici ed Equipaggiamenti, 10072 Caselle (Torinio), It
CULLYER, W.J. Dr	RSRE MOD (PE), St Andrews Rd, Malvern WR14 3PS, UK
DANIEL, J.P. Mr	Thomson CSF, 52 rue Guynemer, 92130 Issy-les-Moulineaux, Fr
DEKKER, G.J. Mr	NLR National Aerospace Laboratory, P.O.Box 90502, NL-1006 BM Amsterdam, Ne
DELACROIX, M. Prof.	IGL, 46 rue de Provence, 75009 Paris, Fr
DELAHAYE, J. Mr	Electronique Serge Dassault, 55 Quai Carnot, 92214 St Cloud, Fr
DeVRIES, J.C.Z. Mr	Fokker B.V., CB-BS, Postbus 7600, 1117 Jz Schiphol Oost, Ne
DeVRIES, M.J. Mr	Fokker B.V. ald CB/LW, Postbus 7600, 1117 ZV Schiphol, Ne
DIAMOND, F. Dr	RADC/CA, Griffiss AFB, NY 13441, US
DIBBLE, R. Mr	Ferranti Computer Systems Ltd, Ty Coch Way, Cwmbran, Gwent NP44 7XA, UK
DIJKSTRA, P.J. Ir.	RNLAF/DMKLU/AWO, Postbus 5953, 2280 Hz Rijswijk, Ne
DOLADILLE, F. Ing.	Electronique Serge Dassault, 55 quai Carnot, 92214 St Cloud, Fr
DOLMAN, W.C. Mr	Lucas Aerospace, York Road, Hall Green, Birmingham BR8 8LN, UK
DORP, W.A. van Mr	National Aerospace Research Lab., Anthony Fokkerweg 2, 1059 CM Amsterdam, Ne
DOVE, Billy Mr	Mail Stop 477, NASA Langley Research Center, Hampton, VA 23665, US
DOWLING, E.J. Mr	Ferranti Computer Systems Ltd, Ty Coch Way, Cwmbran, Gwent NP44 7XX, UK
DUBOIS, B. Major	Etat Major de la Force Aerienne VDT, 1 rue d'Evere, B-1140 Bruxelles, Be
DUNCAN, I. Mr	Ferranti plc, Ferry Road, Edinburgh EH5 2XS, Scotland, UK
ESCAFFRE, F. Mr	*SNIAS DTO/PLE, 316 Rte. Bayonne*, 31060 Toulouse Cedex, Fr
EVAIN, R. Mrs	SFENA, Aerodrome de Villacoublay, 78141 Villacoublay, Fr
FAULKNER, J.A. L/Col.	Aurora Software Development Unit, CFB Greenwood N.S., Ca
FERRANTE, P. Mr	Reparto Sperimentale Di Volo, Aeroporto Pratica di Mare, Roma, It
FIKKERT, D.W. Mr	Physics Laboratory TNO, P.O. Box 208, 2509 JG The Hague, Ne
FOLIGUET, G. Mr	Thomson-CSF, 52 rue Guynemer, 92130 Issy-les-Moulineaux, Fr
FOULON, M. Ing.	MATRA, 17 rue Paul Dautier, 78 Velizy, Fr
FRAEDRICH, W. Mr	Bundesminister der Varteidijeng, Postfach 1328, 5300 Bonn, Ge
FUCHS, J. Mr	Teldix GmbH, Grenzhoferweg 36, POB 105 608, 6900 Heidelberg, Ge
GALLI, S. Miss	LABEN, Via Cassini 15, Milano, It
GERHARDT, L.A. Prof.	School of Engineering, Rensselaer Polytechnic Institute, 110 Eighth Street, Troy, N.Y. 12181, US
GHICOPOULOS, B. Dr	HAF Technology Research Center, KETA Delta Falirou, P. Faliron, Athens, Gr
GIUBOLLINI, M. Mr	Selenia s.p.a., Via dei Castelli Romani 2, Pomezia, It
GROENENDIJK, J.Q.M. Mr	Fokker B.V., Postbus 7600, 1117 ZJ Schiphol-Oost, Ne
GROOT, Th.H. de Mr	Directie Luchtvaartinspectie RLD, Stationsplein Gebouw 144, 1117 AA Schiphol, Ne
GROOTE, H. von Dr	MBB Ottobrun-FE 411, Postfach 80 11 60, D-8000 Munchen 80, Ge
GUILLAUME, J.P. Mr	SFENA, Aerodrome de Villacoublay, 78141 Villacoublay, Fr
GUSMANN, B. Dr	Fa. LITEF (Litton Technische Werke), Abt. EWD, Postfach 774, 7800 Freiburg i. Br., Ge
GIJSBERS, G.A. Mr	KLM Aircraft System Group (SPL/CI), P.O.Box 7700, 11172L Schiphol-Oost, Ne
HAIGH, N.P.H. Mr	MOD (PE) A&AEE Boscombe Down, Salisbury, Wilts SP4 0JF, UK
HALL, W.H. Mr	British Aerospace P.L.C., Brough, HU15 1EQ, North Humberside, UK
HAMBERGER, W. Mr	Eurocontrol, Postbus 78, NL-62 36 ZH Luchthaven Zuidlimburg, Ne
HANSEN, R. Mr	Messerschmitt-Bolkow-Blohm GmbH, P.B. 801160, 8 Munchen 80, Ge

HARRIS, Mr. K.E.	Smiths Industries, Aerospace and Defence Systems Company, Winchester Rd, Basingstoke, Hants RG22 6HP, UK
HARTUNG, W.G.	SHAPE Technical Centre, P.O.Box 174, 2501 CD The Hague, Ne
HELMICH, K. Mr	Bodenseewerk Geratetechnik, 7770 Uberlingen, Postfach 1120, Ge
HELPS, K.A. Mr	Smiths Industries Aerospace & Defence Systems Company, Evesham Rd, Bishops Cleeve, Cheltenham, Glos, GL52 4SF, UK
HENDERSON, I.H.S. Dr	DTA Air, National Defence Hqs, 101 Colonel By Drive, Ottawa, Ontario K1A OZ3, Ca
HESSEL, H. Dr	MBB Ottobrunn, FE 43, Postfach 80 11 60, 8000 Munchen 80, Ge
HIGSON, N. Mr	Airborne Software Division, Marconi Avionics Ltd, Borehamwood, Herts, UK
HOEVE, J.T. Mr	National Aerospace Research Lab, NLR, P.O. Box 90502, 1006 BM Amsterdam, Ne
HOLMES, R.H. Mr	Airborne Display Division, Marconi Avionics Ltd, Airport Works, Rochester, Kent, UK
HOOGERVORST, J.A.P. Mr	KLM N.V., Aircraft Systems Group, SPL/CI, P.O.Box 7700, 1117 ZL Schiphol-Oost, Ne
HUBER, E. Mr	Messerschmitt-Bolkow-Blohm GmbH, Unternehmensbereich Apparate, AE432 P.B. 801160, 8 Munchen 80, Ge
HUGHES, T. Mr	Software Engineer, Smiths Induatries Ltd, Bishop Cleeve, Cheltenham, Glos, UK
HUNT, G.H. Dr	ADXR/E, RAE Farnborough, Farnborough, Hants GU14 6TD, UK
JACK, C.D. Mr	Rolls-Royce Ltd, P.O. Box 31, Derby, UK
JACKSON, P.W. Lt Col.	NDHQ/DA vsSE 3, Ottowa, Ontario K1A 0K4, Ca
JACOBSEN, M. Mr	AEG Telefunken, A 14 V3, Postfach 1730, D-7900 Ulm, Ge
JANSSEN, J.M. Mr	RNLAF/DMKLU/AVL, Postbus 90501, 2509 LM 's-Gravenhage, Ne
JARSCH, V. Mr	SHAPE Technical Centre, P.O. Box 174, 2051 CD The Hague, Ne
JONES, C. Mr	MOD (PE) A&AEE Boscombe Down, Salisbury, Wilts SP4 0JF, UK
JONES, E.P. Mr	Smiths Industries Aerospace & Defence Systems Company, Bishops Cleeve, Cheltenham, UK
JORDAN, D. Mr	Marconi Avionic Systems Ltd, Elstree Way, Borehamwood, Herts, UK
JURKOWSKI, D.M. Major	Project Management Office CF-18, NDHQ, 101 Colonel By Drive K1A OK2, Ca
KENSEY, T.F. Mr	EASAMS Ltd, Hq. Lyon Way, Frimley Rd, Camberley, Surrey, UK
KEPPNER, K. Mr	Standard Elektrik Lorenz AG, Abt/GLS/TS, Hellmuth-Hirth, Sn 42, 7000 Stuttgart 40, Ge
KESSELMAN, M. Mr	SHAPE Technical Centre, P.O. Box 174, 2501 CD The Hague, Ne
KLEINSCHMIDT, M. Dr	Fa. LITEF, Postfach 774, 7800 Freiburg i. Br., Ge
KLEMM, R. Dr	FGAN-FFM, D 5307 Watchberg-Werthhoven, Ge
KLENK, H. Dr rer. nat.	MBB Ottobrun-FE 411, Postfach 80 11 60, 8000 Munchen 80, Ge
KRAMER, J.F. L/Cdr	ADA Joint Program Office, Suite 1210, Ballston Tower II, 801 North Randolph Street, Arlington, VA 22203, US
KRYN, R. Mr	NLR, Anthony Fokkerweg 2, 1059 CM Amsterdam, Ne
KUNY, W. Mr	MBB Ottobrunn FE 4, Postfach 80 11 60, 8000 Munchen 80, Ge
KUNZ, J. Mr	Luftwaffenamt, Postfach 902 500/501/14, 5000 Koln 90, Ge
LAWES, B.R. Mr	Smiths Industries, Aerospace and Defence Systems Company, Winchester Rd, Basingstoke, Hants RG22 6HP, UK
LeROY, G.A. Mr	RNLAF/DMKLU/AVL, Postbus 90501, 2509 LM 's-Gravenhage, Ne
LeCOQ, M. Mrs	3 Allee de la Jointe Genete, 91190 Gif au Yvette, Fr
LEJEUNE, G. Capt.	Etat-Major Force Aerienne VDT/B, Quartier Reine Elisabeth, rue d'Evere 1, B 1140 Bruxelles, Be
LEUTSCHER, A.J. Mr	Physics Laboratory TNO, P.O.Box 96864, 2509 JG The Hague, Ne
LIESHOUT, P.L.J. van Mr	Physics Laboratory TNO, P.O.Box 96864, 2509 JG The Hague, Ne
LONGINOTTI, E. Mr	Costruz. Aeronautiche, G. AGUSTA S.p.A., 21017 Cascina Costa di Samarate, Varese, It
LOONEY, M.J. Mr	PE, MOD, ASWE, Portsdown, Cosham, Portsmouth PO6 4AA, UK

LUSCHNITZ, W. Dipl. Ing.	AEG-Telefunken, Sedanstrasse 10, D-7900 Ulm/Donau, Ge
MacKINTOSH, I.W. Mr	Royal Signals and Radar Establishment, St Andrews Rd, Malvern, Worcs., UK
MacPHERSON, R.W. Dr	National Defence Headquarters, CRAD/DSP-3, 101 Colonel By Drive, Ottawa, Ontario, K1A OK2, Ca
MADERNA, F. Dr	CISE, DOB 12081, 20134 Milano, It
MADSEN, T. Mr	NDRE, Division for Electronics, P.O.Box 25, 2007 Kjeller, No
MALCOLM, R. Chief Eng.	Marconi Avionic Systems Ltd, Elstree Way, Borehamwood, Herts, UK
MANDERS, P.J. Mr	National Aerospace Laboratory NLR, Anthony Fokkerweg 2, 1059 CM Amsterdam, Ne
MARAS, A. Ing.	Via Tiburtina 1210, Roma, It
MARIEN, J. Mr	rue Toussaint Catros, 33160 Le Haillan, Fr
MARQUART, D. Dr	FGAN IFE, Konigstrasse 2, 5307 Wachtberg-Werthhoven, Ge
MARTIN, D.J. Dr	Flight Controls Division, Marconi Avionics Ltd, Airport Works, Rochester, Kent, UK
MASCARENHAS, J.M.B. Major	G. Dirrecao do Servico de Telecom da FA, Rua Escola de Exercito, Lisbon, Po
McTIGUE, T.V. Mr	McDonnell Aircraft (D312/271B), P.O.Box 516, St Louis, Mo 63166, US
MEURS, W.J. van Mr	Javastraat 43, 3531 PN Utrecht, Ne
MEYERRATHKEN, J.J.M. Mr	NLR, Anthony Fokkerweg 2, 1059 CM Amsterdam, Ne
MIENTJES, G.J.A. Mr	Fokker B.V. Certificatie, Posthus 7600, 1117 ZV Schiphol, Ne
MIGNEAULT, G.E. Mr	FTSB/FCSD MS 477, NASA Langley Research Center, Hampton, VA 23665, US
MIRAILLES, B. Mr	STTE/PNI, 129 rue de la Convention, 75731 Paris Cedex 15, Fr
MITCHELL, R.O. Mr	Code 501, Software and Computer Directorate, Naval Air Development Center, Warminster, PA 18974, US
MOLLE, Mr	Societe Crouzet, 25, rue Jules Vedrines, 26027 Valence Cedex, Fr
MONTCHEUIL, J. De ICA	DTEN/STEN, 4 Avenue de la Porte d'Issy, 75025 Paris, Fr
MOREAU, C. ICA	STTE/PNI, 129 rue de la Convention, 75731 Paris Cedex 15, Fr
MOWAT, A.R. Mr	Ferranti Ltd, Silverknowles, Ferry Rd, Edinburgh EH4 4AD, Scotland, UK
NAYLOR, P. Mr	Westland Helicopters Ltd, Yeovil, Somerset, UK
NEGRE, J.M. Ing.	37 Ave. Louis Breguet, 78146 Velizy, Fr
NOUVELLON, J. Mr	SAGEM, Chaussee Jules Cesar, 95 Osny, Fr
O'HARA, D.W. Dr	Smiths Industries Aerospace and Defence Systems Company, Evesham Rd, Bishops Cleeve, Cheltenham, Glos, GL52 4SF UK
OXMAN, S. Mr	SHAPE Technical Center, P.O. Box 174, 2501 CD The Hauge, Ne
PANKHURST, R.V. Mr	MOD (PE) A&AEE Boscombe Down, Salisbury, Wilts SP4 0JF, UK
PEARCE, G. Mr	British Aerospace plc, Richmond Rd, Kingston upon Thames, UK
PEDERSEN, J.T. Dr	A/S Kongsberg Vapenfabrikk, Posthox 25, N-3601 Kongsberg, No
PELISSERO, R. Dr Ing.	Aeritalia, Gruppo Equipaggiamenti, 10072 Caselle (Torino), It
PIJPERS, E.W. Mr	NLR, Anthony Fokkerweg 2, 1059 CM Amsterdam, Ne
PORSIUS, D. Mr	KLM/NV, Aircraft Systems Group, SPL/CI, P.O.Box 7700, 1117 ZL Schiphol-Oost, Ne
POST, J.A. Mr	CB-BS, P.O.Box 7600, 1117 ZJ Schiphol, Ne
PRICE, C.P. Mr	British Aerospace PLC, Aircraft Group, Warton Division, Warton Aerodrome, Preston PR4 1AX, UK
PULFORD, K.J. Chief Eng.	Marconi Avionic Systems Ltd, Elstreee Way, Borehamwood, Herts, UK
PUTZKI, R. Dr	Clo SCS, Oehlecherring 40, 2000 Hamburg 62, Ge
REITZ, S. Mr	Messerschmitt-Bolkow-Blohm GmbH, AE135, P.B. 801160, 8 Munchen 80, Ge
RUNNALLS, A.R. Dr	Marconi Avionics Ltd, Airport Works, Rochester, Kent, UK
SALMON, J.P. Capt.	Etat-Major Force Aerienne VDT/B, Quartier Reine Elisabeth, rue d'Evere 1, B 1140 Bruxelles, Be
SANDNER, N. Dr	Fu LITEF, Lorracher Strasse 18 POB 774, 7800 Frieburg, Ge

SANGLARD, A. Mr	Societe Crouzet, 25 rue Jules Vedrines, 26027 Valence Cedex, Fr
SANSON, P. Mr	Electronique Aerospatiale, B.P. 51, 93350 Le Bourget Principal, Fr
SCHAAFF, H. Mr	Bundesakademie fur Wehrverwaltung und Wehrtechnik, Seckenheimer Landstr. 8–10, 6800 Mannheim 25, Ge
SCHIRLE, P. Mr	Breguet Aviation, 78, Quai Carnot, 92214 St Cloud, Fr
SCHNEIDEWIND, N. Prof.	Code 54Ss, Naval Postgraduate School, Monterey, CA 93940, US
SCHROEPL, H. Dr	Teldix GmbH, Grenzhoferweg 36 POB 105 608, 6900 Heidelberg, Ge
SERGEANT, W.A. Capt.	Software Development Unit, Canadian Forces Base Greenwood, Greenwood, Nova Scotia POB 1NO, Ca
SHAW, R. Mr	AFWAL/AAWP, Wright-Patterson AFB, OH 45433, US
SIMPSON, H.R. Dr	British Aerospace Public Ltd Co., Dynamics Group, Stevenage Division, Six Hills Way, Stevenage, Herts SG1 2DA, UK
SIROT, B. Mr	6 Ave Auguste Dutreux, 78120 La Celle St Cloud, Fr
SKORCZEWSKI, L. Mr	British Aerospace plc, Aircraft Gp, Warton Division, Warton Aerodrome, Preston, Lancs PR4 1AX, UK
STUEBING, H.G. Mr	Code 50 C, Software & Computer Directorate, US Naval Air Development Center, Warminster, PA 18974, US
SUDWORTH, J.P. Dr	Royal Aircraft Est., Farnborough, Hants, UK
SUMMERBELL, F. Mr	British Aerospace plc, Richmond Rd, Kingston-upon-Thames, Surrey, UK
SUNDBERG, G. Mr	Tracor, Inc., 65 West Street Road, C-206, Warminster, PA 18974, US
SUNDSTROM, D.E. Dr	General Dynamics, P.O.Box 748, MZ 2451, Fort Worth, Texas 76101, US
SWANN, T.G. Dr	Marconi Avionics Ltd, Elstree Way, Borehamwood, Herts, UK
SYVERTSEN, A. Mr	RNOAF/MATERIEL Command, PO Box 10, 2007 Kjeller, No
TAILLIBERT, P. Mr	Electronique Serge Dassault, 55 quai Carnot, 92214 St Cloud, Fr
TANNER, G.F. Mr	Rolls-Royce Ltd, P.O.Box 3, Filton, Bristol, BS12 7QE, UK
TIMMERS, H. Ir.	Nat. Aerospace Lab., NLR, Ant. Fokkerweg 2, 1059 CM Amsterdam, Ne
TJOA, H.G. Mr	Fokker B.V., Postbus 7600, 1117 ZJ Schiphol-Oost, Ne
TUINENBURG, H.A. Mr	National Aerospace Research Lab., Anthony Fokkerweg 2, 1059 CM Amsterdam, Ne
TURNER, G.A. Mr	Dowty Rotol Ltd, Cheltenham Rd, Gloucester, UK
VAGNARELLI, F. L/Col.	Aeronautica Militare Italiana, Officio del Delegato Nazionale AGARD, 3 Ple Adenauer, 00144 Roma/EUR, It
VALLBRACHT, G. Mr	SHAPE Technical Centre, P.O.Box 174, 2501 CD The Hague, Ne
VAQUIER, Mme	SFENA, Aerodrome de Villacoublay, 78141 Villacoublay, Fr
VEEZE, J.P. Mr	Rijksluchtvaartdienst, P.O.Box 7555, 1117 ZH Schiphol, Ne
VERBEKE, A. Mr	Societe Crouzet, 25 rue Jules Vedrines, 26027 Valence Cedex, Fr
VICKERY, B.L. Dr	Ferranti plc, Navigation Systems Dept., Silverknowles, Edinburgh EH4 4AD, Scotland, UK
VOGEL, M. Dr	DFVLR, D8031 Oberpfaffenhofen, Ge
VOGELPOEL, A.A.F. Mr	KLM Schiphol East, Dept. SPL/CI, P.O. Box 7700, Ne
VOGL, W. Mr	Messerschmitt-Bolkow-Blohm GmbH, Military Aircraft Division FE52, P.B. 801160, 8 Munchen 80, Ge
VOLES, R. Dr	Thorn EMI Electronics Ltd, 135 Blyth Road, Hayes, Middlesex UB3 1BP, UK
VRIES, J.C.Z. de Mr	Fokker BV, P.O.Box 7600, 1117 ZJ Schiphol Oost, Ne
WAGGOTT, J.G. Capt.	National Defence Headquarters, Ottawa, Ontario K1A OK2, Attention DAEM 2-2-2, Ca
WARE, W. Dr	Rand Corp., 1700 Main St, Santa Monica, CA 90406, US
WASCH, R.B.A. Mr	NLR, P.O.Box 90502, 1006 BM Amsterdam, Ne
WEISS, D.M. Mr	Code 7592, US Naval Research Laboratory, Washington, D.C. 20375, US
WEISS, M.T. Dr	The Aerospace Corporation, P.O.Box 92957, Los Angeles, CA 90009, US
WESTBROOK, R.E. Mr	Code 31903, Naval Weapons Center, China Lake, CA 93555, US
WESTCOTT, P. Mr	Westland Helicopters Ltd, Yeovil, Somerset, UK

WHALLEY, J. Mr	British Aerospace PLC, Woodford Aerodrome, Stockport, Cheshire SK7 1QR, UK
WIEMER, W. Dr	Messerschmitt-Bolkow-Blohm GmbH, UA AE135, Postfach 801149, D-8000 Munchen 80, Ge
WILKINSON, A.H. Mr	Westland Helicopters Ltd,Yeovil, Somerset, UK
WILLIAMSON, A.E. Mr	Marconi Avionics, Airport Works, Rochester Kent ME1 2XX
WILSON, R.E. Mr	Marconi Avionic Systems Ltd, Elstree Way Borehamwood, Herts,UK
ZALUSKI, E. Mr	Director of Air Requirements, 101 Colonel By Drive, Ottawa, Ontario K1A OK2, Ca
ZIEGLER, J. Mr	ERNO Raumfahrttechnik GmbH, Hunefeldstrasse 1 5 Abt. KP5, 2800 Bremen 1 55, Ge

REPORT DOCUMENTATION PAGE

1. Recipient's Reference	2. Originator's Reference	3. Further Reference	4. Security Classification of Document
	AGARD-CP-330	ISBN 92-835-0323-6	UNCLASSIFIED

5. Originator Advisory Group for Aerospace Research and Development
North Atlantic Treaty Organization
7 rue Ancelle, 92200 Neuilly sur Seine, France

6. Title

SOFTWARE FOR AVIONICS

7. Presented at

the Avionics Panel's 44th Symposium held at the Atlantic Hotel, The Hague-Kijkduin, Netherlands, 6–10 September 1982.

8. Author(s)/Editor(s)	9. Date
Various	January 1983
10. Author's/Editor's Address	**11. Pages**
Various	472

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

Avionic system software	Software technology
Avionic embedded computer resources	Ada®
Avionic software life cycle considerations	MASCOT
Avionic software design and development	Avionic software standards

15. Abstract

These Proceedings include the papers and discussions presented at the AGARD Avionics Panel Symposium on "Software for Avionics" held in The Hague-Kijkduin, Netherlands, in September 1982.

The last decade has brought about an explosion-like progress in electronic data processing technology. Computer capabilities have increased while computer size and costs have decreased. The military need for high performance navigation, weapon delivery, flight control, and defensive avionic systems has lead to highly integrated digitized avionic systems. The costs and complexity of the associated software are increasing rapidly.
This symposium addressed the impact of software on avionic systems.

The symposium provided an overview of the software elements associated with embedded computer resources and addressed current issues related to software requirements, design, development, verification and validation. Software life-cycle considerations in the areas of cost, management and maintenance were identified. Trends in software technology and key features contributing to more efficient and more economical software programs in the NATO countries were discussed.

Included with the 44 papers presented are the discussions that followed the presentations of papers. A Technical Evaluation Report has been included by the Editor to summarize and highlight the results of the symposium.

AGARD Conference Proceedings No.330
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE FOR AVIONICS
Published January 1983
472 pages

These Proceedings include the papers and discussions presented at the AGARD Avionics Panel Symposium on "Software for Avionics" held in The Hague-Kijkduin, Netherlands, in September 1982.

The last decade has brought about an explosion-like progress in electronic data processing technology. Computer capabilities have increased while computer size and costs have decreased. The military need for high

P.T.O

AGARD-CP-330

Avionic system software
Avionic embedded computer resources
Avionic software life cycle considerations
Avionic software design and development
Software technology
Ada®
MASCOT
Avionic software standards

AGARD Conference Proceedings No.330
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE FOR AVIONICS
Published January 1983
472 pages

These Proceedings include the papers and discussions presented at the AGARD Avionics Panel Symposium on "Software for Avionics" held in The Hague-Kijkduin, Netherlands, in September 1982.

The last decade has brought about an explosion-like progress in electronic data processing technology. Computer capabilities have increased while computer size and costs have decreased. The military need for high

P.T.O

AGARD-CP-330

Avionic system software
Avionic embedded computer resources
Avionic software life cycle considerations
Avionic software design and development
Software technology
Ada®
MASCOT
Avionic software standards

AGARD Conference Proceedings No.330
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE FOR AVIONICS
Published January 1983
472 pages

These Proceedings include the papers and discussions presented at the AGARD Avionics Panel Symposium on "Software for Avionics" held in The Hague-Kijkduin, Netherlands, in September 1982.

The last decade has brought about an explosion-like progress in electronic data processing technology. Computer capabilities have increased while computer size and costs have decreased. The military need for high

P.T.O

AGARD-CP-330

Avionic system software
Avionic embedded computer resources
Avionic software life cycle considerations
Avionic software design and development
Software technology
Ada®
MASCOT
Avionic software standards

AGARD Conference Proceedings No.330
Advisory Group for Aerospace Research and Development, NATO
SOFTWARE FOR AVIONICS
Published January 1983
472 pages

These Proceedings include the papers and discussions presented at the AGARD Avionics Panel Symposium on "Software for Avionics" held in The Hague-Kijkduin, Netherlands, in September 1982.

The last decade has brought about an explosion-like progress in electronic data processing technology. Computer capabilities have increased while computer size and costs have decreased. The military need for high

P.T.O

AGARD-CP-330

Avionic system software
Avionic embedded computer resources
Avionic software life cycle considerations
Avionic software design and development
Software technology
Ada®
MASCOT
Avionic software standards

performance navigation, weapon delivery, flight control, and defensive avionic systems has lead to highly integrated digitized avionic systems. The costs and complexity of the associated software are increasing rapidly.
This symposium addressed the impact of software on avionic systems.

The symposium provided an overview of the software elements associated with embedded computer resources and addressed current issues related to software requirements, design, development, verification and validation. Software life-cycle considerations in the areas of cost, management and maintenance were identified. Trends in software technology and key features contributing to more efficient and more economical software programs in the NATO countries were discussed.

Included with the 44 papers presented are the discussions that followed the presentations of papers. A Technical Evaluation Report has been included by the Editor to summarize and highlight the results of the symposium.

ISBN 92-835-0323-6

performance navigation, weapon delivery, flight control, and defensive avionic systems has lead to highly integrated digitized avionic systems. The costs and complexity of the associated software are increasing rapidly.
This symposium addressed the impact of software on avionic systems.

The symposium provided an overview of the software elements associated with embedded computer resources and addressed current issues related to software requirements, design, development, verification and validation. Software life-cycle considerations in the areas of cost, management and maintenance were identified. Trends in software technology and key features contributing to more efficient and more economical software programs in the NATO countries were discussed.

Included with the 44 papers presented are the discussions that followed the presentations of papers. A Technical Evaluation Report has been included by the Editor to summarize and highlight the results of the symposium.

ISBN 92-835-0323-6

performance navigation, weapon delivery, flight control, and defensive avionic systems has lead to highly integrated digitized avionic systems. The costs and complexity of the associated software are increasing rapidly.
This symposium addressed the impact of software on avionic systems.

The symposium provided an overview of the software elements associated with embedded computer resources and addressed current issues related to software requirements, design, development, verification and validation. Software life-cycle considerations in the areas of cost, management and maintenance were identified. Trends in software technology and key features contributing to more efficient and more economical software programs in the NATO countries were discussed.

Included with the 44 papers presented are the discussions that followed the presentations of papers. A Technical Evaluation Report has been included by the Editor to summarize and highlight the results of the symposium.

ISBN 92-835-0323-6

performance navigation, weapon delivery, flight control, and defensive avionic systems has lead to highly integrated digitized avionic systems. The costs and complexity of the associated software are increasing rapidly.
This symposium addressed the impact of software on avionic systems.

The symposium provided an overview of the software elements associated with embedded computer resources and addressed current issues related to software requirements, design, development, verification and validation. Software life-cycle considerations in the areas of cost, management and maintenance were identified. Trends in software technology and key features contributing to more efficient and more economical software programs in the NATO countries were discussed.

Included with the 44 papers presented are the discussions that followed the presentations of papers. A Technical Evaluation Report has been included by the Editor to summarize and highlight the results of the symposium.

ISBN 92-835-0323-6

AGARD
NATO OTAN
7 RUE ANCELLE · 92200 NEUILLY-SUR-SEINE
FRANCE
Telephone 745.08.10 · Telex 610176

DISTRIBUTION OF UNCLASSIFIED AGARD PUBLICATIONS

AGARD does NOT hold stocks of AGARD publications at the above address for general distribution. Initial distribution of AGARD publications is made to AGARD Member Nations through the following National Distribution Centres. Further copies are sometimes available from these Centres, but if not may be purchased in Microfiche or Photocopy form from the Purchase Agencies listed below.

NATIONAL DISTRIBUTION CENTRES

BELGIUM
Coordonnateur AGARD – VSL
Etat-Major de la Force Aérienne
Quartier Reine Elisabeth
Rue d'Evere, 1140 Bruxelles

CANADA
Defence Science Information Services
Department of National Defence
Ottawa, Ontario K1A 0K2

DENMARK
Danish Defence Research Board
Østerbrogades Kaserne
Copenhagen Ø

FRANCE
O.N.E.R.A. (Direction)
29 Avenue de la Division Leclerc
92320 Châtillon sous Bagneux

GERMANY
Fachinformationszentrum Energie,
Physik, Mathematik GmbH
Kernforschungszentrum
D-7514 Eggenstein-Leopoldshafen 2

GREECE
Hellenic Air Force General Staff
Research and Development Directorate
Holargos, Athens

ICELAND
Director of Aviation
c/o Flugrad
Reykjavik

ITALY
Aeronautica Militare
Ufficio del Delegato Nazionale all'AGARD
3, Piazzale Adenauer
Roma/EUR

LUXEMBOURG
See Belgium

NETHERLANDS
Netherlands Delegation to AGARD
National Aerospace Laboratory, NLR
P.O. Box 126
2600 A.C. Delft

NORWAY
Norwegian Defence Research Establishment
Main Library
P.O. Box 25
N-2007 Kjeller

PORTUGAL
Direcção do Serviço de Material
da Força Aerea
Rua da Escola Politécnica 42
Lisboa
Attn: AGARD National Delegate

TURKEY
Department of Research and Development (ARGE)
Ministry of National Defence, Ankara

UNITED KINGDOM
Defence Research Information Centre
Station Square House
St. Mary Cray
Orpington, Kent BR5 3RE

UNITED STATES
National Aeronautics and Space Administration (NASA)
Langley Field, Virginia 23365
Attn: Report Distribution and Storage Unit

THE UNITED STATES NATIONAL DISTRIBUTION CENTRE (NASA) DOES NOT HOLD STOCKS OF AGARD PUBLICATIONS, AND APPLICATIONS FOR COPIES SHOULD BE MADE DIRECT TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS) AT THE ADDRESS BELOW.

PURCHASE AGENCIES

Microfiche or Photocopy
National Technical
Information Service (NTIS)
5285 Port Royal Road
Springfield
Virginia 22161, USA

Microfiche
Space Documentation Service
European Space Agency
10, rue Mario Nikis
75015 Paris, France

Microfiche or Photocopy
British Library Lending
Division
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
England

Requests for microfiche or photocopies of AGARD documents should include the AGARD serial number, title, author or editor, and publication date. Requests to NTIS should include the NASA accession report number. Full bibliographical references and abstracts of AGARD publications are given in the following journals:

Scientific and Technical Aerospace Reports (STAR)
published by NASA Scientific and Technical
Information Facility
Post Office Box 8757
Baltimore/Washington International Airport
Maryland 21240, USA

Government Reports Announcements (GRA)
published by the National Technical
Information Services, Springfield
Virginia 22161, USA

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

ISBN 92-835-0323-6

END

DATE
FILMED

5-83

DTIC